SAS® Viya® Workbench Machine Learning Procedures

SAS® Documentation
September 25, 2026

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2024. *SAS® Viya® Workbench: Machine Learning Procedures*. Cary, NC: SAS Institute Inc.

SAS® Viya® Workbench: Machine Learning Procedures

Copyright © 2024, SAS Institute Inc., Cary, NC, USA

All Rights Reserved. Produced in the United States of America.

For a hard-copy book: No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, or otherwise, without the prior written permission of the publisher, SAS Institute Inc.

For a web download or e-book: Your use of this publication shall be governed by the terms established by the vendor at the time you acquire this publication.

The scanning, uploading, and distribution of this book via the internet or any other means without the permission of the publisher is illegal and punishable by law. Please purchase only authorized electronic editions and do not participate in or encourage electronic piracy of copyrighted materials. Your support of others' rights is appreciated.

September 2026

SAS® and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration.

Other brand and product names are trademarks of their respective companies.

SAS software may be provided with certain third-party software, including but not limited to open source software, which is licensed under its applicable third-party software license agreement. For license information about third-party software distributed with SAS software, refer to Third-Party Software Reference | SAS Support.

v_001-P1:vwbcasml

Contents

Chapter 1. Introduction . . . 1
Chapter 2. Shared Concepts . . . 5
Chapter 3. ASSESSBIAS Procedure . . . 29
Chapter 4. ASTORE Procedure . . . 47
Chapter 5. BNET Procedure . . . 89
Chapter 6. DYNBNET Procedure . . . 133
Chapter 7. FACTMAC Procedure . . . 167
Chapter 8. FASTKNN Procedure . . . 193
Chapter 9. FISM Procedure . . . 205
Chapter 10. FOREST Procedure . . . 213
Chapter 11. FPCA Procedure . . . 275
Chapter 12. FPCASCORE Procedure . . . 291
Chapter 13. GMM Procedure . . . 307
Chapter 14. GPCLASS Procedure . . . 331
Chapter 15. GPREG Procedure . . . 359
Chapter 16. GRADBOOST Procedure . . . 391
Chapter 17. GVARCLUS Procedure . . . 453
Chapter 18. KPCA Procedure . . . 475
Chapter 19. LIGHTGRADBOOST Procedure . . . 511
Chapter 20. LIME Procedure . . . 537
Chapter 21. MBANALYSIS Procedure . . . 555
Chapter 22. MTLEARN Procedure . . . 569
Chapter 23. MWPCA Procedure . . . 597
Chapter 24. NNET Procedure . . . 621
Chapter 25. NOMINALDR Procedure . . . 667
Chapter 26. PARTIALDEPEND Procedure . . . 685
Chapter 27. RECASSESS Procedure . . . 705
Chapter 28. RECENGINE Procedure . . . 717
Chapter 29. REGISTERMODEL Procedure . . . 757
Chapter 30. RPCA Procedure . . . 779
Chapter 31. SHAPLEY Procedure . . . 807
Chapter 32. SPARSEML Procedure . . . 825
Chapter 33. STYLEGAN Procedure . . . 841
Chapter 34. SVDD Procedure . . . 863
Chapter 35. SVMACHINE Procedure . . . 931
Chapter 36. TABULARGAN Procedure . . . 983
Chapter 37. TSNE Procedure . . . 1007

Chapter 1
Introduction

Contents

Overview of Machine Learning Procedures **1**
Experimental Software **2**
About This Book **2**
Chapter Organization 2
Typographical Conventions 3
Options Used in Examples 3
Where to Turn for More Information **4**
Online Documentation 4
SAS Technical Support Services 4

Overview of Machine Learning Procedures

This book describes the data mining and machine learning procedures that are available in SAS Viya Workbench. These procedures provide data mining and machine learning algorithms that have been specially developed to take advantage of the environment that SAS Viya Workbench provides. Supervised learning methods that are available include forest and gradient boosting models, neural networks, support vector machines, and factorization machines. Procedures for scoring via an analytic store and for text mining are also included.

In addition to the data mining and machine learning procedures described in this book, SAS Viya Workbench provides procedures for sampling, data exploration, clustering, dimension reduction, model assessment, and additional supervised learning, which are described in *SAS Visual Statistics: Procedures for SAS Viya Workbench*.

Experimental Software

Experimental software is sometimes included as part of a production-release product. It is provided to (sometimes targeted) customers in order to obtain feedback. All experimental uses are marked Experimental in this document. Whenever an experimental procedure, statement, or option is used, a message is printed to the SAS log to indicate that it is experimental.

The design and syntax of experimental software might change before its status becomes production. Experimental software has been tested prior to release, but it has not necessarily been tested to production-quality standards and so should be used with care.

About This Book

This book assumes that you are familiar with Base SAS software and with the books *SAS Programmers Guide: Essentials* and *Base SAS Procedures Guide*. It also assumes that you are familiar with basic SAS System concepts, such as using the DATA step to create SAS data sets and using Base SAS procedures (such as the PRINT and SORT procedures) to manipulate SAS data sets.

Chapter Organization

This book is organized as follows:

Chapter 1, this chapter, provides an overview of the data mining and machine learning procedures that are available in SAS Viya Workbench, and it summarizes related information, products, and services.

Chapter 2 provides information about topics that are common to multiple procedures. Topics include how to use SAS Viya Workbench. This chapter also documents the CODE and PARTITION statements, which are used across a number of procedures.

Subsequent chapters describe the data mining and machine learning procedures. These chapters appear in alphabetical order by procedure name and are organized as follows:

- The “Overview” section briefly describes the analysis provided by the procedure.
- The “Getting Started” section provides a quick introduction to the procedure through a simple example.
- The “Syntax” section describes the SAS statements and options that control the procedure.
- The “Details” section discusses methodology and other topics, such as ODS tables.
- The “Examples” section contains examples that use the procedure.
- The “References” section contains references for the methodology.

Typographical Conventions

This book uses several type styles for presenting information. The following list explains the meaning of the typographical conventions used in this book:

roman	is the standard type style used for most text.
UPPERCASE ROMAN	is used for SAS statements, options, and other SAS language elements when they appear in text. However, you can enter these elements in your own SAS programs in lowercase, uppercase, or a mixture of the two.
UPPERCASE BOLD	is used in the "Syntax" sections' initial lists of SAS statements and options.
oblique	is used in the syntax definitions and in text to represent arguments for which you supply a value.
VariableName	is used for the names of variables and data sets when they appear in text.
bold	is used for matrices and vectors.
italic	is used for terms that are defined in text, for emphasis, and for references to publications.
`monospace bold`	is used for example code. In most cases, this book uses lowercase type for SAS code.

Options Used in Examples

The HTMLBLUE style is used to create the graphs and the HTML tables that appear in the online documentation. The PEARLJ style is used to create the PDF tables that appear in the documentation. A style template controls stylistic elements such as colors, fonts, and presentation attributes. You can specify a style template for an HTML ODS destination as follows:

```
ods html style=HTMLBlue;
```

You can also specify a style template for a PDF ODS destination as follows:

```
ods pdf style=PearlJ;
```

Most of the PDF tables are produced by using the following SAS System option:

```
options papersize=(6.5in 9in);
```

If you run the examples, you might get slightly different output. This is a function of the SAS System options that are used and the precision that your computer uses for floating-point calculations.

Where to Turn for More Information

Online Documentation

You can access the documentation by going to http://support.sas.com/documentation.

SAS Technical Support Services

The SAS Technical Support staff is available to respond to problems and answer technical questions regarding the use of procedures in this book. Go to http://support.sas.com/techsup for more information.

Chapter 2
Shared Concepts

Contents

Introduction to Shared Concepts **5**
Using SAS Viya Workbench 6
Syntax Common to Machine Learning Procedures **6**
AUTOTUNE Statement 6
CODE Statement 20
Loading DATA Step Scoring Code into S3 21
PARTITION Statement 21
SAVESTATE Statement 22
Details for Machine Learning Procedures **23**
Using Validation and Test Data 23
Multithreading 24
Hyperparameter Tuning 24
Data Partitioning 25
Hyperparameter Tuning Process 25
Determining the Number of Parallel Evaluations 27
Lookup Table and History Table 28
References **28**

Introduction to Shared Concepts

This book describes data mining and machine learning procedures that run in SAS Viya Workbench.

The section "Syntax Common to Machine Learning Procedures" on page 6 describes the common syntax elements that are supported by some of the procedures in this book. In some cases, individual procedures implement these common elements in slightly different ways. When this occurs, the differences are described in the respective procedure chapters.

The section "Details for Machine Learning Procedures" on page 23 provides details that are common to some of the procedures in this book.

Using SAS Viya Workbench

SAS Viya Workbench has greatly simplified the SAS programming experience by offering a syntax style nearly identical to that of SAS 9 and earlier releases. However, there are minor exceptions. In the past, SAS Viya procedures required the use of a CAS engine libref. This engine served multiple purposes, but it was primarily used to upload data to the CAS server and manage the data. Its additional purposes included support of new data types and improved performance for large data sets. No CAS server is needed for SAS Viya Workbench, because all computations are done in place for you in a single location.

In the following program, the LIBNAME statement creates a mylib libref that the examples in this chapter use to refer to your current or desired work directory:

```
libname mylib ".";
```

For more information, see *SAS Viya Workbench: Getting Started*.

Syntax Common to Machine Learning Procedures

AUTOTUNE Statement

AUTOTUNE < *options* > **;**

This section applies to the following procedures: BNET, FACTMAC, FOREST, GRADBOOST, MTLEARN, NNET, SEMISUPLEARN, SVMACHINE, and TSNE.

Table 2.1 summarizes the *options* you can specify in the AUTOTUNE statement. All options except the TUNINGPARAMETERS= option are described in detail in the list that follows Table 2.1. For more information about the TUNINGPARAMETERS= option, see the specific procedure chapter.

Table 2.1 AUTOTUNE Statement Options

Option	Description
APPENDLOOKUP	Specifies that the table specified in the HISTORYTABLE= option contain the rows from the table specified in the LOOKUPTABLE= option
CONSTRAINTS=	Specifies the constraints on the output metrics
EVALHISTORY=	Specifies how to report the evaluation history of the tuner
FOLDCOLUMN=	Specifies the column in the data table in which the cross validation fold value is indicated
FRACTION=	Specifies the fraction of observations to use for validation
HISTORYTABLE=	Specifies the table that contains the evaluation history
KFOLD=	Specifies the number of folds for *k*-fold cross validation

Table 2.1 *continued*

Option	Description
LIVEUPDATE	Specifies that the table specified in the HISTORYTABLE= option be updated at every evaluation
LOOKUPTABLE=	Specifies the table to use for evaluation lookup
MAXBAYES=	Specifies the maximum number of points in the kriging model
MAXEVALS=	Specifies the maximum number of evaluations
MAXITER=	Specifies the maximum number of iterations when SEARCHMETHOD=GA or SEARCHMETHOD=BAYESIAN
MAXTIME=	Specifies the maximum time for all iterations
MAXTRAINTIME=	Specifies the maximum time for a model training
NCONVITER=	Specifies the number of convergence iterations
NOGRIDSHUFFLE	Requests that the grid points not be shuffled
NOLOCALSEARCH	Disables local search optimization
NPARALLEL=	Specifies the number of parallel sessions
NSUBSESSIONWORKERS=	Specifies the number of workers in parallel sessions
OBJECTIVE=	Specifies the objective function
POPSIZE=	Specifies the population size when SEARCHMETHOD=GA or SEARCHMETHOD=BAYESIAN
SAMPLESIZE=	Specifies the sample size when SEARCHMETHOD=LHS or SEARCHMETHOD=RANDOM
SEARCHMETHOD=	Specifies the search method that the optimizer uses
SECONDOBJECTIVE=	Specifies the second objective to use for tuning
SELECTINITPOINT	Specifies that the tuner select the best evaluation from the lookup table
TARGETEVENT=	Specifies the target event for ROC-based calculations
TRAINFRACTION=	Specifies the fraction of observations to use for training
TUNINGPARAMETERS=	Specifies the custom tuning parameters
USEPARAMETERS=	Specifies how to handle the TUNINGPARAMETERS= option

APPENDLOOKUP

specifies that the rows from the table specified in the LOOKUPTABLE= option be appended to the output table that is created using the HISTORYTABLE= option.

This option is ignored if you do not specify both the HISTORYTABLE= and LOOKUPTABLE= options.

CONSTRAINTS=(*suboption* **| . . . | <** *suboption* **>)**

specifies constraints on output metrics that the tuner will use during tuning.

You can specify one or more of the following *suboptions*:

ASE (LB=*number* **UB=***number***)**
specifies a constraint on the average square error to be used while tuning the model.

You must specify one or both of the following suboptions:

LB=*number*
specifies the lower bound for the average square error.

UB=*number*
specifies the upper bound for the average square error.

F05 (LB=*number* **UB=***number***)**
specifies a constraint on the F0.5 coefficient to be used while tuning the model.

You must specify one or both of the following suboptions:

LB=*number*
specifies the lower bound for the F0.5 coefficient.

UB=*number*
specifies the upper bound for the F0.5 coefficient.

F1 (LB=*number* **UB=***number***)**
specifies a constraint on the F1 coefficient to be used while tuning the model.

You must specify one or both of the following suboptions:

LB=*number*
specifies the lower bound for the F1 coefficient.

UB=*number*
specifies the upper bound for the F1 coefficient.

FNR (LB=*number* **UB=***number***)**
specifies a constraint on the false negative rate to be used while tuning the model.

You must specify one or both of the following suboptions:

LB=*number*
specifies the lower bound for the false negative rate.

UB=*number*
specifies the upper bound for the false negative rate.

FPR (LB=*number* **UB=***number***)**
specifies a constraint on the false positive rate to be used while tuning the model.

You must specify one or both of the following suboptions:

LB=*number*
specifies the lower bound for the false positive rate.

UB=*number*
specifies the upper bound for the false positive rate.

GAMMA (LB=*number* **UB=***number***)**
specifies a constraint on the gamma coefficient to be used while tuning the model.

You must specify one or both of the following suboptions:

LB=*number*
specifies the lower bound for the gamma coefficient.

UB=*number*
specifies the upper bound for the gamma coefficient.

GINI (LB=*number* **UB=***number***)**
specifies a constraint on the Gini coefficient to be used while tuning the model.

You must specify one or both of the following suboptions:

LB=*number*
specifies the lower bound for the Gini coefficient.

UB=*number*
specifies the upper bound for the Gini coefficient.

KS (LB=*number* **UB=***number***)**
specifies a constraint on the Kolmogorov-Smirnov coefficient to be used while tuning the model.

You must specify one or both of the following suboptions:

LB=*number*
specifies the lower bound for the Kolmogorov-Smirnov coefficient.

UB=*number*
specifies the upper bound for the Kolmogorov-Smirnov coefficient.

MAE (LB=*number* **UB=***number***)**
specifies a constraint on the mean absolute error to be used while tuning the model.

You must specify one or both of the following suboptions:

LB=*number*
specifies the lower bound for the mean absolute error.

UB=*number*
specifies the upper bound for the mean absolute error.

MCE (LB=*number* **UB=***number***)**
specifies a constraint on the misclassification rate to be used while tuning the model.

You must specify one or both of the following suboptions:

LB=*number*
specifies the lower bound for the misclassification rate.

UB=*number*
specifies the upper bound for the misclassification rate.

MCLL (LB=*number* **UB=***number***)**
specifies a constraint on the multiclass log loss to be used while tuning the model.

You must specify one or both of the following suboptions:

LB=*number*
specifies the lower bound for the multiclass log loss.

UB=*number*
specifies the upper bound for the multiclass log loss.

MISC (LB=*number* **UB=***number***)**
specifies a constraint on the misclassification error percentage to be used while tuning the model.

You must specify one or both of the following suboptions:

LB=*number*
specifies the lower bound for the misclassification error percentage.

UB=*number*
specifies the upper bound for the misclassification error percentage.

MSE (LB=*number* **UB=***number***)**
specifies a constraint on the mean square error to be used while tuning the model.

You must specify one or both of the following suboptions:

LB=*number*
specifies the lower bound for the mean square error.

UB=*number*
specifies the upper bound for the mean square error.

MSLE (LB=*number* **UB=***number***)**
specifies a constraint on the mean square logarithmic error to be used while tuning the model.

You must specify one or both of the following suboptions:

LB=*number*
specifies the lower bound for the mean square logarithmic error.

UB=*number*
specifies the upper bound for the mean square logarithmic error.

RASE (LB=*number* **UB=***number***)**
specifies a constraint on the root average square error to be used while tuning the model.

You must specify one or both of the following suboptions:

LB=*number*
specifies the lower bound for the root average square error.

UB=*number*
specifies the upper bound for the root average square error.

RMAE (LB=*number* **UB=***number***)**
specifies a constraint on the root mean absolute error to be used while tuning the model.

You must specify one or both of the following suboptions:

LB=*number*
specifies the lower bound for the root mean absolute error.

UB=*number*
specifies the upper bound for the root mean absolute error.

RMSLE (LB=*number* **UB=***number***)**
specifies a constraint on the root mean square logarithmic error to be used while tuning the model.

You must specify one or both of the following suboptions:

LB=*number*
specifies the lower bound for the root mean square logarithmic error.

UB=*number*
specifies the upper bound for the root mean square logarithmic error.

SCORETIME (LB=*number* **UB=***number***)**
SCORINGTIME (LB=*number* **UB=***number***)**
specifies a constraint on the model scoring time to be used while tuning the model.

You must specify one or both of the following suboptions:

LB=*number*
specifies the lower bound for the model scoring time.

UB=*number*
specifies the upper bound for the model scoring time.

TAU (LB=*number* **UB=***number***)**
specifies a constraint on the tau coefficient to be used while tuning the model.

You must specify one or both of the following suboptions:

LB=*number*
specifies the lower bound for the tau coefficient.

UB=*number*
specifies the upper bound for the tau coefficient.

TNR (LB=*number* **UB=***number***)**
specifies a constraint on the true negative rate to be used while tuning the model.

You must specify one or both of the following suboptions:

LB=*number*
specifies the lower bound for the true negative rate.

UB=*number*
specifies the upper bound for the true negative rate.

TPR (LB=*number* **UB=***number***)**
specifies a constraint on the true positive rate to be used while tuning the model.

You must specify one or both of the following suboptions:

LB=*number*
specifies the lower bound for the true positive rate.

UB=*number*
specifies the upper bound for the true positive rate.

TRAINTIME (LB=*number* **UB=***number***)**
TRAININGTIME (LB=*number* **UB=***number***)**
specifies a constraint on the model training time to be used while tuning the model.

You must specify one or both of the following suboptions:

LB=*number*
specifies the lower bound for the model training time.

UB=*number*
specifies the upper bound for the model training time.

EVALHISTORY=ALL | LOG | NONE | TABLE

specifies how to report the evaluation history of the tuner.

You can specify one of the following values:

ALL reports each evaluation in the log and creates the EvaluationHistory ODS table.

LOG prints the following information to the log for each evaluation: evaluation number, objective value, best objective value up to that point, evaluation time, and elapsed time since the beginning of the tuning process.

NONE suppresses reporting of evaluations in the log and does not create the EvaluationHistory ODS table.

TABLE creates the EvaluationHistory ODS table, which contains all evaluated points. The table contains columns for the evaluation number, all tuning parameters, and the objective function value.

By default, EVALHISTORY=TABLE.

FOLDCOLUMN=*string*

specifies the name of the column in the data table in which the cross validation fold value is indicated for all observations. This column must contain integers between 1 and *number*, where *number* is the desired number of folds. This option has an effect similar to that of the KFOLD=*number* option. If you specify this option, the tuner uses the cross validation process to estimate the accuracy of the models. But instead of dividing the data into multiple folds (partitions), as the KFOLD=*number* option does, the tuner reads the fold numbers from the specified column in the data table and assigns each observation to the specified fold. In cross validation, each model evaluation requires *number* of training executions (on *number*–1 data folds) and *number* of scoring executions (on one holdout fold). Thus, the evaluation time is increased by approximately a factor of *number*. For small to medium data tables or for unbalanced data tables, cross validation typically provides a better representation of error across the entire data table (a better generalization error).

If you specify both this option and the KFOLD= option, the value of the KFOLD= option must match the number of folds in the fold column. If you omit this option, the KFOLD= option, and the FRACTION= option, then the default value of FRACTION=0.3 is used. If you specify a PARTITION statement, the validation partition defined in that statement is used, and the FOLDCOLUMN= option is ignored. You cannot specify the FOLDCOLUMN= option in combination with the FRACTION= option.

FRACTION=*number*

specifies the fraction of all data to use for validation, where *number* must be between 0.01 and 0.99, inclusive. If you specify this option, the tuner uses a single-partition validation for finding the objective value (validation error estimate). Using this option might not be advisable for small or unbalanced data tables, where the random assignment of the validation subset might not provide a good estimate of error. For large, balanced data tables, a single-partition validation is usually sufficient for estimating error; a single partition is more efficient than cross validation in terms of the total execution time.

If you specify a PARTITION statement, the validation partition defined in that statement is used, and this option is ignored. You cannot specify this option in combination with the KFOLD= option.

If you specify the TRAINFRACTION= option, then by default the value of the FRACTION= option is one minus the value of the TRAINFRACTION= option. Otherwise, by default, FRACTION=0.3.

HISTORYTABLE=*libref.data-table* **<(***data-set-options***)>**

specifies the data table created by the tuning process that contains the evaluation history. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the section "Using SAS Viya Workbench" on page 6.

The *data-set-options* are various options that enable you to control how the data table is created. For more information about data set options, see *SAS Data Set Options: Reference*. In the case of the HISTORYTABLE= option, the default value of the REPLACE= data set option is NO. The REPLACE= option specifies whether a new SAS data set can overwrite an existing data set that has the same name. You can override the default value by specifying REPLACE=YES.

KFOLD=*number*

specifies the number of folds (partitions) in the cross validation process, where *number* must be between 2 and 20, inclusive. If you specify this option, the tuner uses cross validation to find the objective value. In cross validation, each model evaluation requires *number* of training executions (on *number*−1 data folds) and *number* of scoring executions (on one holdout fold). Thus, the evaluation time is increased by approximately a factor of *number*. For small to medium data tables or for unbalanced data tables, cross validation typically provides a better representation of error across the entire data table (a better generalization error).

Another option that triggers cross validation is the FOLDCOLUMN= option. If you specify both the KFOLD= option and the FOLDCOLUMN= option, the value of the KFOLD= option must match the number of folds in the fold column.

If you omit the KFOLD= option, the FOLDCOLUMN= option, and the FRACTION= option, then the default value of FRACTION=0.3 is used. If you specify a PARTITION statement, the validation partition defined in that statement is used, and the KFOLD= option is ignored. You cannot specify the KFOLD= option in combination with the FRACTION= option.

LIVEUPDATE

specifies that the table specified in the HISTORYTABLE= option be updated at every evaluation. This option forces the table to be a promoted CAS table so that it can be seen by other sessions.

This option is ignored if you do not specify the HISTORYTABLE= option.

LOOKUPTABLE=*libref.data-table*

specifies the data table previously created by the tuning process that you want to use as the lookup table. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the input data table. For more information about this two-level name, see the section "Using SAS Viya Workbench" on page 6.

MAXBAYES=*number*

specifies the maximum number of points in the Kriging model. This option is honored only when SEARCHMETHOD=BAYESIAN and when *number* has a minimum value of 10.

By default, MAXBAYES=100.

MAXEVALS=*number*

specifies the maximum number of configuration evaluations allowed for the tuner, where *number* must be an integer greater than or equal to 3. When the *number* of evaluations is reached, the tuner terminates the search and returns the results. To produce a single objective function value (validation error estimate), each configuration evaluation requires either a single model training and scoring execution on a validation partition, or a number of training and scoring executions equal to the value of the KFOLD= option for cross validation. The MAXEVALS= option might lead to termination before the value of the MAXITER= option or the MAXTIME= option is reached.

By default, MAXEVALS=50.

MAXITER=*number*

specifies the maximum number of iterations of the optimization tuner, where *number* must be greater than or equal to 1. Each iteration normally involves a number of objective evaluations up to the value of the POPSIZE= option. The MAXITER= option is used only when SEARCHMETHOD=GA or SEARCHMETHOD=BAYESIAN; it is ignored when SEARCHMETHOD=RANDOM or SEARCHMETHOD=LHS. The MAXITER= option might lead to termination before the value of the MAXEVALS= option or the MAXTIME= option is reached.

By default, MAXITER=5.

MAXTIME=*number*

specifies the maximum time (in seconds) allowed for the tuner, where *number* must be greater than or equal to 1. When this value is reached, the tuner terminates the search and returns results. The actual run time for optimization might be longer because it includes the remaining time needed to finish the current evaluation. For long-running model training (large data tables), the actual run time might significantly exceed *number*. The MAXTIME= option might lead to termination before the value of the MAXEVALS= option or the MAXITER= option is reached.

By default, MAXTIME=36000.

MAXTRAINTIME=*number*

specifies the maximum time allowed for a single model train. The model train is terminated if it exceeds this time, and the objective value is set to missing.

By default, there is no maximum time allowed for a single train.

NCONVITER=*number*

specifies the *number* of convergence iterations after which tuning is terminated.

By default, NCONVITER=4.

NOGRIDSHUFFLE

requests that the points that are generated by the grid search method not be shuffled before execution.

NOLOCALSEARCH

disables local search optimization for the tuning process.

By default, local search optimization is enabled.

NPARALLEL=*number*

specifies the number of evaluations to be performed in parallel, where *number* must be greater than or equal to 0.

By default, NPARALLEL=0, which indicates that the value of *number* is determined as follows:

- If SEARCHMETHOD=GA or SEARCHMETHOD=BAYESIAN, then the number of parallel evaluations is equal to the value of the POPSIZE= option minus 1.
- If SEARCHMETHOD=LHS or SEARCHMETHOD=RANDOM, then the number of parallel evaluations is equal to the value of the SAMPLESIZE= option, with a maximum value of 32.

NSUBSESSIONWORKERS=*number*

specifies the number of workers to use in parallel subsessions. When alternative configurations are evaluated in parallel, a number of subsessions are created by the tuner, and each subsession uses multiple workers. By default, the number of workers that are used in a parallel subsession is determined as described in the section "Determining the Number of Parallel Evaluations" on page 27.

OBJECTIVE=*function*

specifies which measure of model performance the tuner uses as the objective function.

You can specify one of the following values for *function*. Some values can be specified only when the target variable is of a particular type, as shown in parentheses.

ASE	specifies average square error as the objective function.
AUC	specifies area under the curve as the objective function (nominal type only).
F05	specifies the F0.5 coefficient as the objective function (nominal type only).
F1	specifies the F1 coefficient as the objective function (nominal type only).
FNR	specifies the false negative rate as the objective function (nominal type only).
FPR	specifies the false positive rate as the objective function (nominal type only).
GAMMA	specifies the gamma coefficient as the objective function (nominal type only).
GINI	specifies the Gini coefficient as the objective function (nominal type only).
KS	specifies the Kolmogorov-Smirnov coefficient as the objective function (nominal type only).
MAE	specifies the mean absolute error as the objective function (interval type only).
MCE	specifies the misclassification rate as the objective function (nominal type only).
MCLL	specifies the multiclass log loss as the objective function (nominal type only).
MISC	specifies the misclassification error percentage as the objective function (nominal type only).
MSE	specifies the mean square error as the objective function (interval type only).
MSLE	specifies the mean square logarithmic error as the objective function (interval type only).
RASE	specifies the root average square error as the objective function.
RMAE	specifies the root mean absolute error as the objective function (interval type only).

RMSLE specifies the root mean square logarithmic error as the objective function (interval type only).

SCORETIME specifies the model scoring time as the objective function.

TAU specifies the tau coefficient as the objective function (nominal type only).

TNR specifies the true negative rate as the objective function (nominal type only).

TPR specifies the true positive rate as the objective function (nominal type only).

TRAINTIME specifies the model training time as the objective function.

By default, OBJECTIVE=MISC for nominal targets, and OBJECTIVE=MSE for interval targets.

POPSIZE=*number*

specifies the maximum population size, where population is the number of configuration evaluations in one iteration (population). The *number* must be greater than or equal to 1. In some cases, the tuner algorithm might generate a number of new configurations that is smaller than *number*. The POPSIZE= option is used only when SEARCHMETHOD=GA or SEARCHMETHOD=BAYESIAN; it is ignored when SEARCHMETHOD=RANDOM or SEARCHMETHOD=LHS.

By default, POPSIZE=10.

SAMPLESIZE=*number*

specifies the total number of evaluations, where *number* must be greater than or equal to 1. This option is only used when SEARCHMETHOD=RANDOM or SEARCHMETHOD=LHS; it is ignored when SEARCHMETHOD=GA or SEARCHMETHOD=BAYESIAN.

By default, SAMPLESIZE=50.

SEARCHMETHOD=BAYESIAN | GA | GRID | LHS | RANDOM

specifies the search method to use for tuning. You can specify the following values:

BAYESIAN builds a kriging surrogate model to approximate the objective value and uses this surrogate model to generate new alternative configurations at each iteration. The kriging model is continuously updated during the search process.

GA uses an initial Latin hypercube sample that seeds a genetic algorithm to generate a new population of alternative configurations at each iteration.

GRID uses all combinations of selected values of the hyperparameters.

LHS uses a Latin hypercube to generate a single sample of configurations that is uniform in each tuning parameter but random in combinations.

RANDOM generates a single sample of purely random configurations.

By default, SEARCHMETHOD=GA.

SECONDOBJECTIVE=*function*

specifies the type of the second objective to use for tuning.

You can specify one of the following values for *function*. Some values can be specified only when the target variable is of a particular type, as shown in parentheses.

NONE does not use a second objective for tuning.

ASE specifies average square error as the second objective.

AUC specifies area under the curve as the second objective (nominal type only).

F05 specifies the F0.5 coefficient as the second objective (nominal type only).

F1 specifies the F1 coefficient as the second objective (nominal type only).

FNR specifies the false negative rate as the second objective (nominal type only).

FPR specifies the false positive rate as the second objective (nominal type only).

GAMMA specifies the gamma coefficient as the second objective (nominal type only).

GINI specifies the Gini coefficient as the second objective (nominal type only).

KS specifies the Kolmogorov-Smirnov coefficient as the second objective (nominal type only).

MAE specifies the mean absolute error as the second objective (interval type only).

MCE specifies the misclassification rate as the second objective (nominal type only).

MCLL specifies the multiclass log loss as the second objective (nominal type only).

MISC specifies the misclassification error percentage as the second objective (nominal type only).

MSE specifies the mean square error as the second objective (interval type only).

MSLE specifies the mean square logarithmic error as the second objective (interval type only).

RASE specifies the root average square error as the second objective.

RMAE specifies the root mean absolute error as the second objective (interval type only).

RMSLE specifies the root mean square logarithmic error as the second objective (interval type only).

SCORETIME specifies the model scoring time as the second objective.

TAU specifies the tau coefficient as the second objective (nominal type only).

TNR specifies the true negative rate as the second objective (nominal type only).

TPR specifies the true positive rate as the second objective (nominal type only).

TRAINTIME specifies the model training time as the second objective.

By default, SECONDOBJECTIVE=NONE.

SELECTINITPOINT

specifies that the tuner select the best evaluation from the table specified in the LOOKUPTABLE= option as the initial point for the tuning process.

This option is ignored if you do not specify the LOOKUPTABLE= option.

TARGETEVENT=*string*

specifies the target event to use for calculating the selected objective function. This option is ignored when the value of the OBJECTIVE= option is not AUC, F1, F05, GINI, GAMMA, TAU, or KS.

If you omit the TARGETEVENT= option, the tuner selects one of the target levels and uses it to calculate the specified objective function.

TRAINFRACTION=*number*

specifies the fraction of all data to be used for training, where *number* must be between 0.01 and 0.99, inclusive. If you specify this option, the tuner uses a single-partition validation to find the objective value (validation error estimate). Using this option might not be advisable for small or unbalanced data tables, where the random assignment of the validation subset might not provide a good estimate of error. For large, balanced data tables, a single-partition validation is usually sufficient for estimating error; a single partition is more efficient than cross validation in terms of the total execution time.

If you specify a PARTITION statement, the validation partition defined in that statement is used, and this option is ignored. You cannot specify this option in combination with the KFOLD= option.

If you specify the FRACTION= option, then by default the value of the TRAINFRACTION= option is one minus the value of the FRACTION= option. Otherwise, by default, TRAINFRACTION=0.7.

TUNINGPARAMETERS=(*suboption* **| . . . | <** *suboption* **>)**

TUNEPARMS=(*suboption* **| . . . | <** *suboption* **>)**

specifies which parameters to tune and which ranges to tune over. If USEPARAMETERS=STANDARD, this option is ignored.

For more information about which tuning *suboptions* are available, see the specific procedure chapters.

USEPARAMETERS=COMBINED | CUSTOM | STANDARD

specifies which set of parameters to tune.

You can specify the following values:

COMBINED tunes the parameters that are specified in the TUNINGPARAMETERS= option and uses default bounds and initial values to tune all other parameters.

CUSTOM tunes only the parameters that are specified in the TUNINGPARAMETERS= option.

STANDARD tunes all parameters by using their default bounds and initial values.

By default, USEPARAMETERS=COMBINED.

CODE Statement

CODE < *options* > **;**

This section applies to the following procedures: FOREST and GRADBOOST.

The CODE statement writes SAS DATA step code for computing predicted values of the fitted model to a file. To score new data, you can then include the file in a DATA step.

Table 2.2 summarizes the *options* that you can specify in the CODE statement.

Table 2.2 CODE Statement Options

Option	Description
COMMENT	Adds comments to the generated code
FILE=	Names the file in which to save the generated code
FORMATWIDTH=	Specifies the numeric format width for the regression coefficients
INDENTSIZE=	Specifies the number of spaces to indent the generated code
LABELID=	Specifies a number used to construct names and labels
LINESIZE=	Specifies the line size for the generated code
NOTRIM	Compares formatted values, including blank padding

If you do not specify the FILE= option and if your SAS client has a default path, then the SAS scoring code is written to an external file named *_code_*. You can specify the following *options* in the CODE statement.

COMMENT
adds comments to the generated code.

FILE=*filename*
names the external file that saves the generated code. When enclosed in a quoted string (for example, FILE="c:\mydir\scorecode.sas"), this option specifies the path and filename for writing the code to an external file. If you do not specify a path but your SAS client has a default path, then the code is written to an external file named *filename* at that location. You can also specify an unquoted *filename* of no more than eight characters. If the *filename* is assigned as a fileref in a Base SAS FILENAME statement, the file that is specified in the FILENAME statement is opened; otherwise, if your SAS client has a default path, an external file named *filename* is created.

FORMATWIDTH=*width*
specifies the width to use in formatting derived numbers such as parameter estimates. You can specify a value in the range 12 to 32; the default is 20.

INDENTSIZE=*n*
specifies the number of spaces to indent the generated code. You can specify a value in the range 0 to 10; the default is 3.

LABELID=*value*
specifies a number used to construct array names and statement labels in the generated code. You can specify a value in the range 0 to 1024; the default is randomly chosen.

LINESIZE=*value*
LS=*value*
specifies the line size for the generated code. You can specify a value in the range 64 to 254; the default is 120.

NOTRIM
bases comparisons of formatted values on the full format width, including blank padding. By default, blanks at the beginning and end of strings are ignored.

Loading DATA Step Scoring Code into S3

The S3 FILENAME statement is not a supported way to download SAS DATA step code that has been generated by the CODE statement. Instead, save the DATA step code to disk. Then, use the S3 procedure to load the file into S3.

The following program shows how you can use PROC S3 to download DATA step code that has been generated by the REGSELECT procedure:

```
proc regselect data=mylib.baseball;
    model logSalary = nAtBat nHits crAtBat crHits;
    code file='/path/to/scorecode.sas';
run;

proc s3 config='/u/user/.tks3';
    put '/path/to/scorecode.sas' '/myBucket/scorecode.sas';
run;
```

PARTITION Statement

PARTITION *partition-option* **;**

This section applies to the following procedures: FOREST and GRADBOOST.

The PARTITION statement specifies how observations in the input data set are logically partitioned into disjoint subsets for model training, validation, and testing. For more information, see the section "Using Validation and Test Data" on page 23. Either you can designate a variable in the input data table and a set of formatted values of that variable to determine the role of each observation, or you can specify proportions to use for randomly assigning observations to each role.

You must specify exactly one of the following *partition-options*:

FRACTION(< TEST=*fraction* **> < VALIDATE=***fraction* **> < SEED=***number* **>)**

randomly assigns specified proportions of the observations in the input data table to the roles. You specify the proportions for testing and validation by using the TEST= and VALIDATE= suboptions. If you specify both the TEST= and VALIDATE= suboptions, then the sum of the specified fractions must be less than 1 and the remaining fraction is the proportion of the observations assigned to the training role. The SEED= option specifies an integer that is used to start the pseudorandom number generator for random partitioning of data for training, testing, and validation. If you do not specify SEED=*number* or if *number* is less than or equal to 0, the seed is generated by reading the time of day from the computer's clock.

ROLE=*variable* **(< TEST='***value***' > < TRAIN='***value***' > < VALIDATE='***value***' >)**

ROLEVAR=*variable* **(< TEST='***value***' > < TRAIN='***value***' > < VALIDATE='***value***' >)**

names the *variable* in the input data table whose values are used to assign roles to each observation. This *variable* cannot also appear as an analysis variable in other statements or options. The TEST=, TRAIN=, and VALIDATE= suboptions specify the formatted values of this variable that are used to assign observation roles. If you do not specify the TRAIN= suboption, then all observations whose role is not determined by the TEST= or VALIDATE= suboption are assigned to the training role.

SAVESTATE Statement

SAVESTATE < **OUT=** >*libref.data-table* **;**

This section applies to the following procedures: BNET, FACTMAC, FOREST, GMM, GPCLASS, GPREG, GRADBOOST, KPCA, LIGHTGRADBOOST, MBANALYSIS, MTLEARN, NNET, OPTBINNING, RPCA, SPARSEML, STYLEGAN, SVDD, SVMACHINE, TABULARGAN, and TEXTMINE.

The SAVESTATE statement saves the context and results of the analysis. The resulting analytic store exists in memory in a binary file format that cannot be modified. The data table that is specified in the SAVESTATE statement references the analytic store in memory. An example of using analytic store technology is to perform a time-consuming analysis and to store its results by using the SAVESTATE statement. Later you can perform specific analysis tasks based on the saved results of the previous analysis, without having to fit the model again. The contents of the analytic store can be processed by the ASTORE procedure (see Chapter 4, "ASTORE Procedure") and your procedure's documentation might describe other ways to use this item store. To save the analytic store on disk, use the DOWNLOAD statement in PROC ASTORE.

Details for Machine Learning Procedures

Using Validation and Test Data

This section applies to the following procedures: FOREST and GRADBOOST.

When you have sufficient data, you can divide your data into three parts called the training, validation, and test data. During the selection process, models are fit on the training data, and the prediction errors for the models so obtained are found by using the validation data. This prediction error on the validation data can be used to decide when to terminate the selection process and to decide which model to select. Finally, after a model has been selected, the test set can be used to assess how the selected model generalizes on data that played no role in selecting the model.

In some cases, you might want to use only training and test data. For example, you might decide to use an information criterion to decide which effects to include and when to terminate the selection process. In this case, no validation data are required, but test data can still be useful in assessing the predictive performance of the selected model. In other cases, you might decide to use validation data during the selection process but forgo assessing the selected model on test data. Hastie, Tibshirani, and Friedman (2001) note that it is difficult to provide a general rule for how many observations you should assign to each role. They note that a typical split might be 50% for training and 25% each for validation and testing.

You use a PARTITION statement to logically subdivide the input data table into separate roles. You can specify the fractions of the data that you want to reserve as test data and validation data. For example, the following statements randomly divide the inData data table, reserving 50% for training and 25% each for validation and testing:

```
proc forest data=mylib.inData;
   partition fraction(test=0.25 validate=0.25);
   ...
run;
```

You can specify the SEED= option in the PARTITION statement to create the same partition data tables.

In some cases, you might need to exercise more control over the partitioning of the input data table. You can do this by naming both a variable in the input data table and a formatted value of that variable for each role. For example, the following statements assign roles to the observations in the inData data table that are based on the value of the variable Group in that data table. Observations whose value of Group is `'Group 1'` are assigned for testing, and those whose value is `'Group 2'` are assigned to training. All other observations are ignored.

```
proc forest data=mylib.inData;
   partition roleVar=Group(test='Group 1' train='Group 2')
   ...
run;
```

When you have reserved observations for training, validation, and testing, a model that is fit on the training data is scored on the validation and test data, and statistics are computed separately for each of these subsets.

Multithreading

Threading refers to the organization of computational work into multiple tasks (processing units that can be scheduled by the operating system). A task is associated with a thread. Multithreading refers to the concurrent execution of threads. When multithreading is possible, substantial performance gains can be realized compared to sequential (single-threaded) execution. The number of threads spawned by a procedure in this book is determined by your installation.

The tasks that are multithreaded by procedures in this book are primarily defined by dividing the data that are processed on a single machine among the threads—that is, the procedures implement multithreading through a data-parallel model. For example, if the input data table has 1,000 observations and the procedure is running on four threads, then 250 observations are associated with each thread. All operations that require access to the data are then multithreaded. These operations include the following (not all operations are required for all procedures):

- variable levelization
- effect levelization
- formation of the initial crossproducts matrix
- formation of approximate Hessian matrices for candidate evaluation during model selection
- objective function calculation
- gradient calculation
- Hessian calculation
- scoring of observations

In addition, operations on matrices such as sweeps can be multithreaded provided that the matrices are of sufficient size to realize performance benefits from managing multiple threads for the particular matrix operation.

Hyperparameter Tuning

This section applies to the following procedures: BNET, FACTMAC, FOREST, GRADBOOST, MTLEARN, NNET, SEMISUPLEARN, SVMACHINE, and TSNE.

The quality of the predictive model that a machine learning algorithm creates depends on the values for various options that govern the training process; these options are called *hyperparameters*. The default values of these hyperparameters might not be suitable for all applications. In order to reduce the manual effort in adjusting these hyperparameters, you can use the AUTOTUNE statement to identify the best settings for them.

To tune hyperparameters, the AUTOTUNE statement directs the procedure to engage a search method (tuner) that searches for the best possible combination of values of these hyperparameters while trying to minimize or maximize an objective function. The objective function is a validation error or accuracy estimate—for example, MISC (misclassification error) for nominal targets or ASE (average square error) for interval targets. The tuning process involves multiple evaluations of the objective function (one for each model hyperparameter configuration it evaluates) and might include multiple iterations, depending on the specified

search method. The tuning process also requires data partitioning in order to avoid overfitting to training data; models are trained on a training partition and validated on a validation partition to obtain the objective function value for each chosen hyperparameter configuration. The options for data partitioning are described in the next section, followed by descriptions of the default hyperparameter tuning process and determination of sizing for parallel tuning.

Data Partitioning

Each evaluation of the objective function can consist of one or several training and scoring (validation) executions, depending on whether single-partition validation or *k*-fold cross validation is used. In both cases, the tuner partitions the data unless the PARTITION statement is used (in this case, the tuner does not partition the training data, but instead uses the data roles that are specified in the PARTITION statement for training and validation).

Single-Partition Validation

By default, the tuner uses single-partition validation. In this process, the tuner partitions all the data into two subsets: one subset for model training and one subset for model validation. For each newly generated configuration of hyperparameters, a new model is trained on the training subset and then the validation subset is scored using the trained model to find the resulting objective function value.

The AUTOTUNE statement calls the **`sampling`** action set, which creates the training and validation partitions. The **`stratified`** action in the **`sampling`** action set is used for a target of nominal type (if all target levels can be included in both the training and validation partitions), and the **`srs`** action in the **`sampling`** action set is used for a target of interval type and for cases in which stratified sampling is not possible. By default, a validation partition of 30% is used and the remaining 70% is used for model training. The validation partition size can be adjusted by specifying the FRACTION= option in the AUTOTUNE statement. For more information about the **`sampling`** action set, see Chapter 24, "Sampling and Partitioning Action Set" (*SAS Visual Statistics: Programming Guide*).

Cross Validation

If *k*-fold cross validation is specified, the tuner partitions all the data into *k* subsets (folds). For each fold, a new model is trained on all folds except the selected (holdout) fold (that is, it is trained on k–1 folds) and then validated using the selected (holdout) fold. The objective function value is averaged over the set of fold validation executions to obtain a single error estimate value. Cross validation is specified using the KFOLD= option in the AUTOTUNE statement.

For efficiency, the cross validation process might be terminated before all *k* folds are evaluated. Cross validation is terminated under the following conditions: the validation score is 2 times worse than the current best score after the first fold, the validation score is 1.75 times worse than the current best score after two folds, or the validation score is 1.5 times worse than the current best score after three or more folds.

Hyperparameter Tuning Process

By default, the tuner chooses the model hyperparameter configurations to evaluate by using a hybrid optimization strategy that is based on a genetic algorithm (GA), which applies principles of natural selection and evolution to find an improved configuration. The tuner performs the following steps:

1. A default model configuration (default values of select model hyperparameters) is evaluated first and designated as Iteration 0. The objective function value is obtained by using either single-partition validation or k-fold cross validation and then recorded for comparison.

2. An initial set of hyperparameter configurations, also called a "population," is generated using a technique called Latin hypercube sampling (LHS). Each configuration of hyperparameters in the Latin hypercube sample is evaluated, and the objective function value for each is again recorded for comparison. This set becomes Iteration 1. All or a portion of the sample configurations can be evaluated in parallel, as described in the section "Determining the Number of Parallel Evaluations" on page 27.

3. The best model configurations from the Iteration 1 are used to generate the next population of model configurations, Iteration 2, which are then evaluated, in parallel when possible (see the section "Determining the Number of Parallel Evaluations" on page 27). This process is repeated for the remaining iterations, as long as the maximum number of evaluations or the maximum time has not been reached. The default number of iterations is 5, and the default maximum time is 36,000 seconds (10 hours).

4. The best model configuration (the one that minimizes or maximizes the objective function) is reevaluated by executing a single model training and scoring, and information about the model training and scoring for this configuration is returned.

5. All evaluated model configurations are ranked, and the hyperparameter and objective function values of the top 10 configurations are returned in the TunerResults table.

You can specify four alternative search methods: Bayesian, random, LHS, and grid. The preceding steps are performed differently, depending on the search method that you use:

- For the Bayesian search method, the preceding steps are performed in the same sequence, with the following changes. Instead of using genetic operations to generate the new combinations of hyperparameter values, an internal surrogate model is created after the first iteration. This surrogate model is then explored for potential candidate configurations, and the new set of configurations is created and evaluated in parallel. After each iteration, the internal surrogate model is updated with the new data and explored again for new potential configurations.

- For the random, LHS, or grid search method, step 3 in the preceding sequence is eliminated; a single sample of candidate configurations is generated and evaluated in step 2. This single sample of configurations is generated in one of the following ways:

  – randomly, for the random search method
  – by using a Latin hypercube sample, for the LHS search method
  – by creating all possible combinations of hyperparameter levels, for the grid search method. If the VALUELIST= option is specified for any hyperparameter, the levels are taken from that list. Otherwise, the tuner takes the lower bound, the upper bound, and the middle value between the two bounds and uses those values as the grid levels for each hyperparameter. The total number of grid configurations is the product of the numbers of levels for each hyperparameter. If the number of hyperparameters and the number of levels is greater than a very small value, then the total number of combinations can be many hundreds or thousands. If the MAXEVALS= option is specified, the tuner uses it to terminate execution of the grid search method before all grid points are evaluated.

If the tuner had partitioned the data for training and scoring (if the PARTITION statement is not specified), then the final (best) model configuration is trained on all the data and scored on all the data; as a result the final model that is returned and its reported validation score might not match the value that is observed during tuning, when the model was trained to a portion of the data and validated on the remaining validation partition.

Determining the Number of Parallel Evaluations

By default, the number of possible parallel evaluations during step 2 and step 3 (if performed) in the section "Hyperparameter Tuning Process" on page 25 is determined by the population size for the GA or Bayesian search method or by the sample size for the random or LHS sampling method. The maximum number of parallel evaluations is limited by the total number of worker nodes that are connected to the server divided by the number of worker nodes that are used by the parallel sessions that are created. For example, if the server is configured with 100 worker nodes and four workers are used in the sessions, at most 25 parallel evaluations are performed by default. This limit can be overridden by a factor of 2, resulting in up to 50 parallel evaluations in this example scenario. Because of the data movement that parallel subsessions produce, repeated runs with the same seed can produce different results.

Specifically, the number of parallel evaluations is determined as follows:

1. If you do not specify the number of workers to use in parallel subsessions by using the NSUBSESSIONWORKERS= option in the AUTOTUNE statement, the number of workers is determined using the size of the data table: NSUBSESSIONWORKERS = 1 + *nDataRows* * *nDataColumns* * 2E–8. If determined on the basis of the data table size, the number of workers in each subsession is fairly aggressively set at one node per 50 million values. The efficiency benefits of tuning in parallel generally outweigh the time savings from distributed training if the data can be managed on fewer nodes. For example, if training a single model on four workers takes 1 minute versus 2 minutes on one worker, then training four models in parallel on one worker each requires only 2 minutes, whereas training four sequential models each on four workers would require 4 minutes.

2. The number of possible parallel evaluations is then determined either as one less than the population size for the GA or Bayesian search method, or as the sampling size for the random or LHS search method.

3. The number of parallel evaluations is then limited by the server configuration:

   - In single-machine mode, if the number of possible parallel evaluations is greater than 4 and not specified in the NPARALLEL= option in the AUTOTUNE statement, it is limited to 4. You can override this limit up to a value of 32 by specifying the NPARALLEL= option (the value of the NPARALLEL= option is reduced if it is greater than 32).
   - In distributed mode, the upper limit for the number of parallel evaluations is calculated as W/n, where W is the number of workers connected to the server and n is the number of workers in the parallel subsessions. You can override this limit by a factor of 2 by specifying the NPARALLEL= option; the new limit is $2W/n$ (the value of the NPARALLEL= option is reduced if it is greater than $2W/n$).

Lookup Table and History Table

Tuning a predictive model often requires multiple runs of a tuning action and various adjustments of the hyperparameter bounds or their initial values. This often means training a large number of model configurations for each new tuning process, some of which might be duplicates of configurations from the previous tuning runs. To avoid rerunning the same configurations, you can use a lookup table, which you specify in the LOOKUPTABLE= option in the AUTOTUNE statement. You can create this lookup table manually, such as by using a DATA step, or you can produce it automatically in any procedure call in which you specify the HISTORYTABLE= option in the AUTOTUNE statement. Before evaluating a new predictive model configuration, the tuner first checks the lookup table for a potential match. The lookup table must contain all the hyperparameter columns and the objective column; if any required columns are not found, the tuner rejects the lookup table. If a matching configuration is found, its objective value is read from the lookup table and the configuration is not evaluated. The tuner matches only the hyperparameter values and ignores all other columns in the lookup table. For convenience, when you do multiple runs of a procedure in which you use autotuning, you can collect the history of all runs in one table by using the APPENDLOOKUP option in the AUTOTUNE statement. This causes the tuner to append all unused configurations from the lookup table to the history table, which can then be used again as the lookup table for the next procedure call in which you use autotuning. It is essential that the lookup table contain only the appropriate data. If any nontuned parameters have changed, or if the data that are used for the predictive model have changed, then the objective values inside the lookup table might no longer be valid for the new tuning process. The tuner is oblivious to any changes in the problem setup between any two runs of the tuning process and will use the objective values from the lookup table as long as the hyperparameter values match the configuration to be evaluated.

References

Hastie, T. J., Tibshirani, R. J., and Friedman, J. H. (2001). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. New York: Springer-Verlag.

Chapter 3
ASSESSBIAS Procedure

Contents

Overview: ASSESSBIAS Procedure **30**
PROC ASSESSBIAS Features 30
Using SAS Viya Workbench 30
Getting Started: ASSESSBIAS Procedure **31**
Syntax: ASSESSBIAS Procedure **33**
PROC ASSESSBIAS Statement 33
FITSTAT Statement 34
FREQ Statement 35
INPUT Statement 35
SENSITIVEVAR Statement 35
TARGET Statement 35
WEIGHT Statement 36
Details: ASSESSBIAS Procedure **36**
Required Specifications 36
Assessment Statistics 37
Nominal Assessment Statistics 37
Interval Assessment Statistics 38
Prediction Statistics 38
Nominal Prediction Statistics 38
Interval Prediction Statistics 38
Bias Metrics 39
Nominal Bias Statistics 39
Interval Bias Statistics 39
Displayed Output 39
Group Information 39
Group Metrics 39
Bias Metrics 39
Max Difference 40
ODS Table Names 40
Examples: ASSESSBIAS Procedure **41**
Example 3.1: Assess Bias of a Model with a Binary Target 41
Example 3.2: Assess Bias of a Model with an Interval Target 43

Overview: ASSESSBIAS Procedure

The ASSESSBIAS procedure evaluates bias in predictive models by calculating statistical metrics and bias measurements.

In predictive modeling, bias occurs when a model's prediction or performance differs for different values of a given variable, referred to here as the sensitive variable. A common example of a sensitive variable is race or gender. Prediction bias is measured by calculating the difference in the average model prediction for various groups that are represented in the sensitive variable, and performance bias is measured by calculating the difference in model performance with respect to bias measurements for those groups.

PROC ASSESSBIAS calculates the performance statistics and average model predictions for each group that is represented in a sensitive variable. The difference in these statistics is calculated either by reporting the maximum pairwise difference among all levels or by comparing each level to a reference group in order to report a single number that represents model bias.

PROC ASSESSBIAS Features

You can call the ASSESSBIAS procedure to evaluate the various assessment metrics of predictive models and report the bias. The following list summarizes some basic features of PROC ASSESSBIAS:

- is multithreaded
- computes and compares the statistical metrics that are related to the receiver operating characteristic (ROC), lift, and fit statistics for each value in the sensitive variable
- calculates most adopted bias metrics, including demographic parity, equalized odds, and equal opportunity

Using SAS Viya Workbench

SAS Viya Workbench has greatly simplified the SAS programming experience by offering a syntax style nearly identical to that of SAS 9 and earlier releases. However, there are minor exceptions. In the past, SAS Viya procedures required the use of a CAS engine libref. This engine served multiple purposes, but it was primarily used to upload data to the CAS server and manage the data. Its additional purposes included support of new data types and improved performance for large data sets. No CAS server is needed for SAS Viya Workbench, because all computations are done in place for you in a single location.

In the following program, the LIBNAME statement creates a mylib libref that the examples in this chapter use to refer to your current or desired work directory:

```
libname mylib ".";
```

For more information, see *SAS Viya Workbench: Getting Started*.

Getting Started: ASSESSBIAS Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

This example demonstrates how to use the ASSESSBIAS procedure to measure bias and compute the difference of various statistical metrics based on a scored data set and a sensitive variable.

The following SAS DATA step creates the SAS data table mylib.score, which simulates a credit score data set that is trained by a supervised model such as a decision tree or neural network model, and loads the table into your SAS session:

```
data mylib.score;
   length credit $4.;
   input ID $ credit $ P_creditgood P_creditbad;
   datalines;
A good 0.83 0.17
A bad  0.65 0.35
A good 0.76 0.24
A good 0.36 0.64
A bad  0.17 0.83
B good 0.55 0.45
B bad  0.22 0.78
B good 0.63 0.37
B good 0.81 0.19
;
```

These statements assume that your SAS libref is named mylib, but you can substitute any appropriately defined SAS libref.

In this data table, the variable credit represents the ground truth label with two values, "good" and "bad". The variables P_creditgood and P_creditbad are the predicted probabilities for the target outcomes "good" and "bad", respectively. The "good" event is considered to be the favored condition in the bias measurement. The variable ID is the sensitive variable, with the attribute values A and B.

The following statements call PROC ASSESSBIAS to produce the reports:

```
proc assessbias data=mylib.score nbins=2;
   var P_creditgood;
   target credit / event="good" level=nominal;
   sensitiveVar ID;
   fitstat pvar=P_creditbad / pevent="bad";
run;
```

The VAR statement specifies that the variable P_creditgood should be analyzed and that the associated event "good" is the event of interest in the bias metrics. The TARGET statement specifies that the variable credit is the target variable for the classification model. The FITSTAT statement lists the other predicted variable, P_creditbad, and the associated event. The SENSITIVEVAR statement specifies that the variable ID is the variable that the population grouping is based on.

Figure 3.1 shows the values of demographic parity, predicted parity, equal accuracy, equalized odds, and equal opportunity for two population groups in the sensitive variable ID.

Figure 3.1 Bias Metrics Based on Variable ID

The ASSESSBIAS Procedure

Bias Metrics					
Bias Statistic	**Bias Statistic Label**	**Bias Statistic Value**	**Base Level**	**Compare Level**	**Note**
DemographicParity	Demographic Parity (Statistical Parity)	0.1500	B	A	
PredictiveParity	Predictive Parity	0.0015	A	B	
EqualAccuracy	Equal Accuracy	0.4000	B	A	
EqualizedOdds	Equalized Odds	0.5000	A	B	The maximum FPR difference is greater than the maximum TPR difference.
EqualOpportunity	Equal Opportunity	0.3333	B	A	

Figure 3.2 shows the difference of a set of statistical metrics that are related to receiver operation characteristic (ROC) curve analysis, lift analysis, and fit statistics analysis for two population groups in the sensitive variable ID.

Figure 3.2 Assessment of Metrics Difference Based on Variable ID

Maximum Difference Between Groups (Base - Compare)				
Metric Name	**Metric Label**	**Maximum Difference**	**Base Level**	**Compare Level**
P_creditbad	Average Predicted: CREDIT=bad	0.0015	B	A
P_creditgood	Average Predicted: CREDIT=good	0.0015	A	B
TPR	True Positive Rate	0.3333	B	A
FPR	False Positive Rate	0.5000	A	B
TNR	True Negative Rate	0.5000	B	A
FNR	False Negative Rate	0.3333	A	B
FDR	False Discovery Rate	0.3333	A	B
ACC	Accuracy	0.4000	B	A
C	Area under ROC	0.1667	B	A
F1	F1 Score	0.3333	B	A
GINI	Gini Coefficient	0.3333	B	A
MISCEVENT	Event Misclassification Rate at Cutoff	0.4000	A	B
MISCEVENTKS	Event Misclassification Rate at Maximum KS	0.2000	A	B
MCE	Misclassification Rate	0.4000	A	B
ASE	Average Squared Error	0.0835	A	B
RASE	Root Average Squared Error	0.1098	A	B
MCLL	Multiclass Log Loss	0.1640	A	B
maxKS	Best Kolmogorov-Smirnov along ROC	0.3333	B	A
cutoffKS	Kolmogorov-Smirnov at Cutoff	0.8333	B	A
GAIN	Gain	0.0000	B	A
LIFT	Lift	0.0000	A	B
RESP	%% Captured Response	0.0000	A	B
CUMRESP	Cumulative %% Captured Response	0.0000	A	B
CUMLIFT	Cumulative Lift	0.0000	A	B
PREDICTED_EVENT	Average Prediction for Event	0.0015	A	B
INTO_EVENT	Proportion into Event Level	0.1500	B	A

Syntax: ASSESSBIAS Procedure

The following statements are available in the ASSESSBIAS procedure:

PROC ASSESSBIAS < *options* > **;**
FITSTAT PVAR=*variables* / **PEVENT=***"event-list"* < *options* > **;**
FREQ *variable* **;**
INPUT *variable* **;**
SENSITIVEVAR *variable* < */options* > **;**
TARGET *variable* < */options* > **;**
WEIGHT *variable* **;**

The PROC ASSESSBIAS, INPUT, SENSITIVEVAR, and TARGET statements are required.

The following sections describe the PROC ASSESSBIAS statement and then describe the other statements in alphabetical order.

PROC ASSESSBIAS Statement

PROC ASSESSBIAS < *options* > **;**

The PROC ASSESSBIAS statement invokes the ASSESSBIAS procedure.

You can specify the following *options*:

CUTOFF=*ratio*
specifies a user-defined threshold for target events for multiclass classification, where *ratio* is between 0 and 1.

By default, CUTOFF=0.5.

DATA=*libref.data-table*
names the input data table for PROC ASSESSBIAS to use. The default is the most recently created data table. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to the data. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 30.

data-table specifies the name of the input data table.

NBINS=*integer*
specifies the number of bins to use in the lift calculation, where *integer* is an integer greater than or equal to 2.

By default, NBINS=20.

NCUTS=*integer*

specifies the number of cuts to use in the ROC calculation, where *integer* is an integer greater than or equal to 2 and less than or equal to 100. The inverse of *integer* represents the step size of the quantile. For example, NCUTS=10 generates 10 intervals for ROC analysis, where each step size is 0.1. You do not need to specify this option when you specify an interval-type response variable in the TARGET statement.

By default, NCUTS=100.

SELECTIONDEPTH=*integer*

specifies the depth to use in lift calculations, where *integer* is an integer greater than or equal to 1 and less than or equal to 100.

By default, SELECTIONDEPTH=10.

FITSTAT Statement

FITSTAT PVAR=*variables* / **PEVENT=***"event-list"* < *options* > **;**

The FITSTAT statement reports the statistical metrics that are related to lift analysis and ROC analysis that the ASSESS procedure performs. This statement is required when the target is nominal. You must specify the following options:

PEVENT=*"event-list"*

specifies the events that correspond to each variable in the PVAR= option. The *"event-list"* cannot include the event that corresponds to the variable in the TARGET statement. The *"event-list"* is case-sensitive, and missing values are not allowed. You must specify the PEVENT= and PVAR= options the same number of times and in the same order. The maximum length of *"event-list"* is 5,000 characters.

PVAR=*variables*

specifies the posterior probability for each level in model prediction except the variable that you specify in the INPUT statement. Duplicate variables are not allowed.

You can also specify the following *option*:

DELIMITER=*"character"*

DLM=*"character"*

specifies the delimiter that is used to separate events that you specify in the PEVENT= option. A delimiter is used when event names contain embedded spaces (such as “Fuel Oil”) or special characters (such as “;” or “*”). You must use quotation marks around *character*. You can specify only one delimiter; combinations of delimiters are not supported. The valid values of the DELIMITER= option are “ ” (space), “;” (semicolon), “*” (asterisk), “.” (period), and “,” (comma).

By default, DELIMITER=“ ” (space).

If you specify a delimiter other than “ ”, PROC ASSESSBIAS treats any spaces (including empty spaces) as valid characters. You are responsible for mapping events correctly in order to obtain correct results.

FREQ Statement

FREQ *variable* **;**

The *variable* in the FREQ statement identifies a numeric variable in the input data table that contains the frequency of occurrence of each observation. PROC ASSESSBIAS treats each observation as if it appears f times, where f is the value of the FREQ *variable* for the observation. If f is not an integer, it is truncated to an integer. If f is less than 1 or missing, the observation is not used in the analysis. When you omit the FREQ statement, each observation is assigned a frequency of 1.

INPUT Statement

INPUT *variable* **;**

VAR *variable* **;**

The INPUT statement specifies the name of one *variable* to be analyzed in model assessment. When you specify a nominal variable in the TARGET statement, *variable* is the posterior probability of the event to be analyzed. When you specify an interval-type variable, *variable* is the predicted value.

SENSITIVEVAR Statement

SENSITIVEVAR *variable* < /*options* > **;**

The SENSITIVEVAR statement specifies the sensitive *variable* in modeling. Only one sensitive *variable* is accepted. You can specify the following *option*:

REFERENCELEVEL="*string***"**
specifies the base level of the sensitive *variable* to be compared with.

If you omit this option, PROC ASSESSBIAS reports the maximum pairwise difference among all levels of the sensitive variable. If you specify this option, PROC ASSESSBIAS reports the maximum pairwise difference between the specified reference level and all other levels of the sensitive variable.

TARGET Statement

TARGET *variable* < /*options* > **;**

The TARGET statement specifies the response *variable* in supervised learning. You can specify the following *options*:

EVENT="*string***"**
specifies the formatted value of the response *variable* that represents the event. For an interval target (LEVEL=INTERVAL), the EVENT= option is ignored. For a nominal target (LEVEL=NOMINAL), the EVENT= option is required.

LEVEL=INTERVAL | NOMINAL
specifies the *variable* type.

You can specify the following types:

INTERVAL specifies that the response *variable* is interval, which must be numeric.

NOMINAL specifies that the response *variable* is nominal, also known as a classification variable, which can be numeric or character.

By default, LEVEL=INTERVAL.

WEIGHT Statement

WEIGHT *variable* **;**

The *variable* in the WEIGHT statement is used as a weight to perform a weighted analysis of ROC information and fit statistics. If specified, the WEIGHT statement is ignored in lift analysis. If you analyze ROC information and fit statistics, nonpositive or missing weights are ignored in the analysis. When you omit the WEIGHT statement, all observations that are used in the analysis are assigned a weight of 1.

Details: ASSESSBIAS Procedure

Required Specifications

The following specifications are required in order to run PROC ASSESSBIAS:

- Specify an input data table in the DATA= option. This table must contain the scored results from a previously trained model.
- Specify a predicted target variable in the INPUT statement. This is the output variable that contains the model's predictions when the scoring function is applied to the input data table. The values of these variables can represent numeric predictions or predicted probabilities for an interval or nominal target, respectively. In many SAS Viya Workbench machine learning models, the name of the variable is constructed as follows: for an interval target, P_ is prefixed to the name of the target variable (for example, P_MSRP); for a nominal target, P_ is prefixed to the name of the target variable, and the value that corresponds to the target event is appended (for example, P_Bad1).
- Specify a sensitive variable in the SENSITIVEVAR statement to assess for bias. The model's performance and average predictions are calculated for each level of the sensitive variable and compared in order to produce bias metrics. This variable must have fewer than 254 levels.
- Specify a response variable in the TARGET statement to use for model assessment. This is the variable that contains the true values of the target variable.

- Specify a list of response levels in the EVENT= and PEVENT= options. For models that predict a nominal response variable, you must also specify the level of the response variable that corresponds to each predicted target variable.

Assessment Statistics

The following assessment statistics are reported for each level of the sensitive variable. For more information about the calculation of a specific statistic, see Chapter 3, "ASSESS Procedure" (*SAS Visual Statistics: Procedures*).

Nominal Assessment Statistics

- TPR (true positive rate)
- FPR (false positive rate)
- TNR (true negative rate)
- FNR (false negative rate)
- FDR (false discovery rate)
- ACC (accuracy)
- C (area under ROC (AUC))
- F1 (F1 score)
- GINI (Gini coefficient)
- MISCEVENT (misclassification rate at cutoff) – the misclassification rate for the event response level based on the CUTOFF= option in the PROC ASSESSBIAS statement
- MISCEVENTKS (misclassification rate at KS cutoff) – the misclassification rate at the cutoff where the maximum KS occurs
- MCE (misclassification rate)
- ASE (average square error)
- RASE (root average square error)
- MCLL (multiclass log loss)
- maxKS (best Kolmogorov-Smirnov along ROC) – the maximum KS statistic that is calculated for all cutoffs along the calculated ROC curve
- cutoffKS (Kolmogorov-Smirnov at cutoff) – the KS statistic that is calculated at the cutoff specified in the CUTOFF= option in the PROC ASSESSBIAS statement
- GAIN (gain)

- LIFT (lift)
- RESP (percentage of captured response)
- CUMRESP (cumulative percentage of captured response)
- CUMLIFT (cumulative lift)

The following assessment statistics depend on the EVENT= and CUTOFF= options in the PROC ASSESSBIAS statement: TP, FP, TN, FN, TPR, FPR, TNR, FNR, FDR, ACC, GINI, MISCEVENT, and cutoffKS.

Interval Assessment Statistics

- ASE (average square error)
- RASE (root average square error)
- MAE (mean absolute error)
- RMAE (root mean absolute error)
- MSLE (mean square logarithmic error)
- RMSLE (root mean square logarithmic error)

Prediction Statistics

The following prediction statistics are reported for each level of the sensitive variable. These statistics are always reported.

Nominal Prediction Statistics

- PREDICTED_EVENT – the average value of the predicted variable that is associated with the level that you specify in the EVENT= option in the TARGET statement
- INTO_EVENT – the proportion of observations that are binned into the level that you specify in the EVENT= option in the TARGET statement. An observation is binned into the event level if the value of the predicted variable that corresponds to the event level is higher than the value that you specify in the CUTOFF= option in the PROC ASSESSBIAS statement.

Interval Prediction Statistics

- PREDICTED – the average value of the predicted variable

Bias Metrics

The following bias metrics are reported for the sensitive variable:

Nominal Bias Statistics

- DemographicParity – maximum pairwise difference in INTO_EVENT
- PredictiveParity – maximum pairwise difference in PREDICTED_EVENT
- EqualAccuracy – maximum pairwise difference in ACC
- EqualizedOdds – larger of the maximum pairwise differences in TPR and FPR
- EqualOpportunity – maximum pairwise difference in TPR

NOTE: For a nominal target, predictive parity is sometimes also called mean parity. This differs from some of the literature, where predictive parity is defined as the difference in positive predictive value (PPV).

Interval Bias Statistics

- PredictiveParity – maximum pairwise difference in PREDICTED
- MaxGroupLoss – maximum average square error (ASE) among all the levels in the sensitive variable

Displayed Output

The output that PROC ASSESSBIAS produces is organized into the following tables, which are presented in the order of their appearance.

Group Information

The “Group Information” table contains both the raw and formatted values of the sensitive variable.

Group Metrics

The “Group Metrics” table contains assessment statistics for the sensitive variable.

Bias Metrics

The “Bias Metrics” table contains bias metrics for the sensitive variable.

Max Difference

The “Max Difference” table contains maximum pairwise difference of assessment statistics among pairs of the sensitive variable.

ODS Table Names

Each table that PROC ASSESSBIAS creates has a name associated with it. You must use this name to refer to the table when you use ODS statements. The name of each table and a short description of the contents are listed in Table 3.1.

Table 3.1 ODS Tables Produced by PROC ASSESSBIAS

Table Name	Description	Statement and Options
BiasMetrics	Bias metrics	FITSTAT statement; VAR= and PEVENT= options for a nominal target; default output for interval target
GroupInfo	Group information	Default output. Table is created only if the sensitive variable has a user-defined format.
GroupMetrics	Assessment statistics for each group in sensitive variable	Default output
MaxDifferences	Maximum difference of assessment statistics between groups in sensitive variable	Default output

Examples: ASSESSBIAS Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

Example 3.1: Assess Bias of a Model with a Binary Target

This example demonstrates how to use the ASSESSBIAS procedure to measure the bias of statistic metrics based on a scored model with a binary target.

The following DATA step simulates a score data table from a previously trained model. For example, the model could be a tree-based model, a neural network, or a support vector machine. Gender represents the sensitive variable and has two attributes: "1" and "0". RF is the risk factor target that the model tries to predict, and it has two values: "high" and "low". P_RFhigh represents the predicted probability of the individual being categorized as "high" risk, and P_RFlow represents the predicted probability of the individual being categorized as "low" risk from the modeling.

```
data mylib.score2;
    length RF $4. Gender $1.;
    call streaminit(123456);
    do i=1 to 1000;
        if (rand("uniform",0,1)>0.5) then Gender = "0";
        else Gender = "1";
        if (rand("uniform",0,1)>0.5) then RF = "low";
        else RF = "high";
        P_RFhigh = rand("uniform", 0, 1);
        P_RFlow = 1 - P_RFhigh;
        output;
    end;
    drop i;
run;
```

The following PROC ASSESSBIAS call uses five bins to do lift analysis and a cutoff threshold of 0.2 for ROC analysis. The VAR statement specifies that the variable P_RFhigh should be analyzed and the associated event "high" is the event of interest in the bias metrics. The TARGET statement specifies that the variable RF is the target variable for the classification model. The FITSTAT statement lists the other predicted variable, P_RFlow, and the associated event. The SENSITIVEVAR statement specifies that Gender is the variable that the population grouping is based on.

```
proc assessbias data=mylib.score2 ncuts=5 nbins=5;
   var P_RFhigh;
   target RF / event="high" level=nominal;
   fitstat pvar=P_RFlow / pevent="low" ;
   sensitiveVar Gender;
run;
```

Output 3.1.1 shows nominal target–related assessment metrics for each group of Gender.

Output 3.1.1 Group Metrics

The ASSESSBIAS Procedure

Group Metrics								
Gender	Number of Observations	True Positive Rate	False Positive Rate	Area Under ROC	Misclassification Rate	Multi-Class Log Loss	Best Kolmogorov-Smirnov Along ROC	Cumulative Lift
0	500	0.5238	0.4960	0.5248	0.4860	1.0009	0.0683	1.1310
1	500	0.4833	0.5577	0.4417	0.5380	1.0524	0.0000	0.8125

Group Metrics			
Gender	Cumulative Response	Proportion Into Event Level	Average Prediction for Event
0	22.6190	0.5100	0.5088
1	16.2500	0.5220	0.5093

Output 3.1.2 shows nominal target–related bias metrics based on Gender.

Output 3.1.2 Bias Metrics

Bias Metrics					
Bias Statistic	Bias Statistic Label	Bias Statistic Value	Base Level	Compare Level	Note
DemographicParity	Demographic Parity (Statistical Parity)	0.0120	1	0	
PredictiveParity	Predictive Parity	0.0005	1	0	
EqualAccuracy	Equal Accuracy	0.0520	0	1	
EqualizedOdds	Equalized Odds	0.0617	1	0	The maximum FPR difference is greater than the maximum TPR difference.
EqualOpportunity	Equal Opportunity	0.0405	0	1	

Output 3.1.3 shows the maximum difference in the nominal target–related assessment metrics based on Gender.

Output 3.1.3 Maximum Difference

Maximum Difference Between Groups (Base - Compare)				
Metric Name	**Metric Label**	**Maximum Difference**	**Base Level**	**Compare Level**
P_RFlow	Average Predicted: RF=low	0.0005	0	1
P_RFhigh	Average Predicted: RF=high	0.0005	1	0
TPR	True Positive Rate	0.0405	0	1
FPR	False Positive Rate	0.0617	1	0
TNR	True Negative Rate	0.0617	0	1
FNR	False Negative Rate	0.0405	1	0
FDR	False Discovery Rate	0.0732	1	0
ACC	Accuracy	0.0520	0	1
C	Area under ROC	0.0831	0	1
F1	F1 Score	0.0576	0	1
GINI	Gini Coefficient	0.1662	0	1
MISCEVENT	Event Misclassification Rate at Cutoff	0.0520	1	0
MISCEVENTKS	Event Misclassification Rate at Maximum KS	0.0520	1	0
MCE	Misclassification Rate	0.0520	1	0
ASE	Average Squared Error	0.0269	1	0
RASE	Root Average Squared Error	0.0229	1	0
MCLL	Multiclass Log Loss	0.0515	1	0
maxKS	Best Kolmogorov-Smirnov along ROC	0.0683	0	1
cutoffKS	Kolmogorov-Smirnov at Cutoff	0.1022	0	1
GAIN	Gain	0.3185	0	1
LIFT	Lift	0.3185	0	1
RESP	%% Captured Response	6.3690	0	1
CUMRESP	Cumulative %% Captured Response	6.3690	0	1
CUMLIFT	Cumulative Lift	0.3185	0	1
PREDICTED_EVENT	Average Prediction for Event	0.0005	1	0
INTO_EVENT	Proportion into Event Level	0.0120	1	0

Example 3.2: Assess Bias of a Model with an Interval Target

This example demonstrates how to use the ASSESSBIAS procedure to measure the bias of statistic metrics based on a scored model with an interval target.

The following DATA step creates a scored data table from a previously trained regression model. For example, the model could be a tree-based model, a neural network, or linear regression. Gender represents the sensitive variable and has two attributes: “0” and “1”. GPA is the interval target for the modeling. P_GPA represents the predicted GPA values for individuals.

```
data mylib.score3;
    length Gender $1.;
    call streaminit(123456);
    do i=1 to 1000;
        if (rand("uniform",0,1)>0.5) then Gender = "0";
        else Gender = "1";
        GPA = rand("uniform",1,5);
```

```
        P_GPA = rand("uniform",1,5);
        output;
    end;
    drop i;
run;
```

The following PROC ASSESSBIAS call uses five bins to do the lift analysis. The VAR statement specifies that the variable P_GPA should be analyzed. The TARGET statement specifies that the variable GPA is the target variable for the regression model. The SENSITIVEVAR statement specifies that the variable Gender is the variable that the population grouping is based on.

```
proc assessbias data=mylib.score3 nbins=5;
   var P_GPA;
   target GPA;
   sensitiveVar Gender;
run;
```

Output 3.2.1 shows the interval target–related assessment metrics for each group of Gender.

Output 3.2.1 Group Metrics

The ASSESSBIAS Procedure

Group Metrics

Gender	Number of Observations	Average Squared Error	Root Average Squared Error	Mean Absolute Error	Root Mean Absolute Error	Mean Squared Logarithmic Error	Root Mean Squared Logarithmic Error	Average Prediction
0	500	2.8343	1.6835	1.3841	1.1765	0.1998	0.4470	3.0353
1	500	2.4239	1.5569	1.2415	1.1142	0.1732	0.4161	3.0373

Output 3.2.2 shows the interval target–related bias metrics based on Gender.

Output 3.2.2 Bias Metrics

Bias Metrics

Bias Statistic	Bias Statistic Label	Bias Statistic Value	Base Level	Compare Level
PredictiveParity	Predictive Parity	0.0019	1	0

Output 3.2.3 shows the maximum difference in the interval target–related assessment metrics based on Gender.

Output 3.2.3 Maximum Difference

Maximum Difference Between Groups (Base - Compare)				
Metric Name	**Metric Label**	**Maximum Difference**	**Base Level**	**Compare Level**
P_GPA	Average Predicted: GPA	0.0019	1	0
ASE	Average Squared Error	0.4104	0	1
RASE	Root Average Squared Error	0.1267	0	1
MAE	Mean Absolute Error	0.1426	0	1
RMAE	Root Mean Absolute Error	0.0622	0	1
MSLE	Mean Squared Logarithmic Error	0.0267	0	1
RMSLE	Root Mean Squared Logparithmic Error	0.0309	0	1
PREDICTED	Average Prediction	0.0019	1	0

Chapter 4
ASTORE Procedure

Contents

Overview: ASTORE Procedure **48**
PROC ASTORE Features 48
Using SAS Viya Workbench 49
Getting Started: ASTORE Procedure **49**
Home Equity Data Set 49
Creating an Analytic Store 50
Using the Analytic Store 51
Syntax: ASTORE Procedure **52**
PROC ASTORE Statement 52
CHECK Statement 52
DESCRIBE Statement 53
DOWNLOAD Statement 54
EXTRACT Statement 55
SAVEAS Statement 55
SCORE Statement 60
SETOPTION Statement 61
UPLOAD Statement 69
Details: ASTORE Procedure **69**
ODS Table Names 70
Procedures Producing Analytic Stores 71
Backward and Forward Compatibility 72
Examples: ASTORE Procedure **73**
Example 4.1: Scoring a Single Record 73
Example 4.2: Describing the Store 74
Example 4.3: Describing and Scoring with Options 76
Example 4.4: Describing the Open Neural Network Exchange (ONNX) Model 78
Example 4.5: Downloading the Store to the Local File System 79
Example 4.6: Uploading the Local Store from the Local File System 79
Example 4.7: Scoring with Multiple Stores Using the EPCODE= Option 79
Example 4.8: Scoring with Multiple Stores That Form a Directed Acyclic Graph (DAG) 81
Example 4.9: Saving an Open Neural Network Exchange (ONNX) Model 82
Example 4.10: Saving a YOLO Model 84
Example 4.11: Extracting the Saved Open Neural Network Exchange (ONNX) Model 88

Overview: ASTORE Procedure

The ASTORE procedure is an interactive procedure in which each statement runs immediately. PROC ASTORE enables you to describe an analytic store, manage the analytic store, and use the analytic store to score new data. The analytic store is the result of a SAVESTATE statement from another analytic procedure; it exists in memory in a binary file format that contains that procedure's state after the training phase of data analysis is completed. The data table that is specified in the SAVESTATE statement references the analytic store in memory. To save the analytic store on disk, use the DOWNLOAD statement.

PROC ASTORE Features

The ASTORE procedure enables you to do the following:

- describe limited information about the analytic store
- produce different types of DS2 scoring code that can run using the DS2 procedure
- score an input data table and produce an output data table by using a specified analytic store and optional DS2 scoring code that uses the analytic store
- use multiple specified analytic stores and DS2 scoring code to score an input data table and produce an output data table
- consume code that is created by a DESCRIBE statement; you can edit the code and send it again in a SCORE statement (because PROC ASTORE is interactive)
- score an input data table and produce an output data table by using multiple specified analytic stores
- check an Open Neural Network Exchange (ONNX) model
- describe certain information about an ONNX model
- save an ONNX model to an analytic store
- extract the ONNX model that is saved to an analytic store
- set an option by using a string value or a list of doubles
- score new data by using an ONNX model on GPUs

Using SAS Viya Workbench

SAS Viya Workbench has greatly simplified the SAS programming experience by offering a syntax style nearly identical to that of SAS 9 and earlier releases. However, there are minor exceptions. In the past, SAS Viya procedures required the use of a CAS engine libref. This engine served multiple purposes, but it was primarily used to upload data to the CAS server and manage the data. Its additional purposes included support of new data types and improved performance for large data sets. No CAS server is needed for SAS Viya Workbench, because all computations are done in place for you in a single location.

In the following program, the LIBNAME statement creates a mylib libref that the examples in this chapter use to refer to your current or desired work directory:

```
libname mylib ".";
```

For more information, see *SAS Viya Workbench: Getting Started*.

Getting Started: ASTORE Procedure

Home Equity Data Set

Several examples in this chapter use the home equity data set hmeq, which is available in the Sampsio library that SAS provides. The data set contains observations for 5,960 mortgage applicants. A variable named Bad indicates whether the customer has paid on his or her loan or has defaulted on it. Table 4.1 describes the variables in Hmeq.

Table 4.1 Variables in the Home Equity (Hmeq) Data Set

Variable	Role	Level	Description
Bad	Response	Binary	1 = customer defaulted on the loan or is seriously delinquent 0 = customer is current on loan payments
CLAge	Predictor	Interval	Age of oldest credit line in months
CLNo	Predictor	Interval	Number of credit lines
DebtInc	Predictor	Interval	Debt-to-income ratio
Delinq	Predictor	Interval	Number of delinquent credit lines
Derog	Predictor	Interval	Number of major derogatory reports
Job	Predictor	Nominal	Occupational category
Loan	Predictor	Interval	Requested loan amount
MortDue	Predictor	Interval	Amount due on existing mortgage
nInq	Predictor	Interval	Number of recent credit inquiries
Reason	Predictor	Binary	`'DebtCon'` = debt consolidation `'HomeImp'` = home improvement
Value	Predictor	Interval	Value of current property
YoJ	Predictor	Interval	Years at present job

You can load the sampsio.hmeq data set by specifying your libref in the following DATA step. These statements assume that your libref is named mylib (as in the section "Using SAS Viya Workbench" on page 49), but you can substitute any appropriately defined libref.

This DATA step includes an id variable, which is bound to the observation number _N_. The id variable is used to join the input records with their corresponding scores.

```
data mylib.hmeq;
   set sampsio.hmeq;
   id = _N_;
run;
```

One or more variables in the input data table should act as the record identifier to resolve any differences in data order in the scoring output data table. Here the record identifier is a single variable id.

Creating an Analytic Store

When you score complex analytic models, you must use a different approach from the usual simple scoring code. Instead you can save the state of the model in a binary file called an analytic store. This binary file can be used later to score the model. This example shows how to produce an analytic store that is created by the SVMACHINE procedure. For more information about the SVMACHINE procedure, see Chapter 35, "SVMACHINE Procedure."

The following PROC SVMACHINE call specifies two INPUT statements: one specifies the variables loan, mortdue, value as interval variables; the other specifies the variables reason, job, delinq, and ninq as classification variables. The TARGET statement indicates that the variable bad is chosen as the target. The ID statement indicates that the id variable must be present in the output data table in order to join records from the input table to their corresponding record in the output table. The SAVESTATE statement saves the state of the SVMACHINE procedure in the analytic store, which is stored in the table mylib.savehmeq.

```
proc svmachine data=mylib.hmeq;
   input loan mortdue value /level=interval;
   input reason job delinq ninq /level=nominal;
   target bad;
   id id;
   savestate rstore=mylib.savehmeq;
run;
```

NOTE: PROC ASTORE does not have to run immediately after PROC SVMACHINE.

Two different procedure runs at different times will produce two analytic stores that have different keys, even though the two runs might appear to be identical. Running the preceding code twice will produce two stores with different key identifiers.

Using the Analytic Store

The most important task of the ASTORE procedure is to score an input table by using the information in the analytic store, which is stored in a data table.

In this example, the input data table is mylib.hmeq, the output data table is mylib.scoreout1, and the analytic store is in the data table mylib.savehmeq.

```
proc astore;
   score data=mylib.hmeq
         out=mylib.scoreout1
         rstore=mylib.savehmeq;
quit;
data scoreout1;
   set mylib.scoreout1;
run;
proc sort data=scoreout1;
   by id;
run;
```

The following statements print the observations, as shown in Output 4.1.

```
proc print data=scoreout1(obs=5);
run;
```

Figure 4.1 Scoring with PROC ASTORE

Obs	id	_P_	P_BAD1	P_BAD0	I_BAD	_WARN_
1	1	1.00000	0.00000	1.00000	0	
2	2	1.00000	0.00000	1.00000	0	
3	3	1.00000	0.00000	1.00000	0	
4	4	-0.84616	0.64103	0.35897	1	M
5	5	1.00000	0.00000	1.00000	0	

Syntax: ASTORE Procedure

The following statements are available in the ASTORE procedure:

PROC ASTORE ;
  CHECK *check-options* ;
  DESCRIBE *describe-options* ;
  DOWNLOAD *download-options* ;
  EXTRACT *extract-options* ;
  SAVEAS *saveas-options* ;
  SCORE *score-options* ;
  SETOPTION *name value* ;
  UPLOAD *upload-options* ;

PROC ASTORE is interactive: each statement is executed immediately.

The following sections describe the PROC ASTORE statement and then describe the other statements in alphabetical order.

PROC ASTORE Statement

PROC ASTORE ;

The PROC ASTORE statement invokes the procedure and does not require any options.

CHECK Statement

CHECK ONNX=*local-file-name* ;

The CHECK statement checks whether a file that you specify by *local-file-name* is a valid Open Neural Network Exchange (ONNX) model.

You must specify the following option:

ONNX=*local-file-name*
specifies either the file reference or the full path of the local file.

You must specify at least one ONNX= option. You can specify as many additional ONNX= options as you need.

DESCRIBE Statement

DESCRIBE STORE=*local-file-name* < *describe-options* > **;**

DESCRIBE RSTORE=*libref.data-table* < *describe-options* > **;**

DESCRIBE ONNX=*local-file-name* **;**

The DESCRIBE statement specifies the name or identifier of an analytic store or an Open Neural Network Exchange (ONNX) model. You can specify an analytic store either in the local file system by using the STORE= option or in a data table by using the RSTORE= option. You can specify an ONNX model in the local file system only by using the ONNX= option. The DESCRIBE statement can also produce DS2 basic scoring code if you specify an analytic store. You can edit the basic scoring code to add transformations to the input variables, flag or override the decision made for the record, work with ensembles, and so on.

Because PROC ASTORE is interactive, you can edit the result from the DESCRIBE statement and submit it in a subsequent SCORE statement. The edited file must comply with the DS2 language syntax. For more information about the DS2 language, see *SAS DS2 Language Reference*.

You must specify exactly one of the following options:

STORE=*local-file-name*
specifies either the file reference or the full path of a valid store file that was created earlier by some analytic engine and exists in the local file system.

RSTORE=*libref.data-table*
specifies the table that contains the analytic store. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the input data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 49.

ONNX=*local-file-name*
specifies either the file reference or the full path of a valid ONNX file.

You can also specify the following *describe-option* if you specify an analytic store:

EPCODE=*code-file-name*
sends DS2 language code to an external code file that is identified by *code-file-name*, which is either the file reference or the full path name and member name of the external code file.

NOTE: The DS2 code enables you to score concurrently with multiple analytic stores as long as they share the same input and output variables. The store key identifier plays an important role in managing multiple analytic stores in a single run.

You can also specify the following *describe-option* if you want the input variable declarations in the generated DS2 code to have the properties that are specified in the input data table:

DATA=*libref.data-table*
names the input data table for PROC ASTORE to use. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to the data. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 49.

data-table specifies the name of the input data table.

You can also specify the following *describe-option* if you want to copy some variables from the input data table to the output data table while scoring with the generated DS2 code:

COPYVARS=*variable* | (*variables*)
COPYVAR=*variable* | (*variables*)
copies one or more *variables* from the input data table that you specify in the DATA= option to the output data table. If you specify them inside parentheses, you can abbreviate the *variables* as a *SAS variable* list. For more information about *SAS variable* lists, see SAS Variable Lists.

NOTE: The COPYVARS= option must follow the DATA= option, which provides the input data table.

DOWNLOAD Statement

DOWNLOAD RSTORE=*libref.data-table* **STORE=***store-file-name* **;**

The DOWNLOAD statement retrieves an external binary analytic store that was produced by another procedure and stores it in the local file system.

You must specify the following options in any order:

RSTORE=*libref.data-table*
specifies a data table that contains the state to be downloaded. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the input data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 49.

STORE=*local-file-name*
specifies either the file reference or the full path of the local analytic store file to which the contents of the data table are to be downloaded. By default, the downloaded file will be located in the directory where SAS is running.

EXTRACT Statement

EXTRACT ONNX=*local-file-name* **RSTORE=***libref.data-table* **;**

EXTRACT ONNX=*local-file-name* **STORE=***local-file-name* **;**

The EXTRACT statement extracts the Open Neural Network Exchange (ONNX) model that is saved to an analytic store and stores it in the local file system. You can specify an analytic store either in the local file system by using the STORE= option or in a data table by using the RSTORE= option. You can specify an ONNX model in the local file system only by using the ONNX= option.

You must specify the following options, in any order:

ONNX=*local-file-name*
specifies either the file reference or the full path of the local ONNX file in which the contents of the extracted ONNX model are to be stored.

RSTORE=*libref.data-table*
specifies a data table that contains the saved ONNX model. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the input data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 49.

STORE=*local-file-name*
specifies either the file reference or the full path of a valid store file that contains the saved ONNX model.

NOTE: You must specify either one RSTORE= option or one STORE= option.

SAVEAS Statement

SAVEAS ANCHORBOXES=(*anchors***) DATA=***libref.data-table* **INPUT=(***input-options***)** < . . . > < **INPUT=(***input-options***)** > **INTOFORMATTEDLEN=***into-formatted-len* **LABELS=(***output-labels***) MAXOBJECTS=***max-objects* **MODELTYPE=***model-type* **NUMANCHORS=***num-anchors* **ONNX=***local-file-name* **OUTPUT=(***output-options***)** < . . . > < **OUTPUT=(***output-options***)** > **ORIGIMGDIMVARS=(***variables***) RSTORE=***libref.data-table* **STRIDES=(***strides***) STORE=***local-file-name* **XYSCALE=(***xyscale***) ;**

The SAVEAS statement enables you to save an Open Neural Network Exchange (ONNX) model to an analytic store.

You must specify the following values:

DATA=*libref.data-table*
names the input data table for PROC ASTORE to use. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to the data. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 49.

data-table specifies the name of the input data table.

INPUT=(*input-options***)**

INPUTS=(*input-options***)**

specifies the mapping information for the input value information of the ONNX model that you specify in ONNX= option. You can specify the following *input-options*:

DECODED

specifies that the input image be decoded in pixels. By default, the input image is encoded.

INPUTCOLORORDER=RGB | BGR

specifies the order of the image colors of the input variable that you specify in the VARS= option. When INPUTCOLORORDER=RGB, the input variable has image colors in the following order: red, green, and blue. When INPUTCOLORORDER=BGR, the input variable has image colors in the following order: blue, green, and red. By default, INPUTCOLORORDER=RGB.

INPUTSHAPEORDER=NCHW | NHWC

specifies the order of the image dimensions of the input variable that you specify in the VARS= option. When INPUTSHAPEORDER=NCHW, the input variable has image dimensions in the following order: number of observations, color channels, height, and width. When INPUTSHAPEORDER=NHWC, the input variable has image dimensions in the following order: number of observations, height, width, and color channels. By default, INPUTSHAPEORDER=NCHW.

MODELCOLORORDER=RGB | BGR

specifies the order of the image colors of the input tensor for the ONNX model. When MODELCOLORORDER=RGB, the input tensor has image colors in the following order: red, green, and blue. When MODELCOLORORDER=BGR, the input tensor has image colors in the following order: blue, green, and red. By default, MODELCOLORORDER=RGB.

MODELSHAPEORDER=NCHW | NHWC

specifies the order of the image dimensions of the input tensor. When MODELSHAPEORDER=NCHW, the input tensor has image dimensions in the following order: number of observations, color channels, height, and width. When MODELSHAPEORDER=NHWC, the input tensor has image dimensions in the following order: number of observations, height, width, and color channels. By default, MODELSHAPEORDER=NCHW.

NORMFACTOR=*number*

specifies the factor that is used to normalize the input. If you specify this *number* together with the PREPROCESS=NORMALIZE option, each input is divided by *number*.

NORMMEAN=(*number-list***)**

specifies the mean for each color that is used to normalize the input. You must specify *number-list* in the following order: red, green, blue.

NORMSTD=(*number-list***)**

specifies the standard deviation for each color that is used to normalize the input. You must specify *number-list* in the following order: red, green, blue.

If you specify *number-list* together with the PREPROCESS=NORMALIZE and NORMMEAN=*number-list* options, each color pixel is standardized by the mean and standard deviation that you specify in *number-list*.

PREPROCESS=NORMALIZE
specifies the method that is used to preprocess the input and normalize the input.

SHAPE=(*number-list***)**
specifies the size of each dimension of the input tensor. Use this option only when the input tensor has a missing dimension size. If a model accepts a variable-sized input tensor such as Faster R-CNN, then *number-list* is the maximum size of each dimension of the input tensor.

VALUE=(*number-list***)**
VALUES=(*number-list***)**
specifies the initialization values for the input tensor. The number of values in *number-list* must match the size of the input tensor.

NOTE: If you specify both the VARS= and VALUES=*number-list* options, the VARS= option takes precedence.

VARS=*variable* **|** **(***variables***)**
VAR=*variable* **|** **(***variables***)**
specifies one or more variables that are mapped to an input tensor of the ONNX model.

VARBINARYTYPE= DOUBLE | FLOAT | INT8 | INT16 | INT32 | INT64 | UINT8 | UINT16 | UINT32 | UINT64

specifies the data type of the variable that you specify in the VARS=*variables* option. You can specify the following data types:

DOUBLE	specifies that the data type is a 64-bit double-precision floating-point number.
FLOAT	specifies that the data type is a 32-bit floating-point number.
INT8	specifies that the data type is a signed 8-bit integer.
INT16	specifies that the data type is a signed 16-bit integer.
INT32	specifies that the data type is a signed 32-bit integer.
INT64	specifies that the data type is a signed 64-bit integer.
UINT8	specifies that the data type is an unsigned 8-bit integer.
UINT16	specifies that the data type is an unsigned 16-bit integer.
UINT32	specifies that the data type is an unsigned 32-bit integer.
UINT64	specifies that the data type is an unsigned 64-bit integer.

This VARBINARYTYPE= option applies only when you specify one variable for the input tensor. By default, VARBINARYTYPE=UINT8.

If you specify more than one INPUT= option, the order must match that of the input value information in the DESCRIBE statement. The number of INPUT= options must match the number of input tensors of the ONNX model.

NOTE: If you specify the INPUT= option, it must follow the DATA= option.

ONNX=*local-file-name*
specifies either the file reference or the full path of a valid ONNX file.

OUTPUT=(*output-options***)**
OUTPUTS=(*output-options***)**
specifies the mapping information for the output value information of the ONNX model that you specify in the ONNX= option. You can specify the following *output-options*:

LABEL=(*string-list***)**
LABELS=(*string-list***)**
specifies the labels for the output tensor. The number of labels must match the total dimension size that you specify in the LABELDIM=(*number-list*) option.

LABELDIM=(*number-list***)**
LABELDIMS=(*number-list***)**
specifies the dimension indices for the LABEL=(*string-list*) option. Use the LABELDIM=(*number-list*) option when you specify the LABEL=(*string-list*) option.

NAME=*var-name*
specifies the name of the output variable that is mapped to an output tensor of the ONNX model.

POSTPROCESS=SOFTMAX
applies the SOFTMAX method to the output.

SHAPE=(*number-list***)**
specifies the size of each dimension of the output tensor. Use this *number-list* only when the output tensor has a missing dimension size.

If you specify more than one OUTPUT= option, the order must match that of the output value information in the DESCRIBE statement. The number of OUTPUT= options must match the number of output tensors of the ONNX model.

NOTE: If you specify the MODELTYPE= option, the OUTPUT= option is ignored.

RSTORE=*libref.data-table*
specifies the data table to contain the analytic store that saves the ONNX model that you specify in the ONNX= option.

libref.data-table is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the input data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 49.

STORE=*local-file-name*
specifies the local file to contain the analytic store that saves the ONNX model that you specify in the ONNX= option.

NOTE: You must specify either one RSTORE= option or one STORE= option.

You can also specify the following options:

ANCHORBOXES=(*anchors*)
specifies the shape of the anchor boxes for the model that you specify in the ONNX= option. The shape of an anchor box consists of the width and height of the box. The order of values in the *number-list* is as follows: the width of the first anchor box, the height of the first anchor box, the width of the second anchor box, the height of the second anchor box, and so on. The number of values in the *number-list* must be 2 times the *number* that you specify in the NUMANCHORS= option.

INTOFORMATTEDLEN=*into-formatted-len*
specifies the length of the INTO column for scoring. You must specify this option if you want the INTO column to appear in the scoring output for classification models.

LABELS=(*output-labels*)
LABEL=(*output-labels*)
specifies the labels for the output classes. The number of labels must match that of the ONNX model.

MAXOBJECTS=*max-objects*
specifies the maximum number of objects to be detected. Use this option when you specify the MODELTYPE= option. By default, MAXOBJECTS=10.

MODELTYPE=FASTERRCNN | SSD | YOLOV2 | YOLOV3 | YOLOV4
specifies the model type for the ONNX file that you specify in the ONNX= option. You can specify the following model types:

FASTERRCNN	specifies a Faster R-CNN model.
SSD	specifies a single-stage detector model.
YOLOV2	specifies a YOLO version 2 model.
YOLOV3	specifies a YOLO version 3 model.
YOLOV4	specifies a YOLO version 4 model.

If you specify the MODELTYPE= option, the model-specific postprocessing is applied during scoring.

NUMANCHORS=*num-anchors*
specifies the number of anchor boxes for the model that you specify in the ONNX= option. Use this option when you specify the MODELTYPE=YOLOV2 or YOLOV4 option.

ORIGIMGDIMVARS=(*variables*)
specifies variables that correspond to the dimensions of the original image. You must specify the height as the first variable, then specify the width.

STRIDES=*stride* | (*strides*)
STRIDE=*stride* | (*strides*)
specifies the stride(s) of the output tensor(s) for the model that you specify in the ONNX= option. Use this option when you specify the MODELTYPE=YOLOV4 option.

XYSCALE=(*xyscale*)
specifies the scale for the X and Y coordinates of the bounding boxes for the model that you specify in the ONNX= option. Use this option when you specify the MODELTYPE=YOLOV4 option.

SCORE Statement

SCORE DATA=*libref.data-table* **OUT=***libref.data-table* < . . . > < **OUT=***libref.data-table* > **RSTORE=***libref.data-table* < . . . > < **RSTORE=***libref.data-table* > < **COPYVARS=(***variables***)** > < **EPCODE< = >***code-file-name* > **;**

The SCORE statement enables you to score both simple and complex models.

You must specify the following options:

DATA=*libref.data-table*
names the input data table for PROC ASTORE to use. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to the data. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 49.

data-table specifies the name of the input data table.

NOTE: If you use the GENOBS option when you score a generative analytic store, the DATA= option is not required.

OUT=*libref.data-table*
names the output data table where PROC ASTORE saves the scored results for the specified input data and analytic store. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to where the data table is to be stored. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 49.

data-table specifies the name of the output data table.

You must specify one OUT= option for each model except the text analytical models. You can specify additional OUT= options, one for each table that the model produces. You can use the DESCRIBE statement to see how many output tables a model produces. If a model produces more than one output table, you must specify the OUT= options in the order in which they are listed in the DESCRIBE statement.

You must specify one of the two following options. You can specify as many additional STORE= or RSTORE= options as you need (but you cannot specify both the STORE= and RSTORE= options). When you specify more than one STORE= or RSTORE= option and you omit the EPCODE= option, you must make sure that the dependence among these analytic stores forms a directed acyclic graph (DAG). An analytic store A is said to depend on an analytic store B when at least one input variable of store A is an output variable of store B.

STORE=*local-file-name*
specifies either the file reference or the full path of a valid store file that was created earlier by some analytic engine and exists in the local file system.

RSTORE=*libref.data-table*
specifies the table that contains the analytic store. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the input data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 49.

You can also specify the following options:

COPYVARS=*variable* | **(***variables***)**

COPYVAR=*variable* | **(***variables***)**
transfers one or more *variables* from the input data table that you specify in the DATA= option to the output data table that you specify in the OUT= option. If you specify them inside parentheses, you can abbreviate the *variables* as a *SAS variable* list. For more information about *SAS variable* lists, see SAS Variable Lists.

NOTE: If you specify the COPYVARS= option, it must follow the DATA= option.

EPCODE=*code-file-name*
names the location of the optional scoring code file (which was created by the DESCRIBE statement) and loads that file for scoring. You can use this option when you have changed the contents of the scoring code.

SETOPTION Statement

SETOPTION *name value* **;**

The SETOPTION statement associates a *value* to a specified *name*. During scoring, PROC ASTORE uses the *value* for the option that corresponds to *name*.

name
specifies an identifier for the option, where *name* must begin with an alphabetic character or underscore, can contain only alphanumerics and underscores, and is limited to 200 characters.

value
specifies a *value* for the named option.

Table 4.2 lists the available SETOPTION statement *names* for certain stores, grouped by the procedure that is used to create the store. The *name* and *value* descriptions follow the table.

Table 4.2 SETOPTION Statement *name value* Pairs

Procedure	*name*	*value*
ASTORE	DETECTION_THRESHOLD	*detectionThreshold*
(SAVEAS	IOU_THRESHOLD	*iouThreshold*
Statement)	ONNX_CUDA_DEVICE_ID	*cudaDeviceId*

Table 4.2 *continued*

Procedure	*name*	*value*
	ONNX_EXECUTION_PROVIDER	*execProvider*
	ONNX_EXECUTION_RESUME_ON_CPU	*resumeOnCpu*
	ONNX_OPENVINO_DEVICE_ID	*openvinoDeviceId*
	ONNX_OPENVINO_DEVICE_TYPE	*openvinoDeviceType*
BART	ALPHA	*alpha*
	COMPUTE_CREDIBLE_INTERVAL	*credlimit*
	OUTPUT_SAMPLES	*outsamp*
	SET_INTO_CUTPOINT	*cutpoint*
FOREST	QUANTILES	*quantiles*
	TREESHAP	*treeshap*
	VOTETYPE	*votetype*
GAMMOD	ALPHA	*alpha*
	COMPUTE_COMPONENTWISE_STATS	*compstat*
	COMPUTE_CONFIDENCE_LIMIT	*conflimit*
	COMPUTE_INTO	*into*
	COMPUTE_LINEAR_PREDICTOR	*linpred*
	COMPUTE_STANDARD_ERROR	*stderr*
	SET_INTO_CUTPOINT	*cutpoint*
GAMSELECT	COMPUTE_COMPONENTWISE_STATS	*compstat*
	COMPUTE_INTO	*into*
	COMPUTE_LINEAR_PREDICTOR	*linpred*
	SET_INTO_CUTPOINT	*cutpoint*
GENSELECT	SET_INTO_CUTPOINT	*cutpoint*
GRADBOOST	TREESHAP	treeshap
KPCA	ITER_NPC	*iternpc*
	ITER_NUM	*iternum*
	ITER_THRESH	*iterthresh*
	KPCA_NPC	*kpcanpc*
	SCORING_MODE	*scoringmode*
LOGSELECT	SET_INTO_CUTPOINT	*cutpoint*
MBANALYSIS	ARM_SCORING_ALGO	*algo*
	NORM	*norm*
RECENGINE	AGGREGATION_METHOD	*aggregationmethod*
	REC_ITEM_FILTER	*recitemfilter*
	REC_TOP_N	*rectopn*
REGSELECT	ALPHA	*alpha*
	COMPUTE_CONFIDENCE_LIMIT	*conflimit*
	USE_ALL_VARS	*useallvars*
RPCA	RPCA_ANOMALYDETECTION_METHOD	*admethod*

Table 4.2 *continued*

Procedure	*name*	*value*
	RPCA_NUMSIGVARS	*numsigvars*
	RPCA_PROJECTION_TYPE	*projtype*
	RPCA_SIGMACOEF	*sigmacoef*
	RPCA_USEMATRIX	*usematrix*
	USEGPU	*usegpu*
STYLEGAN	DEVICE0	*device0*
	DEVICE1	*device1*
	DEVICEk	*devicek*
	GENOBS	*genobs*
	NDEVICES	*ndevices*
	NUMTRUNCATIONLATENTS	*numtruncationlatents*
	SEED	*seed*
	TRUNCATION	*truncation*
	USEGPU	*usegpu*
SVDD	BWCOL	*bwcol*
	MODELID	*modelid*
TABULARGAN	DEVICE0	*device0*
	DEVICE1	*device1*
	DEVICEk	*devicek*
	GENOBS	*genobs*
	NDEVICES	*ndevices*
	SEED	*seed*
	USEGPU	*usegpu*
TEXTBERT	DEVICE0	*device0*
	DEVICE1	*device1*
	DEVICEk	*devicek*
	DOCUMENT_BATCH_SIZE	*documentbatchsize*
	INCLUDE_ALL_SCORES	*includeallscores*
	NDEVICES	*ndevices*
	SEED	*seed*
	USEGPU	*usegpu*

The following *name value* pairs are available for the procedures that are listed in Table 4.2:

AGGREGATION_METHOD *aggregationmethod*

specifies the type of aggregation to apply in pooling similarity scores of nearest neighbors. If you specify a value of 0, similarity scores for repeated nearest neighbors are summed. If you specify a value of 1, similarity scores for repeated nearest neighbors are averaged. If you specify a value of 2, similarity scores for repeated nearest neighbors are summed with mean adjustment. If you specify a value of 3, similarity scores for repeated nearest neighbors are summed with Z-score adjustment. (For more information, see the **recKnn** action in *SAS Viya: Machine Learning Programming Guide*.) The default value is 0. **NOTE:** This option is available only when you specify the KNN statement in the RECENGINE procedure run that is used to train the model.

ALPHA *alpha*
sets the significance level to be used for the construction of confidence intervals. You can specify a value between 0 and 1. The default value is 0.05. This parameter influences only the confidence intervals and does not affect the predicted values.

ARM_SCORING_ALGO *algo*
specifies which algorithm to use for scoring. You can specify a value of 1 (use Association Rules) or 2 (use Exclusive Recommendation) or 3 (use Recommendation). The default value is 1.

BWCOL *bwcol*
specifies whether to output the bandwidth columns. You can specify a value of 0 (do not output) or 1 (output). The default value is 0.

COMPUTE_COMPONENTWISE_STATS *compstat*
specifies whether to compute the following componentwise statistics for each spline term, if they have been requested: linear predictor, standard error, and confidence band. You can specify a value of 0 (do not compute) or 1 (compute). The default value is 0.

COMPUTE_CONFIDENCE_LIMIT *conflimit*
specifies whether to produce additional statistics such as confidence intervals, where *conflimit* is either 0 or 1. If set to 1, additional statistics are produced for each observation, including the confidence intervals for generalized linear models; for generalized additive models, Bayesian confidence bands are computed. The default value is 0.

COMPUTE_CREDIBLE_INTERVAL *credlimit*
specifies whether to produce additional statistics such as equal-tail credible intervals. You can specify a value of 0 (do not output) or 1 (output). The default value is 0.

COMPUTE_INTO *into*
specifies whether to compute the predicted response level. You can specify a value of 0 (do not compute) or 1 (compute). The default value is 0.

COMPUTE_LINEAR_PREDICTOR *linpred*
specifies whether to compute the predicted response level. You can specify a value of 0 (do not compute) or 1 (compute). The default value is 0.

COMPUTE_STANDARD_ERROR *stderr*
specifies whether to compute the standard error of the linear predictor. You can specify a value of 0 (do not compute) or 1 (compute). The default value is 0.

DEVICE0 *device0*
enables the first GPU device in the system. This applies only when the value of USEGPU *usegpu* is set to 1 and the value of NDEVICES *ndevices* is set to be greater than or equal to 1.

DEVICE1 *device1*
enables the second GPU device in the system. This applies only when the value of USEGPU *usegpu* is set to 1 and the value of NDEVICES *ndevices* is set to be greater than or equal to 2.

DEVICEk *devicek*
enables the $k + 1$th GPU device in the system. This applies only when the value of USEGPU *usegpu* is set to 1 and the value of NDEVICES *ndevices* is set to be greater than or equal to $k + 1$.

DETECTION_THRESHOLD *detectionThreshold*
specifies the threshold of the score for detecting objects. The valid range is from 0 to 1. The default value is 0.2.

DOCUMENT_BATCH_SIZE *documentbatchsize*
specifies the quantity of documents to process at a time. This enables you to control the procedure's memory use. This option's influence on the final model is only stochastic, because the training batch size (number of documents that are required to update the weights) is fixed at 16 as an internal hyperparameter. You can specify a value between 1 and 1,024. The default value is 16.

GENOBS *genobs*
specifies the number of output observations to generate when you score a generative model without input data. The valid value is a positive integer.

INCLUDE_ALL_SCORES *includeallscores*
specifies an option to include a score for each classifier label in the output table. You can specify a value of 0 (output only the highest score) or 1 (output all scores). The default value is 0.

IOU_THRESHOLD *iouThreshold*
specifies the threshold of intersection-over-union for matching the bounding boxes. The valid range is from 0 to 1. The default value is 0.3.

ITER_NPC *iternpc*
specifies the number of principal components to use in the KPCA iterative pre-image method. The valid value is a positive integer no greater than the rank of the kernel matrix (the rank is stored in the state). PROC ASTORE issues an error if *iternpc* is greater than the kernel matrix rank. If the user inputs decimals, the number is cast to an integer. The default value is the same as the number of principal components for the pre-image that you specify during the KPCA training.

ITER_NUM *iternum*
specifies the maximum number of iterations to use in the KPCA iterative pre-image method. The valid value is a positive integer. If the user inputs decimals, the number is cast to an integer. The default value is 1,000.

ITER_THRESH *iterthresh*
specifies the convergence threshold to use in the KPCA iterative pre-image method. The valid value is a nonnegative number less than 1. The default value is 1E–5.

KPCA_NPC *kpcanpc*
specifies the number of principal components to obtain from scoring new data. The valid value is a positive integer no greater than the rank of the kernel matrix (the rank is stored in the state). PROC ASTORE issues an error if *kpcanpc* is greater than the kernel matrix rank. If the user inputs decimals, the number is cast to an integer. The default value is the same as the number of principal components that you specify during the KPCA training.

MODELID *modelid*
specifies the ID of the model that is used for scoring, if multiple bandwidths are used to train the models for support vector data description (SVDD). You can specify *modelid* as a number between 0 and the number of bandwidths that is used to train the models for SVDD. The default value is 0, which means that scoring uses all models for SVDD.

NDEVICES *ndevices*
specifies the number of GPU devices to use. This applies only when the value of USEGPU *usegpu* is set to 1.

NORM *norm*
specifies whether to normalize the input while scoring. You can specify a value of 0 (do not normalize) or 1 (normalize). The default value is 0.

NUMTRUNCATIONLATENTS *numtruncationlatents*
specifies the number of latents to use in calculating the truncation mean.

ONNX_CUDA_DEVICE_ID *cudaDeviceId*
specifies the ID of the GPU device in the system when *execProvider* is set to 'cuda'. The value of *cudaDeviceId* must be less than the number of GPU devices in the system. The default value is 0.

ONNX_EXECUTION_PROVIDER *execProvider*
specifies the execution provider for the ONNX run time. The value of *execProvider* must be 'cpu', 'cuda', 'openvino', or 'tensorrt'. The value is case-insensitive. The default value is 'cpu'.

ONNX_EXECUTION_RESUME_ON_CPU *resumeOnCpu*
specifies whether to resume execution on the CPU. The value of *resumeOnCpu* must be either 0 or 1. If you set *resumeOnCpu* to 1, the execution resumes on the CPU when there is no such device set by *execProvider* in the system. Otherwise, the execution stops with an error. The default value is 0.

ONNX_OPENVINO_DEVICE_ID *openvinoDeviceId*
specifies the ID of the hardware device in the system when *execProvider* is set to 'openvino'. The value of *openvinoDeviceId* must be the ID of a valid OpenVINO device in the system. The value can be 'CPU', 'GPU', 'GNA', or 'VPU', depending on the hardware devices available in the system. The value is case-sensitive. If you omit this *name value* pair, an arbitrary free device is automatically selected by OpenVINO Runtime.

ONNX_OPENVINO_DEVICE_TYPE *openvinoDeviceType*
specifies the hardware device type and precision when *execProvider* is set to 'openvino'. The value of *openvinoDeviceType* must be a valid OpenVINO device type, and it overrides the hardware type and precision at run time. The value can be 'CPU_FP32', 'GPU_FP32', 'GPU_FP16', 'MYRIAD_FP16', 'VAD-M_FP16', 'VAD-F_FP16', any valid Hetero combination, or any valid Multi or Auto device combination, depending on the hardware devices available in the system. The value is case-sensitive. If you omit this *name value* pair, the default hardware and precision are used.

OUTPUT_SAMPLES *outsamp*
specifies whether to output predictions from each sample that is saved for prediction in addition to the sample average prediction. You can specify a value of 0 (do not output) or 1 (output). The default value is 0.

QUANTILES *quantiles*
specifies a list of quantile values to compute while you are scoring a quantile regression forest, where each value in *quantiles* is a number between 0 and 1 (inclusive) that represents a percentile that is expressed as a decimal. The FOREST procedure must have been trained with the QUANTILE-REGRESSION option in order for this option to take effect.

REC_ITEM_FILTER *recitemfilter*
specifies the type of item filtering to apply to the recommended items. If you set *recitemfilter* to 0, no filtering is applied. If you set *recitemfilter* to a positive value, only the items that the user has not previously observed are recommended.

REC_TOP_N *rectopn*
specifies the number of items to recommend to each input user who is listed in the input scoring data set. The default value is 50.

RPCA_ANOMALYDETECTION_METHOD *admethod*
specifies the method of anomaly detection, where *admethod* must be either 0, 1, 2, or 3. If you set *admethod* to 1, a method called R4S that is based on the root sum of squares of values in the sparse part of the scoring observation is applied. If you set *admethod* to 0, a method called SIGVARS that counts the number of significant values in the sparse part of the scoring observation is applied to the scoring observations. If you set *admethod* to 3, a method called ICANORMS that is based on the norms of the latent sources of the sparse part of the scoring observation is applied. If you set *admethod* to 2, a method called ICASIGVARS that counts the number of significant values in the latent sources of the sparse part of the scoring observation is applied to the scoring observations. The default value is what you specify in the RPCA procedure.

RPCA_NUMSIGVARS *numsigvars*
specifies the minimum number of variables that must have significant values in the sparse matrix for the observation to be considered an anomaly. The default value is what you specify in the RPCA procedure.

RPCA_PROJECTION_TYPE *projtype*
specifies the type of projection, where *projtype* must be either 0, 1, or 2. If you set *projtype* to 1, the scoring observations are projected onto the low-rank subspace. If you set *projtype* to 2, the scoring observations are projected onto the low-rank space, but the sparse part of the observations are stored in the resulting scoring table. If you set *projtype* to 0, the scoring observations are projected onto the principal component space. You cannot set *projtype* to 0 if you specify image inputs during RPCA training. The default value is 0 for non-image RPCA and 1 for image RPCA.

RPCA_SIGMACOEF *sigmacoef*
specifies how many standard deviations away from the mean of the column an entry in the sparse matrix must be to be considered a significant value. The default value is what you specify in the RPCA procedure.

RPCA_USEMATRIX *usematrix*
specifies which matrix to use for standardizing the values of the sparse part of the scoring observations for anomaly detection, where *usematrix* must be either 0 or 1. If you set *usematrix* to 1, the sparse part of the scoring observations is standardized on the basis of the standard deviation of the sparse matrix for anomaly detection. If you set *usematrix* to 0, the sparse part of the scoring observations is standardized on the basis of the standard deviation of the input data for anomaly detection. The default value is what you specify in the RPCA procedure.

SCORING_MODE *scoringmode*
specifies the mode of KPCA scoring. The valid value is either 0 (for the standard KPCA score) or 1 (for the KPCA pre-image score).

SEED *seed*
specifies the seed to use for the random number generator for scoring. A value of 0 means to use the time of day from the computer's clock as the seed.

SET_INTO_CUTPOINT *cutpoint*
specifies the cutpoint value that is used in binary and binomial response models or generalized additive models to classify observations, where *cutpoint* is a number between 0 and 1. If the predicted probability of an observation equals or exceeds *cutpoint*, the observation is classified as an event; otherwise, it is classified as a nonevent. The default value is 0.5. If you specify a value of 0, the default value is used.

TREESHAP *treeshap*
specifies whether to calculate TreeSHAP values to explain the predictions in scoring. You can specify a value of 0 (do not calculate TreeSHAP values) or 1 (calculate TreeSHAP values). The default value is 0. If you set the value to 1 while also specifying the DESCRIBE statement, PROC ASTORE generates a TreeSHAP name mapping table, which maps the calculated TreeSHAP variables to the input variable and predicted target name that they explain. For more information about TreeSHAP values, see Chapter 11, "FOREST Procedure" (*SAS Viya: Machine Learning Procedures*), and Chapter 17, "GRADBOOST Procedure" (*SAS Viya: Machine Learning Procedures*).

TRUNCATION *truncation*
specifies the truncation ratio for image generation.

USE_ALL_VARS *useallvars*
specifies whether to use all predictor variables and the response variable to calculate the statistics for general linear models (GLM). You can specify a value of 0 (do not use) or 1 (use). The default value is 0. You must use the original training data for scoring.

USEGPU *usegpu*
specifies whether to use GPU devices to perform the calculations for scoring. You can specify a value of 0 (do not use) or 1 (use). The default value is 0. **NOTE:** For the STYLEGAN procedure and for the TABULARGAN procedure, only one GPU device is allowed. By default, the GPU device ID is 0. You can change the device ID by setting the value of DEVICEk *devicek* to 1 and the value of NDEVICES *ndevices* to $k+1$, where k is the ID of the device. **NOTE:** For *name value* pairs specific to SAS Event Stream Processing for deep learning, see *SAS Event Stream Processing: Using Streaming Analytics*.

VOTETYPE *votetype*
specifies the voting strategy for forest classification. You can specify a value of 0 (use the majority vote to predict) or 1 (use the average probability to predict). The default value is the voting strategy that you specify in the FOREST procedure.

UPLOAD Statement

UPLOAD *upload-options* **;**

The UPLOAD statement moves an analytic store from the local file system into a data table.

You must specify the following *upload-options* in any order:

RSTORE=*libref.data-table*
specifies the table to which the store is sent. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 49.

STORE=*local-file-name*
specifies either the file reference or the full path of the valid store file that was created earlier by some analytic engine and exists in the local file system.

Details: ASTORE Procedure

The DESCRIBE statement displays the following information about the store:

- store key
- basic information to describe the model
- input variables
- output variables

The DESCRIBE statement displays the following information about the ONNX model:

- basic information to describe the model
- input value information
- output value information

If you specify the EPCODE= option in the DESCRIBE statement, PROC ASTORE produces basic DS2 language statements, which it sends to the SAS log unless you specify an optional *code-file-name* to send them to a specified file. The DS2 code contains empty method blocks like the following:

```
method preScoreRecord();
end;
method postScoreRecord();
end;
method run();
     set sasep.in;          /* read in the record */
     preScoreRecord();      /* Optional: process the input variables as needed */
```

```
    sc.scoreRecord();      /* score one record */
    postScoreRecord();     /* Optional: process the output variables as needed */
end;
```

You can use the **preScoreRecord** method block to transform the input variables, and you can change or flag the scores in the **postScoreRecord** method block.

ODS Table Names

Each table that the ASTORE procedure creates has a name associated with it. You must use this name to refer to the table when you use ODS statements. These names are listed in Table 4.3.

Table 4.3 ODS Tables Produced by PROC ASTORE

Table Name	Description	Statement
Description	Name of the component that saved the state and the time it was saved or information about an ONNX model	DESCRIBE
InputVariables	List of input variables from the procedure that saved the state	DESCRIBE
OutputVariables	List of output variables from the procedure that saved the state	DESCRIBE
InputValueInfo	List of inputs of an ONNX model	DESCRIBE
OutputValueInfo	List of outputs of an ONNX model	DESCRIBE
StoreKey	Key information from the DESCRIBE statement	DESCRIBE
Timing	Timing details **NOTE:** Analytic stores can become very large; breaking down the different stages of scoring a data table is important and informative.	SCORE

Procedures Producing Analytic Stores

You can use the procedures in the following list to produce an analytic store.

- ASTORE (specifically the "SAVEAS Statement" on page 55)
- BART
- BNET
- FACTMAC
- FOREST
- GAMMOD
- GAMSELECT
- GENSELECT
- GMM
- GPCLASS
- GPREG
- GRADBOOST
- HPFOREST
- HPSVM
- KCLUS
- KPCA
- LIGHTGRADBOOST
- LMIXED
- LOGSELECT
- MBANALYSIS
- MBC
- MTLEARN
- NNET
- NOMINALDR
- PCA
- QTRSELECT

- RECENGINE
- REGSELECT
- RPCA
- SPARSEML
- STYLEGAN
- SVDD
- SVMACHINE
- TABULARGAN
- TEXTBERT
- TREESPLIT

Backward and Forward Compatibility

Backward compatibility is guaranteed for analytic stores. Analytic store files that are trained on older SAS platforms and releases are guaranteed to work with PROC ASTORE on newer SAS platforms and releases.

However, forward compatibility is not guaranteed for analytic stores. Analytic store files that are trained on newer SAS platforms and releases are not guaranteed to work with PROC ASTORE on older SAS platforms and releases. For example, an analytic store file that is trained on the newer SAS Viya 4 platform is not guaranteed to work on the older SAS Viya 3.5 (and earlier) and SAS 9.4 (and earlier) platforms. This does not mean that *all* analytic store files that are trained on newer SAS platforms and releases will not work on older SAS platforms and releases; in many cases they will work. Rather, it means that there are some cases where an analytic store that is trained on a newer SAS platform or release will not work with PROC ASTORE on an older SAS platform or release.

Examples: ASTORE Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

Example 4.1: Scoring a Single Record

This is a simple example that creates a table that contains one record whose ID is 100 and uses an analytic store that was produced by the SVMACHINE procedure in the section "Creating an Analytic Store" on page 50.

The following DATA step creates the mylib.hmeq1 table by loading the hmeq data table that was created in section "Home Equity Data Set" on page 49 and extracting the record whose ID is 100:

```
data mylib.hmeq1;
   set mylib.hmeq;
   if (id = 100);
run;
```

The following statements score the mylib.hmeq1 table with the analytic store in the table mylib.savehmeq to produce the output table mylib.hmeq1out, which is shown in Output 4.1.1.

```
proc astore;
   score data=mylib.hmeq1
         rstore=mylib.savehmeq
         out=mylib.hmeq1out;
quit;
proc print data=mylib.hmeq1out ;
run;
```

Output 4.1.1 Scoring a Single Record

Obs	id	_P_	P_BAD1	P_BAD0	I_BAD	_WARN_
1	100	1.00000	6.2878E-8	1.00000	0	M

Example 4.2: Describing the Store

In this simple example, the DESCRIBE statement in the following code produces tables that describe some of the contents of the analytic store as shown in Output 4.2.1 and sends the basic code to the file *svmepcode.sas*:

```
proc astore;
   describe rstore=mylib.savehmeq
            epcode="svmepcode.sas";
quit;
```

Output 4.2.1 includes the following ODS tables:

- The "Store Key" table displays the string identifier of the store. This is the same string that is contained in the code that the EPCODE= option in the DESCRIBE statement produces.
- The "Basic Information" table displays the analytic engine that produced the store and the time when the store was created by processing a SAVESTATE statement.
- The "Input Variables" table displays the input variables.
- The "Output Variables" table displays the output variables.

Output 4.2.1 Output Tables from the DESCRIBE Statement

The ASTORE Procedure

Store Key
2A568F844D26EAFE0AB8E7CA01CA384DB7CF75FF

Basic Information	
Analytic Engine	svmachine
Time Created	18Aug2021:10:33:03

Input Variables

Name	Length	Role	Type	RawType	FormatName
LOAN	8	Input	Interval	Num	
MORTDUE	8	Input	Interval	Num	
VALUE	8	Input	Interval	Num	
REASON	7	Input	Classification	Character	
JOB	7	Input	Classification	Character	
DELINQ	8	Input	Classification	Num	
NINQ	8	Input	Classification	Num	
BAD	8	Target	Classification	Num	
id	8	Id		Num	

Output 4.2.1 *continued*

Output Variables			
Name	**Length**	**Type**	**Label**
id	8	Num	
_P_	8	Num	Decision Function
P_BAD1	8	Num	Predicted: BAD=1
P_BAD0	8	Num	Predicted: BAD=0
I_BAD	12	Character	Into: BAD
WARN	4	Character	Warnings

The following statements show the contents of the *svmepcode.sas* file; this is DS2 language code:

```
data sasep.out;
     dcl package score sc();
     dcl double "LOAN";
     dcl double "MORTDUE";
     dcl double "VALUE";
     dcl nchar(7) "REASON";
     dcl nchar(7) "JOB";
     dcl double "DELINQ";
     dcl double "NINQ";
     dcl double "BAD";
     dcl double "id";
     dcl double "_P_" having label n'Decision Function';
     dcl double "P_BAD1" having label n'Predicted: BAD=1';
     dcl double "P_BAD0" having label n'Predicted: BAD=0';
     dcl nchar(12) "I_BAD" having label n'Into: BAD';
     dcl nchar(4) "_WARN_" having label n'Warnings';
     keep "id" "_P_" "P_BAD1" "P_BAD0" "I_BAD" "_WARN_";
     varlist allvars[_all_];
     method init();
             sc.setvars(allvars);
             sc.setKey(n'14F2E0A7B9A73581F79904C6E900D60ABCC51EDB');
     end;
     method preScoreRecord();
     end;
     method postScoreRecord();
     end;
     method term();
     end;
     method run();
             set sasep.in;
             preScoreRecord();
             sc.scoreRecord();
             postScoreRecord();
     end;
enddata;
```

NOTE: The `sc.setKey` in the `method init` method block contains a string that identifies an analytic store. Every time you produce a new store, the key changes, even if you think the runs are identical.

You can view or edit the *svmepcode.sas* file that resides in the local file system.

```
method preScoreRecord();
   /* insert input variable transformations here */
end;
method postScoreRecord();
   /* change or flag the decisions here */
end;
method run();
     set sasep.in;      /* read in the record */
     preScoreRecord();  /* Optional: process the input variables as needed */
     sc.scoreRecord();  /* score one record */
     postScoreRecord(); /* Optional: process the output variables as needed */
end;
```

Transformations of an input variable should be in the **preScoreRecord** method block. You can alter the decisions that are made from scoring one record in the **postScoreRecord** method block. If you do not intend to alter the contents of the DS2 code in either of these method blocks, then you do not need to specify the EPCODE= option in the SCORE statement.

NOTE: The store key will be different every time you use the SAVESTATE statement in a procedure that supports it.

Example 4.3: Describing and Scoring with Options

This example uses the SETOPTION statement to set options when you describe and score an analytic store. It first creates an analytic store that can be used together with the SETOPTION statement, and then it uses the ASTORE procedure to describe and score the analytic store in accordance with the options that you set.

The following DATA step code creates the data table mylib.baseball by loading the baseball data set, which is available in the Sashelp library that SAS provides:

```
data mylib.baseball;
   set sashelp.baseball;
run;
```

The following PROC REGSELECT call trains a GLM model on the mylib.baseball table. The model predicts the natural logarithm of a Major League Baseball player's salary on the basis of their league, number of hits, and number of home runs. The STORE statement saves the model's state in an analytic store, which is stored in the mylib.baseballsave table.

```
proc regselect data=mylib.baseball;
   class league;
   model logSalary = league nHome nHits;
   store mylib.baseballsave;
run;
```

The following PROC ASTORE call sets the COMPUTE_CONFIDENCE_LIMIT option to 1 via the SETOPTION statement and then describes the contents of the analytic store via the DESCRIBE statement:

```
proc astore;
   setoption COMPUTE_CONFIDENCE_LIMIT 1;
   describe rstore=mylib.baseballsave;
```

```
quit;
```

Output 4.3.1 includes the following ODS tables:

- The "Store Key" table displays the string identifier of the store.
- The "Basic Information" table displays the analytic engine that produced the store and the time when the store was created by processing a SAVESTATE statement.
- The "Input Variables" table displays the input variables.
- The "Output Variables" table displays the output variables. Note that there are additional variables related to the confidence limit because the COMPUTE_CONFIDENCE_LIMIT option was set to 1.

Output 4.3.1 Output Tables from the DESCRIBE Statement

The ASTORE Procedure

Store Key
2E6649EED53BF66EFFFD2A9A2B86AA9FD91D1984

Basic Information	
Analytic Engine	glm
Time Created	22May2023:14:42:16

Input Variables

Name	Length	Role	Type	RawType	FormatName
League	8	Input	Classification	Character	
nHits	8	Input	Interval	Num	
nHome	8	Input	Interval	Num	

Output Variables

Name	Length	Type	Label
P_logSalary	8	Num	Predicted: logSalary
_h_	8	Num	Leverage
lcl	8	Num	Lower Bound of 95% C.I.(Individual Pred)
ucl	8	Num	Upper Bound of 95% C.I.(Individual Pred)
lclm	8	Num	Lower Bound of 95% C.I. for Mean
uclm	8	Num	Upper Bound of 95% C.I. for Mean
stdi	8	Num	Standard Error of Individual Prediction
stdp	8	Num	Standard Error of Mean Predicted Value

The following PROC ASTORE call sets the ALPHA option to 0.01 and the COMPUTE_CONFIDENCE_LIMIT option to 1 via the SETOPTION statement, and then it scores the mylib.baseball table on the analytic store to produce the output data table mylib.baseballout. Note that multiple options are set using multiple SETOPTION statements. The first observation of the mylib.baseballout table is shown in Output 4.3.2. You can see the additional output variables there because the COMPUTE_CONFIDENCE_LIMIT option is set to 1, and the confidence interval values that are shown are influenced by the ALPHA option being set to 0.01.

```
proc astore;
   setoption ALPHA 0.01;
   setoption COMPUTE_CONFIDENCE_LIMIT 1;
   score data=mylib.baseball
         rstore=mylib.baseballsave
         out=mylib.baseballout;
quit;
proc print data= mylib.baseballout (obs=1);
run;
```

Output 4.3.2 First Record of the Scoring Results

Obs	P_logSalary	_h_	_lcl_	_ucl_	_lclm_	_uclm_	_stdi_	_stdp_
1	5.31435	0.015819	3.31077	7.31793	5.06432	5.56438	0.77211	0.096353

Example 4.4: Describing the Open Neural Network Exchange (ONNX) Model

This example uses the DESCRIBE statement to produce tables that describe some of the contents of the Open Neural Network Exchange (ONNX) model that are shown in Output 4.4.1. The following code describes the MNIST Handwritten Digit Recognition model:

```
proc astore;
   describe onnx="&PATH_TO_MNIST_ONNX";
quit;
```

Output 4.4.1 shows the following ODS tables:

- The “Basic Information” table displays basic information about the ONNX model.
- The “Input Value Information” table displays the input information.
- The “Output Value Information” table displays the output information.

Output 4.4.1 Output Tables from the DESCRIBE Statement

The ASTORE Procedure

Basic Information	
IR Version	3
Model Version	1
Domain	ai.cntk
Doc String	
Producer Name	CNTK
Producer Version	2.5.1
Operator Set Domain	
Operator Set Version	8

Input Value Information

Name	Length	Value Type	Shape	Element Type
Input3	3136	Tensor	(1, 1, 28, 28)	FLOAT

Output 4.4.1 *continued*

Output Value Information				
Name	Length	Value Type	Shape	Element Type
Plus214_Output_0	40	Tensor	(1, 10)	FLOAT

Example 4.5: Downloading the Store to the Local File System

The following statements extract the analytic store that is saved in the data table mylib.savehmeq to the local file *svmstore.sasast* in the local file system:

```
proc astore;
   download rstore=mylib.savehmeq
            store="svmstore.sasast";
quit;
```

In addition to downloading the actual file, PROC ASTORE writes a note in the log that indicates how many bytes were downloaded.

Example 4.6: Uploading the Local Store from the Local File System

The following statements send the analytic store *svmstore.sasast* from the local file system to the data table mylib.savehmeq:

```
proc astore;
  upload rstore=mylib.savehmeqnew
         store="svmstore.sasast";
quit;
```

Example 4.7: Scoring with Multiple Stores Using the EPCODE= Option

This example first shows how to produce another analytic store that is created by the FOREST procedure. For more information about the FOREST procedure, see Chapter 10, "FOREST Procedure."

This example uses the same home equity data table (mylib.hmeq) that is loaded as described in section "Home Equity Data Set" on page 49. The SAVESTATE statement in the following FOREST procedure code saves the state in an analytic store, which is stored in the table mylib.savehmeq2:

```
proc forest data=mylib.hmeq seed=17;
   input loan mortdue value /level=interval;
   input reason job delinq ninq /level=nominal;
   target bad;
   id id;
   savestate rstore=mylib.savehmeq2;
run;
```

The following statements score the mylib.hmeq table by using two analytic stores, which are stored in the tables mylib.savehmeq and mylib.savehmeq2, to produce the output table mylib.hmeq2out:

```
proc astore;
   score data=mylib.hmeq rstore=mylib.savehmeq rstore=mylib.savehmeq2
         epcode="epcode2.sas" out=mylib.hmeq2out;
quit;
```

The contents of the *epcode2.sas* file are provided in the following code. This DS2 language code has been modified to score the table by using both analytic stores concurrently; each is referenced by its store key. Both analytic stores equally contribute their computed probability that the "bad" event occurs. This is indicated by the P_BAD1 output variable in the first analytic store (mylib.savehmeq) and the P_BAD output variable in the second analytic store (mylib.savehmeq2), which are consolidated into the P_BAD variable in the overall scoring output.

```
data sasep.out;
   dcl package score sc();
   dcl package score sc2();
   dcl double "LOAN";
   dcl double "MORTDUE";
   dcl double "VALUE";
   dcl nchar(7) "REASON";
   dcl nchar(7) "JOB";
   dcl double "DELINQ";
   dcl double "NINQ";
   dcl double "BAD";
   dcl double "id";
   dcl double "_P_" having label n'Decision Function';
   dcl double "P_BAD1" having label n'Predicted: BAD=1';
   dcl double "P_BAD0" having label n'Predicted: BAD=0';
   dcl nchar(12) "I_BAD" having label n'Into: BAD';
   dcl nchar(4) "_WARN_" having label n'Warnings';
   dcl double "P_BAD";
   dcl double "tempP";
   keep "id" "P_BAD";
   varlist allvars[_all_];
   method init();
         sc.setvars(allvars);
         sc.setKey(n'2A568F844D26EAFE0AB8E7CA01CA384DB7CF75FF');
         sc2.setvars(allvars);
         sc2.setKey(n'FC7E82345DF91C75364F7BFD177BF93AA9129A26');
   end;
   method preScoreRecord();
   end;
   method postScoreRecord();
         P_BAD = 0.5*(P_BAD + tempP);
   end;
   method term();
   end;
   method run();
         set sasep.in;
         preScoreRecord();
         sc.scoreRecord();
         tempP = P_BAD1;
         sc2.scoreRecord();
         postScoreRecord();
```

```
    end;
 enddata;
```

Example 4.8: Scoring with Multiple Stores That Form a Directed Acyclic Graph (DAG)

This example uses two analytic stores that exhibit a directed acyclic graph (DAG) dependence to perform a two-step scoring operation.

The example uses the junkmail data set available in the Sashelp library that SAS provides. The junkmail data set contains 59 variables:

- test: variable that indicates whether an email is labeled for the training or test set
- make to captotal: 57 predictor variables that record the frequencies of some common words and characters and the lengths of uninterrupted sequences of capital letters in emails
- class: binary response variable that indicates whether an email is considered spam or not

The following DATA step code creates the training data table mylib.junkmail_train by loading the training observations from the junkmail data set:

```
data mylib.junkmail_train;
   set sashelp.junkmail;
   if (test = 0);
   drop test;
run;
```

The following PROC PCA call trains a principal component analysis model on the mylib.junkmail_train table. Principal component analysis reduces the dimensionality of data. Here PROC PCA reduces 57 predictor variables to five principal components and then saves the resulting output to the mylib.junkmail_train_reduced data table. The STORE statement saves the model's state in an analytic store, which is stored in the mylib.reduce_save table.

```
proc pca data=mylib.junkmail_train n=5;
   var make--captotal;
   output out=mylib.junkmail_train_reduced copyvar=class;
   store mylib.reduce_save;
run;
```

The following PROC SVMACHINE call specifies the input data table mylib.junkmail_train_reduced, which is the output data table from the preceding PROC PCA call. It also specifies the input variables score1 to score5, which are the five principal components that are extracted there. The SAVESTATE statement saves the state of PROC SVMACHINE in another analytic store, which is stored in the mylib.score_reduced_save table.

```
proc svmachine data=mylib.junkmail_train_reduced;
   input score1-score5 /level=interval;
   target class;
   savestate rstore=mylib.score_reduced_save;
```

```
run;
```

The following DATA step code creates the test data table mylib.junkmail_test by loading the test observations from the junkmail data set:

```
data mylib.junkmail_test;
   set sashelp.junkmail;
   if (test = 1);
   drop test class;
run;
```

The following PROC ASTORE call scores the test data table mylib.junkmail_test by using both analytic stores, which are stored in the tables mylib.reduce_save and mylib.score_reduced_save, to produce the output table mylib.out_test. Notice that the dependence between the two analytic stores forms a DAG. Store mylib.score_reduced_save depends on the store mylib.reduce_save, because the input variables score1 to score5 of the store mylib.score_reduced_save are the principal components that are extracted as the output variables in the store mylib.reduce_save.

```
proc astore;
   score rstore=mylib.score_reduced_save rstore=mylib.reduce_save
         data=mylib.junkmail_test out=mylib.out_test;
run;
```

Example 4.9: Saving an Open Neural Network Exchange (ONNX) Model

This example uses the SAVEAS statement to save an Open Neural Network Exchange (ONNX) model to an analytic store. The following code saves the MNIST Handwritten Digit Recognition model. Notice that a LIBNAME statement creates a mysvlib libref that uses the SASVIYA engine, because the LOADIMAGES procedure uses new data types that require this advanced engine.

```
libname imglib "&PATH_TO_MNIST_IMAGES";
libname mysvlib sasviya;

proc loadimages libref=imglib;
   output out=mysvlib.mnist_valid;
run;

proc astore;
   saveas onnx="&PATH_TO_MNIST_ONNX"
      rstore=mylib.mniststore data=mysvlib.mnist_valid
      input=(vars=(_image_) preprocess=NORMALIZE normFactor=255)
      output=(name="digit" label=("0" "1" "2" "3" "4" "5" "6" "7" "8" "9")
         labeldim=(1))
      intoformattedlen = 2
      ;
quit;
```

After you save the model to an analytic store, you can describe the store as follows:

```
proc astore;
   describe rstore=mylib.mniststore;
quit;
```

Output 4.9.1 contains the following ODS tables:

- The "Store Key" table displays the string identifier of the store.
- The "Basic Information" table displays the analytic engine and the time when the store was created.
- The "Input Variables" table displays the input variables.
- The "Output Variables" table displays the output variables.

Output 4.9.1 Output Tables from the DESCRIBE Statement

The ASTORE Procedure

Store Key
1573EA6C69EE56ABC83A70F679129A65167AD19B

Basic Information	
Analytic Engine	onnx
Time Created	18Aug2021:10:33:22

Input Variables

Name	Length	Role	Type	RawType	FormatName
image	784	Id		Character	

Output Variables

Name	Length	Type	Label
P_digit0	8	Num	Predicted: digit=0
P_digit1	8	Num	Predicted: digit=1
P_digit2	8	Num	Predicted: digit=2
P_digit3	8	Num	Predicted: digit=3
P_digit4	8	Num	Predicted: digit=4
P_digit5	8	Num	Predicted: digit=5
P_digit6	8	Num	Predicted: digit=6
P_digit7	8	Num	Predicted: digit=7
P_digit8	8	Num	Predicted: digit=8
P_digit9	8	Num	Predicted: digit=9
I_digit	2	Character	Into: digit

Example 4.10: Saving a YOLO Model

This example uses the SAVEAS statement to save a You Only Look Once (YOLO) model to an analytic store. The following code describes the YOLO version 3 model:

```
proc astore;
   describe onnx="&PATH_TO_YOLO_ONNX";
quit;
```

Output 4.10.1 shows the following ODS tables:

- The "Basic Information" table displays basic information about the ONNX model.
- The "Input Value Information" table displays the input information.
- The "Output Value Information" table displays the output information.

Output 4.10.1 Output Tables from the DESCRIBE Statement

The ASTORE Procedure

Basic Information	
IR Version	6
Model Version	0
Domain	onnx
Doc String	
Producer Name	keras2onnx
Producer Version	1.6.0
Operator Set Domain	
Operator Set Version	11

Input Value Information				
Name	**Length**	**Value Type**	**Shape**	**Element Type**
input_1	.	Tensor	(., 3, ., .)	FLOAT
image_shape	.	Tensor	(., 2)	FLOAT

Output Value Information				
Name	**Length**	**Value Type**	**Shape**	**Element Type**
yolonms_layer_1	.	Tensor	(1, ., 4)	FLOAT
yolonms_layer_1:1	.	Tensor	(1, 80, .)	FLOAT
yolonms_layer_1:2	.	Tensor	(1, ., 3)	INT32

The following code saves the YOLO version 3 model to an analytic store. Notice that a LIBNAME statement creates a mysvlib libref that uses the SASVIYA engine, because the LOADIMAGES and PROCESSIMAGES procedures use new data types that require this advanced engine.

```
libname imglib "&PATH_TO_CIFAR10_IMAGES";
libname mysvlib sasviya;

proc loadimages libref=imglib;
   output out=mysvlib.yolo_raw addvars=(HEIGHT WIDTH);
run;

proc processimages data=mysvlib.yolo_raw;
   resize type=LETTERBOX width=416 height=416 paddingmethod=CONSTANT
          b=128 g=128 r=128;
   output out=mysvlib.yolo_input copyvars=(_height_ _width_);
run;

%let labels = %str("person" "bicycle" "car" "motorbike" "aeroplane" "bus"
"train" "truck" "boat" "traffic light" "fire hydrant" "stop sign"
"parking meter" "bench" "bird" "cat" "dog" "horse" "sheep" "cow" "elephant"
"bear" "zebra" "giraffe" "backpack" "umbrella" "handbag" "tie" "suitcase"
"frisbee" "skis" "snowboard" "sports ball" "kite" "baseball bat"
"baseball glove" "skateboard" "surfboard" "tennis racket" "bottle"
"wine glass" "cup" "fork" "knife" "spoon" "bowl" "banana" "apple"
"sandwich" "orange" "broccoli" "carrot" "hot dog" "pizza" "donut" "cake"
"chair" "sofa" "pottedplant" "bed" "diningtable" "toilet" "tvmonitor"
"laptop" "mouse" "remote" "keyboard" "cell phone" "microwave" "oven"
"toaster" "sink" "refrigerator" "book" "clock" "vase" "scissors" "teddy bear"
"hair drier" "toothbrush");

proc astore;
   saveas onnx="&PATH_TO_YOLO_ONNX"
      rstore=mylib.yolostore data=mysvlib.yolo_input
      inputs=(var=_image_ inputShapeOrder=NHWC inputColorOrder=BGR
         shape=(1 3 416 416) preprocess=NORMALIZE normFactor=255
         modelShapeOrder=NCHW modelColorOrder=RGB)
      inputs=(vars=(_width_ _height_) shape=(1 2))
      modelType=YOLOV3
      maxObjects=6
      labels=(&labels)
      ;
quit;
```

After you save the model to an analytic store, you can describe the store as follows:

```
proc astore;
   describe rstore=mylib.yolostore;
quit;
```

Output 4.10.2 contains the following ODS tables:

- The “Store Key” table displays the string identifier of the store.
- The “Basic Information” table displays the analytic engine and the time when the store was created.
- The “Input Variables” table displays the input variables.
- The “Output Variables” table displays the output variables.

Output 4.10.2 Output Tables from the DESCRIBE Statement

The ASTORE Procedure

Store Key
B5A63D7D4D416F4C7169A669B20146DFF9C6B5E6

Basic Information	
Analytic Engine	onnx
Time Created	18Aug2021:10:33:49

Input Variables

Name	Length	Role	Type	RawType	FormatName
image	519168	Id		Character	
width	8	Id		Num	
height	8	Id		Num	

Output 4.10.2 *continued*

Output Variables			
Name	**Length**	**Type**	**Label**
_P_Object0_	8	Num	Predicted: Object=0_
_Object0_x	8	Num	Predicted: Object=0_x
_Object0_y	8	Num	Predicted: Object=0_y
_Object0_width	8	Num	Predicted: Object=0_width
_Object0_height	8	Num	Predicted: Object=0_height
_P_Object1_	8	Num	Predicted: Object=1_
_Object1_x	8	Num	Predicted: Object=1_x
_Object1_y	8	Num	Predicted: Object=1_y
_Object1_width	8	Num	Predicted: Object=1_width
_Object1_height	8	Num	Predicted: Object=1_height
_P_Object2_	8	Num	Predicted: Object=2_
_Object2_x	8	Num	Predicted: Object=2_x
_Object2_y	8	Num	Predicted: Object=2_y
_Object2_width	8	Num	Predicted: Object=2_width
_Object2_height	8	Num	Predicted: Object=2_height
_P_Object3_	8	Num	Predicted: Object=3_
_Object3_x	8	Num	Predicted: Object=3_x
_Object3_y	8	Num	Predicted: Object=3_y
_Object3_width	8	Num	Predicted: Object=3_width
_Object3_height	8	Num	Predicted: Object=3_height
_P_Object4_	8	Num	Predicted: Object=4_
_Object4_x	8	Num	Predicted: Object=4_x
_Object4_y	8	Num	Predicted: Object=4_y
_Object4_width	8	Num	Predicted: Object=4_width
_Object4_height	8	Num	Predicted: Object=4_height
_P_Object5_	8	Num	Predicted: Object=5_
_Object5_x	8	Num	Predicted: Object=5_x
_Object5_y	8	Num	Predicted: Object=5_y
_Object5_width	8	Num	Predicted: Object=5_width
_Object5_height	8	Num	Predicted: Object=5_height
Object0	14	Character	Into: Object=0_
Object1	14	Character	Into: Object=1_
Object2	14	Character	Into: Object=2_
Object3	14	Character	Into: Object=3_
Object4	14	Character	Into: Object=4_
Object5	14	Character	Into: Object=5_
nObjects	8	Num	

Example 4.11: Extracting the Saved Open Neural Network Exchange (ONNX) Model

This example uses the EXTRACT statement to extract the Open Neural Network Exchange (ONNX) model that is saved to an analytic store.

```
proc astore;
   extract rstore=mylib.mniststore onnx="mnist.onnx";
quit;
```

After you extract the ONNX model from the analytic store, you can describe the ONNX model as follows:

```
proc astore;
   describe onnx="mnist.onnx";
quit;
```

Output 4.11.1 shows the following ODS tables:

- The “Basic Information” table displays basic information about the ONNX model.
- The “Input Value Information” table displays the input information.
- The “Output Value Information” table displays the output information.

Output 4.11.1 Output Tables from the DESCRIBE Statement

The ASTORE Procedure

Basic Information	
IR Version	3
Model Version	1
Domain	ai.cntk
Doc String	
Producer Name	CNTK
Producer Version	2.5.1
Operator Set Domain	
Operator Set Version	8

Input Value Information

Name	Length	Value Type	Shape	Element Type
Input3	3136	Tensor	(1, 1, 28, 28)	FLOAT

Output Value Information

Name	Length	Value Type	Shape	Element Type
Plus214_Output_0	40	Tensor	(1, 10)	FLOAT

Chapter 5
BNET Procedure

Contents

Overview: BNET Procedure **90**
- PROC BNET Features 90
- Using SAS Viya Workbench 91

Getting Started: BNET Procedure **91**
- Structure Learning 92
- Probability Table 93
- Variable Selection 93
- Guiding Network 94

Syntax: BNET Procedure **97**
- PROC BNET Statement 97
- AUTOTUNE Statement 103
- CODE Statement 109
- CROSSVALIDATION Statement 110
- FREQ Statement 110
- ID Statement 110
- INPUT Statement 111
- OUTPUT Statement 111
- PARTITION Statement 112
- SAVESTATE Statement 113
- TARGET Statement 113

Details: BNET Procedure **113**
- Independence Tests 113
- Variable Selection 115
- Structure Learning 115
- Parameter Learning 117
- *k*-fold Cross Validation 117
- Displayed Output 118
- ODS Table Names 119

Examples: BNET Procedure **120**
- Example 5.1: Naive Bayesian Network 120
- Example 5.2: Tree-Augmented Naive Bayesian Network 121
- Example 5.3: Parent-Child Bayesian Network 122
- Example 5.4: Markov Blanket 122
- Example 5.5: Bayesian Network—Augmented Naive Bayesian Network 123
- Example 5.6: Bayesian Network—General Network 124
- Example 5.7: Model Selection 125

Example 5.8: Tuning a Bayesian Network Model 129
References **132**

Overview: BNET Procedure

The BNET procedure learns a Bayesian network from an input data table in SAS Viya. A Bayesian network is a directed acyclic graphical model in which nodes represent random variables and the links between nodes represent direct dependency among random variables. Variables that are connected through intermediary variables in the network are conditionally independent given the intermediary variables. Because the Bayesian network provides conditional independence structure and a conditional probability table at each node, the model has been used successfully as a predictive model in supervised data mining. For more information about Bayesian networks, see Pearl (1988).

The BNET procedure can learn different types of Bayesian network structures, including naive, tree-augmented naive (TAN), Bayesian network-augmented naive (BAN), parent-child Bayesian network, general Bayesian network, and Markov blanket. PROC BNET performs efficient variable selection through independence tests, and it selects the best model automatically from the specified parameters. It also generates SAS DATA step code or an analytic store to score data.

PROC BNET Features

The BNET procedure has the following features:

- structure learning through efficient local learning algorithms
- efficient variable selection through independence tests
- automatic selection of the best parameters by using a validation data subset
- learning of different types of Bayesian network structures
- capability to specify links to include in or exclude from a network
- handling of both nominal and interval input variables
- binning of the interval input variables
- handling of missing values
- multithreading during the training and scoring phases

Using SAS Viya Workbench

SAS Viya Workbench has greatly simplified the SAS programming experience by offering a syntax style nearly identical to that of SAS 9 and earlier releases. However, there are minor exceptions. In the past, SAS Viya procedures required the use of a CAS engine libref. This engine served multiple purposes, but it was primarily used to upload data to the CAS server and manage the data. Its additional purposes included support of new data types and improved performance for large data sets. No CAS server is needed for SAS Viya Workbench, because all computations are done in place for you in a single location.

In the following program, the LIBNAME statement creates a mylib libref that the examples in this chapter use to refer to your current or desired work directory:

```
libname mylib ".";
```

For more information, see *SAS Viya Workbench: Getting Started*.

Getting Started: BNET Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

Consider a study of the analgesic effects of treatments on elderly patients who have neuralgia. Two test treatments and a placebo are compared. The response variable is whether the patient reported pain or not. Researchers recorded the age and gender of 60 patients and the duration of complaint before the treatment began. The following DATA step creates the data set Neuralgia:

```
Data Neuralgia;
   input Treatment $ Sex $ Age Duration Pain $ @@;
   datalines;
P  F  68   1  No   B  M  74  16  No  P  F  67  30  No
P  M  66  26  Yes  B  F  67  28  No  B  F  77  16  No
A  F  71  12  No   B  F  72  50  No  B  F  76   9  Yes
A  M  71  17  Yes  A  F  63  27  No  A  F  69  18  Yes
B  F  66  12  No   A  M  62  42  No  P  F  64   1  Yes
A  F  64  17  No   P  M  74   4  No  A  F  72  25  No
P  M  70   1  Yes  B  M  66  19  No  B  M  59  29  No
A  F  64  30  No   A  M  70  28  No  A  M  69   1  No
B  F  78   1  No   P  M  83   1  Yes B  F  69  42  No
B  M  75  30  Yes  P  M  77  29  Yes P  F  79  20  Yes
A  M  70  12  No   A  F  69  12  No  B  F  65  14  No
B  M  70   1  No   B  M  67  23  No  A  M  76  25  Yes
P  M  78  12  Yes  B  M  77   1  Yes B  F  69  24  No
P  M  66   4  Yes  P  F  65  29  No  P  M  60  26  Yes
A  M  78  15  Yes  B  M  75  21  Yes A  F  67  11  No
P  F  72  27  No   P  F  70  13  Yes A  M  75   6  Yes
B  F  65   7  No   P  F  68  27  Yes P  M  68  11  Yes
P  M  67  17  Yes  B  M  70  22  No  A  M  65  15  No
P  F  67   1  Yes  A  M  67  10  No  P  F  72  11  Yes
```

```
A  F  74   1  No   B  M  80  21  Yes A  F  69   3  No
;
```

The Neuralgia data set contains five variables: Treatment, Sex, Age, Duration, and Pain. The last variable, Pain, is the target variable. Pain=Yes indicates that the patient felt pain, and Pain=No indicates no pain. The variable Treatment is a nominal variable that has three levels: A and B represent the two test treatments, and P represents the placebo treatment. The gender of the patients is indicated by the nominal variable Sex. The variable Age is the age of the patients, in years, when treatment began. The duration of complaint, in months, before the treatment began is indicated by the variable Duration.

You can load the Neuralgia data set into your SAS library and by using your libref in the first statement of the following DATA step:

```
data mylib.neuralgia;
   set neuralgia;
run;
```

These statements assume that your SAS libref is named mylib, but you can substitute any appropriately defined SAS libref.

Structure Learning

The following statements use the BNET procedure to learn a Bayesian network with Treatment and Sex as nominal variables and Age and Duration as interval variables. The two interval variables are binned into three equal-width levels.

```
proc bnet data=mylib.neuralgia numbin=3 outnetwork=mylib.network;
    target Pain;
    input Treatment Sex/level=nominal;
    input Age Duration/level=interval;
    ods output varselect=varselect;
run;
```

The following statements produce Figure 5.1, which shows the network structure that PROC BNET has learned. There are three variables in the network: Treatment is the parent of Pain, and Pain is the parent of Age. From the structure, you can infer that Pain is dependent on Treatment and is also dependent on Age, but is (conditionally) independent of Sex or Duration.

```
proc print data=mylib.network noobs label;
    var _parentnode_ _childnode_;
    where _type_="STRUCTURE";
run;
```

Figure 5.1 Network Structure

Parent Node	Child Node
Treatment	Pain
Pain	Age

Probability Table

The following statements produce Figure 5.2, which shows the conditional probability table for each node in the network. You can use these probability tables for scoring or inferences or both. The conditional probability tables together with the network structure determine the Bayesian network.

```
proc print data=mylib.network noobs label;
    var _parentnode_ _parentcond_ _childnode_ _childcond_ _value_;
    where _type_="PROBABILITY";
run;
```

Figure 5.2 Probability Table

Parent Node	Parent Condition	Child Node	Child Condition	Value
		Treatment	A	0.33333
		Treatment	B	0.33333
		Treatment	P	0.33333
Treatment	A	Pain	Yes	0.27273
Treatment	A	Pain	No	0.72727
Treatment	B	Pain	Yes	0.27273
Treatment	B	Pain	No	0.72727
Treatment	P	Pain	Yes	0.72727
Treatment	P	Pain	No	0.27273
Pain	Yes	Age	<67	0.17857
Pain	Yes	Age	<75	0.35714
Pain	Yes	Age	>=75	0.46429
Pain	No	Age	<67	0.31579
Pain	No	Age	<75	0.60526
Pain	No	Age	>=75	0.07895

Variable Selection

The network shown in Figure 5.1 does not include the variable Sex or Duration, because PROC BNET automatically removes these variables from the network by using independence tests. PROC BNET produces the table in Figure 5.3, which shows the results of independence tests that are conducted to select variables. Variables that are found to be marginally independent or conditionally independent of the target variable are removed from the network. PROC BNET removes Duration from the network because the p-value of the chi-square and G-square statistics of Duration are greater than 0.05 (the default value of the ALPHA= option). Sex is conditionally independent of Pain given Treatment; therefore, PROC BNET removes Sex from the network.

Figure 5.3 Variable Selection

The BNET Procedure

Variable Selection								
Variable	**Selected**	**Chi-Square**	**Pr > ChiSq**	**G-Square**	**Pr > GSq**	**Mutual Information**	**DF**	**Conditional Variables**
Sex	No	7.20000	0.0658	7.59454	0.0552	0.34481	3	Treatment
Treatment	Yes	13.71429	0.0011	14.02297	0.0009	0.45652	2	
Age	Yes	14.60003	0.0007	15.27118	0.0005	0.47404	2	
Duration	No	2.25795	0.3234	3.34851	0.1874	0.23298	2	

Guiding Network

As you review the network, you notice that Pain is the parent of Age. Given the data, there is a strong relationship between Age and Pain that explains this connection, but the direction is not as desired. You can use the INNETWORK= option as in the following statements to guide the network so that both Treatment and Age become parents of Pain:

```
Data Innetwork;
    length parent $ 10;
    input Parent $ Child $ Flag;
datalines;
Treatment  Pain   1
Age        Pain   1

data mylib.innetwork;
   set innetwork;
run;

 proc bnet data=mylib.neuralgia numbin=3 outnetwork=mylib.network
     innetwork=mylib.innetwork;
     target Pain;
     input Treatment Sex/level=nominal;
     input Age Duration/level=interval;
     ods output varselect=varselect;
 run;


 proc print data=mylib.network noobs label;
     var _parentnode_ _childnode_;
     where _type_="STRUCTURE";
 run;
```

The revised network is as desired, with Treatment and Age as parents of Pain, as shown in Figure 5.4.

Figure 5.4 Structure with INNETWORK= Option

Parent Node	Child Node
Treatment	Pain
Age	Pain

You can get the revised conditional probabilities associated with this network by running the following code:

```
proc print data=mylib.network noobs label;
    var _parentnode_ _parentcond_ _childnode_ _childcond_ _value_;
    where _type_="PROBABILITY";
run;
```

Figure 5.5 shows the revised conditional probabilities, given this network.

Figure 5.5 Probability Table with INNETWORK= Option

Parent Node	Parent Condition	Child Node	Child Condition	Value
		Treatment	A	0.33333
		Treatment	B	0.33333
		Treatment	P	0.33333
		Age	<67	0.25397
		Age	<75	0.50794
		Age	>=75	0.23810
Treatment	A	Pain	Yes	0.14286
Age	<67	Pain	Yes	0.14286
Treatment	A	Pain	No	0.85714
Age	<67	Pain	No	0.85714
Treatment	A	Pain	Yes	0.21429
Age	<75	Pain	Yes	0.21429
Treatment	A	Pain	No	0.78571
Age	<75	Pain	No	0.78571
Treatment	A	Pain	Yes	0.80000
Age	>=75	Pain	Yes	0.80000
Treatment	A	Pain	No	0.20000
Age	>=75	Pain	No	0.20000
Treatment	B	Pain	Yes	0.14286
Age	<67	Pain	Yes	0.14286
Treatment	B	Pain	No	0.85714
Age	<67	Pain	No	0.85714
Treatment	B	Pain	Yes	0.10000
Age	<75	Pain	Yes	0.10000
Treatment	B	Pain	No	0.90000
Age	<75	Pain	No	0.90000
Treatment	B	Pain	Yes	0.66667
Age	>=75	Pain	Yes	0.66667
Treatment	B	Pain	No	0.33333
Age	>=75	Pain	No	0.33333
Treatment	P	Pain	Yes	0.71429
Age	<67	Pain	Yes	0.71429
Treatment	P	Pain	No	0.28571
Age	<67	Pain	No	0.28571
Treatment	P	Pain	Yes	0.61538
Age	<75	Pain	Yes	0.61538
Treatment	P	Pain	No	0.38462
Age	<75	Pain	No	0.38462
Treatment	P	Pain	Yes	0.83333
Age	>=75	Pain	Yes	0.83333
Treatment	P	Pain	No	0.16667
Age	>=75	Pain	No	0.16667

Syntax: BNET Procedure

The following statements are available in the BNET procedure:

PROC BNET < *options* > **;**
  AUTOTUNE < *options* > **;**
  CODE FILE=*filename* **;**
  CROSSVALIDATION < *options* > **;**
  FREQ *variable* **;**
  ID *variables* **;**
  INPUT *variables* / < **LEVEL=INTERVAL** | **NOMINAL** > **;**
  OUTPUT OUT=*libref.data-table* < *option* > **;**
  PARTITION *partition-option* **;**
  SAVESTATE RSTORE=*libref.data-table* **;**
  TARGET *variable* **;**

The PROC BNET statement, the TARGET statement, and the INPUT statement are required. You can specify only one TARGET statement, but you can specify multiple INPUT statements. The following sections describe the PROC BNET statement and then describe the other statements in alphabetical order.

PROC BNET Statement

PROC BNET < *options* > **;**

The PROC BNET statement invokes the procedure. Table 5.1 summarizes important *options* in the PROC BNET statement by function.

Table 5.1 PROC BNET Statement Options

Option	Description
Data Options	
DATA=	Specifies the input data set
MISSINGINT=	Specifies how to handle missing values for interval variables
MISSINGNOM=	Specifies how to handle missing values for nominal variables
NUMBIN=	Specifies the number of binning levels for interval variables
PRESCREENING=	Specifies the initial screening for the input variables
VARSELECT=	Specifies the selection for the input variables
Independence Test Options	
ALPHA=	Specifies the significance level for independence tests by using chi-square or G-square statistics
INDEPTEST=	Specifies the method to use for independence tests
MIALPHA=	Specifies the significance level for independence tests by using mutual information
Structure Learning Options	
INNETWORK=	Specifies the input table that contains the included and excluded arcs that are defined using pairs of parent and child variables

Table 5.1 *continued*

Option	Description
MAXPARENTS=	Specifies the maximum number of parents for each node in the network
PARENTING=	Specifies the structure learning method
STRUCTURE=	Specifies the network structure type
Model Selection Options	
BESTMODEL	Requests that the best model be selected
Network Output Options	
OUTNETWORK=	Specifies the output table that contains the final network and the associated probabilities

You can specify the following *options*:

ALPHA=*number*

specifies the significance level for independence tests by using chi-square or G-square statistics. The valid range is 0 to 1, inclusive. If you want to choose the best model among several, you can specify up to five *numbers*, separated by spaces. If you specify multiple *numbers* but you do not specify the BESTMODEL option, PROC BNET uses the first *number* and ignores the remaining *numbers*.

By default, ALPHA=0.05.

BESTMODEL

selects the best model by using a validation data subset. You can specify the validation data subset by using the PARTITION statement. If you specify this option, you can specify multiple values for the ALPHA=, PRESCREENING=, VARSELECT=, STRUCTURE=, and PARENTING= options. PROC BNET uses the misclassification errors that arise from the validation data to automatically decide the best set of parameter values among these options. If you supply multiple values of these options in the absence of the BESTMODEL option, then only the first value among the multiple values is used.

By default, the final network is produced for only one set of parameters. You must specify the BESTMODEL option to invoke a search for the best network by using all possible combinations of the specified parameters.

NOTE: If you specify the BESTMODEL option in the PROC BNET statement, then the AUTOTUNE statement is ignored.

DATA=*libref.data-table*

names the input data table for PROC BNET to use. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to the data. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 91.

data-table specifies the name of the input data table.

INDEPTEST=ALL | CHIGSQUARE | CHISQUARE | GSQUARE | MI

specifies the method to use for independence tests. You can specify the following values:

ALL uses the chi-square, the G-square statistics, and the normalized mutual information for independence tests. A variable is independent of the target if both the *p*-values of the chi-square and the G-square statistics are greater than the specified ALPHA= value and the normalized mutual information is less than the value that is specified in MIALPHA= option.

CHIGSQUARE uses both the chi-square and the G-square statistics for independence tests. A variable is independent of the target if both the *p*-values of the chi-square and the G-square statistics are greater than the specified ALPHA= value.

CHISQUARE uses the chi-square statistics for independence tests. A variable is independent of the target if the *p*-value of the statistics is greater than the specified ALPHA= value.

GSQUARE uses the G-square statistics for independence tests. A variable is independent of the target if the *p*-value of the statistics is greater than the specified ALPHA= value.

MI uses the normalized mutual information for independence tests. A variable is independent of the target if the normalized mutual information is less than the value that is specified in the MIALPHA= option.

By default, INDEPTEST=CHIGSQUARE.

INNETWORK=*libref.data-table*

INNET=*libref.data-table*

names the input data table that contains the arcs defined as parent-child variable pairs to be included in or excluded from the final network. The *flag* column in this table indicates that the arc is an included arc if the value of *flag* is 1, and the arc is an excluded arc if the value of *flag* is 0. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 91.

MAXPARENTS=*integer*

specifies the maximum number of parents that are allowed for each node in the network structure. The valid range is 1 to 16, inclusive. If you specify the BESTMODEL option, PROC BNET calculates from 1 to *integer* and decides the best number of parents.

By default, MAXPARENTS=5.

MIALPHA=*number*

specifies the threshold for independence tests by using mutual information. The valid range is 0 to 1, inclusive.

By default, MIALPHA=0.05.

MISSINGINT=IGNORE | IMPUTE

specifies how to handle missing values for all interval input variables. This option applies to training data, validation data, testing data, and any data that are used for scoring. You can specify the following values:

IGNORE ignores the observations that have missing values in any of the interval variables.

IMPUTE replaces the missing values in any interval variable by the mean of the variable.

By default, MISSINGINT=IGNORE.

MISSINGNOM=IGNORE | IMPUTE | LEVEL
specifies how to handle the missing values for all nominal input variables. You can specify the following values:

IGNORE ignores the observations that have missing values in any of the nominal variables.

IMPUTE replaces the missing values in any nominal variable by the mode of the variable.

LEVEL treats the missing values in any nominal variable as a separate level of the variable.

By default, MISSINGNOM=IGNORE.

NTHREADS=*number-of-threads*
specifies the number of threads to use. The default is the minimum CPU count of all the nodes.

NUMBIN=*integer*

NBIN=*integer*
specifies the number of binning levels for all interval variables. PROC BNET bins each interval variable into *integer* equal-width levels. The valid range of *integer* is 2 to 1,024, inclusive.

By default, NUMBIN=5.

OUTNETWORK=*libref.data-table*

OUTNET=*libref.data-table*
names the output data table to contain the network structure and the probability distributions. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 91.

PARENTING=BESTONE | BESTSET
specifies the algorithm for orienting the network structure. You can specify the following values:

BESTONE uses a greedy approach to determine the parents of each node; that is, for each node, the best candidate is added as a parent of the node in each iteration.

BESTSET determines the best set of variables among possible candidate sets as the parents of each node; that is, instead of adding one variable in an iteration, PROC BNET tests multiple sets of variables together and chooses the best set as the parents of the node.

If you want to choose between the two methods, you can specify both of them and also specify the BESTMODEL option. If you specify both methods but you do not specify the BESTMODEL option, PROC BNET uses the first specified method, and ignores the other.

By default, PARENTING=BESTSET.

PRESCREENING=0 | 1

specifies the initial screening for the input variables. You can specify the following values:

0 uses all the input variables.

1 uses only the input variables that are dependent on the target.

If you want to choose the best model with or without prescreening, you can specify PRESCREENING=0 1 or PRESCREENING=1 0 and also specify the BESTMODEL option. There should be at least one blank space between the two values. If you specify both values but you do not specify the BESTMODEL option, PROC BNET uses the first specified value and ignores the other.

If you specify STRUCTURE=GENERAL, then any setting of the PRESCREENING= option other than 1 is ignored and PRESCREENING=1 is used.

By default, PRESCREENING=1.

PRINTTARGET

generates the "Predicted Probability Variables" table, which displays the target variable and the predicted probability variables, and the "Predicted Target Variable" table, which displays the predicted target variable.

By default, these two tables are not generated.

STRUCTURE=*values*

specifies the network structure. Together with the MAXPARENTS= option, this option determines which network structure the procedure learns from the training data. You can specify one or more of the following *values*, separated by spaces:

GENERAL | GN learns a general Bayesian network over the target and input variables. PROC BNET learns a general Bayesian network by using a search algorithm that grows partial networks to completion by exploring among possible completions. For the GENERAL network structure, any specification of the PRESCREENING= option is ignored and PRESCREENING=1 is used. In addition, any specification of the VARSELECT= option is ignored and VARSELECT=0 is used. See the VARSELECT= option for more information.

MB learns the Markov blanket of the target variable. The Markov blanket includes the parents, the children, and the other parents of the children. After learning the Markov blanket, PROC BNET further determines the parents of the target, the links from the parents to the children, and the links among the children. When you specify STRUCTURE=MB, the procedure learns the Markov blanket regardless of the values of the PRESCREENING= and VARSELECT= options.

NAIVE assumes a naive Bayesian network structure (that is, the target has a direct link to each input variable). When MAXPARENTS=1, the structure is a naive Bayesian network (NB). When the value of the MAXPARENTS= option is greater than 1, the structure is a Bayesian network-augmented naive Bayesian network (BAN).

PC learns the parent-child Bayesian network structure (PC). STRUCTURE=PC differs from STRUCTURE=NAIVE in that some input variables could be learned as the parents of the target variable. In addition, links from the parents to the children and among the children are also possible in the PC structure.

TAN learns the tree-augmented naive Bayesian network structure. The TAN structure includes a direct link from the target to each input variable plus a tree structure among the input variables.

If you want to choose the best structure among several structures, you can specify multiple values in any combination, separated by spaces, and also specify the BESTMODEL option. If you specify multiple structures but you do not specify the BESTMODEL option, PROC BNET uses the first value that you specify and ignores the rest.

By default, STRUCTURE=PC.

VARSELECT=0 | 1 | 2 | 3

specifies how input variables are selected beyond the prescreening. You can specify the following values:

0 uses all input variables that remain after the initial screening is performed as specified in the PRESCREENING= option.

1 tests each input variable for conditional independence of the target variable, given any other input variable. This type of selection uses only the variables that are not rendered conditionally independent of the target given any other input variable.

2 tests each input variable further for conditional independence of the target variable, given any subset of other input variables. This type of selection uses only the variables that are not rendered conditionally independent of the target, given any subset of other input variables.

3 determines the Markov blanket of the target variable and uses only the variables in the Markov blanket.

By default, VARSELECT=1.

If you specify VARSELECT=1, 2, or 3, PROC BNET automatically tests each input variable for unconditional independence of the target regardless of the value of the PRESCREENING= option. If no variables are left at a particular variable selection level, PROC BNET reverts to the previous level. For example, if you specify VARSELECT=3 and there are no variables in the Markov blanket of the target, PROC BNET uses the variables from the previous level, VARSELECT=2.

If you want to choose the best model among different levels of variable selections, you can specify any combination of values for the VARSELECT= option and also specify the BESTMODEL option. There should be at least one blank space between the specified values. If you specify multiple values for the VARSELECT= option but you do not specify the BESTMODEL option, PROC BNET uses the first specified value and ignores the remaining values.

If you specify STRUCTURE=GENERAL, then any setting of the VARSELECT= option other than 0 is ignored and VARSELECT=0 is used.

The INNETWORK= option enables you to override the selections that you make in the PRESCREENING= and VARSELECT= options. Thus a variable that is selected by independence tests might not be in the network if you exclude it by using the INNETWORK= option. Similarly, a variable that is deemed independent of the target might get into the network if you include it by using the INNETWORK= option. Hence the selections that you make by using independence tests are not final, and specifications of the INNETWORK= option can override these selections.

AUTOTUNE Statement

AUTOTUNE < *options* > **;**

The AUTOTUNE statement searches for the best combination of values for the ALPHA=, INDEPTEST=, MAXPARENTS=, MIALPHA=, MISSINGINT=, MISSINGNOM=, NUMBIN=, PARENTING=, PRESCREENING=, STRUCTURE=, and VARSELECT= options in the PROC BNET statement. You cannot specify both the AUTOTUNE statement and the CROSSVALIDATION statement in the same procedure run.

Table 5.2 summarizes the *options* that you can specify in the AUTOTUNE statement. For more information about all options except the TUNINGPARAMETERS= and SEED= options, see the option's description in the section "AUTOTUNE Statement" on page 6 in Chapter 2, "Shared Concepts." The TUNINGPARAMETERS= and SEED= options are described following Table 5.2.

NOTE: Processing the AUTOTUNE statement is computationally expensive and requires a significant amount of time.

NOTE: If you specify both the AUTOTUNE statement and the BESTMODEL option in the PROC BNET statement, the AUTOTUNE statement is ignored.

Table 5.2 AUTOTUNE Statement Options

Option	Description
APPENDLOOKUP	Specifies that the table specified in the HISTORYTABLE= option contain the rows from the table specified in the LOOKUPTABLE= option
CONSTRAINTS=	Specifies the constraints on the output metrics
EVALHISTORY=	Specifies how to report the evaluation history of the tuner
FOLDCOLUMN=	Specifies the column in the data table in which the cross validation fold value is indicated
FRACTION=	Specifies the fraction of observations to use for validation
HISTORYTABLE=	Specifies the table that contains the evaluation history
KFOLD=	Specifies the number of folds for *k*-fold cross validation
LIVEUPDATE	Specifies that the table specified in the HISTORYTABLE= option be updated at every evaluation
LOOKUPTABLE=	Specifies the table to use for evaluation lookup
MAXBAYES=	Specifies the maximum number of points in the kriging model
MAXEVALS=	Specifies the maximum number of evaluations
MAXITER=	Specifies the maximum number of iterations when SEARCHMETHOD=GA or SEARCHMETHOD=BAYESIAN
MAXTIME=	Specifies the maximum time for all iterations
MAXTRAINTIME=	Specifies the maximum time for a model training
NCONVITER=	Specifies the number of convergence iterations
NOGRIDSHUFFLE	Requests that the grid points not be shuffled
NOLOCALSEARCH	Disables local search optimization
NPARALLEL=	Specifies the number of parallel sessions
NSUBSESSIONWORKERS=	Specifies the number of workers in parallel sessions
OBJECTIVE=	Specifies the objective function

Table 5.2 *continued*

Option	Description
POPSIZE=	Specifies the population size when SEARCHMETHOD=GA or SEARCHMETHOD=BAYESIAN
SAMPLESIZE=	Specifies the sample size when SEARCHMETHOD=LHS or SEARCHMETHOD=RANDOM
SEARCHMETHOD=	Specifies the search method that the optimizer uses
SECONDOBJECTIVE=	Specifies the second objective to use for tuning
SEED=	Specifies the random seed to use for tuning
SELECTINITPOINT	Specifies that the tuner select the best evaluation from the lookup table
TARGETEVENT=	Specifies the target event for ROC-based calculations
TRAINFRACTION=	Specifies the fraction of observations to use for training
TUNINGPARAMETERS=	Specifies the custom tuning parameters
USEPARAMETERS=	Specifies how to handle the TUNINGPARAMETERS= option

SEED=*number*

specifies the random seed value to use for the tuning process, where *number* must be a positive integer. If you do not specify a seed, or if you specify a value equal to zero, the seed is generated by reading the time of day from the computer's clock.

TUNINGPARAMETERS=(*suboption* **| . . . | <** *suboption* **>)**

TUNEPARMS=(*suboption* **| . . . | <** *suboption* **>)**

specifies which parameters to tune and which ranges to tune over. If you specify USEPARAMETERS=STANDARD, this option is ignored.

You can specify one or more of the following *suboptions*:

ALPHA (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**

specifies information about the significance level for independence tests by using chi-square or G-square statistics, where *number* or any value in the *value-list* must be a real number in the range 0 to 1, inclusive. For more information, see the ALPHA= option in the PROC BNET statement.

You can specify the following additional suboptions:

LB=*number*

specifies the minimum significance level to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=0.01.

UB=*number*

specifies the maximum significance level to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=0.99.

VALUES=*value-list*
specifies a list of significance levels to consider during tuning, where *value-list* is a space-separated list of numbers in the range 0 to 1. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial significance level for the tuner to use.

By default, INIT=0.05.

EXCLUDE
excludes the significance level from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

INDEPTEST (VALUES=*value-list* **INIT=***value* **EXCLUDE)**
specifies information about the method to use for independence tests. For more information, see the INDEPTEST= option in the PROC BNET statement.

You can specify the following additional suboptions:

VALUES=*value-list*
specifies a list of methods to use for independence tests, where *value-list* is a space-separated list of CHIGSQUARE, CHISQUARE, MI, and GSQUARE.

INIT=CHIGSQUARE | CHISQUARE | MI | GSQUARE
specifies the initial method to use for independence tests.

By default, INIT=CHIGSQUARE.

EXCLUDE
excludes the INDEPTEST suboption from the tuning process. If you specify EXCLUDE, any specified VALUES= and INIT= suboptions are ignored.

MAXPARENTS (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**
specifies information about the maximum number of parents that are allowed for each node in the network structure. The valid range is 1 to 16, inclusive. For more information, see the MAXPARENTS= option in the PROC BNET statement.

You can specify the following additional suboptions:

LB=*number*
specifies a lower bound on the maximum number of parents to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=1.

UB=*number*
specifies an upper bound on the maximum number of parents to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=16.

VALUES=*value-list*
specifies a list of values to consider for the maximum number of parents in the network structure, where *value-list* is a space-separated list of numbers. If you specify this suboption, you cannot specify the LB= and UB= suboptions.

INIT=*number*
specifies the initial maximum number of parents in the network structure.

By default, INIT=5.

EXCLUDE
excludes the maximum number of parents from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

MIALPHA (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**
specifies information about the threshold for independence tests by using mutual information. For more information, see the MIALPHA= option in the PROC BNET statement.

You can specify the following additional suboptions:

LB=*number*
specifies the minimum threshold value to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=0.

UB=*number*
specifies the maximum threshold value to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=1.

VALUES=*value-list*
specifies a list of threshold values to consider during tuning, where *value-list* is a space-separated list of numbers in the range 0 to 1. If you specify this suboption, you cannot specify the LB= and UB= suboptions.

INIT=*number*
specifies the initial threshold value for the tuner to use.

By default, INIT=0.05.

EXCLUDE
excludes the threshold value from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

MISSINGINT (VALUES=*value-list* **INIT=***value* **EXCLUDE)**
specifies information about how to handle missing values for interval variables during the tuning process. For more information, see the MISSINGINT= option in the PROC BNET statement.

You can specify the following additional suboptions:

VALUES=*value-list*
specifies a list of values for the tuner to try for the MISSINGINT suboption, where you can specify IMPUTE or IGNORE (or both) in a space-separated *value-list*.

INIT=IMPUTE | IGNORE
specifies whether to start tuning by imputing or ignoring missing values of interval variables.

By default, INIT=IGNORE.

EXCLUDE
excludes the MISSINGINT suboption from the tuning process. If you specify EXCLUDE, any specified VALUES= and INIT= suboptions are ignored.

MISSINGNOM (VALUES=*value-list* **INIT=***value* **EXCLUDE)**
specifies information about how to handle missing values for nominal variables during the tuning process. For more information, see the MISSINGNOM= option in the PROC BNET statement.

You can specify the following additional suboptions:

VALUES=*value-list*
specifies a list of values for the tuner to handle missing values for nominal variables, where *value-list* is a combination of one or more of the following values in a space-separated list: IMPUTE, IGNORE, and LEVEL.

INIT=IMPUTE | IGNORE | LEVEL
specifies the initial value to use in tuning the MISSINGNOM suboption.

By default, INIT=IGNORE.

EXCLUDE
excludes the MISSINGNOM suboption from the tuning process. If you specify EXCLUDE, any specified VALUES= and INIT= suboptions are ignored.

NUMBIN (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**
specifies information about the number of binning levels for all interval variables. The valid range is 2 to 20. For more information, see the NUMBIN= option in the PROC BNET statement.

You can specify the following additional suboptions:

LB=*number*
specifies a lower bound of binning levels to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=2.

UB=*number*
specifies an upper bound of binning levels to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=20.

VALUES=*value-list*
specifies a list of values of binning levels to consider, where *value-list* is a space-separated list of numbers. If you specify this suboption, you cannot specify the LB= and UB= suboptions.

INIT=*number*
specifies the initial number of binning levels.

By default, INIT=5.

EXCLUDE
excludes the number of binning levels from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

PARENTING (VALUES=*value-list* **INIT=***value* **EXCLUDE)**
specifies information about the algorithm for orienting the network structure. For more information, see the PARENTING= option in the PROC BNET statement.

You can specify the following additional suboptions:

VALUES=*value-list*
specifies a list of algorithms for orienting the network structure during tuning, where *value-list* is a space-separated list of BESTONE and BESTSET.

INIT=BESTONE | BESTSET
specifies the initial algorithm for orienting the network structure for the tuner to use.

By default, INIT=BESTSET.

EXCLUDE
excludes the algorithm selection for orienting the network structure from the tuning process. If you specify this suboption, any specified VALUES= and INIT= suboptions are ignored.

PRESCREENING (VALUES=*value-list* **INIT=***value* **EXCLUDE)**
specifies information about the initial screening for the input variables. For more information, see the PRESCREENING= option in the PROC BNET statement.

You can specify the following additional suboptions:

VALUES=*value-list*
specifies a list of initial screening values for the input variables, where *value-list* is a space-separated list of 0 and 1.

INIT=*number*
specifies the initial screening for the input variables.

By default, INIT=1.

EXCLUDE
excludes the initial screening for the input variables from the tuning process. If you specify this suboption, any specified VALUES= and INIT= suboptions are ignored.

STRUCTURE (VALUES=*value-list* **INIT=***value* **EXCLUDE)**
specifies information about the network structure. For more information, see the STRUCTURE= option in the PROC BNET statement.

You can specify the following additional suboptions:

VALUES=*value-list*
specifies a list of structures, where *value-list* is a space-separated list of MB, NAIVE, PC, and TAN. You cannot specify the value GENERAL (or GN) for the VALUES= or INIT= suboption in the AUTOTUNE statement.

INIT=MB | NAIVE | PC | TAN
specifies the initial network structure.

By default, INIT=PC.

EXCLUDE
excludes the network structure from the tuning process. If you specify this suboption, any specified VALUES= and INIT= suboptions are ignored.

VARSELECT (VALUES=*value-list* **INIT=***value* **EXCLUDE)**
specifies information about how to select input variables during the tuning process after the prescreening. For more information, see the VARSELECT= option in the PROC BNET statement.

You can specify the following additional suboptions:

VALUES=*value-list*
specifies a list of input variables to consider for the tuning process after the prescreening, where *value-list* is a space-separated list of 0, 1, 2, and 3.

INIT=*number*
specifies the initial value to consider for the tuning process after the prescreening.

By default, INIT=1.

EXCLUDE
excludes the VARSELECT suboption from the tuning process. If you specify EXCLUDE, any specified VALUES= and INIT= suboptions are ignored.

CODE Statement

CODE FILE=*filename* **;**

The CODE statement is optional in PROC BNET. If you use a CODE statement, SAS DATA step code is generated and stored in a file that can be used for scoring purposes.

CROSSVALIDATION Statement

CROSSVALIDATION < *options* > **;**

The CROSSVALIDATION statement performs *k*-fold cross validation to find the average estimated validation error. You cannot specify this statement if you specify the PARTITION statement.

You can specify the following *options*:

KFOLD=*number*
specifies the number of partition folds in the cross validation process, where *number* is between 2 and 20, inclusive. By default, KFOLD=5.

NOPARALLEL
specifies that *k*-fold cross validation not be run in parallel. By default, the process runs in parallel.

SEED=*number*
specifies an integer that seeds the pseudorandom number generator for partitioning the data into folds (subsets). If you omit this option or if the *number* is 0, the seed is generated using the time of day from the computer's clock.

FREQ Statement

FREQ *variable* **;**

The *variable* in the FREQ statement identifies a numeric variable in the data set that contains the frequency of occurrence for each observation. The BNET procedure treats each observation as if it appeared *n* times, where *n* is the value of the *variable* for the observation. If *n* is not an integer, it is truncated to an integer. If *n* is less than 1 or is missing, the observation is ignored. When the FREQ statement is not specified, each observation is assigned a frequency of 1.

ID Statement

ID *variables* **;**

The optional ID statement lists one or more variables from the input data set to be copied to the prediction output data set. The ID statement accepts both numeric and character variables. The variables in an ID statement can also appear in any other statements.

INPUT Statement

INPUT *variables* </**LEVEL=INTERVAL | NOMINAL** > **;**

The INPUT statement specifies one or more *variables* as input variables. You can specify multiple INPUT statements. If the INPUT statement contains a duplicate variable, PROC BNET considers the variable as input variable and ignores the repetitions. You can specify the following option in each INPUT statement:

LEVEL=INTERVAL | NOMINAL
specifies the type of all the variables in the current INPUT statement. You can specify the following values:

NOMINAL treats all the variables in the current INPUT statement as nominal variables.

INTERVAL treats all the variables in the current INPUT statement as interval variables.

By default, LEVEL=INTERVAL for numeric variables and LEVEL=NOMINAL for categorical variables.

OUTPUT Statement

OUTPUT OUT=*libref.data-table* < *options* > **;**

The OUTPUT statement creates a data table to contain the predicted target values of the input table.

You must specify the following option:

OUT=*libref.data-table*
names the output data table for PROC BNET to use. You must specify this option before any other options. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to where the data table is to be stored. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 91.

data-table specifies the name of the output data table.

This table includes variables that are specified either in the COPYVARS= option or in the ID statement. If you specify PARTITION statement, the output includes one more column, _ROLE_. _ROLE_ is a reserved name. If it exist in the input data table and you specify it in the COPYVARS= option or in the ID statement, you need to use the ROLE= option to change the generated column's name.

You can also specify the following *options*:

COPYVAR=*variable*

COPYVARS=(*variables***)**

lists one or more variables from the input data table to be copied to the output data table.

ROLE=*rolename*

renames the generated column _ROLE_ in the output data table to the specified *role*.

PARTITION Statement

PARTITION *partition-option* **;**

The PARTITION statement specifies how observations in the input data set are logically partitioned into disjoint subsets for model training, validation, and testing. For more information, see the section "Using Validation and Test Data" on page 23 in Chapter 2, "Shared Concepts." Either you can designate a variable in the input data table and a set of formatted values of that variable to determine the role of each observation, or you can specify proportions to use for randomly assigning observations to each role.

You must specify exactly one of the following *partition-options*:

FRACTION(< TEST=*fraction* **> < VALIDATE=***fraction* **> < SEED=***number* **>)**

randomly assigns specified proportions of the observations in the input data table to the roles. You specify the proportions for testing and validation by using the TEST= and VALIDATE= suboptions. If you specify both the TEST= and VALIDATE= suboptions, then the sum of the specified fractions must be less than 1 and the remaining fraction is the proportion of the observations assigned to the training role. The SEED= option specifies an integer that is used to start the pseudorandom number generator for random partitioning of data for training, testing, and validation. If you do not specify SEED=*number* or if *number* is less than or equal to 0, the seed is generated by reading the time of day from the computer's clock.

ROLE=*variable* **(< TEST='***value***' > < TRAIN='***value***' > < VALIDATE='***value***' >)**

ROLEVAR=*variable* **(< TEST='***value***' > < TRAIN='***value***' > < VALIDATE='***value***' >)**

names the *variable* in the input data table whose values are used to assign roles to each observation. This *variable* cannot also appear as an analysis variable in other statements or options. The TEST=, TRAIN=, and VALIDATE= suboptions specify the formatted values of this variable that are used to assign observation roles. If you do not specify the TRAIN= suboption, then all observations whose role is not determined by the TEST= or VALIDATE= suboption are assigned to the training role.

SAVESTATE Statement

SAVESTATE RSTORE=*libref.data-table* **;**

The SAVESTATE statement creates an analytic store for the model and saves it as a binary object in a data table. You can use the analytic store in the ASTORE procedure to score new data. For more information, see Chapter 4, "ASTORE Procedure."

You must specify the following option:

RSTORE=*libref.data-table*
specifies a data table in which to save the analytic store for the model. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 91.

TARGET Statement

TARGET *variable* **;**

The TARGET statement names the *variable* that PROC BNET predicts. PROC BNET treats the TARGET *variable* as nominal.

The target values are levelized in descending order. If multiple target statements are supplied then the first statement applies.

Details: BNET Procedure

Independence Tests

Both variable selection and structure learning require either independence tests between two variables or conditional independence tests given some other variables. PROC BNET supports independence tests by using the chi-square statistic, G-square statistic, normalized mutual information, or some combination of them.

Given two nominal variables X and Y (or interval variables after binning) that have levels r and c, respectively, the chi-square statistic is computed as

$$\chi^2 = \sum_{i=1}^{r} \sum_{j=1}^{c} \frac{(O_{ij} - E_{ij})^2}{E_{ij}}$$

where O_{ij} is an observed frequency in a cell of the contingency table of the two variables and E_{ij} is the expected frequency of the cell. The degrees of freedom for the test is $(r-1) \times (c-1)$. If the p-value of the test statistic is greater than the specified significance level, the two variables are considered to be independent.

Similarly, the G-square statistic is calculated as

$$G^2 = 2\sum_{i=1}^{r}\sum_{j=1}^{c} O_{ij}\ln(\frac{O_{ij}}{E_{ij}})$$

where O_{ij} is an observed frequency, E_{ij} is the expected frequency, and the degrees of freedom for the test is $(r-1)\times(c-1)$. If the p-value of the test statistic is greater than the specified significance level, the two variables are considered to be independent.

The mutual information between X and Y is defined as

$$I(X,Y) = \sum_{x=1}^{r}\sum_{y=1}^{c} p(x,y)\ln(\frac{p(x,y)}{p(x)p(y)})$$

where $p(x,y) = \frac{O_{xy}}{N}$ is the joint distribution function of X and Y; $p(x) = \frac{\sum_{y=1}^{c} O_{xy}}{N}$ and $p(y) = \frac{\sum_{x=1}^{r} O_{xy}}{N}$ are the marginal probability distributions of X and Y, respectively; and N is the total number of observations in the training data.

The mutual information is then normalized to be between 0 and 1 as

$$NI(X,Y) = \sqrt{1-e^{-2I(X,Y)}}$$

where $I(X,Y)$ is the mutual information between X and Y.

If the value of the normalized mutual information is less than the specified significance level, the two variables are considered to be independent.

To test the conditional independence between two variables X and Y given a subset S of other variables ($X \notin S$ and $Y \notin S$), the corresponding statistics are summed for each value combination of S, and the corresponding degrees of freedom for the chi-square and G-square statistics is $(r-1)\times(c-1)\times q$, where q is the number of value combinations for S.

The following PROC BNET options are related to independence tests:

- The INDEPTEST= option specifies which test statistic or combination of them to use.
- The ALPHA= option specifies the significance level for the chi-square and the G-square statistics.
- The MIALPHA= option specifies the significance level for the normalized mutual information.

Variable Selection

A Bayesian network is a graphical model that consists of two parts, $< G, P >$, where G is a directed acyclic graph (DAG) whose nodes correspond to the random variables in U (U is the set of input variables plus the target variable in PROC BNET and P is a set of local probability distributions, one for each node conditional on each value combination of the parents). The joint probability distribution of U can be factorized to the product of the local probability distributions; that is, $p(U) = \Pi_{X \in U}\, p(X|\pi(X))$, where $\pi(X)$ are the parents of X. It is assumed that the network structure G and the probability distribution P are faithful to each other; that is, every conditional independence in the structure G is also present in P, and vice versa.

Given a target variable T, a Markov blanket of T is defined as a subset of input variables $MB \subseteq U - \{T\}$ such that T is conditionally independent of each of the remaining input variables $X \in U - MB - \{T\}$ given MB. Under the faithful assumption, the Markov blanket of T is unique. According to the definition of the Markov blanket, the probability distribution of T is completely determined by its Markov blanket; therefore, the Markov blanket can be used for variable selection.

PROC BNET supports two types of variable selection: one by independence tests between each input variable and the target (when PRESCREENING=1) and the other by conditional independence tests between each input variable and the target given any subset of other input variables (when VARSELECT=1, 2, or 3). When a STRUCTURE option is set to GENERAL then VARSELECT=0 is used and any other setting of VARSELECT is ignored. Similarly, when a STRUCTURE option is set to GENERAL then PRESCREENING=1 is used and any other setting of PRESCREENING is ignored.

Specifying the INNETWORK= option in the PROC BNET statement enables you to override the variable selections that are made by the PRESCREENING= and VARSELECT= options. Thus, a variable that is selected by independence tests might not be in the network if you have excluded it by using the INNETWORK= option. Similarly, a variable that is deemed independent of the target might get in the network if you have included it by using the INNETWORK= option. Hence, the selections that are made using independence tests are not final, and the INNETWORK= option specifications can override these selections.

PROC BNET uses specialized data structures to efficiently compute the contingency tables for any variable combination, and it uses dynamic candidate generation to reduce the false candidates (variable combinations).

Structure Learning

In general, there are two approaches to learning the network structure: one is score-based, and the other is constraint-based. The score-based approach uses a score function to measure how well a structure fits the training data and tries to find the structure that has the best score. The constraint-based approach uses independence tests to determine the edges and the directions.

PROC BNET uses both score-based and constraint-based approaches to learn the network structure. It uses the BIC (Bayesian information criterion) score, which is defined as

$$\mathrm{BIC}(G, D) = N \sum_{i=1}^{n} \sum_{j=1}^{q_i} \sum_{k=1}^{r_i} p(\pi_{ij}) p(X_i = v_{ik}|\pi_{ij}) \ln p(X_i = v_{ik}|\pi_{ij}) - \frac{M}{2} \ln N$$

where G is a network, D is the training data set, N is the number of observations in D, n is the number of variables, X_i is a random variable, r_i is the number of levels for X_i, v_{ik} is the kth value of X_i, q_i is

the number of value combinations of X_i's parents, π_{ij} is the jth value combination of X_i's parents, and $M = \sum_{i=1}^{n}(r_i - 1) \times q_i$ is the number of parameters for the probability distributions.

PROC BNET uses independence tests to determine the edges and the directions as follows. Assume that you have three variables, X, Y, and Z, and that it has been determined (using independent tests) that there are edges between X and Z and Y and Z, but no edge between X and Y. If X is conditionally dependent of Y given any subset of variables $S = \{Z\} \bigcup S'$, $S' \subseteq U - \{X, Y, Z\}$, then the direction between X and Z is $X \rightarrow Z$ and the direction between Y and Z is $Y \rightarrow Z$. Notice that using independence tests alone might not be able to orient all edges because some structures are equivalent with respect to conditional independence tests. For example, $X \leftarrow Y \leftarrow Z$, $X \rightarrow Y \rightarrow Z$, and $X \leftarrow Y \rightarrow Z$ belong to the same equivalence class. In these cases, PROC BNET uses the BIC score to determine the directions of the edges.

PROC BNET learns different types of network structures: naive Bayesian (NB), tree-augmented naive (TAN), Bayesian network-augmented naive (BAN), parent-child Bayesian network (PC), Markov blanket (MB), and general Bayesian network (GENERAL). Based on the network structure that is specified, it uses different algorithms. For example, if you specify STRUCTURE=TAN, the procedure uses the maximum spanning tree to learn the tree structure, where the weight for an edge is the mutual information between the two nodes. PROC BNET uses either the BESTONE or BESTSET value of the PARENTING= option to learn the other network structures (BAN, PC, MB).

PROC BNET orders the input variables based on the BIC score with the target. The BIC score of an input variable X with the target is defined as

$$\mathrm{BIC}(X, T) = \max(\mathrm{BIC}(X \rightarrow T), \mathrm{BIC}(T \rightarrow X))$$

where $\mathrm{BIC}(X \rightarrow T)$ is the BIC score when X is the parent of T (ignoring all the remaining variables) and $\mathrm{BIC}(T \rightarrow X)$ is the BIC score when X is the child of T (ignoring all the remaining variables).

PROC BNET learns the parents of the target first for structures PC and MB. Then it learns the parents of the input variable that has the highest BIC score with the target. It continues learning the parents of the input variable that has the next highest BIC score, and so on. When learning the parents of a node, it first determines the edges by using independence tests. Then it orients the edges by using both independence tests and the BIC score. PROC BNET uses the BIC score not only for orienting the edges but also for controlling the network complexity, because a complex network that has more parents is penalized in the BIC score.

Both the BESTONE and BESTSET value of the PARENTING= option try to find the local optimum structure for each node. BESTONE adds the best candidate variable to the parents at each iteration, whereas BESTSET tries to pick the best set of variables among the candidate sets.

If you have many input variables, structure learning can be time consuming, because the number of variable combinations is exponential. Therefore, variable selection is strongly recommended.

The general Bayesian network relaxes the assumption of a naive network that all input variables are directly connected to the target variable. It thus requires more computing but is capable of producing networks that you can interpret by using intuition and prior knowledge of network connections. Combined with the INNETWORK= option, you can use the knowledge that you glean from the data together with your subject expertise to arrive at a satisfactory network.

Parameter Learning

Parameter learning determines the probability distribution for each node in a network structure. In PROC BNET, the probability distribution is discrete because the interval variables are binned.

You can use the resulting probability distribution table to score an observation $(x_1, x_2, \ldots, x_{n-1})$ as

$$\begin{aligned} \mathrm{argmax}_c\, p(T = c | x_1, x_2, \ldots, x_{n-1}) &= p(x_1, x_2, \ldots, x_{n-1} | T = c) \times K \\ &= \Pi_i\, p(x_i | \pi(X_i)) \times K \end{aligned}$$

where c is a level of the target variable, $\pi(X_i)$ are the parents of X_i, K is a constant, and $X_n = T$ (target) for convenience.

To estimate the parameters $p(x_i | \pi(X_i))$, PROC BNET uses an additive smoothing technique:

$$\hat{p}(x_i | \pi(X_i)) = \frac{\mathrm{counts_of}(xi, pi(X_i)) + \alpha}{\mathrm{counts_of}(pi(X_i)) + \alpha \times n_i}$$

where α is the smoothing parameter ($\alpha = 0$ corresponds to no smoothing, and PROC BNET uses $\alpha = 1$) and n_i is the number of possible values for X_i.

The general reason for smoothing is to avoid overfitting the data. The case where the count of some class is 0 is just a particular case of overfitting. PROC BNET still smooths the probabilities when every class is observed.

k-fold Cross Validation

The CROSSVALIDATION statement performs *k*-fold cross validation to assess the accuracy of a model. During cross validation, all data are divided into *k* subsets (folds), where *k* is the value of the KFOLD= option. For each fold, a new model is trained on the (*k*–1) folds and then validated using the selected (holdout) fold. The assessment metrics are then averaged across all the holdout folds. The CROSSVALIDATION statement creates a table that has *k*+1 rows. The first *k* rows contain the assessment metrics for each holdout fold, and the last row contains the average across all the holdout folds. Which metrics the table includes depends on whether the target is nominal or interval. For nominal targets, the table includes the values of square error, mean consequential error, and multiclass log loss. For interval targets, the table includes the values of square error, absolute error, and square logarithmic error.

Displayed Output

The following sections describe the output that PROC BNET produces by default. The output is organized into various tables, which are discussed in the order of their appearance.

Model Information

The "Model Information" table contains the initial training settings, such as significance level, structure, and number of bins.

Fit Statistics

The "Fit Statistics" table contains the fit statistics of the Bayesian network.

Number of Observations

The "Number of Observations" table contains the total number of observations and the number of observations that are used.

Variable Level

The "Variable Level" table contains the details of each level of the variables. The columns include the observed target, predicted event, predicted nonevent, and total numbers of events or nonevents for the training data.

Variable Order

The "Variable Order" table contains the order of the input variables based on the BIC score with the target.

Variable Information

The "Variable Information" table contains the variable information, such as number of levels, number of missing values, and so on.

Variable Selection Report

The "Variable Selection Report" table contains the variable selection results as determined by independence tests. You can override these results by specifying the INNETWORK= option in the PROC BNET statement. This table shows the selection status and test results for the variable list, based on the settings for prescreening and variable selection prior to the rollback. This information helps you determine the reason for the rollback.

Validation Information

The "Validation Information" table contains the validation results. If you specify the PARTITION statement, then the misclassification errors are misclassification errors in the validation data; if not, then the misclassification errors are misclassification errors in the training data.

ODS Table Names

Each table that the BNET procedure creates has a name associated with it. You must use this name to refer to the table when you use the ODS statements. These names are listed in Table 5.3.

Table 5.3 ODS Tables Produced by PROC BNET

Table Name	Description	Statement	Option
BestConfiguration	Hyperparameters and objective function values for the best configuration	AUTOTUNE	Default
CrossValidateMLFitStat	Per-fold and average assessment metrics of *k*-fold cross validation	CROSSVALIDATION	Default
EvaluationHistory	Values of the hyperparameters and the objective function for all configurations	AUTOTUNE	Default
FitStatistics	Contains the fit statistics of the network	PROC BNET	Default
ModelInfo	Contains the initial training settings, such as significance level, structure, and number of bins	PROC BNET	Default
NObs	Contains the number of observations for training, validation and testing, and so on	PROC BNET	Default
PredIntoName	Predicted target variable	PROC BNET	PRINTTARGET
PredProbName	Predicted probability variables	PROC BNET	PRINTTARGET
TunerInfo	Setup values that are used by the tuner	AUTOTUNE	Default
TunerResults	Values of the hyperparameters, the objective function for the default configuration (iteration 0), and up to 10 best configurations found	AUTOTUNE	Default
TunerSummary	Statistics about the tuning process	AUTOTUNE	Default
TunerTiming	Total time spent on different tasks during tuning	AUTOTUNE	Default
ValidInfo	Contains the validation results	PROC BNET	BESTMODEL
VarInfo	Contains the variable information such as number of levels, number of missing values, and so on	PROC BNET	Default
VarLevel	Contains the details of each level of the variables	PROC BNET	Default

Table 5.3 *continued*

Table Name	Description	Statement	Option
VarOrder	Contains the order of the input variables	PROC BNET	Default
VarSelect	Contains the variable selection results using independence tests	PROC BNET	Default

Examples: BNET Procedure

Example 5.1: Naive Bayesian Network

This example shows how you can use PROC BNET to learn a naive Bayesian network for the Iris data that is available in the Sashelp library. The following DATA step loads the Iris data into your SAS library. This DATA step assumes that your SAS libref is named mylib, but you can substitute any appropriately defined SAS libref.

```
data mylib.iris;
   set sashelp.iris;
run;
```

The following statements specify MAXPARENTS=1, PRESCREENING=0, and VARSELECT=0 to request that PROC BNET use only one parent for each node and use all the input variables:

```
proc bnet data=mylib.iris numbin=3 structure=Naive maxparents=1
        prescreening=0 varselect=0
        outnetwork=mylib.network;
    target Species;
    input PetalWidth PetalLength SepalLength SepalWidth/level=interval;
run;
```

The following statements produce Output 5.1.1, which shows the network structure. In the structure, Species is the parent of PetalWidth, PetalLength, SepalLength, and SepalWidth.

```
proc print data=mylib.network noobs label;
    var _parentnode_ _childnode_;
    where _type_="STRUCTURE";
run;
```

Output 5.1.1 Naive Bayesian Network Structure

Parent Node	Child Node
Species	PetalLength
Species	PetalWidth
Species	SepalLength
Species	SepalWidth

Example 5.2: Tree-Augmented Naive Bayesian Network

This example also uses the Iris data set that is available in the Sashelp library. In the following statements, STRUCTURE=TAN results in a tree-augmented Bayesian network:

```
data mylib.iris;
   set sashelp.iris;
run;

 proc bnet data=mylib.iris numbin=3 structure=TAN
            prescreening=0 varselect=0
            outnetwork=mylib.network;
     target Species;
     input PetalWidth PetalLength SepalLength SepalWidth/level=interval;
 run;
```

The following statements produce Output 5.2.1, which shows the network structure. In the structure, Species is a parent of PetalWidth, PetalLength, SepalLength, and SepalWidth. In addition, PetalWidth is a parent PetalLength, SepalWidth is a parent of SepalLength, and PetalWidth is a parent of SepalWidth.

```
 proc print data=mylib.network noobs label;
     var _parentnode_ _childnode_;
     where _type_="STRUCTURE";
 run;
```

Output 5.2.1 TAN Network Structure

Parent Node	Child Node
Species	PetalLength
PetalWidth	PetalLength
Species	PetalWidth
Species	SepalLength
PetalLength	SepalLength
Species	SepalWidth
SepalLength	SepalWidth

Example 5.3: Parent-Child Bayesian Network

This example also uses the Iris data set that is available in the Sashelp library. In the following statements, STRUCTURE=PC results in a parent-child Bayesian network:

```
data mylib.iris;
   set sashelp.iris;
run;

 proc bnet data=mylib.iris numbin=3 structure=PC
             prescreening=0 varselect=0
             outnetwork=mylib.network;
     target Species;
     input PetalWidth PetalLength SepalLength SepalWidth/level=interval;
 run;
```

The following statements produce Output 5.3.1, which shows the network structure. In the structure, PetalLength is the parent of Species, and Species is the parent of PetalWidth, SepalLength, and SepalWidth.

```
 proc print data=mylib.network noobs label;
     var _parentnode_ _childnode_;
     where _type_="STRUCTURE";
 run;
```

Output 5.3.1 Parent-Child Network Structure

Parent Node	Child Node
PetalLength	Species
Species	PetalWidth
Species	SepalLength
Species	SepalWidth

Example 5.4: Markov Blanket

This example uses the HMEQ sample data set that is available in the Sampsio library to learn a Markov blanket Bayesian network, which is specified by STRUCTURE=MB:

```
data mylib.hmeq;
   set sampsio.hmeq;
run;

 proc bnet data=mylib.hmeq indeptest=MI mialpha=0.2 structure=MB nbin=5
         missingint=IMPUTE missingnom=LEVEL
         outnetwork=mylib.network;
     target Bad;
     input Reason Job Delinq Derog Ninq/level=nominal;
     input Loan Mortdue Value Yoj Clage Clno Debtinc/level=interval;
 run;
```

The following statements produce Output 5.4.1, which shows the network structure. In the structure, Bad is a parent of Delinq and Derog, Delinq and Ninq are the other parents of Derog, and Ninq is the other parent of Delinq.

```
proc print data=mylib.network noobs label;
    var _parentnode_ _childnode_;
    where _type_="STRUCTURE";
run;
```

Output 5.4.1 Markov Blanket Network Structure

Parent Node	Child Node
BAD	DELINQ
NINQ	DELINQ
BAD	DEROG
DELINQ	DEROG
NINQ	DEROG

Example 5.5: Bayesian Network—Augmented Naive Bayesian Network

This example also uses the HMEQ sample data set that is available in the Sampsio library to learn a BAN structure, which is specified by STRUCTURE=NAIVE:

```
data mylib.hmeq;
   set sampsio.hmeq;
run;

 proc bnet data=mylib.hmeq numbin=10 alpha=0.1 structure=Naive
            missingint=IMPUTE missingnom=LEVEL
            outnetwork=mylib.network;
     target Bad;
     input Reason Job Delinq Derog Ninq/level=nominal;
     input Loan Mortdue Value Yoj Clage Clno Debtinc/level=interval;
 run;
```

The following statements produce Output 5.5.1, which shows the network structure. In the structure, Bad is a parent of Delinq, Derog, Job, Ninq, Clage, Clno, Loan, and Mortdue. In addition, Delinq is the other parent of Derog, and Job is a parent of both Mortdue and Clno.

```
proc print data=mylib.network noobs label;
    var _parentnode_ _childnode_;
    where _type_="STRUCTURE";
run;
```

Output 5.5.1 BAN Network Structure

Parent Node	Child Node
BAD	DELINQ
BAD	DEROG
DELINQ	DEROG
BAD	JOB
BAD	NINQ
BAD	CLAGE
BAD	CLNO
JOB	CLNO
BAD	LOAN
BAD	MORTDUE
JOB	MORTDUE

Example 5.6: Bayesian Network—General Network

This example also uses the HMEQ sample data set that is available in the Sampsio library to learn a general structure, which is specified by STRUCTURE=GN:

```
data mylib.hmeq;
   set sampsio.hmeq;
run;

 proc bnet data=mylib.hmeq numbin=10 alpha=0.1 structure=GN
             missingint=IMPUTE missingnom=LEVEL
             outnetwork=mylib.network;
     target Bad;
     input Reason Job Delinq Derog Ninq/level=nominal;
     input Loan Mortdue Value Yoj Clage Clno Debtinc/level=interval;
 run;
```

The following statements produce the table in Output 5.6.1, which shows the network structure. In the structure, Bad is embedded in a general network, showing its relationship to other variables.

```
 proc print data=mylib.network noobs label;
     var _parentnode_ _childnode_;
     where _type_="STRUCTURE";
 run;
```

Output 5.6.1 General Network Structure

Parent Node	Child Node
DELINQ	BAD
DEROG	DELINQ
CLNO	JOB
DEROG	NINQ
LOAN	REASON
BAD	CLAGE
CLAGE	CLNO
BAD	DEBTINC
VALUE	LOAN
VALUE	MORTDUE
JOB	VALUE
REASON	YOJ

Example 5.7: Model Selection

This example uses the German Credit sample data that is available in the Sampsio library to learn the best Bayesian network model among all network structures: naive, TAN, PC, and MB, with or without variable selection. PROC BNET also tries to choose the best value for the MAXPARENTS= option. About 30% of the input data is used for validation.

```
data dmagecr_part;
  set sampsio.dmagecr;
  seed=12345;
  if ranuni(seed) < 0.7 then partind=1;
  else partind=0;
  id=_N_;
run;

data mylib.dmagecr;
   set dmagecr_part;
run;

 proc bnet data=mylib.dmagecr numbin=10 alpha=0.05
          structure=Naive TAN PC MB varselect=0 1 bestmodel
          outnetwork=mylib.network;
     target Good_Bad;
     input  Checking History Purpose Savings Employed Installp Marital Coapp
            Resident Property Other Housing Existcr Job Depends Telephon
            Foreign/level=nominal;
     input Age Amount Duration/level=interval;
     partition rolevar= PartInd (TRAIN='1' VALIDATE='0' TEST='2');
     ods output nobs=nobs fitstatistics=fit validinfo=validinfo;
 run;
```

Output 5.7.1 shows the information of number of observations: 695 observations are used for training and 305 are used for validation.

Output 5.7.1 Model Selection: Number of Observations

The BNET Procedure

Number of Observations	
Number of Observations Read	1000
Number of Observations Used	1000
Number of Observations Used for Training	695
Number of Observations Used for Validation	305
Number of Observations Used for Testing	0

Output 5.7.2 shows the fit statistics. In the resulting network, there are 13 nodes and 13 links between the nodes, and the number of parameters is 129.

Output 5.7.2 Model Selection: Fit Statistics

Fit Statistics	
Number of Nodes	13
Number of Links	13
Average Degree	2
Maximum Number of Parents in Network	2
Number of Parameters	129
Score	-11092.55
Validation Misclassification Rate	0.23934426
Test Misclassification Rate	.

Output 5.7.3 shows the validation results for each parameter combination. The PC Bayesian network structure has misclassified 73 observations out of 305 validation observations when VARSELECT=0 and the value of the MAXPARENTS= option is greater than or equal to 2. The TAN structure has 81 misclassification errors when VARSELECT=0 and MAXPARENTS=2. The naive Bayesian network has 81 misclassification errors when VARSELECT=0 and the value of the MAXPARENTS= option is greater than or equal to 2. The MB Bayesian network structure has 113 misclassification errors.

Output 5.7.3 Model Selection: Validation Information

Validation Information								
Best Model	Misclassification Rate	Misclassification Errors	N Observations for Assessment	Significance Threshold	Prescreening	Variable Selection	Structure	Parenting Method
Yes	0.23934	73	305	0.05	1	0	PC	BestSet
	0.23934	73	305	0.05	1	0	PC	BestSet
	0.23934	73	305	0.05	1	0	PC	BestSet
	0.23934	73	305	0.05	1	0	PC	BestSet
	0.26557	81	305	0.05	1	0	TAN	BestSet
	0.26557	81	305	0.05	1	0	Naive	BestSet
	0.26557	81	305	0.05	1	0	Naive	BestSet
	0.26557	81	305	0.05	1	0	Naive	BestSet
	0.26557	81	305	0.05	1	0	Naive	BestSet
	0.26885	82	305	0.05	1	1	PC	BestSet
	0.26885	82	305	0.05	1	1	PC	BestSet
	0.26885	82	305	0.05	1	1	PC	BestSet
	0.26885	82	305	0.05	1	1	PC	BestSet
	0.28197	86	305	0.05	1	0	PC	BestSet
	0.28197	86	305	0.05	1	0	Naive	BestSet
	0.34754	106	305	0.05	1	1	PC	BestSet
	0.34754	106	305	0.05	1	1	Naive	BestSet
	0.34754	106	305	0.05	1	1	Naive	BestSet
	0.34754	106	305	0.05	1	1	Naive	BestSet
	0.34754	106	305	0.05	1	1	Naive	BestSet
	0.34754	106	305	0.05	1	1	Naive	BestSet
	0.35738	109	305	0.05	1	1	TAN	BestSet
	0.37049	113	305	0.05	1	3	MB	BestSet
	0.37049	113	305	0.05	1	3	MB	BestSet
	0.37049	113	305	0.05	1	3	MB	BestSet
	0.37049	113	305	0.05	1	3	MB	BestSet
	0.37049	113	305	0.05	1	3	MB	BestSet

Output 5.7.3 *continued*

Validation Information Max N Parents
2
3
4
5
2
2
3
4
5
2
3
4
5
1
1
1
1
2
3
4
5
2
1
2
3
4
5

The following statements produce Output 5.7.4, which shows that PROC BNET has learned a PC Bayesian network structure. In the structure, Checking is the parent of Good_Bad, Housing is the parent of Property, and Good_Bad is the parent of all the selected input variables except its parent Checking.

```
proc print data=mylib.network noobs label;
    var _parentnode_ _childnode_;
    where _type_="STRUCTURE";
run;
```

Output 5.7.4 Model Selection: Best Structure

Parent Node	Child Node
checking	good_bad
good_bad	employed
good_bad	foreign
good_bad	history
good_bad	housing
good_bad	other
good_bad	property
housing	property
good_bad	purpose
good_bad	savings
good_bad	age
good_bad	amount
good_bad	duration

Example 5.8: Tuning a Bayesian Network Model

This example illustrates how you can use the AUTOTUNE statement to automatically tune the hyperparameters of a Bayesian network classifier model from observations in the German credit benchmark data, which are available in the sampsio.dmagecr data set. This data set contains 1,000 observations, each of which contains an applicant's information, including the applicant's credit rating ("GOOD" or "BAD"). The binary target is named GOOD_BAD. Other input variables are Checking, Duration, History, and so on.

The contents of the sampsio.dmagecr data set are described at http://support.sas.com/documentation/cdl/en/emgs/59885/HTML/default/a001026918.htm.

You can load the sampsio.dmagecr data set into your SAS library by specifying your SAS libref in the following DATA step. These statements assume that your SAS libref is named mylib, but you can substitute any appropriately defined SAS libref.

```
data mylib.dmagecr;
   set sampsio.dmagecr;
   seed=12345;
   if ranuni(seed) < 0.7 then partind=1;
   else                       partind=0;
run;

proc bnet
   data=mylib.dmagecr outnetwork=mylib.network;
   target Good_Bad;
   input Checking History Purpose Savings Employed Installp
      Marital Coapp Resident Property Other Housing Existcr
      Job Depends Telephon Foreign / level=nominal;
   input Age Amount Duration / level=interval;
   partition rolevar= PartInd (TRAIN='1' VALIDATE='0');
   autotune
      /* Tuning Parameters
      You do not need to specify any tuning parameters for the default
      tuning process. If you want to make adjustments to the default
      tuning process, uncomment the following block of code and change
```

```
      any of the tuning parameters' attributes.

      tuningParameters=(
         ALPHA         ( lb=0.01  ub=0.99  init=0.5 )
         INDEPTEST     ( values=CHISQUARE GSQUARE MI CHIGSQUARE init=CHIGSQUARE )
         MAXPARENTS    ( lb=1     ub=5     init=5   )
         MIALPHA       ( lb=0.0   ub=1.0   init=0.05  )
         MISSINGINT    ( values=IMPUTE IGNORE       init=IGNORE exclude )
         MISSINGNOM    ( values=IMPUTE IGNORE LEVEL init=IGNORE exclude )
         NUMBIN        ( lb=2     ub=20    init=5  )
         PARENTING     ( values=BESTONE BESTSET init=BESTSET )
         PRESCREENING  ( values=0 1 init=1 )
         STRUCTURE     ( values=NAIVE TAN PC MB init=PC )
         VARSELECT     ( values=0 1 2 3 init=1 )
      )
      */
   ;
   /* Remove this line to see all results */
   ods select BestConfiguration EvaluationHistoryPlot IterationHistoryPlot;
run;
```

The preceding statements produce the table and the plots shown in Output 5.8.1 through Output 5.8.3. The table in Output 5.8.1 displays the evaluation number, the values of the tuning parameters, and the error metric value for the best Bayesian network classifier model that the tuner found. Note that the ODS SELECT statement limits the displayed results to a single table and two plots. You can remove this statement to display all tables and plots. For the full list of ODS tables that PROC BNET produces, see Table 5.3.

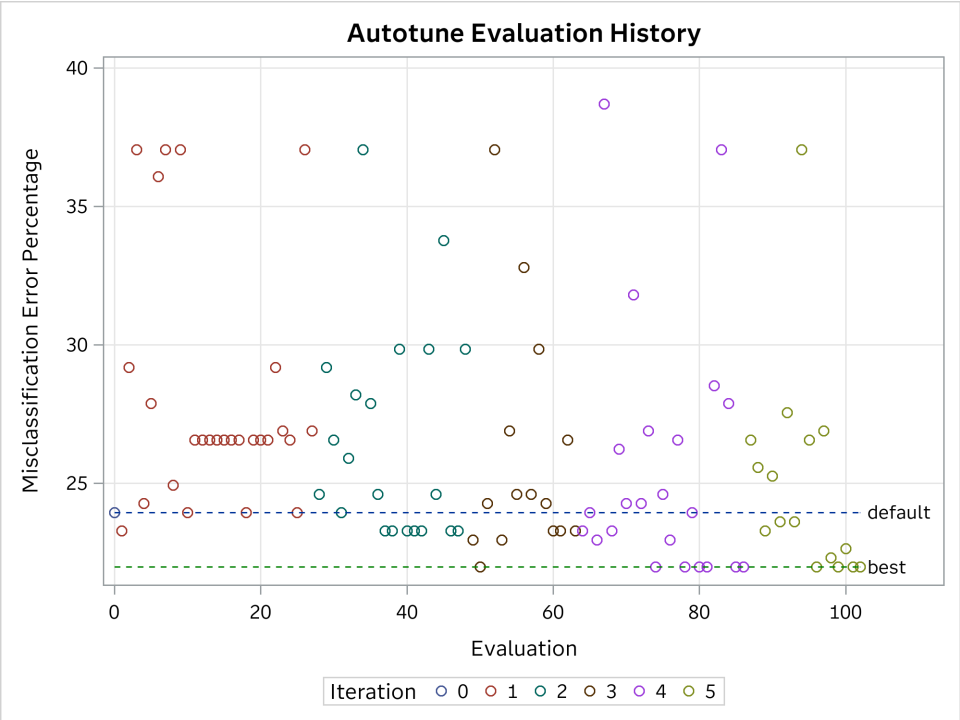
Output 5.8.1 Best Configuration Table

The BNET Procedure

Best Configuration	
Evaluation	50
Alpha	0.90143412
Maximum Number of Parents	3
Parenting Algorithm	BESTSET
Prescreening	ZERO
Network Structure	PC
Variable Selection	TWO
Number of Bins	7
Mutual Information Alpha	0.25282
Independence Test Method	CHIGSQUARE
Misclassification Error Percentage	21.97

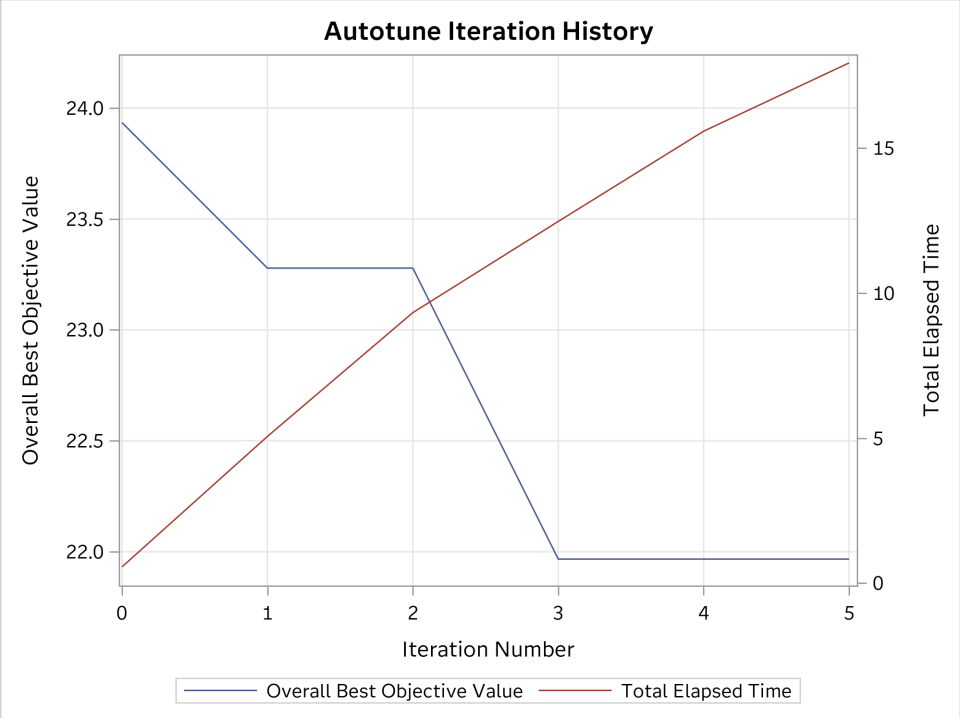
Output 5.8.2 displays a scatter plot of all configurations that the tuner tried. The objective values are shown on the Y axis, and the evaluation numbers are shown on the X axis.

Output 5.8.2 Evaluation History Plot



The plot in Output 5.8.3 displays how the best found objective value and the elapsed time changed with each iteration of the tuner.

Output 5.8.3 Iteration History Plot



References

Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. San Francisco: Morgan Kaufmann.

Chapter 6
DYNBNET Procedure

Contents

Overview: DYNBNET Procedure **134**
- PROC DYNBNET Features 134
- Using SAS Viya Workbench 134

Getting Started **135**

Syntax: DYNBNET Procedure **135**
- PROC DYNBNET Statement 135
- CLIQUES Statement 138
- INITIALBELIEFS Statement 139
- LINKS Statement 140
- OUTPUT Statement 141
- VARIABLELEVELS Statement 141
- VARIABLEODES Statement 142
- VARIABLEROLES Statement 142
- VARIABLESYNCPARMS Statement 143

Details: DYNBNET Procedure **144**
- Two-Time-Period Bayesian Network 144
- Specifying Models by Using the FCMP Procedure 146
- Canonical Form of a Multivariate Normal Distribution 147
- Reformulating a Bayesian Network as a Clique Tree 148
- Approximating Nonlinear Models by Using Linear Models 149
- Filtering Algorithm 150
- Displayed Output 155
- ODS Table Names 156

Example: DYNBNET Procedure **156**
- Example 6.1: Fault Detection in a System of Two Tanks 156
- Example 6.2: Use of an Existing Model for Fault Detection 165

References **165**

Overview: DYNBNET Procedure

Bayesian networks consist of variables, which are represented as nodes, and relationships among these variables, which are represented as directed edges. The variables are a mix of hidden and observed variables. The DYNBNET procedure is a dynamic procedure that computes probabilistic distributions of hidden variables, given the values of observed variables. This procedure is referred to as dynamic because it is used to update the distributions of hidden variables as new values of observed variables become available over time. Arranging variables in a network provides significant computational savings, enabling you to solve problems that could not be tackled otherwise.

PROC DYNBNET Features

PROC DYNBNET has the following features:

- It filters observed variables by time to generate outgoing beliefs in hidden variables.
- The output includes outgoing beliefs at two levels of detail: beliefs for the most probable level of a nominal variable, and beliefs for all possible levels of nominal variables with their associated probabilities.
- Output is generated for cliques that are obtained by processing the network for use in subsequent runs of the procedure in order to avoid generating cliques each time.
- A table is generated to output the information that serves as input in the subsequent runs of the procedure with the availability of new observations.

Using SAS Viya Workbench

SAS Viya Workbench has greatly simplified the SAS programming experience by offering a syntax style nearly identical to that of SAS 9 and earlier releases. However, there are minor exceptions. In the past, SAS Viya procedures required the use of a CAS engine libref. This engine served multiple purposes, but it was primarily used to upload data to the CAS server and manage the data. Its additional purposes included support of new data types and improved performance for large data sets. No CAS server is needed for SAS Viya Workbench, because all computations are done in place for you in a single location.

In the following program, the LIBNAME statement creates a mylib libref that the examples in this chapter use to refer to your current or desired work directory:

```
libname mylib ".";
```

For more information, see *SAS Viya Workbench: Getting Started*.

Getting Started

The DYNBNET procedure uses the values of observed variables to calculate beliefs in hidden variables. You can use it to estimate unknown failure probabilities of components in a system of interacting parts in various settings. Probabilities of hidden faults are estimated and can be flagged for possible intervention well before a system shuts down with a severe malfunction. Other scenarios are possible where detection of shifts in modes of operation is useful in a system of interacting components. In such scenarios, you can estimate the probabilities of different modes of operation as these modes evolve over time.

Syntax: DYNBNET Procedure

The following statements are available in the DYNBNET procedure:

PROC DYNBNET < *options* > **;**
  CLIQUES DATA=*libref.data-table* < / *options* > **;**
  INITIALBELIEFS DATA=*libref.data-table* < / *options* > **;**
  LINKS DATA=*libref.data-table* < / *options* > **;**
  OUTPUT OUT=*libref.data-table* **;**
  VARIABLELEVELS DATA=*libref.data-table* < / *options* > **;**
  VARIABLEODES DATA=*libref.data-table* < / *options* > **;**
  VARIABLEROLES DATA=*libref.data-table* < / *options* > **;**
  VARIABLESYNCPARMS DATA=*libref.data-table* < / *options* > **;**

The PROC DYNBNET statement and the OUTPUT statement are required. The INITIALBELIEFS, LINKS, and VARIABLEROLES statements are also required, unless you specify a data table by using the INPUTSTATE= option in the PROC DYNBNET statement. The following sections describe the PROC DYNBNET statement and then describe the other statements in alphabetical order.

PROC DYNBNET Statement

PROC DYNBNET < *options* > **;**

The PROC DYNBNET statement invokes the procedure. Table 6.1 summarizes important *options* in the PROC DYNBNET statement by function.

Table 6.1 PROC DYNBNET Statement Options

Option	Description
Data Options	
DATA=	Specifies the input data table that contains observations ordered by time
LINEARAPPROXIMATIONTYPE=	Specifies the technique to be used to generate a linear approximation of the nonlinear relationships
ODEAPPROXIMATIONTYPE=	Specifies which type of ODE approximation to use

Table 6.1 *continued*

Option	Description
SYNCTHRESHOLD=	Specifies the synchronization threshold
Input Data Table Option	
INPUTSTATE=	Specifies the input table that contains the saved state of the model, including the variables, network, and initial beliefs
Output Data Table Options	
OUTPUTCLIQUES=	Specifies the output table that contains the cliques in the clique tree obtained by processing the Bayesian network
OUTPUTDETAILS=	Specifies the output table that contains the outgoing beliefs of hidden variables specific to the branches (values) of hidden nominal variables. Thus, this table provides beliefs of interval variables over less probable values of nominal variables as well.
OUTPUTSTATE=	Specifies the output table that contains the variables, network, and outgoing beliefs

You can specify the following *options*:

DATA=*libref.data-table*

names the input data table for PROC DYNBNET to use. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to the data. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 134.

data-table specifies the name of the input data table.

INPUTSTATE=*libref.data-table*

specifies the input table that contains the saved state of the model, including the variables, network, and initial beliefs. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the input data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 134.

If you specify this option, then the DATA= option in the following statements is ignored: CLIQUES, INITIALBELIEFS, LINKS, VARIABLELEVELS, VARIABLEODES, VARIABLEROLES, and VARIABLESYNCPARMS.

LINEARAPPROXIMATIONTYPE=EXTENDED | UNSCENTED

LINAPPROX=EXTENDED | UNSCENTED

specifies the method to be used to generate a linear approximation of the nonlinear relationships. You can specify the following methods:

EXTENDED uses the extended method of linear approximation.

UNSCENTED uses the unscented method of linear approximation.

By default, LINEARAPPROXIMATIONTYPE=EXTENDED.

ODEAPPROXIMATIONTYPE=ALL | GRADIENT | NONE

ODEAPPROX=ALL | GRADIENT | NONE

specifies which type of ordinary differential equation (ODE) approximation to use. You can specify the following values:

ALL approximately solves the ODEs that are used in the mean and gradient calculations required in linearization.

GRADIENT approximately solves the ODEs that are used in the gradient calculations required in linearization.

NONE does not use any approximation to solve the ODEs that are used in the mean and gradient calculations required in linearization.

By default, ODEAPPROXIMATIONTYPE=GRADIENT.

OUTPUTCLIQUES=*libref.data-table*

OUTCLIQUES=*libref.data-table*

specifies the output table that contains the cliques in the clique tree that is obtained by processing the Bayesian network. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 134.

OUTPUTDETAILS=*libref.data-table*

OUTDETAILS=*libref.data-table*

specifies the output table that contains the outgoing beliefs of hidden variables specific to the branches (values) of hidden nominal variables. Thus, this table provides beliefs of interval variables over less probable values of nominal variables as well. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 134.

OUTPUTSTATE=*libref.data-table*

specifies the output table where PROC DYNBNET saves information about the model, including the variables, network, and outgoing beliefs. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 134.

SYNCTHRESHOLD=*number*

THRESHOLD=*number*

specifies the synchronization threshold. Probabilities over the levels of a nominal variable define a point in the probability space at each time. Euclidean distance (L2-norm) between two points that specify probabilities at the beginning and end of a time period is calculated and compared to the synchronization threshold. If the distance exceeds this threshold, then the beliefs in interval variables at all levels of the nominal variable are synchronized over the branches that correspond to the values of nominal variables. The valid range of *number* is 0 to 1, inclusive.

By default, SYNCTHRESHOLD=0.005.

CLIQUES Statement

CLIQUES DATA=*libref.data-table*< / *options*> **;**

The CLIQUES statement specifies the table of the factors and the cliques that form the clique tree that is obtained by processing the Bayesian network of variables. A clique tree permits an exact two-pass belief propagation over the tree. If cliques are not provided by this table, then the network is processed to form cliques, and you can use the OUTPUTCLIQUES= option in the PROC DYNBNET statement to save the cliques for use in subsequent runs of the filtering algorithm. It is recommended that you use the cliques that are generated by a previous run of the procedure instead of specifying cliques by using customized logic. The information in this table can alternatively be provided by a table that you specify in the INPUTSTATE= option in the PROC DYNBNET statement. The table has the following columns:

id	contains a clique ID or the ID of a factor in the clique. The model of the clique is obtained from the models of the factors that form a clique.
type	indicates whether the ID parameter value corresponds to a clique or a factor.
varname	contains the variable name when the ID parameter value corresponds to a belief factor for a variable.
stage	contains a 1 for a variable in the current time period or a 2 for a variable in the previous time period.
toid	contains the ID of the clique that receives the message.

You must specify the following option:

DATA=*libref.data-table*

names the input data table for PROC DYNBNET to use. *libref.data-table* is a two-level name, where

libref	refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to the data. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 134.
data-table	specifies the name of the input data table.

When the column names are not id, type, varname, stage, and toid, you can also specify the following *options* after a slash (/) for the column mapping of the CLIQUES data table:

ID=*id*

specifies the name of the id column.

TYPE=*type*

specifies the name of the type column.

VARNAME=*varname*

specifies the name of the varname column.

STAGE=*stage*
specifies the name of the stage column.

TOID=*toid*
specifies the name of the toid column.

INITIALBELIEFS Statement

INITIALBELIEFS DATA=*libref.data-table*< / *options*> **;**

The INITIALBELIEFS statement specifies the table of incoming beliefs in hidden variables from the previous run of the filter. The information in this table can alternatively be specified by a table that you specify in the INPUTSTATE= option in the PROC DYNBNET statement. The table has the following columns:

intervalvar	contains the name of the interval variable.
nominalvar	contains the name of the nominal variable.
varlevel	contains the level of the nominal variable.
mean	contains the mean of the interval variable or the probability of the nominal variable.
std	contains the standard deviation of the interval variable.

You must specify the following option:

DATA=*libref.data-table*
names the input data table for PROC DYNBNET to use. *libref.data-table* is a two-level name, where

libref	refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to the data. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 134.
data-table	specifies the name of the input data table.

When the column names are not intervalvar, nominalvar, varlevel, mean, and std, you can also specify the following *options* after a slash (/) for the column mapping of the INITIALBELIEFS data table:

INTERVALVAR=*intervalvar*
specifies the name of the intervalvar column.

NOMINALVAR=*nominalvar*
specifies the name of the nominalvar column.

VARLEVEL=*varlevel*
specifies the name of the varlevel column.

MEAN=*mean*
specifies the name of the mean column.

STD=*std*
specifies the name of the std column.

LINKS Statement

LINKS DATA=*libref.data-table*< / *options* > **;**

The LINKS statement specifies the table of parent-child relationships among variables in a two-time-period network. The information in this table can alternatively be provided by a table that you specify in the INPUTSTATE= option in the PROC DYNBNET statement. The expected structure of this table consists of a *child* column that contains the name of the child variable, a *parent* column that contains the name of the parent variable, and a *stage* column that has values of 1 for the current time period and 2 for the previous time period.

You must specify the following option:

DATA=*libref.data-table*
names the input data table for PROC DYNBNET to use. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to the data. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 134.

data-table specifies the name of the input data table.

When the column names are not *child*, *parent*, and *stage*, you can also specify the following *options* after a slash (/) for the column mapping of the LINKS data table:

CHILD=*child*
specifies the name of the child column.

PARENT=*parent*
specifies the name of the parent column.

STAGE=*stage*
specifies the name of the stage column.

OUTPUT Statement

OUTPUT OUT=*libref.data-table* **;**

The OUTPUT statement creates a data table that contains the outgoing beliefs of hidden variables. Beliefs of interval variables differ by levels of nominal variables. This table provides the interval variable beliefs for the most probable levels of the nominal variables.

You must specify the following option:

OUT=*libref.data-table*
names the output data table for PROC DYNBNET to use. You must specify this option before any other options. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to where the data table is to be stored. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 134.

data-table specifies the name of the output data table.

VARIABLELEVELS Statement

VARIABLELEVELS DATA=*libref.data-table*< / *options*> **;**

The VARIABLELEVELS statement specifies the table of valid levels of nominal variables. The information in this table can alternatively be provided by a table that you specify in the INPUTSTATE= option in the PROC DYNBNET statement. The expected structure of this table consists of a *varname* column that contains the name of a nominal variable and a *varlevel* column that contains a level of the nominal variable.

You must specify the following option:

DATA=*libref.data-table*
names the input data table for PROC DYNBNET to use. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to the data. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 134.

data-table specifies the name of the input data table.

When the column names are not *varname* and *varlevel*, you can also specify the following *options* after a slash (/) for the column mapping of the VARIABLELEVELS data table:

VARNAME=*varname*
specifies the name of the varname column.

VARLEVEL=*varlevel*
specifies the name of the varlevel column.

VARIABLEODES Statement

VARIABLEODES DATA=*libref.data-table*< / *options* > **;**

The VARIABLEODES statement specifies the table of variables by systems of ordinary differential equations (ODEs). Each system has a unique group number. The information in this table can alternatively be provided by a table that you specify in the INPUTSTATE= option in the PROC DYNBNET statement. The expected structure of this table consists of a *varname* column that contains the name of the variable in an ODE system and a *system* column that contains the ODE system's unique group number.

You must specify the following option:

DATA=*libref.data-table*
names the input data table for PROC DYNBNET to use. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to the data. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 134.

data-table specifies the name of the input data table.

When the column names are not *varname* and *system*, you can also specify the following *options* after a slash (/) for the column mapping of the VARIABLEODES data table:

VARNAME=*varname*
specifies the name of the varname column.

SYSTEM=*system*
specifies the name of the system column.

VARIABLEROLES Statement

VARIABLEROLES DATA=*libref.data-table*< / *options* > **;**

The VARIABLEROLES statement specifies the table of variables and their properties as time ID, control, hidden, or observed. The information in this table can alternatively be provided by a table that you specify in the INPUTSTATE= option in the PROC DYNBNET statement. The expected structure of this table consists of a *varname* column that contains the name of a variable and a *varrole* column that contains the role of the variable.

You must specify the following option:

DATA=*libref.data-table*
names the input data table for PROC DYNBNET to use. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to the data. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 134.

data-table specifies the name of the input data table.

When the column names are not *varname* and *varrole*, you can also specify the following *options* after a slash (/) for the column mapping of the VARIABLEROLES data table:

VARNAME=*varname*
specifies the name of the varname column.

VARROLE=*varrole*
specifies the name of the varrole column.

VARIABLESYNCPARMS Statement

VARIABLESYNCPARMS DATA=*libref.data-table*< / *options* > **;**

The VARIABLESYNCPARMS statement specifies the table of parameters that are used in synchronizing beliefs of hidden interval variables for different levels of hidden nominal variables. The information in this table can alternatively be provided by a table that you specify in the INPUTSTATE= option in the PROC DYNBNET statement. The table has the following columns:

intervalvar contains the name of the interval variable.

nominalvar contains the name of the nominal variable.

varlevel contains the level of the nominal variable.

mean contains the mean of the interval variable for synchronization or the frequency of synchronization for the nominal variable.

std contains the standard deviation of the interval variable.

You must specify the following option:

DATA=*libref.data-table*
names the input data table for PROC DYNBNET to use. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to the data. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 134.

data-table specifies the name of the input data table.

When the column names are not intervalvar, nominalvar, varlevel, mean, and std, you can also specify the following *options* after a slash (/) for the column mapping of the VARIABLESYNCPARMS data table:

INTERVALVAR=*intervalvar*
specifies the name of the intervalvar column.

NOMINALVAR=*nominalvar*
specifies the name of the nominalvar column.

VARLEVEL=*varlevel*
specifies the name of the varlevel column.

MEAN=*mean*
specifies the name of the mean column.

STD=*std*
specifies the name of the std column.

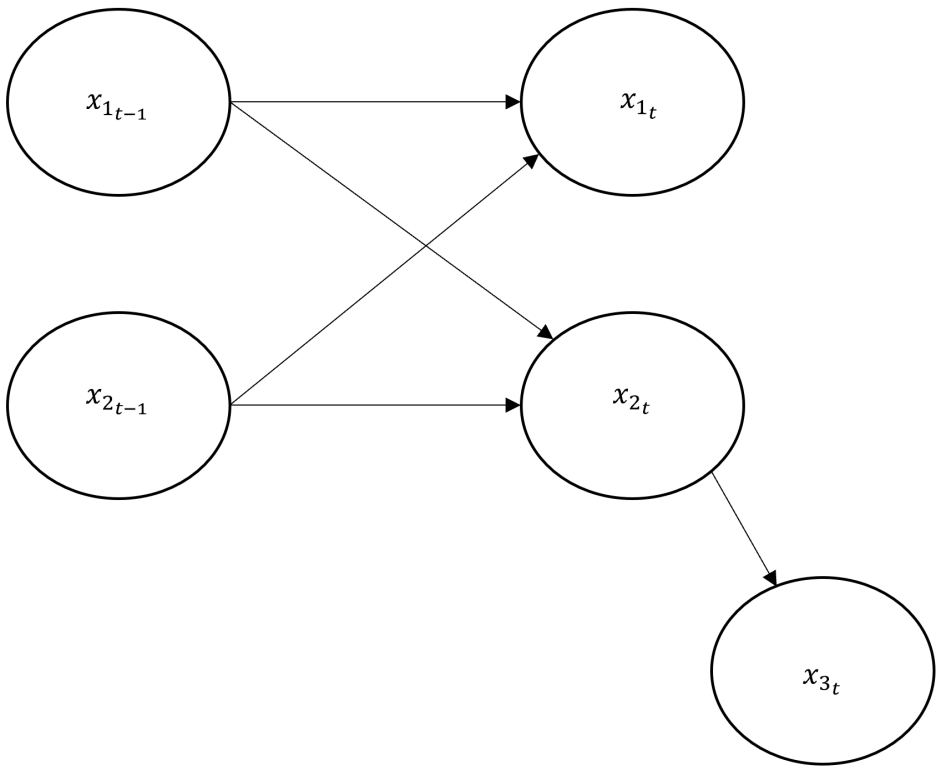
Details: DYNBNET Procedure

Two-Time-Period Bayesian Network

The dynamic nature of a problem is captured by modeling the Bayesian network over two time periods. Distributions of variables evolve from one time period to the next by using the relationships that are specified by directed edges in the network. The direction of an edge is specified as an arrow and connects a parent variable to a child variable. Thus, a child variable is a function of all parent variables, in which arrows point from parents to child. As expected, a parent-child relationship never points backward in time; an arrow points to a child variable in either the same time period or the next time period. Conversely, a child variable is a function of parent variables in the same and/or previous time period. Figure 6.1 shows a network over three variables, x_1, x_2, and x_3, where the second part of the subscripts, $t-1$ and t, denotes the time periods. This network has five variables over two time periods. Three functions, one for each child variable, are specified as follows, using the directed edges:

- x_{1_t} is a function of $x_{1_{t-1}}$ and $x_{2_{t-1}}$
- x_{2_t} is a function of $x_{1_{t-1}}$ and $x_{2_{t-1}}$
- x_{3_t} is a function of x_{2_t}

Figure 6.1 Two-Period Network



A consideration of nominal and/or interval variables is needed in order to examine these relationships. Let an interval child variable ic be the child of an interval parent variable ip and a nominal parent variable np. Let there be two possible values for the nominal parent: np_1 and np_2. In this scenario, a function determines the value of the interval child ic as a function of the interval parent ip for each value of the nominal parent np:

$$ic = f(ip, np), \text{ where } np = np_1 \text{ or } np_2$$

In general, functions that relate interval child variables to interval parent variables produce different results for different settings of the nominal parent variables. For example, a chemical reaction follows a different relationship in the absence or presence of a fault, where a fault is a nominal parent variable whose two possible values are *absent* and *present*. Therefore, different settings of nominal variables lead to a network whose models are specific to different nominal variable settings. Hence, the distributions over hidden variables are calculated separately for different settings of the nominal variables, which are referred to as branches in the subsequent discussion.

For a given branch, you can specify relationships among interval variables in two ways:

- An interval child variable in the current time period is a function (linear or nonlinear) of the interval parent variables from previous and/or current periods, where the function specifies the evolution of the child variable.
- A set of ordinary differential equations (ODEs) specifies the evolution of interval variables from one time period to the next.

For the network shown in Figure 6.1, either of the following options could be the appropriate model of the network, depending on its suitability to the underlying behavior:

- x_{1_t} and x_{2_t} are functions of $x_{1_{t-1}}$ and $x_{2_{t-1}}$, respectively, with added noise terms:

  $$x_{1_t} = f_1(x_{1_{t-1}}, x_{2_{t-1}}) + \epsilon_1, \text{ where } \epsilon_1 \sim \mathcal{N}(0, \sigma_1^2)$$

  $$x_{2_t} = f_2(x_{1_{t-1}}, x_{2_{t-1}}) + \epsilon_2, \text{ where } \epsilon_2 \sim \mathcal{N}(0, \sigma_2^2)$$

- A system of differential equations is specified as derivatives of x_1 and x_2 with respect to time, along with the initial conditions of x_1 and x_2 at time $t-1$:

  $$\frac{dx_1}{dt} = f_1(t, x_1, x_2)$$

  $$\frac{dx_2}{dt} = f_2(t, x_1, x_2)$$

  $$x_{1_{t-1}} = v_1 \text{ and } x_{2_{t-1}} = v_2$$

  Noise terms of ϵ_1 and ϵ_2 are added to x_{1_t} and x_{2_t}, the solutions of ODEs, where

  $$\epsilon_1 \sim \mathcal{N}(0, \sigma_1^2) \text{ and } \epsilon_2 \sim \mathcal{N}(0, \sigma_2^2)$$

In both cases,

$$x_{3_t} = f_3(x_{2_t}) + \epsilon_3, \text{ where } \epsilon_3 \sim \mathcal{N}(0, \sigma_3^2)$$

Specifying Models by Using the FCMP Procedure

You must specify a model for every interval variable that is a child in the Bayesian network. Such a variable is related to its parents by using either a function or an ordinary differential equation (ODE). This function or ODE is the model of the child variable, expressed as a conditional distribution of the child variable for given values of parent variables. Different settings of nominal parent variables might require different models for the interval child variable as functions of its interval parents. You can use the FCMP procedure to specify the models as subroutines for these variables by using one of that procedure's supported programming languages.

You use subroutines to specify the models as follows:

- If the child variable is a single variable (that is, not an array variable):
  - If the child variable is a function of parent variables:
    - Name of the subroutine should be the name of the child variable.
    - First set of arguments are the parent variables of this child variable.
    - Next-to-last argument is the mean of the conditional distribution of the child variable computed by the subroutine, given the values of the parent variables.
    - Last argument is the model noise computed as the standard deviation by the subroutine.
  - If the model of the child variable is an ODE over the parent variables:

    * Name of the subroutine should be written as d_CHVARNAME, where CHVARNAME is the name of the child variable.
    * First set of arguments are the parent variables of this child variable.
    * Next-to-last argument is the derivative of the child variable with respect to time computed by the subroutine, given the values of the parent variables.
    * Last argument is the model noise computed as the standard deviation by the subroutine.

- If the child variable is an array of variables:
  - If the child variable array is a function of parent variables:
    * Name of the subroutine should be the name of the child variable array.
    * First set of arguments are the parent variables of this child variable to be identified by its index in the array in the next argument.
    * Second-to-last argument is the index of the child variable in the array.
    * Next-to-last argument is the mean of the conditional distribution of the child variable with the specified index computed by the subroutine, given the values of the parent variables.
    * Last argument is the model noise computed as the standard deviation by the subroutine.
  - If the model of the child variable array is an ODE over the parent variables:
    * Name of the subroutine should be written as d_CHVARNAME, where CHVARNAME is the name of the child variable array.
    * First set of arguments are the parent variables of this child variable array.
    * Next-to-last argument is the array of derivatives for the array of child variables with respect to time computed by the subroutine, given the values of the parent variables.
    * Last argument is the array of model noises for the array of child variables computed as the standard deviation by the subroutine.

The last argument that is related to model noise is optional and can be dropped from the subroutine. If you omit the last argument in a subroutine, then only one argument remains as the value that is computed by the subroutine. The remaining computed argument relates to either means or derivatives, depending on whether the model is a function or an ODE. In this case, you can code a function instead of a subroutine, and the function uses its return value to store means or derivatives instead of the argument that the subroutine uses.

Canonical Form of a Multivariate Normal Distribution

Processing of models at a child node in the two-time-period Bayesian network leads to multivariate normal distributions. The operations for generating updated beliefs at the end of the second time period in the network are streamlined using a canonical form of these distributions.

Let $\mathcal{N}(\boldsymbol{\mu}; \Sigma)$ be a multivariate normal distribution, where $\boldsymbol{\mu}$ is the vector of means for variables $\boldsymbol{X}$ and Σ is the variance-covariance matrix. The form of the distribution can be restated as

$$\frac{1}{(2\pi)^{n/2}|\Sigma|^{1/2}} \exp\left(-\frac{1}{2}(\boldsymbol{X}-\boldsymbol{\mu})^T \Sigma^{-1}(\boldsymbol{X}-\boldsymbol{\mu})\right)$$

$$= \exp\left(-\frac{1}{2}\boldsymbol{X}^T \Sigma^{-1}\boldsymbol{X} + \boldsymbol{\mu}^T \Sigma^{-1}\boldsymbol{X} - \frac{1}{2}\boldsymbol{\mu}^T \Sigma^{-1}\boldsymbol{\mu} - \log\left((2\pi)^{n/2}|\Sigma|^{1/2}\right)\right)$$

Define the terms K, $\boldsymbol{h}$, and g as follows:

$$K = \Sigma^{-1}$$

$$\boldsymbol{h} = \Sigma^{-1}\boldsymbol{\mu}$$

$$g = -\frac{1}{2}\boldsymbol{\mu}^T\Sigma^{-1}\boldsymbol{\mu} - \log\left((2\pi)^{n/2}|\Sigma|^{1/2}\right)$$

Then the normal distribution can be rewritten as

$$\exp\left(-\frac{1}{2}\boldsymbol{X}^T K\boldsymbol{X} + \boldsymbol{h}^T\boldsymbol{X} + g\right)$$

Stating the normal distribution in its canonical form, given by K, $\boldsymbol{h}$, and g, streamlines the operations in the processing of the Bayesian network, as discussed in the section "Filtering Algorithm" on page 150.

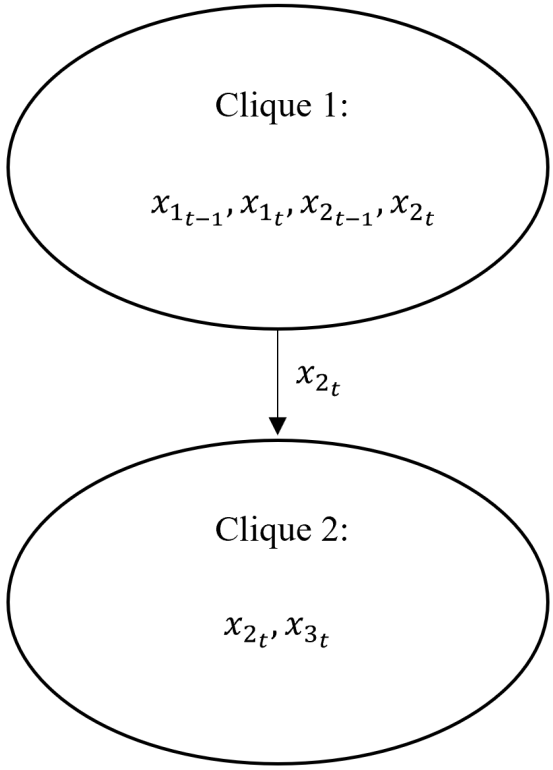
Reformulating a Bayesian Network as a Clique Tree

The process of updating the beliefs of hidden variables, given the previous beliefs of hidden variables and the current values of observed variables, requires an algorithm that propagates the incoming beliefs through the model and then adjusts the model-generated beliefs by using Bayesian logic to account for the observed values. You can streamline this process in a two-pass algorithm (Lauritzen and Spiegelhalter 1988) over the network by reformulating the Bayesian network as a clique tree. Cliques (Koller and Friedman 2009) in a clique tree are obtained by combining incoming beliefs and models. A clique combines incoming beliefs and models in such a way that the following statements are true:

- Each incoming belief and model belongs to one and only one clique.
- The clique tree satisfies the property that if a variable is in two cliques, then it is in all cliques on the path in the tree that connects those two cliques.

A clique tree is generated by the minimum-weight clique tree algorithm (Koller and Friedman 2009), which uses a heuristic to arrive at a tree that has small clique sizes, where the size of a clique is the square of the number of interval variables in the clique. The network shown in Figure 6.1 leads to the clique tree shown in Figure 6.2. This clique tree consists of two cliques, and a message that passes between the cliques corresponds to the common variable between the two cliques—namely, x_{2_t}.

Figure 6.2 Two-Period-Network Clique Tree



Approximating Nonlinear Models by Using Linear Models

The models that capture the evolution of variables over time can be nonlinear. You make a linear approximation to determine the multivariate normal distributions and the corresponding canonical forms over these variables. You can use the following approximation methods (Koller and Friedman 2009) for this purpose:

- extended linear transformation
- unscented linear transformation

Recall the nonlinear function that relates variable x_{1_t} to variables $x_{1_{t-1}}$ and $x_{2_{t-1}}$ in Figure 6.1:

$$x_{1_t} = f_1(x_{1_{t-1}}, x_{2_{t-1}}) + \epsilon_1, \text{ where } \epsilon_1 \sim \mathcal{N}(0, \sigma_1^2)$$

The linear approximation of this nonlinear function takes the form

$$x_{1_t} = b_{1_t} + w_{1_t} x_{1_{t-1}} + w_{2_t} x_{2_{t-1}} + \epsilon_1, \text{ where } \epsilon_1 \sim \mathcal{N}(0, \sigma_1^2)$$

In the extended linear transformation, w_{1_t} and w_{2_t} are estimated as point estimates of slopes at the means of $x_{1_{t-1}}$ and $x_{2_{t-1}}$. In the unscented linear transformation, w_{1_t} and w_{2_t} are estimated as weighted sums of point estimates of slopes at the means and other points that are selected around the means, as required by the unscented transformation (Koller and Friedman 2009).

The estimation of w_{1_t} and w_{2_t} is performed by either the extended or unscented method, relating to the evolution of $x_{1_{t-1}}$ and $x_{2_{t-1}}$ in accordance with the nonlinear function f_1.

In a similar way, you can linearize the ODE form by estimating w_{1_t} and w_{2_t}, using either the extended or unscented method, where the evolution of $x_{1_{t-1}}$ and $x_{2_{t-1}}$ to time t is computed by solving the related system of ODEs.

Filtering Algorithm

The updated beliefs of hidden variables at time period t are obtained by applying the filtering algorithm (Lauritzen and Spiegelhalter 1988) to the incoming beliefs from time period $t-1$ and the values of observed variables at time period t. This algorithm performs two passes over the cliques in the clique tree to produce updated beliefs. The cliques are ordered in such a way that clique i is not processed until the incoming messages to clique i have been generated by the parent cliques of clique i. The concept of factors is important in understanding these passes. A factor is a distribution or a conditional distribution over a set of variables. Here are a few examples of factors:

- incoming beliefs over interval variables from the previous time period
- conditional distribution of a child variable for given values of the parent variables
- distribution over a set of interval variables that are common between two adjacent cliques in a clique tree denoted as a message between cliques

The two passes are performed for each branch that is defined by the settings of the nominal variables.

Pass One: Process the Cliques in Ascending Order

1. Gather factors for creating a clique in the form of incoming beliefs, incoming messages, and models of child variables that are given by conditional distributions of child variables in this clique.
2. Perform the belief multiplication operation to combine the factors in this clique.
3. If the clique has one or more observed variables, then perform the instantiation operation for the observed values of these variables. Instantiation updates the probability of the branch for the given observed variables.
4. If the clique sends a message to a child clique, then perform the marginalization operation to produce the factor that represents the outgoing message.

Pass Two: Process the Cliques in Descending Order

1. Gather factors for returning messages from cliques that are children of this clique. For each returning message, perform the netting operation to produce the net message.
2. Perform the belief multiplication operation to combine the existing clique model from pass one with net messages.

3. If the clique received a message from a parent clique in pass one, then perform the marginalization operation to produce the factor for the returning message.

4. If the clique has an outgoing belief, then perform the marginalization operation to produce the factor that represents the belief for the relevant variable at time period t.

The following sections describe the operations that the algorithm uses.

Belief Multiplication

The $K, \boldsymbol{h}$, and g of all gathered factors in canonical forms are summed to obtain the result of belief multiplication, which is the canonical for the clique. Multiplication of beliefs in the distribution space is equivalent to summation in the canonical space.

For Clique 1 in Figure 6.2, there are incoming beliefs for $x_{1_{t-1}}$ and $x_{2_{t-1}}$, given by $(K_1, \boldsymbol{h}_1, g_1)$ and $(K_2, \boldsymbol{h}_2, g_2)$, respectively. There are models of child variables $x_{1_t} = f_1(x_{1_{t-1}}, x_{2_{t-1}})$ and $x_{2_t} = f_2(x_{1_{t-1}}, x_{2_{t-1}})$, given by $(K_3, \boldsymbol{h}_3, g_3)$ and $(K_4, \boldsymbol{h}_4, g_4)$, respectively. The multiplication operation determines the model for the clique as

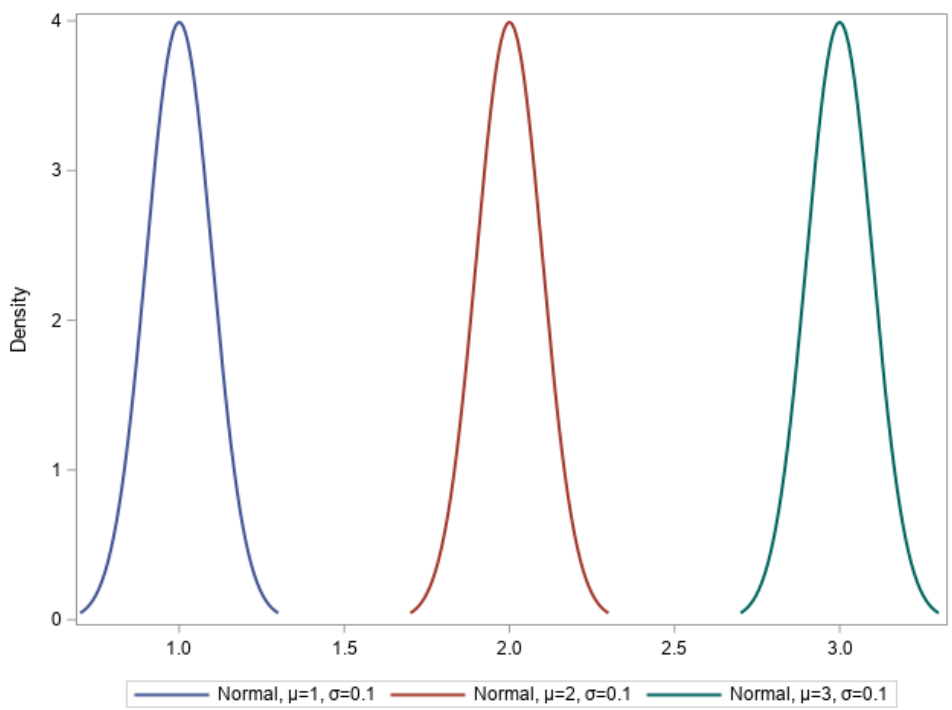
$$K = K_1 + K_2 + K_3 + K_4 \qquad \boldsymbol{h} = \boldsymbol{h}_1 + \boldsymbol{h}_2 + \boldsymbol{h}_3 + \boldsymbol{h}_4 \qquad g = g_1 + g_2 + g_3 + g_4$$

To illustrate belief multiplication, the following conditional linear Gaussian is used as the model of variable y given x:

$$y = x + \epsilon, \text{ where } \epsilon \sim \mathcal{N}(0, 0.1^2)$$

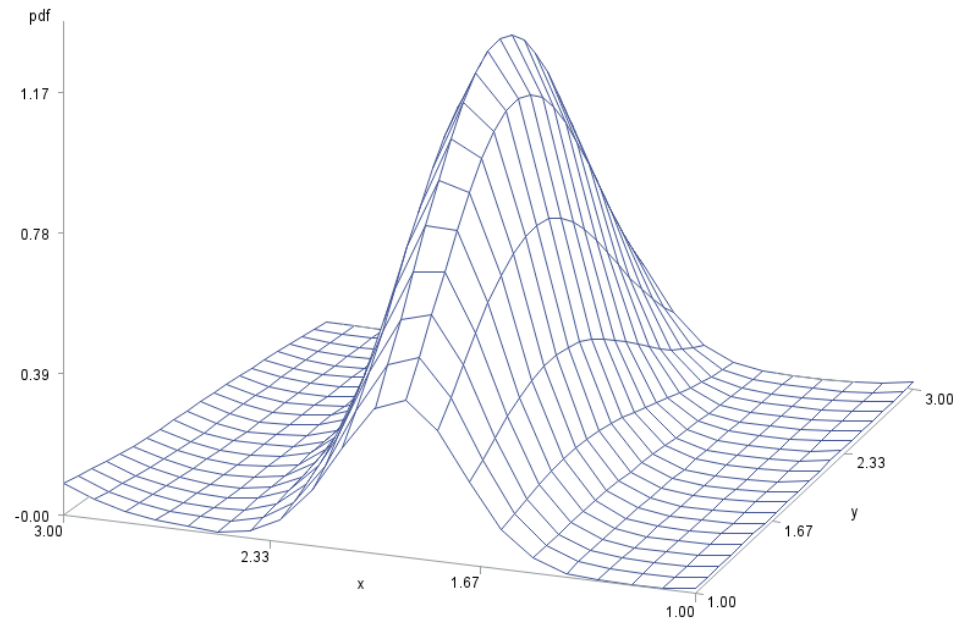
Figure 6.3 shows the distributions of y conditioned on three different x values, 1, 2, and 3.

Figure 6.3 Conditional Linear Gaussian Distributions of $y|x$



Let the incoming belief of x be given by $\mathcal{N}(2, 0.2^2)$. Multiplying the incoming belief of x by the belief of y, given as a conditional linear Gaussian of $y|x$, produces a joint normal distribution over x and y, as shown in Figure 6.4. Because the incoming belief of x is centered at 2, the bivariate normal distribution of $\{x, y\}$ is centered at (2,2).

Figure 6.4 Bivariate Normal Distribution of $\{x, y\}$



Marginalization

Let $\mathcal{C}(K, \boldsymbol{h}, g)$ be the canonical form over $\{\boldsymbol{X}, \boldsymbol{Y}\}$, where

$$K = \begin{bmatrix} K_{\boldsymbol{XX}} & K_{\boldsymbol{XY}} \\ K_{\boldsymbol{YX}} & K_{\boldsymbol{YY}} \end{bmatrix} \qquad \boldsymbol{h} = \begin{pmatrix} \boldsymbol{h_X} \\ \boldsymbol{h_Y} \end{pmatrix}$$

Marginalization of variables $\boldsymbol{Y}$ to obtain the model over $\boldsymbol{X}$ only is given as follows:

$$K' = K_{\boldsymbol{XX}} - K_{\boldsymbol{XY}} K_{\boldsymbol{YY}}^{-1} K_{\boldsymbol{YX}}$$

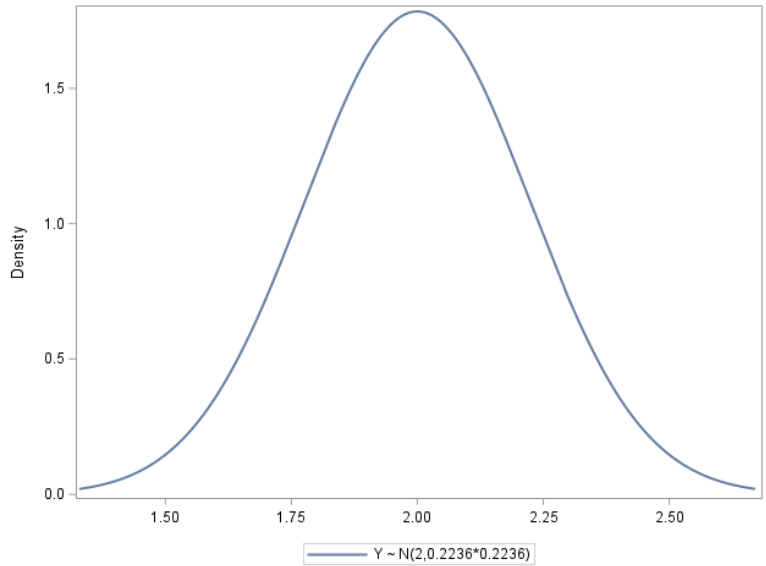
$$\boldsymbol{h}' = \boldsymbol{h_X} - K_{\boldsymbol{XY}} K_{\boldsymbol{YY}}^{-1} \boldsymbol{h_Y}$$

$$g' = g + \frac{1}{2}\left(\log\left|2\pi K_{\boldsymbol{YY}}^{-1}\right| + \boldsymbol{h}_{\boldsymbol{Y}}^{T} K_{\boldsymbol{YY}}^{-1} \boldsymbol{h_Y} \right)$$

For Clique 1 in Figure 6.2, the canonical $(K, \boldsymbol{h}, g)$ is defined over the variables $(x_{1_{t-1}}, x_{1_t}, x_{2_{t-1}}, x_{2_t})$. The first three variables are marginalized to produce the canonical for the outgoing message to Clique 2.

To illustrate marginalization, the bivariate normal distribution of $\{x, y\}$ that is obtained by belief multiplication is marginalized over x, as shown in Figure 6.5. The resulting belief in y is $\mathcal{N}(2, 0.2236^2)$.

Figure 6.5 Marginal Normal Distribution of y



Instantiation

Let $\mathcal{C}(K, \boldsymbol{h}, g)$ be the canonical form over $\{\boldsymbol{X}, \boldsymbol{Y}\}$, where

$$K = \begin{bmatrix} K_{\boldsymbol{XX}} & K_{\boldsymbol{XY}} \\ K_{\boldsymbol{YX}} & K_{\boldsymbol{YY}} \end{bmatrix} \qquad \boldsymbol{h} = \begin{pmatrix} \boldsymbol{h}_{\boldsymbol{X}} \\ \boldsymbol{h}_{\boldsymbol{Y}} \end{pmatrix}$$

The instantiation of $\boldsymbol{Y} = \boldsymbol{y}$ is given as follows:

$$K' = K_{\boldsymbol{XX}}$$

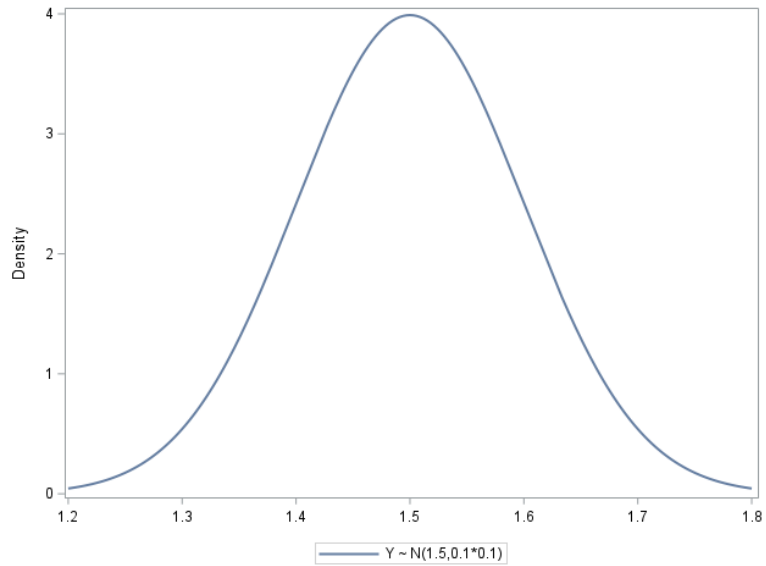
$$\boldsymbol{h}' = \boldsymbol{h}_{\boldsymbol{X}} - K_{\boldsymbol{XY}}\boldsymbol{y}$$

$$g' = g + \boldsymbol{h}_{\boldsymbol{Y}}^T\boldsymbol{y} - \frac{1}{2}\boldsymbol{y}^T K_{\boldsymbol{YY}}\boldsymbol{y}$$

For Clique 2 in Figure 6.2, x_{3_t} is instantiated to produce a canonical over x_{2_t}.

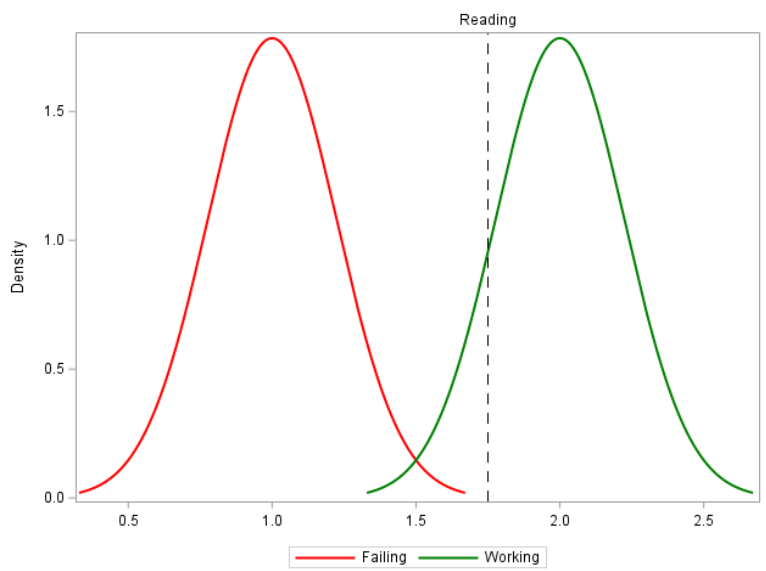
To illustrate instantiation, the bivariate normal distribution of $\{x, y\}$ that is obtained by belief multiplication is instantiated at $x = 1.5$, as shown in Figure 6.6. The resulting belief in y is $\mathcal{N}(1.5, 0.1^2)$.

Figure 6.6 Instantiated Normal Distribution of *y*



In the presence of multiple branches over the values of nominal variables, instantiation is performed on each branch. Figure 6.7 illustrates the marginal distribution of a variable that represents a sensor reading on two branches, which are associated with the Working and Failing branches of a nominal variable. The observed value of the sensor reading is represented by a vertical line. As the figure shows, the observed value is in the tail of the distribution of the Failing branch and agrees with the distribution of the Working branch. Instantiation leads to different distributions of the sensor reading on the two branches and assigns a high probability to the Working branch.

Figure 6.7 Marginal Distribution of Reading over Two Branches



Slowly Evolving Variables

In the case of variables that evolve slowly over time, it can be difficult to detect differences in the behavior of branches in only one time period. The filtering algorithm maintains beliefs by branches to evolve separately over multiple time periods. The number of these periods is controlled by values of the mean variable (frequency of synchronization for the nominal variable) that you specify in the input data table VARIABLESYNCPARMS. If probabilities of the branches have not evolved by more than the minimum threshold that you specify in the SYNCTHRESHOLD= option in the PROC DYNBNET statement within the specified synchronization frequency (mean variable in the input data table VARIABLESYNCPARMS), then branch probabilities are considered to be stable and beliefs are reconciled over the branches. If evolution of probabilities exceeds the specified threshold, then separate beliefs are maintained over branches to allow a clearer detection of branches that separate in probabilities over time.

Applying the Filtering Algorithm to a Simple Clique Tree

The flow of the filtering algorithm is outlined as follows, using the clique tree in Figure 6.2:

1. Belief multiplication is performed to combine the incoming beliefs of $x_{1_{t-1}}$ and $x_{2_{t-1}}$ with the models for x_{1_t} and x_{2_t}.
2. The Clique 1 model is marginalized to produce a message for x_{2_t} to go to Clique 2.
3. Belief multiplication is performed to combine the incoming message for x_{2_t} with the Clique 2 model that contains x_{3_t}.
4. The Clique 2 model is instantiated with the observed value of x_{3_t} to produce the returning message of x_{2_t}. The probability of the current branch is adjusted in response to this instantiation.
5. The netting operation is performed to adjust the returning message in light of the incoming message.
6. Belief multiplication is performed to combine the net message of x_{2_t} with the Clique 1 model.
7. Belief marginalization is performed to produce the outgoing beliefs in x_{1_t} and x_{2_t}.

Displayed Output

PROC DYNBNET produces the following output by default.

Filter Information

The “Filter Information” table contains the status of the filter run and the number of threads that are used.

ODS Table Names

PROC DYNBNET creates one ODS table named FilterInfo. You must use this name to refer to the table when you use ODS statements.

Example: DYNBNET Procedure

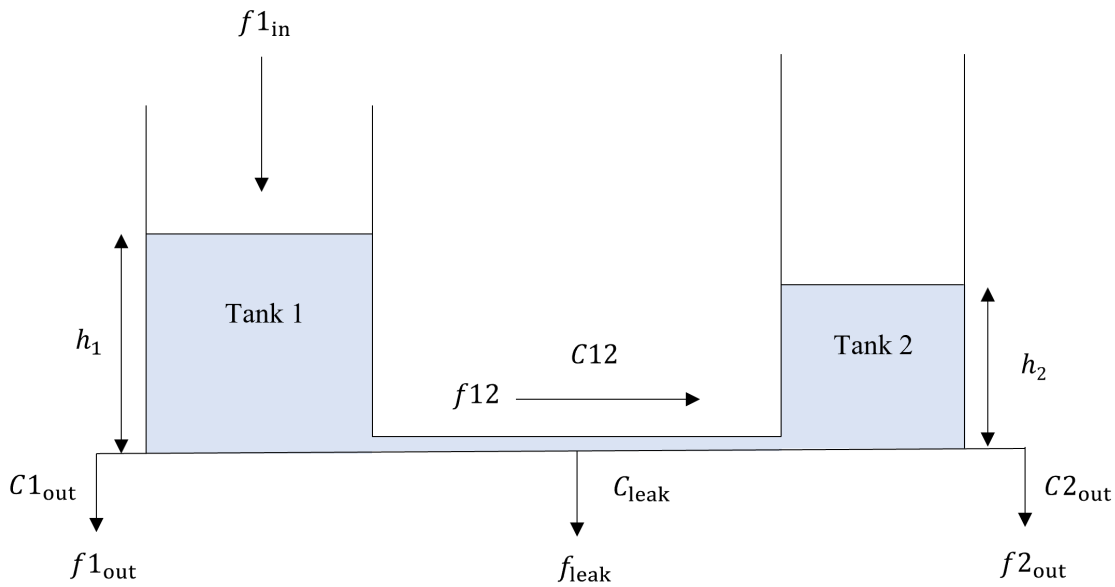
Example 6.1: Fault Detection in a System of Two Tanks

This example uses a system of two interconnected tanks to illustrate how to detect faults in the connecting pipes by using the DYNBNET procedure. The example (see Figure 6.8) is developed using a similar scenario of five tanks from Lerner et al. (2000) to illustrate the filtering algorithm. The numerical values are generated using a simulation model for this two-tank scenario. The following notation is used to build the model:

- A_1 = cross-sectional area of tank 1
- A_2 = cross-sectional area of tank 2
- h_1 = height of liquid in tank 1
- h_2 = height of liquid in tank 2
- $f1_{\text{in}}$ = flow rate of liquid coming into tank 1
- $f1_{\text{out}}$ = flow rate of liquid going out of tank 1
- $C1_{\text{out}}$ = conductance (reciprocal of resistance) of pipe going out of tank 1
- $C12$ = conductance of pipe going from tank 1 to tank 2
- $f12$ = flow rate of liquid going from tank 1 to tank 2
- C_{leak} = conductance of leak in pipe going from tank 1 to tank 2
- f_{leak} = flow rate of leak in pipe going from tank 1 to tank 2
- $C2_{\text{out}}$ = conductance of pipe going out of tank 2
- $f2_{\text{out}}$ = flow rate of liquid going out of tank 2

Adding the subscript t to a notation denotes its value at time period t.

Figure 6.8 Two-Tank Model



Flows are computed using these relationships:

$$f1_{out} = C1_{out} \times h_1$$

$$f12 = C12 \times (h_1 - h_2)$$

$$f_{leak} = C_{leak} \times h_1$$

$$f2_{out} = C2_{out} \times h_2$$

Material balance in tank 1 is represented by this relationship:

$$A_1 \times \frac{dh_1}{dt} = f1_{in} - f12 - f_{leak} - f1_{out}$$

Similarly, material balance in tank 2 is represented by the following relationship:

$$A_2 \times \frac{dh_2}{dt} = f12 - f2_{out}$$

Rearranging terms in these equations gives the following derivatives for ODEs:

$$\frac{dh_1}{dt} = \frac{1}{A_1}\left[f1_{in} - (C12 + C_{leak} + C1_{out})h_1 + (C12)h_2\right]$$

$$\frac{dh_2}{dt} = \frac{1}{A_2}\left[(C12)h_1 - (C12 + C2_{out})h_2\right]$$

A block that develops over time in the pipe that connects tank 1 to tank 2 is specified as

$$\frac{dC12}{dt} = -block_rate$$

where $block_rate$ is the rate at which the block is developing. A leak that develops over time in the pipe that connects tank 1 to tank 2 is specified as

$$\frac{dC_{\text{leak}}}{dt} = leak_rate$$

where $leak_rate$ is the rate at which the leak is developing. The following branches are used to consider behavior under different scenarios for the pipe that connects tank 1 to tank 2:

- L1Working: no block or leak in the pipe
- L2Blocked: a sudden block appearing in the pipe that reduces the conductance to half of normal conductance
- L3BwDrift: a gradual block appearing in the pipe
- L4LwDrift: a gradual leak appearing in the pipe
- L5B_LwD: a sudden block with a gradual leak appearing in the pipe
- L6BwD_LwD: a gradual block with a gradual leak appearing in the pipe

The two-time-period Bayesian network for the two-tank model in Figure 6.8 is shown in Figure 6.9, where $f1_{\text{outRd}_t}$ and $f2_{\text{outRd}_t}$ are observed variables at time t. The models for $f1_{\text{outRd}_t}$ and $f2_{\text{outRd}_t}$ are as follows:

$$f1_{\text{outRd}_t} = C1_{\text{out}} \times h_{1_t} + \epsilon_1, \text{ where } \epsilon_1 \sim \mathcal{N}(0, \sigma_1^2) \text{ and } \sigma_1^2 \text{ is the variance in the sensor to observe } f1_{\text{out}}$$

$$f2_{\text{outRd}_t} = C2_{\text{out}} \times h_{2_t} + \epsilon_2, \text{ where } \epsilon_2 \sim \mathcal{N}(0, \sigma_2^2) \text{ and } \sigma_2^2 \text{ is the variance in the sensor to observe } f2_{\text{out}}$$

The clique tree for the Bayesian network in Figure 6.9 is shown in Figure 6.10.

Figure 6.9 Two-Period Tank Model

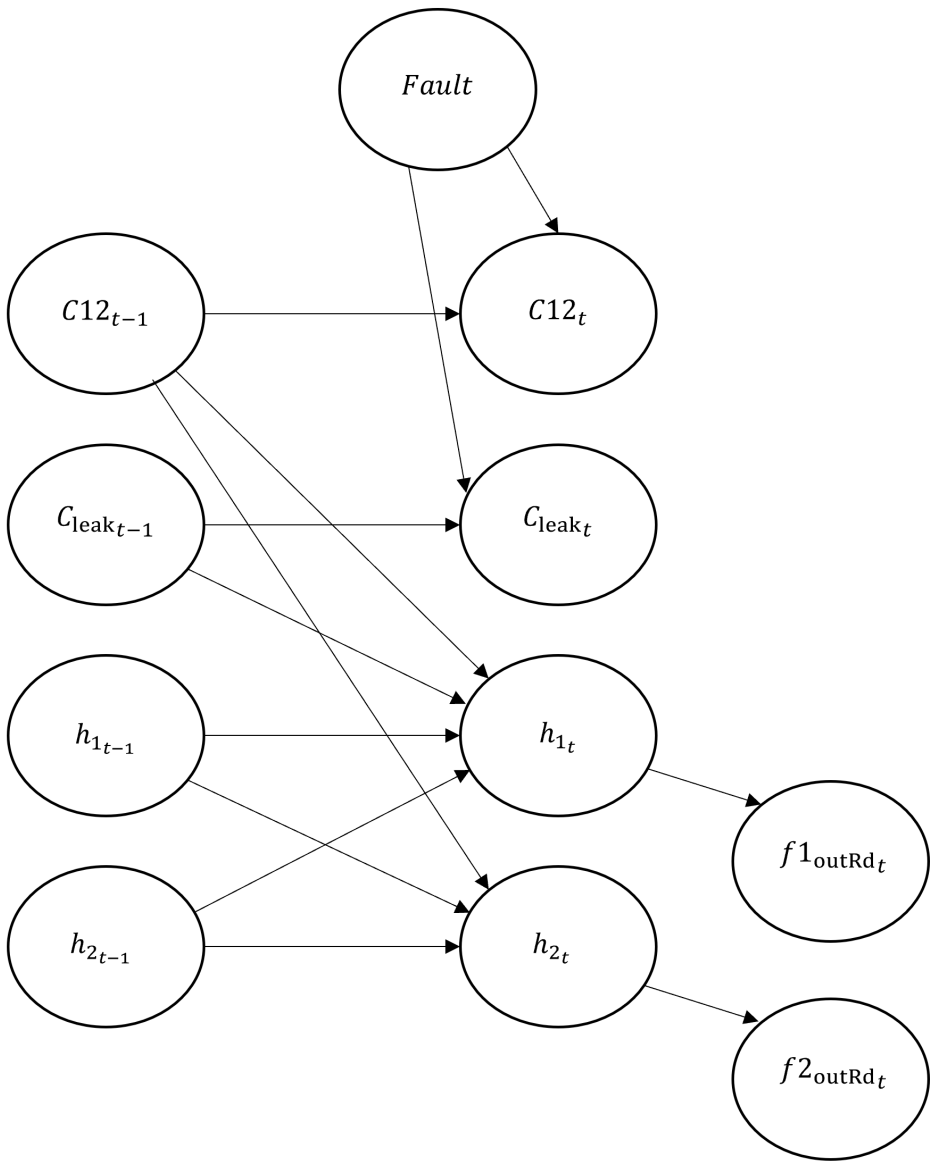
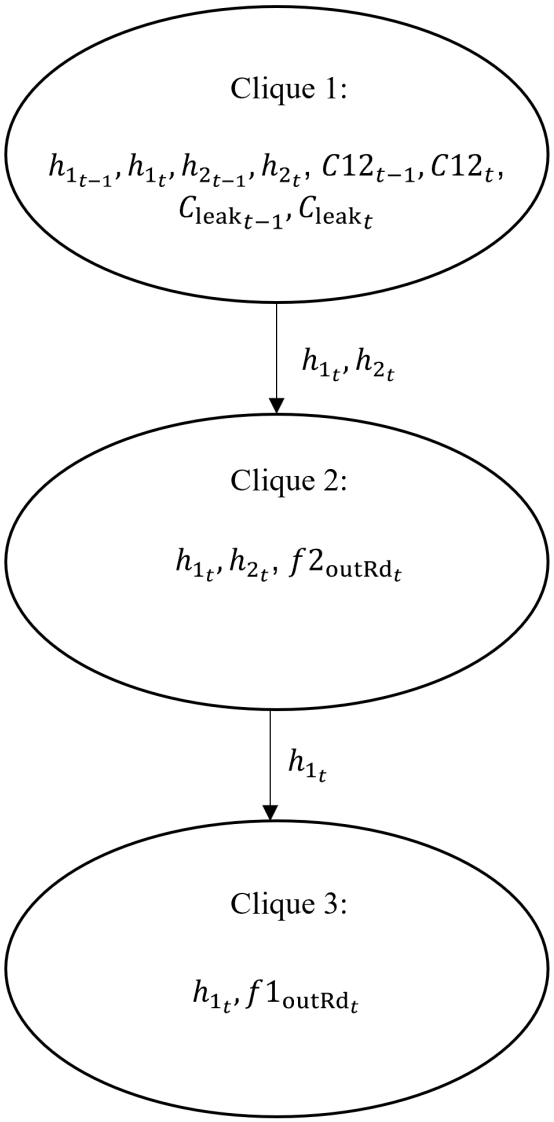


Figure 6.10 Two-Tank Clique Tree



PROC DYNBNET requires multiple input data tables. The following statements assume that your SAS libref is named mylib, but you can substitute any appropriately defined SAS libref.

The variables in the two-time-period Bayesian network and their roles are defined in the mylib.variableRoles data table as follows:

```
data mylib.variableRoles;
   length varname $20;
   input  varname $        varrole $;
datalines;
Time                TIMEID
H1                  HIDDEN
H2                  HIDDEN
C12                 HIDDEN
```

```
Cleak                HIDDEN
Fault                HIDDEN
F1out_rd             OBSERVED
F2out_rd             OBSERVED
;
```

Levels of nominal variables are defined in the mylib.variableLevels data table as follows:

```
data mylib.variableLevels;
   length varname $20;
   length varlevel $20;
   input  varname $   varlevel $;
datalines;
Fault           L1Working
Fault           L2Blocked
Fault           L3BwDrift
Fault           L4LwDrift
Fault           L5B_LwD
Fault           L6BwD_LwD
;
```

The parent-child relationships in the two-time-period Bayesian network are defined using the mylib.links data table as follows, where stage defines the parent to be in the current time period (1) or previous time period (2):

```
data mylib.links;
   length parent $20 child $20;
   input  parent $ child $ stage;
datalines;
Time            C12         1
C12             C12         2
Fault           C12         1
Time            Cleak       1
Cleak           Cleak       2
Fault           Cleak       1
Time            H1          1
C12             H1          2
Cleak           H1          2
H1              H1          2
H2              H1          2
Time            H2          1
C12             H2          2
H1              H2          2
H2              H2          2
H1              F1out_rd    1
H2              F2out_rd    1
;
```

The systems of ordinary differential equations in the two-time-period Bayesian network are defined using the mylib.variableOdes data table as follows:

```
data mylib.variableOdes;
   length varname $20;
   input  varname $  system;
datalines;
```

```
C12         1
Cleak       1
H1          1
H2          1
;
```

The initial beliefs that come from the previous time period are defined in the mylib.initialBeliefs data table as follows. For nominal variables, incoming probabilities are defined for the levels. For interval variables, the incoming mean and standard deviation are optionally defined by levels of nominal variables.

```
data mylib.initialBeliefs;
   length intervalvar $20   nominalvar $20    varlevel $20;
   input  Time    intervalvar $    mean    std        nominalvar $   varlevel $;
datalines;
1       H1         20               0.001           .               .
1       H2          2               0.001           .               .
1       C12         1               0.0001          .               .
1       C12         0.5             0.0001          Fault           L2Blocked
1       C12         0.5             0.0001          Fault           L5B_LwD
1       Cleak       0               0.0001          .               .
1       .           0.95             .              Fault           L1Working
1       .           0.01             .              Fault           L2Blocked
1       .           0.01             .              Fault           L3BwDrift
1       .           0.01             .              Fault           L4LwDrift
1       .           0.01             .              Fault           L5B_LwD
1       .           0.01             .              Fault           L6BwD_LwD
;
```

The observations in the mylib.tank_data data table, shown in the following code, are for the two observed variables; they cover the time periods from 1 to 10. Time period 1 corresponds to the maximum time period in the mylib.initialBeliefs data table. Hence, the future observations are from time periods 2 to 10. PROC DYNBNET generates outgoing beliefs for time periods 2 to 10, given the input beliefs from time period 1 and the observations from time periods 2 to 10. If a model of any child variable uses a parent that is an observed variable from the previous time period, then the procedure also uses the observed values from time period 1.

```
data mylib.tank_data;
   input Time f1out_rd f2out_rd;
datalines;
1       10.000        2.000
2        9.955        2.069
3        9.910        2.137
4        9.866        2.203
5        9.822        2.268
6        9.779        2.332
7        9.736        2.394
8        9.692        2.455
9        9.650        2.514
10       9.607        2.572
;
```

You need to use the FCMP procedure to define the models in the two-time-period Bayesian network.

Next, you need to invoke PROC DYNBNET as follows:

```
proc dynbnet data = mylib.tank_data
           linapprox = EXTENDED
           odeapprox = NONE
           outdetails = mylib.tank_outd
           outputstate = mylib.tank_state;
     varroles data = mylib.variableRoles;
     varlevels data = mylib.variableLevels;
     links data = mylib.links;
     varodes data = mylib.variableOdes;
     initbeliefs data = mylib.initialBeliefs;
     output out = mylib.tank_out;
run;
```

You need to invoke the ASTORE procedure as follows to save the state in a file on disk so it can be accessed as input in future invocations of PROC DYNBNET:

```
proc astore;
     download rstore=mylib.tank_state store='\Documents\tank_state';
run;
```

The mylib.tank_data data table contains values of observed variables every minute from $t = 2$ to $t = 10$ minutes. The data are obtained by simulating the fault for the branch L2Blocked, where the fault appears at $t = 2$ minutes and remains. Models are linearized using the extended approximation method when you specify LINAPPROX=EXTENDED, and ODEs are not approximated when you specify ODEAPPROX=NONE.

The incoming beliefs specify L1Working (branch 1) to have a probability of 0.95 and all other branches to have a probability of 0.01.

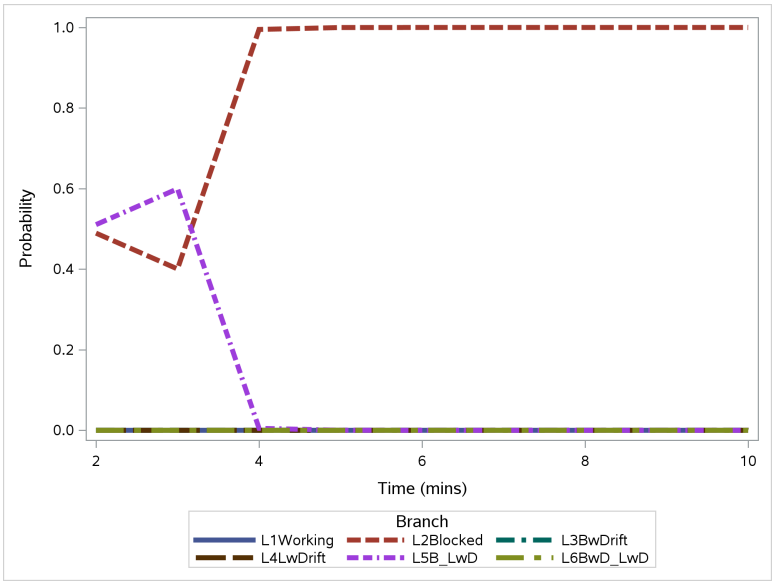
The following code displays the probabilities of branches over time:

```
data tank_out;
   set mylib.tank_out;
by Time;
Branch = "L1Working";
prob = Fault_L1Working;
output;
Branch = "L2Blocked";
prob = Fault_L2Blocked;
output;
Branch = "L3BwDrift";
prob = Fault_L3BwDrift;
output;
Branch = "L4LwDrift";
prob = Fault_L4LwDrift;
output;
Branch = "L5B_LwD";
prob = Fault_L5B_LwD;
output;
Branch = "L6BwD_LwD";
prob = Fault_L6BwD_LwD;
output;
run;
```

```
ods graphics on / attrpriority=none;
proc sgplot data = tank_out;
   series x = Time y = prob / group = Branch lineattrs=(thickness=4);
   xaxis label = "Time (mins)";
   yaxis label = "Probability";
run;
```

Output 6.1.1 shows the results.

Output 6.1.1 Probabilities of Fault Branches



At $t = 3$ minutes, the probability of L2Blocked (branch 2) is 0.40 and the probability of L5B_Lwd (branch 5) is 0.60. A division of probabilities between L2Blocked and L5B_Lwd is explained by the fact that L5B_Lwd also has a block in the pipe that connects tank 1 to tank 2. At $t = 4$ minutes, the probability of L2Blocked is 0.97 and the probability of L5B_Lwd is 0.03. The higher probability of L2Blocked is explained by the fact that the observations of the variables $f1_{\text{out}}$ and $f2_{\text{out}}$ at time $t = 4$ minutes are closer to the L2Blocked branch than to the L5B_Lwd branch.

Example 6.2: Use of an Existing Model for Fault Detection

This example shows how you can use the DYNBNET procedure with a previously saved model to detect faults in new observations. The following data table contains the new observations from time periods 11 to 20:

```
data mylib.tank_data_2;
   input Time f1out_rd f2out_rd;
datalines;
11        9.565        2.629
12        9.523        2.685
13        9.482        2.739
14        9.440        2.792
15        9.399        2.844
16        9.358        2.895
17        9.318        2.944
18        9.277        2.993
19        9.237        3.040
20        9.198        3.086
;
```

You need to invoke the ASTORE procedure as follows to upload the saved state from a file on disk so it can be accessed as input by PROC DYNBNET:

```
proc astore;
     upload rstore=mylib.tank_state_in store='\Documents\tank_state';
run;
```

Finally, you invoke PROC DYNBNET as follows:

```
proc dynbnet data = mylib.tank_data_2
           inputstate = mylib.tank_state_in;
     output out = mylib.tank_out_2;
run;
```

References

Koller, D., and Friedman, N. (2009). *Probabilistic Graphical Models: Principles and Techniques*. Cambridge, MA: MIT Press.

Lauritzen, S. L., and Spiegelhalter, D. J. (1988). "Local Computations with Probabilities on Graphical Structures and Their Application to Expert Systems." *Journal of the Royal Statistical Society, Series B* 50:157–224.

Lerner, U., Parr, R., Koller, D., and Biswas, G. (2000). "Bayesian Fault Detection and Diagnosis in Dynamic Systems." In *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence*, 531–537. New York: AAAI Press.

Chapter 7
FACTMAC Procedure

Contents

Overview: FACTMAC Procedure . . . **168**
- PROC FACTMAC Features . . . 168
- Using SAS Viya Workbench . . . 169

Getting Started: FACTMAC Procedure . . . **169**

Syntax: FACTMAC Procedure . . . **172**
- PROC FACTMAC Statement . . . 172
- AUTOTUNE Statement . . . 174
- CODE Statement . . . 178
- CROSSVALIDATION Statement . . . 178
- DISPLAY Statement . . . 178
- DISPLAYOUT Statement . . . 179
- FREQ Statement . . . 180
- ID Statement . . . 180
- INPUT Statement . . . 180
- OUTPUT Statement . . . 181
- PARTITION Statement . . . 182
- SAVESTATE Statement . . . 182
- TARGET Statement . . . 183
- WEIGHT Statement . . . 183

Details: FACTMAC Procedure . . . **183**
- *k*-fold Cross Validation . . . 184
- Displayed Output . . . 184
- ODS Table Names . . . 185
- Output Data Tables . . . 187

Examples: FACTMAC Procedure . . . **187**
- Example 7.1: Running PROC FACTMAC with the MovieLens Data Set . . . 187
- Example 7.2: Tuning a Factorization Machine Model . . . 189

References . . . **192**

Overview: FACTMAC Procedure

The FACTMAC procedure implements the factorization machine model in SAS Viya Workbench. The flexible factorization machine model has applications in predictive modeling and recommendation (Rendle 2012). Factorization machines generalize matrix factorization, among other techniques. You can use the FACTMAC procedure to perform factorization in parallel by making full use of multicore computers computing environments.

The FACTMAC procedure estimates factors for each of the nominal input variables you specify, in addition to estimating a global bias and a bias for each level of those nominal input variables. You also specify an interval target variable. The procedure computes the biases and factors by using the stochastic gradient descent (SGD) algorithm, which minimizes the root mean square error (RMSE) criterion on the input data table that you provide. In this method, each iteration attempts to reduce the RMSE. The SGD algorithm proceeds until the maximum number of iterations is reached.

PROC FACTMAC stores the results of the factorization an output data table, which is produced by the OUTMODEL= option. This data table contains the factors in addition to the global bias and the biases for all the levels of the input variables, in addition to the factors. The corresponding level names are listed for ease of reference. The biases and factors are used for scoring.

PROC FACTMAC Features

PROC FACTMAC enables you to use parallel execution for factorization.

The following list summarizes the basic features of PROC FACTMAC:

- is multithreaded
- learns a factorization machine model based on a parallel implementation of the SGD optimization algorithm

Using SAS Viya Workbench

SAS Viya Workbench has greatly simplified the SAS programming experience by offering a syntax style nearly identical to that of SAS 9 and earlier releases. However, there are minor exceptions. In the past, SAS Viya procedures required the use of a CAS engine libref. This engine served multiple purposes, but it was primarily used to upload data to the CAS server and manage the data. Its additional purposes included support of new data types and improved performance for large data sets. No CAS server is needed for SAS Viya Workbench, because all computations are done in place for you in a single location.

In the following program, the LIBNAME statement creates a mylib libref that the examples in this chapter use to refer to your current or desired work directory:

```
libname mylib ".";
```

For more information, see *SAS Viya Workbench: Getting Started*.

Getting Started: FACTMAC Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

This example shows how to use the FACTMAC procedure to learn a factorization machine model from observations in a SAS data table. This example uses the cars data set in the Sashelp library and illustrates the prediction of gas mileage of cars based on make and model. The analysis uses four variables: car make, car model, car type, and a variable named mpg_city, which measures the car's fuel usage (in miles per gallon) for city driving. The remaining variables in the data table are not used.

You can load the cars data set into your SAS library in the following DATA step:

```
data mylib.cars;
   set sashelp.cars;
run;
```

These statements assume that your SAS libref is named mylib, as in the section "Using SAS Viya Workbench" on page 169, but you can substitute any appropriately defined SAS libref.

The following statements run PROC FACTMAC and output the results to ODS tables:

```
proc factmac data=mylib.cars outmodel=mylib.factors maxiter=50
             nfactors=5  learnstep=0.002;
   input make model type /level=nominal;
   target mpg_city  /level=interval;
   output out=mylib.score_out copyvars=(make model type mpg_city);
run;

proc print data=mylib.factors(obs=48);
run;
```

The NFACTORS= option requests that the model estimate five factors, the LEARNSTEP= option sets the optimization learning step to 0.002, the MAXITER= option requests that the optimization stop after 50

iterations, and the OUTMODEL= option requests that the model parameters be written to the mylib.factors data table. The INPUT statement specifies that the make, model, and type variables are to be used as nominal inputs. The TARGET statement specifies that mpg_city is the target variable to be predicted. The OUTPUT statement requests that the predictions be written to the data table mylib.score_out and that the make, model, type and mpg_city variables be copied from the mylib.cars data table to the mylib.score_out data table.

Figure 7.1 shows the global bias, the bias values for each level, and the list of the factors for the first 48 observations.

Figure 7.1 Bias Values and Factors

Obs	Variable	Level	Bias	Factor1	Factor2	Factor3	Factor4	Factor5
1	_GLOBAL_		20.0607	0.00000	0.00000	0.00000	0.00000	0.00000
2	Make	Acura	-0.6322	-1.65189	-2.34213	0.09553	-4.06551	0.72563
3	Make	Audi	-1.5871	-0.23131	0.07828	-0.35298	-0.45827	-0.20166
4	Make	BMW	-1.3607	0.00777	0.05455	-0.19466	-0.07272	0.22275
5	Make	Buick	-1.1719	2.96493	0.10815	3.14048	-4.74293	-0.28692
6	Make	Cadillac	-3.5607	-1.07033	-0.99187	1.49699	0.58245	0.95669
7	Make	Chevrolet	-0.3941	0.01216	0.00688	0.05321	0.00923	-0.03507
8	Make	Chrysler	-0.1941	0.56168	-0.90681	-0.77899	-1.38498	1.48220
9	Make	Dodge	-0.6761	0.53603	0.05044	0.03614	0.79867	0.46522
10	Make	Ford	-0.7999	-0.29532	-0.16454	0.05845	-0.27550	0.10528
11	Make	GMC	-4.6857	1.03171	-1.97310	-0.91069	-4.54418	2.00952
12	Make	Honda	7.7628	0.84902	-0.54073	-0.37149	-0.87898	-0.31334
13	Make	Hummer	-10.0607	-3.26652	2.06044	-3.31811	-1.32709	-6.03068
14	Make	Hyundai	2.9393	0.30075	0.72757	-0.78057	0.01371	-0.13113
15	Make	Infiniti	-2.8107	-1.20270	2.23762	6.27459	-3.97131	0.51608
16	Make	Isuzu	-4.0607	6.31966	-4.61052	2.32465	-5.98882	-2.33495
17	Make	Jaguar	-2.5607	-2.18175	-1.36395	-1.72034	2.88573	1.22202
18	Make	Jeep	-2.7274	1.34136	-4.71543	5.68740	0.25748	-5.30474
19	Make	Kia	1.8483	-5.30755	3.53864	-4.75619	5.24446	2.55098
20	Make	Land Rover	-6.0607	-5.98657	1.73501	3.16050	0.16469	-3.89031
21	Make	Lexus	-2.6062	-0.97068	1.83440	1.61661	0.77815	0.44383
22	Make	Lincoln	-3.2830	1.32684	0.16185	-0.68375	-0.66196	1.17094
23	Make	MINI	6.4393	-0.18316	1.88514	-0.79716	-2.62557	5.78783
24	Make	Mazda	1.3938	-0.91171	1.95718	-1.13241	0.70124	0.06483
25	Make	Mercedes-Benz	-2.7146	-0.16738	0.07799	-0.20517	-0.25945	-0.08443
26	Make	Mercury	-2.5052	0.93393	2.25607	0.17357	0.25786	1.27973
27	Make	Mitsubishi	0.8623	-0.24432	0.35635	0.16703	0.32479	-0.02810
28	Make	Nissan	-0.3549	-0.00132	-0.02561	0.00751	0.00230	-0.02652
29	Make	Oldsmobile	0.9393	0.21073	3.49856	5.92790	-6.28979	2.92394
30	Make	Pontiac	0.4847	-0.13061	-1.70502	-1.35086	1.14599	1.41535
31	Make	Porsche	-2.6322	-4.90259	4.56269	4.18555	0.66543	-4.09505
32	Make	Saab	0.3678	-0.18666	0.00714	0.16181	0.80439	0.61889
33	Make	Saturn	4.3143	1.90340	-3.38020	3.73003	0.31785	-0.34106
34	Make	Scion	11.4393	6.31009	-6.22994	6.26170	-1.74481	5.77012
35	Make	Subaru	0.2120	0.48494	-0.36550	-0.41698	-0.49696	1.03908
36	Make	Suzuki	2.0643	-1.72773	1.93579	-0.77099	-0.01766	-1.50695
37	Make	Toyota	4.3678	0.29650	-0.08574	-0.13681	0.20477	-0.54623
38	Make	Volkswagen	1.3393	-0.05236	0.70879	-1.49191	-1.33924	0.19614
39	Make	Volvo	-0.3107	0.91585	-0.20479	-0.64416	-0.23166	0.74650
40	Model	3.5 RL 4dr	-2.0607	4.29008	6.32127	6.32609	-6.32203	-4.13052
41	Model	3.5 RL w/Navigation 4dr	-2.0607	-2.53358	1.07033	6.32244	4.64185	0.88841
42	Model	300M 4dr	-2.0607	6.33258	6.32102	-1.46105	6.32544	6.32405
43	Model	300M Special Edition 4dr	-2.0607	-6.33347	4.20158	-6.32730	-0.68946	4.77917
44	Model	325Ci 2dr	-0.0607	0.75687	-6.32171	6.32257	-6.32459	6.32309
45	Model	325Ci convertible 2dr	-1.0607	6.33216	-4.54598	-0.49266	6.32496	6.32270
46	Model	325i 4dr	-0.0607	-0.11359	4.98979	2.21139	5.50177	6.32416
47	Model	325xi 4dr	-1.0607	-6.32539	1.49304	5.52169	-6.13325	6.32568
48	Model	325xi Sport	-1.0607	6.32095	-5.18676	-0.45827	-4.11622	-3.12254

Syntax: FACTMAC Procedure

The following statements are available in the FACTMAC procedure:

```
PROC FACTMAC <options> ;
    AUTOTUNE <options> ;
    CODE FILE=filename ;
    CROSSVALIDATION <options> ;
    DISPLAY <table-list> </ options> ;
    DISPLAYOUT table-spec-list </ options> ;
    FREQ variable ;
    ID variables ;
    INPUT variables <LEVEL=INTERVAL | NOMINAL> ;
    OUTPUT OUT=libref.data-table <options> ;
    PARTITION partition-option ;
    SAVESTATE RSTORE=libref.data-table ;
    TARGET variable <LEVEL=INTERVAL> ;
    WEIGHT variable ;
```

The PROC FACTMAC statement, an INPUT statement, and the TARGET statement are required. You can specify multiple INPUT statements.

The following sections describe the PROC FACTMAC statement and then describe the other statements in alphabetical order.

PROC FACTMAC Statement

```
PROC FACTMAC <options> ;
```

The PROC FACTMAC statement invokes the procedure. Table 7.1 summarizes the *options* available in this statement.

Table 7.1 PROC FACTMAC Statement Options

Option	Description
Input Data Table Options	
DATA=	Specifies the input data table
Factorization Options	
LEARNSTEP=	Specifies the learning step size for the SGD algorithm
MAXITER=	Specifies the maximum number of iterations
NFACTORS=	Specifies the number of factors to estimate for the model
NONNEGATIVE	Requests nonnegative factorization
NTHREADS=	Specifies the number of threads to use
OUTMODEL=	Specifies the output model data table to contain the computed factor parameters
SEED=	Specifies the seed to be used for pseudorandom number generation
Output Options	
NOPRINT=	Suppresses ODS output

You can specify the following *options*:

DATA=*libref.data-table*

names the input data table for PROC FACTMAC to use. The default is the most recently created data table. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to the data. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 169.

data-table specifies the name of the input data table.

LEARNSTEP=*number*

specifies the learning step size for the stochastic gradient descent (SGD) algorithm, where *number* is a positive real number. The learning step size controls the amount by which the factors are updated at each iteration.

By default, LEARNSTEP=0.001. This value can be tuned with the AUTOTUNE statement.

MAXITER=*number*

specifies the maximum number of iterations for the algorithm to perform, where *number* is an integer greater than or equal to 1. In each iteration of the SGD method, the factors are recomputed.

By default, MAXITER=1. This value can be tuned with the AUTOTUNE statement.

NFACTORS=*number*

specifies the number of factors to estimate for the model, where *number* is an integer greater than or equal to 1.

By default, NFACTORS=1. This value can be tuned with the AUTOTUNE statement.

NONNEGATIVE
performs nonnegative factorization, in which the estimated factors are greater than or equal to 0 and the estimated biases are 0.

By default, nonnegative factorization is not performed.

NOPRINT
suppresses ODS output.

NTHREADS=*number-of-threads*
specifies the number of threads to use for the computation, where *number-of-threads* is an integer from 1 to 64, inclusive. The default value is the maximum number of available threads.

OUTMODEL=*libref.data-table*
specifies the output model data table to contain the computed factor parameters.

SEED=*random-seed*
specifies an integer that is used to start the pseudorandom number generator. This option enables you to reproduce the same sample output, but only when NTHREADS=1. If you do not specify a seed or you specify a value less than or equal to 0, the seed is generated from reading the time of day from the computer's clock.

By default, SEED=0.

AUTOTUNE Statement

AUTOTUNE < *options* > **;**

The AUTOTUNE statement searches for the best combination of values of the NFACTORS=, LEARNSTEP=, and MAXITER= options in the PROC FACTMAC statement. You cannot specify both the OUTPUT and AUTOTUNE statements in the same run of PROC FACTMAC. You cannot specify both the AUTOTUNE statement and the CROSSVALIDATION statement in the same run of PROC FACTMAC.

Table 7.2 summarizes the *options* that you can specify in the AUTOTUNE statement. For more information about all options except the TUNINGPARAMETERS= option, see the option's description in the section "AUTOTUNE Statement" on page 6 in Chapter 2, "Shared Concepts." The TUNINGPARAMETERS= option is described following Table 7.2.

Table 7.2 AUTOTUNE Statement Options

Option	Description
APPENDLOOKUP	Specifies that the table specified in the HISTORYTABLE= option contain the rows from the table specified in the LOOKUPTABLE= option
CONSTRAINTS=	Specifies the constraints on the output metrics
EVALHISTORY=	Specifies how to report the evaluation history of the tuner
FOLDCOLUMN=	Specifies the column in the data table in which the cross validation fold value is indicated
FRACTION=	Specifies the fraction of observations to use for validation

Table 7.2 *continued*

Option	Description
HISTORYTABLE=	Specifies the table that contains the evaluation history
KFOLD=	Specifies the number of folds for *k*-fold cross validation
LIVEUPDATE	Specifies that the table specified in the HISTORYTABLE= option be updated at every evaluation
LOOKUPTABLE=	Specifies the table to use for evaluation lookup
MAXBAYES=	Specifies the maximum number of points in the kriging model
MAXEVALS=	Specifies the maximum number of evaluations
MAXITER=	Specifies the maximum number of iterations when SEARCHMETHOD=GA or SEARCHMETHOD=BAYESIAN
MAXTIME=	Specifies the maximum time for all iterations
MAXTRAINTIME=	Specifies the maximum time for a model training
NCONVITER=	Specifies the number of convergence iterations
NOGRIDSHUFFLE	Requests that the grid points not be shuffled
NOLOCALSEARCH	Disables local search optimization
NPARALLEL=	Specifies the number of parallel sessions
NSUBSESSIONWORKERS=	Specifies the number of workers in parallel sessions
OBJECTIVE=	Specifies the objective function
POPSIZE=	Specifies the population size when SEARCHMETHOD=GA or SEARCHMETHOD=BAYESIAN
SAMPLESIZE=	Specifies the sample size when SEARCHMETHOD=LHS or SEARCHMETHOD=RANDOM
SEARCHMETHOD=	Specifies the search method that the optimizer uses
SECONDOBJECTIVE=	Specifies the second objective to use for tuning
SELECTINITPOINT	Specifies that the tuner select the best evaluation from the lookup table
TARGETEVENT=	Specifies the target event for ROC-based calculations
TRAINFRACTION=	Specifies the fraction of observations to use for training
TUNINGPARAMETERS=	Specifies the custom tuning parameters
USEPARAMETERS=	Specifies how to handle the TUNINGPARAMETERS= option

TUNINGPARAMETERS=(*suboption* | **...** | **<** *suboption* **>)**

TUNEPARMS=(*suboption* | **...** | **<** *suboption* **>)**

specifies which parameters to tune and which ranges to tune over. If USEPARAMETERS=STANDARD, this option is ignored.

You can specify one or more of the following *suboptions*:

NFACTORS (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**

specifies information about the number of factors to use for tuning the factorization machine model. For more information, see the NFACTORS= option in the PROC FACTMAC statement.

You can specify the following additional suboptions:

LB=*number*
specifies the minimum number of factors to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=5.

UB=*number*
specifies the maximum number of factors to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=30.

VALUES=*value-list*
specifies a list of values to consider for the number of factors during tuning, where *value-list* is a space-separated list of integer numbers greater than or equal to 1. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial number of factors for the tuner to use.

By default, INIT=5.

EXCLUDE
excludes the number of factors from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

LEARNSTEP (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**
specifies information about the learning step to use for tuning the factorization machine model. For more information, see the LEARNSTEP= option in the PROC FACTMAC statement.

You can specify the following additional suboptions:

LB=*number*
specifies the minimum learning step to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=0.001.

UB=*number*
specifies the maximum learning step to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=1.

VALUES=*value-list*
specifies a list of learning steps to consider during tuning, where *value-list* is a space-separated list of numbers greater than 0. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial learning step for the tuner to use.

By default, INIT=0.001.

EXCLUDE
excludes the learning step from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

MAXITER (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**
specifies information about the maximum number of iterations to use for tuning the factorization machine model. For more information, see the MAXITER= option in the PROC FACTMAC statement.

You can specify the following additional suboptions:

LB=*number*
specifies the minimum number of iterations to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=10.

UB=*number*
specifies the maximum number of iterations to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=200.

VALUES=*value-list*
specifies a list of numbers of trees to consider during tuning, where *value-list* is a space-separated list of positive integers. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial number of iterations for the tuner to use.

By default, INIT=30.

EXCLUDE
excludes the number of iterations from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

CODE Statement

CODE FILE=*filename* **;**

The CODE statement generates SAS DATA step code that mimics the computations that are performed. The generated SAS DATA step code can be used for scoring new observations. Only one CODE statement is processed. If you specify multiple CODE statements, only the first one is used.

You must specify the following option:

FILE=*filename*
specifies the name of the file to write the SAS score code to.

The CODE statement is optional. If you do not include a CODE statement, no score code is generated.

CROSSVALIDATION Statement

CROSSVALIDATION < *options* > **;**

The CROSSVALIDATION statement performs *k*-fold cross validation to find the average estimated validation error. You cannot specify this statement if you specify the PARTITION statement.

You can specify the following *options*:

KFOLD=*number*
specifies the number of partition folds in the cross validation process, where *number* is between 2 and 20, inclusive. By default, KFOLD=5.

NOPARALLEL
specifies that *k*-fold cross validation not be run in parallel. By default, the process runs in parallel.

DISPLAY Statement

DISPLAY < *table-list* > < / *options* > **;**

The DISPLAY statement enables you to specify a list of display tables to display or exclude. This statement is similar to the ODS SELECT, ODS EXCLUDE, and ODS TRACE statements. However, the DISPLAY statement can improve performance when a large number of tables could be generated (such as in BY-group processing). The DISPLAY statement produces only the requested tables, but ODS statements generate all the tables and then keep the requested tables.

If you use both DISPLAY and ODS statements together, the DISPLAY statement takes precedence over the ODS statements. Note that the ODS EXCLUDE statement processes tables after they have been filtered by the DISPLAY statement. In some cases, it might appear that the ODS EXCLUDE statement is taking precedence because it can further filter the tables. For more information about ODS, see *SAS Output Delivery System: Procedures Guide*.

You can specify the *table-list* as a list of table names, paths, partial pathnames, and regular expressions.

The table names that you can specify are listed in the section "ODS Table Names" on page 185. A path is a table name that is prefixed with dot-separated grouping information. For example, a SelectionSummary table that a procedure produces during a selection routine might have the path *Bygroup1.Summary.SelectionSummary*. A partial pathname does not include all groups; for example, *SelectionSummary* and *Summary.SelectionSummary* are partial pathnames for *Bygroup1.Summary.SelectionSummary*.

When you specify a table name or partial pathname, all display tables whose paths end in the specified name are selected for display or exclusion. For example, both *SelectionSummary* and *Summary.SelectionSummary* select *Bygroup1.Summary.SelectionSummary*.

A regular expression is enclosed in forward slashes (/). For example, specifying "/tions/" selects all pathnames that contain the substring "tions"; in particular, the *Bygroup1.Summary.SelectionSummary* table is selected. Specifying "!/tions/" selects all pathnames that do not contain the substring "tions"; in particular, the *Bygroup1.Summary.SelectionSummary* table is not selected.

You can specify the following *options* after a slash (/):

CASESENSITIVE
performs a case-sensitive comparison of table names in the *table-list* to display table names when tables are subsetted for display. To preserve case, you must enclose table names in the *table-list* in quotation marks.

EXCLUDE
displays all display tables except those that you specify in the *table-list*.

EXCLUDEALL
suppresses display of all tables. This option takes precedence over the other options.

TRACE
displays the display table names, labels, and paths.

DISPLAYOUT Statement

DISPLAYOUT *table-spec-list* < / *options* > **;**

The DISPLAYOUT statement enables you to create output tables from your displayed output. This statement is similar to the ODS OUTPUT statement. For more information about ODS, see *SAS Output Delivery System: Procedures Guide*.

The *table-spec-list* specifies a list of output tables to create. Each entry in the list has either a *key*=*value* format or a *key* format:

key=*value* specifies *key* as the ODS table name, path, or partial pathname, and specifies *value* as the output table name.

key specifies *key* as the ODS table name and also as the output table name.

The ODS table names that you can specify are listed in the section "ODS Table Names" on page 185. You cannot specify the ODS table named OutputCasTables in the *table-spec-list*.

Table names and partial pathnames are discussed under the DISPLAY statement. The DISPLAYOUT statement does not support regular expressions.

You can specify the following *options* after a slash (/):

INCLUDEALL
creates output tables for all display tables. The name of the created output table is the same as the name of the corresponding display table. If you specify this option, the *table-spec-list* specification is ignored.

NOREPLACE
does not replace any existing output table of the same name.

FREQ Statement

FREQ *variable* **;**

The *variable* in the FREQ statement identifies a numeric variable in the input data table that contains the frequency of occurrence of each observation. If the value of the *variable* for an observation is not an integer, it is truncated to an integer; if the value is less than 1 or missing, the observation is not used in the analysis. When the FREQ statement is omitted, each observation is assigned a frequency of 1.

ID Statement

ID *variables* **;**

The ID statement lists one or more variables that are to be copied from the input data table to the output data tables that are specified in the OUT= option in the OUTPUT statement and the RSTORE= option in the SAVESTATE statement.

INPUT Statement

INPUT *variables* < **LEVEL=INTERVAL | NOMINAL** > **;**

The INPUT statement specifies the names of the *variables* to be used in the factorization. It names one or more input variables that use common options. If you want to use different options for different variables, you can specify multiple INPUT statements.

You can include the following option in each INPUT statement:

LEVEL=INTERVAL | NOMINAL
specifies the level of measurement of the *variables*. You can specify the following values:

INTERVAL specifies that the level of measurement of the *variables* is interval.

NOMINAL specifies that the level of measurement of the *variables* is nominal.

By default, LEVEL=INTERVAL for numeric variables and LEVEL=NOMINAL for categorical variables.

You must specify at least two nominal input variables. You can also specify any number of interval input variables.

OUTPUT Statement

OUTPUT OUT=*libref.data-table* < *options* > **;**

The OUTPUT statement creates an output data table to contain the results of the procedure run. You cannot specify both the OUTPUT and AUTOTUNE statements in the same run of PROC FACTMAC.

You must specify the following *option*:

OUT=*libref.data-table*
names the output data table for PROC FACTMAC to use. You must specify this option before any other options. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to where the data table is to be stored. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 169.

data-table specifies the name of the output data table.

You can also specify the following *options*:

COPYVAR=*variable*
COPYVARS=(*variables***)**
lists one or more *variables* from the input data table to be transferred to the output data table.

ROLE< =*variable* **>**
specifies a numeric *variable* that indicates the role that each observation plays in fitting the model. The default name is _ROLE_. Table 7.3 shows how this *variable* is interpreted for each observation.

Table 7.3 Role Interpretation

Value	Observation Role
0	Not used
1	Training
2	Validation
3	Testing

If you do not partition the input data by specifying a PARTITION statement, then the role variable value is 1 for all observations.

PARTITION Statement

PARTITION *partition-option* **;**

The PARTITION statement specifies how observations in the input data set are logically partitioned into disjoint subsets for model training, validation, and testing. For more information, see the section "Using Validation and Test Data" on page 23 in Chapter 2, "Shared Concepts." Either you can designate a variable in the input data table and a set of formatted values of that variable to determine the role of each observation, or you can specify proportions to use for randomly assigning observations to each role.

You must specify exactly one of the following *partition-options*:

FRACTION(< TEST=*fraction* **> < VALIDATE=***fraction* **> < SEED=***number* **>)**
randomly assigns specified proportions of the observations in the input data table to the roles. You specify the proportions for testing and validation by using the TEST= and VALIDATE= suboptions. If you specify both the TEST= and VALIDATE= suboptions, then the sum of the specified fractions must be less than 1 and the remaining fraction is the proportion of the observations assigned to the training role. The SEED= option specifies an integer that is used to start the pseudorandom number generator for random partitioning of data for training, testing, and validation. If you do not specify SEED=*number* or if *number* is less than or equal to 0, the seed is generated by reading the time of day from the computer's clock.

ROLE=*variable* **(< TEST='***value***' > < TRAIN='***value***' > < VALIDATE='***value***' >)**
ROLEVAR=*variable* **(< TEST='***value***' > < TRAIN='***value***' > < VALIDATE='***value***' >)**
names the *variable* in the input data table whose values are used to assign roles to each observation. This *variable* cannot also appear as an analysis variable in other statements or options. The TEST=, TRAIN=, and VALIDATE= suboptions specify the formatted values of this variable that are used to assign observation roles. If you do not specify the TRAIN= suboption, then all observations whose role is not determined by the TEST= or VALIDATE= suboption are assigned to the training role.

SAVESTATE Statement

SAVESTATE RSTORE=*libref.data-table* **;**

The SAVESTATE statement creates an analytic store for the model and saves it as a binary object in a data table. You can use the analytic store in the ASTORE procedure to score new data. For more information, see Chapter 4, "ASTORE Procedure."

You must specify the following option:

RSTORE=*libref.data-table*
specifies a data table in which to save the analytic store for the model.

TARGET Statement

TARGET *variable* < **LEVEL=INTERVAL** > **;**

The TARGET statement names the target variable whose values PROC FACTMAC predicts. The target must be interval and must be different from the variables in the INPUT statement. You can include the following option in the OUTPUT statement:

LEVEL=INTERVAL
specifies the level of measurement of the variables.

PROC FACTMAC currently accepts only interval target variables.

WEIGHT Statement

WEIGHT *variable* **;**

The *variable* in the WEIGHT statement is used as a weight to perform a weighted analysis of the data. Observations that have nonpositive or missing weights are not included in the analysis. When the WEIGHT statement is omitted, all observations that are used in the analysis are assigned a weight of 1.

Details: FACTMAC Procedure

The factorization machines model is defined as

$$\hat{y}(\mathbf{x}) = w_0 + \sum_{j=1}^{p} w_j x_j + \sum_{j=1}^{p} \sum_{j'=j+1}^{p} x_j x_{j'} \sum_{f=1}^{k} v_{jf} v_{j'f}$$

where $\mathbf{x} = (x_1, \ldots, x_p)$ is an observed p-dimensional input feature vector, $\hat{y}$ is the predicted target, w_0 is a global bias, w_j are per-feature biases, and v_{jf} denotes coordinate f of the vector $\mathbf{v}_j \in \mathbb{R}^k$. The overall factor matrix $\mathbf{V} \in \mathbb{R}^{p \times k}$ is the concatenation of the row vectors $\mathbf{v}_j$ for $j = 1, \ldots, p$. The number of factors is k. PROC FACTMAC estimates the model parameters $w_0, w_1, \ldots, w_p$ and $\mathbf{V}$. The estimation is done by minimizing the root mean square error (RMSE), which is defined by

$$\text{RMSE} = \sqrt{\frac{1}{n} \sum_{i=1}^{n} (\hat{y}_i - y_i)^2}$$

over the training set, subject to the max-norm regularization constraint

$$\|\mathbf{v}_j\|_\infty < B, j = 1, \ldots, p$$

The optimization uses a projected-gradient version of the stochastic gradient descent (SGD) algorithm. The constant B is automatically set, based on the range of the input and target variables.

The results of running PROC FACTMAC are reproducible only if you specify a value greater than 0 for the SEED= option and specify NTHREADS=1, because PROC FACTMAC uses a threaded SGD solver that purposefully uses shared memory without locks. The variability between runs is nevertheless expected to be small.

k-fold Cross Validation

The CROSSVALIDATION statement performs *k*-fold cross validation to assess the accuracy of a model. During cross validation, all data are divided into *k* subsets (folds), where *k* is the value of the KFOLD= option. For each fold, a new model is trained on the (*k*–1) folds and then validated using the selected (holdout) fold. The assessment metrics are then averaged across all the holdout folds. The CROSSVALIDATION statement creates a table that has *k*+1 rows. The first *k* rows contain the assessment metrics for each holdout fold, and the last row contains the average across all the holdout folds. Which metrics the table includes depends on whether the target is nominal or interval. For nominal targets, the table includes the values of square error, mean consequential error, and multiclass log loss. For interval targets, the table includes the values of square error, absolute error, and square logarithmic error.

Displayed Output

The FACTMAC procedure displays various tables that are related to the factorization. The following sections describe the output tables in the order of their appearance when the related options are specified.

Model Information

The “Model Information” table displays basic information about the parameters that are used in the factorization analysis. This information includes the maximum number of iterations, learning step, number of factors, and seed value.

Number of Observations

The “Number of Observations” table displays the number of observations that are read from the input data table and used.

Iteration History

The “Iteration History” table displays the iteration history and approximate loss when the variables that are specified in the INPUT statement are interval.

The “displayed loss” is an approximation for computational efficiency reasons. The final exact loss is shown in the “Final Exact Loss” table.

Final Exact Loss

The “Final Exact Loss” table displays the actual, exact mean square error (MSE) and the root mean square error (RMSE) of the learned factorization machines model solution, which are computed on the training data.

Interval Variables

The “Interval Variables” table shows the mean and the standard deviation for the interval variables.

OutputCasTables Table

The OutputCasTables table is a special table that has information about each table that is created. The information for each table consists of the table name, the library in which the table resides, and the number of columns and rows in the table. Because this table is not a typical ODS table that contains analytical results, you cannot include it in the *table-spec-list* in the DISPLAYOUT statement.

ODS Table Names

Each table created by the FACTMAC procedure has a name associated with it, and you must use this name to refer to the table when you use ODS statements. The names of each table and a short description of the contents are listed in Table 7.4.

Table 7.4 ODS Tables Produced by PROC FACTMAC

Table Name	Description	Statement	Option
BestConfiguration	Hyperparameters and objective function values for the best configuration	AUTOTUNE	Default
CrossValidateMLFitStat	Per-fold and average assessment metrics of *k*-fold cross validation	CROSSVALIDATION	Default
DescStatsInt	Descriptive statistics for interval variables	PROC FACTMAC	Default
EvaluationHistory	Values of the hyperparameters and the objective function for all configurations	AUTOTUNE	Default
FinalLoss	Final exact loss	PROC FACTMAC	Default
HyperparameterImportance	Relative importance of hyperparameters	AUTOTUNE	Default
IterationHistory	Tuner iteration history	AUTOTUNE	Default
ModelInfo	Model information	PROC FACTMAC	Default
NObs	Number of observations	PROC FACTMAC	Default
OptIterHistory	Iteration history	PROC FACTMAC	Default
OutputCASTables	See the section "OutputCasTables Table" on page 185	PROC FACTMAC	Default
TunerCasOutputTables	Tables created during tuning	AUTOTUNE	Default
TunerInfo	Setup values that are used by the tuner	AUTOTUNE	Default
TunerResults	Values of the hyperparameters, the objective function for the default configuration (iteration 0), and up to 10 best configurations found	AUTOTUNE	Default
TunerSummary	Statistics about the tuning process	AUTOTUNE	Default
TunerTiming	Total time spent on different tasks during tuning	AUTOTUNE	Default

Output Data Tables

The FACTMAC procedure creates a data table to which it writes the global biases and the factors. You specify the name of this data table in the OUTMODEL= option in the PROC FACTMAC statement. Details about the data table are listed in Table 7.5.

Table 7.5 Output Data Table Produced by PROC FACTMAC

Data Table	Description
FACTORS	Lists the global bias, the name of each input variable, each level and the values of the estimated factors

Examples: FACTMAC Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

Example 7.1: Running PROC FACTMAC with the MovieLens Data Set

This example draws on data that are derived from companies that provide movies for online viewing. A company wants to offer its customers recommendations of movies that they might like. These recommendations are based on ratings that are provided by users. The MovieLens data set was developed by the GroupLens project at the University of Minnesota and is available at `http://grouplens.org/datasets/movielens`. This example uses the MovieLens 100K version.

There are four columns in the MovieLens 100K data set: user ID, item ID (each item is a movie), timestamp, and rating. This example predicts the rating for a specified user ID and an item ID. The data set is very sparse because most combinations of users and movies are not rated.

You can download the compressed archive file from the website at `http://files.grouplens.org/datasets/movielens/ml-100k.zip` and use any third-party unzip tool to extract all the files in the archive to the destination directory of your choice.[1] The file that contains the ratings is *u.data*. The following statements load the data table into your SAS library:

[1]Disclaimer: SAS may reference other websites or content or resources for use at Customer's sole discretion. SAS has no control over any websites or resources that are provided by companies or persons other than SAS. Customer acknowledges and agrees that SAS is not responsible for the availability or use of any such external sites or resources, and does not endorse any advertising, products, or other materials on or available from such websites or resources. Customer acknowledges and agrees that SAS is not liable for any loss or damage that may be incurred by Customer or its end users as a result of the availability or use of those external sites or resources, or as a result of any reliance placed by Customer or its end users on the completeness, accuracy, or existence of any advertising, products, or other materials on, or available from, such websites or resources.

```
proc import
    datafile="u.data"
    /*or other user-defined location*/
    out=mylib.movlens(rename=(var1=userid var2=itemid
        var3=rating var4=timestamp))
    dbms=tab replace ;
    getnames=no;
run;
```

The following statements show how to use PROC FACTMAC to predict movie ratings:

```
proc factmac data=mylib.movlens nfactors=10 learnstep=0.15
             maxiter=20  outmodel=mylib.factors;

   input userid itemid /level=nominal;
   target rating /level=interval;
   output out=mylib.out1 copyvars=(userid itemid rating);
run;
```

The following statements print the first 10 observations in the mylib.factors data table, which is specified in the OUTMODEL= option in the PROC FACTMAC statement. The output is shown in Output 7.1.1.

```
proc print data=mylib.factors(obs=10);
run;
```

Output 7.1.1 Bias Values and Factors

Obs	Variable	Level	Bias	Factor1	Factor2	Factor3	Factor4	Factor5	Factor6	Factor7	Factor8	Factor9	Factor10
1	_GLOBAL_		3.52986	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
2	userid	1	0.08043	0.14034	0.36798	-0.52760	-0.31465	0.22486	-0.28397	-0.87780	0.24885	0.02000	-0.14646
3	userid	2	0.17982	0.54746	0.00647	-0.47194	-0.39421	-0.19759	0.51829	-0.10209	0.01729	0.26124	-0.12959
4	userid	3	-0.73356	-0.54006	0.01826	0.31498	0.11264	0.73452	0.31893	-0.06499	-0.63882	-0.45544	0.18285
5	userid	4	0.80347	0.26031	0.04870	-0.25062	-0.01312	-0.29526	0.53290	-0.58693	0.00283	0.36615	0.40131
6	userid	5	-0.65557	0.51211	0.07824	-0.08614	-0.01463	0.46066	-0.30982	-0.21790	0.37157	-1.06146	-0.29942
7	userid	6	0.10521	0.17515	0.18334	-0.51516	0.53364	-0.55709	-0.16770	-0.05254	0.39754	0.29666	0.23568
8	userid	7	0.43540	0.25194	0.07348	-0.05654	-0.02345	-0.24364	0.14093	-0.03300	-0.46717	0.51100	-0.19197
9	userid	8	0.26675	0.11208	0.57564	0.02860	-0.77657	-0.16036	-0.41215	-0.06069	0.76389	-0.00608	-0.11334
10	userid	9	0.74287	0.18856	-0.10608	0.22230	-1.01906	-0.18707	-0.08458	-0.02147	-0.82411	-0.21252	-0.09411

The following statements print the predicted movie ratings for the first 20 observations, as shown in Output 7.1.2.

```
proc print data=mylib.out1(obs=20);
run;
```

Output 7.1.2 Predicted Movie Ratings

Obs	userid	itemid	rating	P_rating
1	196	242	3	4.09834
2	186	302	3	3.78284
3	22	377	1	1.42463
4	244	51	2	3.20907
5	166	346	1	2.58391
6	298	474	4	4.60470
7	115	265	2	3.43183
8	253	465	5	4.57718
9	305	451	3	2.96174
10	6	86	3	4.44866
11	62	257	2	3.08217
12	286	1014	5	3.08536
13	200	222	5	4.46180
14	210	40	3	3.30407
15	224	29	3	3.15151
16	303	785	3	3.03463
17	122	387	5	4.47396
18	194	274	2	2.57941
19	291	1042	4	3.47518
20	234	1184	2	1.62934

Example 7.2: Tuning a Factorization Machine Model

This example illustrates how you can use the AUTOTUNE statement to automatically tune the hyperparameters of a factorization machine model from observations in the MovieLens data table. The data table is described in Example 7.1.

You can download the compressed archive file from the website at `http://files.grouplens.org/datasets/movielens/ml-100k.zip` and use any third-party unzip tool to extract all the files in the archive to the destination directory of your choice.[2] The file that contains the ratings is *u.data*.

The following statements load the data table into your SAS library:

```
proc import
    datafile  = "/my/data/u.data" /* full path to the data file */
    out       = mylib.movlens(rename=(var1=userid var2=itemid
                var3=rating var4=timestamp))
    dbms      = tab replace ;
    getnames  = no;
run;
```

[2]Disclaimer: SAS may reference other websites or content or resources for use at Customer's sole discretion. SAS has no control over any websites or resources that are provided by companies or persons other than SAS. Customer acknowledges and agrees that SAS is not responsible for the availability or use of any such external sites or resources, and does not endorse any advertising, products, or other materials on or available from such websites or resources. Customer acknowledges and agrees that SAS is not liable for any loss or damage that may be incurred by Customer or its end users as a result of the availability or use of those external sites or resources, or as a result of any reliance placed by Customer or its end users on the completeness, accuracy, or existence of any advertising, products, or other materials on, or available from, such websites or resources.

The following statements show how you can use the FACTMAC procedure with the AUTOTUNE statement to automatically tune the hyperparameters of a factorization machine model that is trained on the mylib.movlens data table:

```
proc factmac
   data=mylib.movlens outmodel=mylib.factors;
   input userid itemid /level=nominal;
   target rating /level=interval;
   output out=mylib.out1 copyvars=(userid itemid rating);
   autotune
      /* Tuning Parameters
      You do not need to specify any tuning parameters for the default
      tuning process. If you want to make adjustments to the default
      tuning process, uncomment the following block of code and change
      any of the tuning parameters' attributes.

      tuningParameters=(
         NFACTORS   ( values=5 10 15 20 25 30                 init=5    )
         MAXITER    ( values=10 20 30 40 50 60 70 80 90 100
                      110 120 130 140 150 160 170 180 190 200 init=30   )
         LEARNSTEP  ( values=1E-6 1E-5 1E-4 1E-3 1E-2 1E-1 1  init=1E-3 )
      )
      */
   ;
   /* Remove this line to see all results */
   ods select BestConfiguration EvaluationHistoryPlot IterationHistoryPlot;
run;
```

The preceding statements produce the table and the plots shown in Output 7.2.1 through Output 7.2.3. The table in Output 7.2.1 displays the evaluation number, the values of the tuning parameters, and the error metric value for the best factorization machine model that the tuner found. Note that the ODS SELECT statement limits the displayed results to a single table and two plots. You can remove this statement to display all tables and plots. For the full list of ODS tables that PROC FACTMAC produces, see Table 7.4.

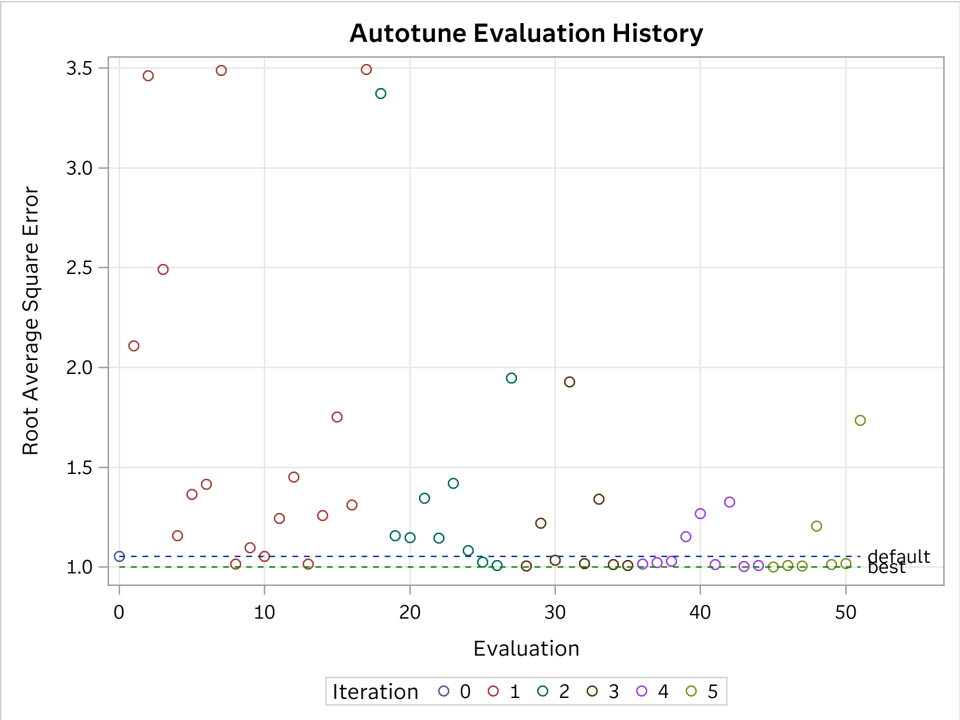
Output 7.2.1 Best Configuration Table

The FACTMAC Procedure

Best Configuration	
Evaluation	45
Number of Factors	5
Maximum Number of Iterations	50
Learning Step Size	1
Root Average Square Error	1.0016766983

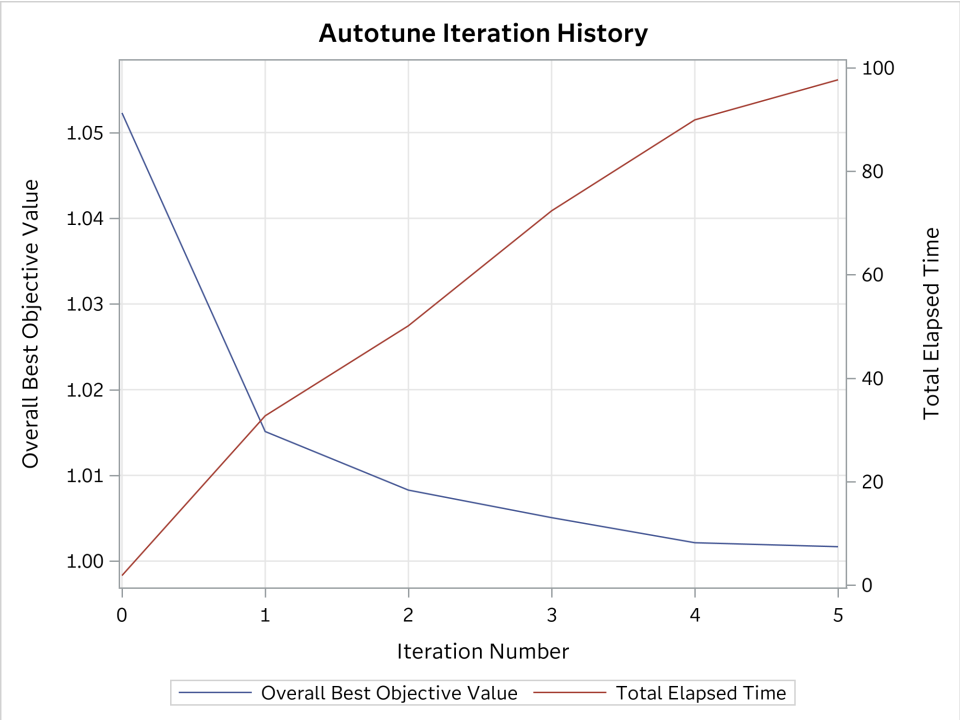
Output 7.2.2 displays a scatter plot of all configurations that the tuner tried. The objective values are shown on the Y axis, and the evaluation numbers are shown on the X axis.

Output 7.2.2 Evaluation History Plot



The plot in Output 7.2.3 displays how the best found objective value and the elapsed time changed with each iteration of the tuner.

Output 7.2.3 Iteration History Plot



References

Rendle, S. (2012). "Factorization Machines with libFM." *ACM Transactions on Intelligent Systems and Technology* 3:1–22.

Chapter 8
FASTKNN Procedure

Contents

Overview: FASTKNN Procedure **193**
PROC FASTKNN Features 194
Using SAS Viya Workbench 194
Getting Started: FASTKNN Procedure **194**
Syntax: FASTKNN Procedure **195**
PROC FASTKNN Statement 196
DISPLAY Statement 199
DISPLAYOUT Statement 200
ID Statement 200
INPUT Statement 201
OUTPUT Statement 201
Details: FASTKNN Procedure **202**
Displayed Output 203
ODS Table Names 203
Output Data Tables 204

Overview: FASTKNN Procedure

The FASTKNN procedure implements the *k*-nearest neighbor (*k*-NN) search algorithm in SAS Viya Workbench. The *k*-NN algorithm has numerous applications, including recommendation systems, image search, fingerprint recognition, and clustering. You can use this procedure to perform search in parallel by making full use of multicore computers computing environments.

The FASTKNN procedure operates on a query data table and an input data table. For each observation in the query data table, it returns the *k* observations in the input data table that are nearest. The procedure uses the Euclidean distance metric to compute distances between the input and query observations, and it stores the results of the search in the output data table that is specified by the OUTPUT statement. PROC FASTKNN can use exact search or approximate search. Approximate search is faster, but it is not guaranteed to return all *k* neighbors. In addition, the procedure stores the distances between the query and input observations in the output data table that is specified by the OUTDIST= option in the PROC FASTKNN statement. You can request that the procedure exclude distances that are greater than a specified threshold value. You can also control the amount of distance information that is output by using the USETOPKOUTDIST option, which restricts the output to only the top *k* nearest distances. By default, all distances are output to the OUTDIST= option table.

PROC FASTKNN Features

PROC FASTKNN enables you to use parallel execution for *k*-NN search.

The following list summarizes the basic features of PROC FASTKNN:

- is multithreaded
- returns *k*-nearest neighbors based on a parallel implementation of the *k*-NN search algorithm

Using SAS Viya Workbench

SAS Viya Workbench has greatly simplified the SAS programming experience by offering a syntax style nearly identical to that of SAS 9 and earlier releases. However, there are minor exceptions. In the past, SAS Viya procedures required the use of a CAS engine libref. This engine served multiple purposes, but it was primarily used to upload data to the CAS server and manage the data. Its additional purposes included support of new data types and improved performance for large data sets. No CAS server is needed for SAS Viya Workbench, because all computations are done in place for you in a single location.

In the following program, the LIBNAME statement creates a mylib libref that the examples in this chapter use to refer to your current or desired work directory:

```
libname mylib ".";
```

For more information, see *SAS Viya Workbench: Getting Started*.

Getting Started: FASTKNN Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

This example shows how to use the FASTKNN procedure to search for the *k*-nearest neighbors, from observations in a SAS data table to a set of query observations. In this case, the data are from the hmeq data set. This data set contains information about mortgage applicants. The example selects 4,000 applicants for the input data table, and 100 applicants for the query data table. The FASTKNN procedure returns the input applicants that are most similar to each of the query applicants. The analysis uses eleven variables: bad, loan, mortdue, value, yoj, derog, delinq, clage, ninq, clno, and debtinc. The remaining variables in the data table are not used.

You can load the hmeq data set into your SAS library in the following DATA step. These statements assume that your libref is named mylib, but you can substitute any appropriately defined libref.

```
data mylib.hmeq;
    set sampsio.hmeq(obs=4000);
    id=_n_;
run;

data mylib.query;
    set sampsio.hmeq(firstobs=4001 obs=4100);
```

```
    id=_n_;
run;
```

The following statements run PROC FASTKNN and output the results to ODS tables:

```
proc fastknn
   data            = mylib.hmeq
   query           = mylib.query
   outdist         = mylib.dist_out
   K               = 4
   usetopkoutdist
   threshdist      = 10000;
   id                id;
   input             bad loan mortdue value yoj derog delinq clage ninq clno debtinc;
   output out      = mylib.knn_out;
run;
```

The K=4 option requests that the model return four neighbors; the USETOPKOUTDIST option requests that only the top *k* nearest distances be output to the OUTDIST= option table. the THRESHDIST=10000 option requests that the model exclude distances that are greater than 10,000; the ID statement specifies that the id variable be used as a record identifier; and the INPUT statement specifies that the variables bad, loan, mortdue, value, yoj, derog, delinq, clage, ninq, clno, and debtinc be used as inputs. The OUTPUT statement requests that the neighbors be written to the data table mylib.knn_out. The OUTDIST= option requests that the distances from the query observations to the input observations be written to the data table mylib.dist_out.

Syntax: FASTKNN Procedure

The following statements are available in the FASTKNN procedure:

PROC FASTKNN < *options* > **;**
  DISPLAY < *table-list* > < / *options* > **;**
  DISPLAYOUT *table-spec-list* < / *options* > **;**
  ID *variables* **;**
  INPUT *variables* < /**LEVEL=INTERVAL** > **;**
  OUTPUT OUT=*libref.data-table* < *options* > **;**

The PROC FASTKNN statement, the ID statement, and at least one INPUT statement are required. You can specify multiple INPUT statements.

The following sections describe the PROC FASTKNN statement and then describe the other statements in alphabetical order.

PROC FASTKNN Statement

PROC FASTKNN < *options* > **;**

The PROC FASTKNN statement invokes the procedure. Table 8.1 summarizes the *options* available in the PROC FASTKNN statement.

Table 8.1 PROC FASTKNN Statement Options

Option	Description
Input Data Table Options	
DATA=	Specifies the input data table
QUERY=	Specifies the query data table
Nearest Neighbor Options	
DISTANCEMETRIC=	Specifies the metric to be used to measure the distance between points in *k*-nearest neighbor calculations
EFCONSTRUCTION=	Specifies the number of neighbors to consider during graph construction
EFSEARCH=	Specifies the number of candidate nodes to explore during the graph search phase
IMPUTE	Specifies that observations with missing values in the query data table be imputed by using the *k*-nearest-neighbors method
K=	Specifies the number of neighbors to return for each query
MAXNEIGHBORS=	Specifies the maximum number of connections that each node can have to other nodes within a layer
METHOD=	Specifies the method to be used to compute nearest neighbors
NOPRINT	Suppresses ODS output
NTHREADS=	Specifies the number of threads to use
OUTDIST=	Specifies the output data table in which to store the computed distances
OUTIMPUTE=	Specifies the output data table in which to save the query data after imputing missing values
SEED=	Specifies the seed to use for pseudorandom number generation
THRESHDIST=	Specifies the threshold on the distance
USETOPKOUTDIST	Specifies that only the top *k* nearest distances be output to the OUTDIST= option table

You can specify the following *options*:

DATA=*libref.data-table*

names the input data table for PROC FASTKNN to use. The default is the most recently created data table. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to the data. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 194.

data-table specifies the name of the input data table.

DISTANCEMETRIC=COSINE | IP | L2

DISTMETRIC=COSINE | IP | L2

specifies the metric to be used to measure the distance between points in *k*-nearest neighbor calculations. You can specify the following values:

COSINE uses cosine distance in the *k*-nearest neighbor calculation.

IP uses the inner product in the *k*-nearest neighbor calculation.

L2 uses Euclidean distance in the *k*-nearest neighbor calculation.

By default, DISTANCEMETRIC=L2.

EFCONSTRUCTION=*number*

EF=*number*

specifies the number of neighbors to consider during graph construction, where *number* is an integer greater than 0. You can specify this option only when METHOD=APPROXIMATE.

By default, EFCONSTRUCTION=200.

EFSEARCH=*number*

specifies the number of candidate nodes to explore during the graph search phase, where *number* is an integer greater than 0. You can specify this option only when METHOD=APPROXIMATE.

By default, EFSEARCH=10.

IMPUTE

specifies that observations with missing values in the query data table be imputed by using the *k*-nearest-neighbors method.

By default, missing-value imputation is not performed.

K=*number*

specifies the number of neighbors to return, where *number* is a positive integer.

By default, K=2.

MAXNEIGHBORS=*number*

M=*number*

specifies the maximum number of connections that each node can have to other nodes within a layer, where *number* is an integer between 2 and 10,000, inclusive. You can specify this option only when METHOD=APPROXIMATE.

By default, MAXNEIGHBORS=16.

METHOD=APPROXIMATE | EXACT

specifies the method for the algorithm to use. You can specify the following values:

APPROXIMATE uses approximate search.

EXACT uses exact search.

By default, METHOD=EXACT.

NOPRINT

suppresses ODS output.

NTHREADS=*number-of-threads*

specifies the number of threads to use for the computation, where *number-of-threads* is an integer from 1 to 64, inclusive. The default value is the maximum number of available threads.

OUTDIST=*libref.data-table*

specifies the output model data table to contain the computed pairwise distances between query and input observations. When METHOD=EXACT, all pairwise distances are computed. When METHOD=APPROXIMATE, not all distances are guaranteed to be computed.

OUTIMPUTE=*libref.data-table*

specifies the output data table in which to save the query data after imputing missing values. You must specify the IMPUTE option in order for this table to contain the imputed data.

QUERY=*libref.data-table*

specifies the input data table that contains the query observations.

SEED=*random-seed*

specifies a real number that is used to start the pseudorandom number generator. This option enables you to reproduce the same sample output. If you do not specify a seed or you specify a value less than or equal to 0, the seed is generated by reading the time of day from the computer's clock. You can specify this option only when METHOD=APPROXIMATE.

By default, SEED=0.

THRESHDIST=*number*

specifies the distance threshold for the model, where *number* is a real number greater than 0. PROC FASTKNN excludes from the computation distances greater than *number*. When METHOD=EXACT, all pairwise distances less than or equal to *number* are computed. When METHOD=APPROXIMATE, it is possible that only a subset of those distances less than or equal to *number* will be computed. This option is ignored when DISTANCEMETRIC=IP or COSINE.

By default, THRESHDIST=100.

USETOPKOUTDIST

specifies that only the top k nearest distances be output to the OUTDIST= option table.

By default, all distances are written to the OUTDIST= option table.

DISPLAY Statement

DISPLAY < *table-list* > < / *options* > **;**

The DISPLAY statement enables you to specify a list of display tables to display or exclude. This statement is similar to the ODS SELECT, ODS EXCLUDE, and ODS TRACE statements. However, the DISPLAY statement can improve performance when a large number of tables could be generated (such as in BY-group processing). The DISPLAY statement produces only the requested tables, but ODS statements generate all the tables and then keep the requested tables.

If you use both DISPLAY and ODS statements together, the DISPLAY statement takes precedence over the ODS statements. Note that the ODS EXCLUDE statement processes tables after they have been filtered by the DISPLAY statement. In some cases, it might appear that the ODS EXCLUDE statement is taking precedence because it can further filter the tables. For more information about ODS, see *SAS Output Delivery System: Procedures Guide*.

You can specify the *table-list* as a list of table names, paths, partial pathnames, and regular expressions.

The table names that you can specify are listed in the section "ODS Table Names" on page 203. A path is a table name that is prefixed with dot-separated grouping information. For example, a SelectionSummary table that a procedure produces during a selection routine might have the path *Bygroup1.Summary.SelectionSummary*. A partial pathname does not include all groups; for example, *SelectionSummary* and *Summary.SelectionSummary* are partial pathnames for *Bygroup1.Summary.SelectionSummary*.

When you specify a table name or partial pathname, all display tables whose paths end in the specified name are selected for display or exclusion. For example, both *SelectionSummary* and *Summary.SelectionSummary* select *Bygroup1.Summary.SelectionSummary*.

A regular expression is enclosed in forward slashes (/). For example, specifying "/tions/" selects all pathnames that contain the substring "tions"; in particular, the *Bygroup1.Summary.SelectionSummary* table is selected. Specifying "!/tions/" selects all pathnames that do not contain the substring "tions"; in particular, the *Bygroup1.Summary.SelectionSummary* table is not selected.

You can specify the following *options* after a slash (/):

CASESENSITIVE
performs a case-sensitive comparison of table names in the *table-list* to display table names when tables are subsetted for display. To preserve case, you must enclose table names in the *table-list* in quotation marks.

EXCLUDE
displays all display tables except those that you specify in the *table-list*.

EXCLUDEALL
suppresses display of all tables. This option takes precedence over the other options.

TRACE
displays the display table names, labels, and paths.

DISPLAYOUT Statement

DISPLAYOUT *table-spec-list* < / *options* > **;**

The DISPLAYOUT statement enables you to create output tables from your displayed output. This statement is similar to the ODS OUTPUT statement. For more information about ODS, see *SAS Output Delivery System: Procedures Guide*.

The *table-spec-list* specifies a list of output tables to create. Each entry in the list has either a *key*=*value* format or a *key* format:

key=*value* specifies *key* as the ODS table name, path, or partial pathname, and specifies *value* as the output table name.

key specifies *key* as the ODS table name and also as the output table name.

The ODS table names that you can specify are listed in the section "ODS Table Names" on page 203. You cannot specify the ODS table named OutputCasTables in the *table-spec-list*.

Table names and partial pathnames are discussed under the DISPLAY statement. The DISPLAYOUT statement does not support regular expressions.

You can specify the following *options* after a slash (/):

INCLUDEALL
creates output tables for all display tables. The name of the created output table is the same as the name of the corresponding display table. If you specify this option, the *table-spec-list* specification is ignored.

NOREPLACE
does not replace any existing output table of the same name.

ID Statement

ID *variable* **;**

The ID statement lists one variable to be copied from the input data table to the output data tables that are specified in the OUT= option in the OUTPUT statement and in the OUTDIST= option in the PROC FASTKNN statement. You must specify one *variable*, which must be numeric. The procedure expects each observation to have a unique value for the ID *variable*.

INPUT Statement

INPUT *variables* **;**

The INPUT statement specifies the *variables* to be used in the nearest neighbor search. The *variables* must be of interval type. You must specify at least one input *variable*.

OUTPUT Statement

OUTPUT OUT=*libref.data-table* < *option* > **;**

The OUTPUT statement creates a nearest-neighbors data table to contain the results of the procedure run.

You must specify the following option:

OUT=*libref.data-table*
names the output data table for PROC FASTKNN to use. You must specify this option before any other options. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to where the data table is to be stored. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 194.

data-table specifies the name of the output data table.

You can also specify the following *option*:

COPYVAR=*variable*
COPYVARS=(*variables***)**
lists one or more *variables* from the input data table to be transferred to the output data table.

Details: FASTKNN Procedure

The distance metrics that the FASTKNN procedure uses in nearest neighbor calculations can be defined as follows:

- The Euclidean distance is expressed as

  $$D_{p,q} = \sqrt{\sum_{m=1}^{d} (p_m - q_m)^2}$$

  where D is the Euclidean distance metric, d is the dimension, and p and q are the input and query observations, respectively.

- The inner product (IP) distance is expressed as

  $$D_{p,q} = 1 - \sum_{m=1}^{d} p_m q_m$$

  where D is the inner product distance metric, d is the dimension, and p and q are the input and query observations, respectively.

- The cosine distance is expressed as

  $$D_{p,q} = 1 - \frac{\sum_{m=1}^{d} p_m q_m}{\sqrt{\sum_{m=1}^{d} p_m^2}\sqrt{\sum_{m=1}^{d} q_m^2}}$$

  where D is the cosine distance metric, d is the dimension, and p and q are the input and query observations, respectively.

The distances are sorted in ascending order, and the k neighbors that correspond to the smallest distances are returned. When you specify the METHOD=EXACT option in the PROC FASTKNN statement, all input observations are considered. When you specify METHOD=APPROXIMATE, the procedure uses the hierarchical navigable small world (HNSW) method, which stores input observations in a graph structure across multiple layers. The search begins at the top layer and traverses the graph to approximate the nearest neighbors efficiently. Key options for the HNSW method include three options in the PROC FASTKNN statement: the MAXNEIGHBORS= option, which sets the maximum number of neighbors per node; the EFCONSTRUCTION= option, which controls the candidate list size during graph construction to balance accuracy and performance; and the EFSEARCH= option, which specifies the number of candidate nodes to explore during the graph search phase.

In addition to nearest-neighbor search, PROC FASTKNN provides options for imputing missing values in the query data. You enable this feature by specifying the IMPUTE option, which estimates missing values by using the k-nearest neighbors. You can write the imputed query data to an output table by using the OUTIMPUTE= option, which saves the query data with missing values replaced. By default, this imputation is not performed unless you specify the IMPUTE option.

Displayed Output

The FASTKNN procedure displays various tables that are related to the *k*-nearest neighbor search algorithm. The following sections describe the output tables in the order of their appearance when the related options are specified.

Model Information

The "Model Information" table displays basic information about the parameters that are used in the *k*-NN algorithm. This information includes the number of neighbors (*k*), the method that is used for the *k*-NN search, and the value of the distance threshold that is specified in the THRESHDIST= option in the PROC FASTKNN statement. In addition, when you specify the METHOD=APPROXIMATE option in the same statement, the "Model Information" table includes the following values, which are also specified in options in the PROC FASTKNN statement: the seed that is used for random number generation (in the SEED= option), the maximum number of node connections (in the MAXNEIGHBORS= option), the construction neighbor count (in the EFCONSTRUCTION= option), and the search candidate count (in the EFSEARCH= option).

Number of Observations

The "Number of Observations" table displays the number of observations that are read from the input data table and used.

OutputCasTables Table

The OutputCasTables table is a special table that has information about each table that is created. The information for each table consists of the table name, the library in which the table resides, and the number of columns and rows in the table. Because this table is not a typical ODS table that contains analytical results, you cannot include it in the *table-spec-list* in the DISPLAYOUT statement.

ODS Table Names

Each table that the FASTKNN procedure creates has a name associated with it, and you must use this name to refer to the table when you use ODS statements. The names of each table and a short description of its contents are listed in Table 8.2.

Table 8.2 ODS Tables Produced by PROC FASTKNN

Table Name	Description	Statement	Option
ModelInfo	Model information	PROC FASTKNN	Default
NObs	Number of observations	PROC FASTKNN	Default
OutputCASTables	See the section "OutputCasTables Table" on page 203	PROC FASTKNN	Default

Output Data Tables

The FASTKNN procedure creates data tables to which it writes the *k*-nearest neighbors and the pairwise distances between query and input, as well as the imputed values of the missing query data. You specify the names of these data tables in the OUTPUT statement and in the OUTDIST= and OUTIMPUTE= options of the PROC FASTKNN statement, respectively. Details about the data tables are listed in Table 8.3.

Table 8.3 Output Data Tables Produced by PROC FASTKNN

Data Table	Description
DIST_OUT	Lists the distances from query observations to the input observations
IMPUTE_OUT	Lists the query data after imputing missing values
KNN_OUT	Lists the nearest neighbors

Chapter 9
FISM Procedure

Contents

Overview: FISM Procedure **205**
PROC FISM Features 206
Using SAS Viya Workbench 206
Getting Started: FISM Procedure **206**
Syntax: FISM Procedure **207**
PROC FISM Statement 207
CUSTOMER Statement 208
OUTPUT Statement 208
TARGET Statement 209
Details: FISM Procedure **209**
FP-Growth Algorithm 209
Displayed Output 210
Results 211
Limitations 211
Examples: FISM Procedure **211**
Example 9.1: Identifying Frequent Item Sets 211
References **212**

Overview: FISM Procedure

The FISM procedure performs frequent item set mining, which looks for frequent patterns in a large database. The FISM procedure finds frequent patterns by using the FP-growth (frequent-pattern growth) algorithm of Han, Pei, and Yin (2000). The FP-growth algorithm uses a special data structure called the frequent-pattern tree (FP-tree), which retains the item set association information. For more information about the FP-growth algorithm, see the section "FP-Growth Algorithm" on page 209.

The FISM procedure is used by other procedures such as the MBANALYSIS procedure for association rule mining.

PROC FISM Features

The FISM procedure has the following features:

- supports the FP-growth algorithm, which is currently one of the fastest approaches to frequent item set mining
- supports parallel implementation of the FP-growth algorithm for faster performance
- stores large databases in compact FP-tree format, which uses memory efficiently

Using SAS Viya Workbench

SAS Viya Workbench has greatly simplified the SAS programming experience by offering a syntax style nearly identical to that of SAS 9 and earlier releases. However, there are minor exceptions. In the past, SAS Viya procedures required the use of a CAS engine libref. This engine served multiple purposes, but it was primarily used to upload data to the CAS server and manage the data. Its additional purposes included support of new data types and improved performance for large data sets. No CAS server is needed for SAS Viya Workbench, because all computations are done in place for you in a single location.

In the following program, the LIBNAME statement creates a mylib libref that the examples in this chapter use to refer to your current or desired work directory:

```
libname mylib ".";
```

For more information, see *SAS Viya Workbench: Getting Started*.

Getting Started: FISM Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

This example finds frequent item sets in the sampsio.assocs data set. This data set contains 1,000 observations, each of which contains products that are purchased by a customer.

You can load the sampsio.assocs data set into your SAS library in the following DATA step:

```
data mylib.assocs;
   set sampsio.assocs;
run;
```

These statements assume that your libref is named mylib, but you can substitute any appropriately defined libref.

The following statements execute the FISM procedure on the mylib.assocs data table and generate output in mylib.out data table, which is specified in the OUTPUT statement:

```
proc fism data=mylib.assocs items=3 support=100;
 output out=mylib.out outfreq=mylib.outfreq;
 customer Customer;
 target product;
 run;
```

The PROC FISM statement specifies that input data are contained in the mylib.assocs table and specifies the rule generation criteria: number of items and level of support. The OUTPUT statement names the output tables for storing results. The CUSTOMER statement identifies Customer as the identification variable for transactions. The TARGET statement defines Product as the target variable for transaction items.

This procedure produces a list of frequent items, which is stored in the outfreq table, and frequent item sets, which are stored in the out table.

Syntax: FISM Procedure

The following statements are available in the FISM procedure:

PROC FISM *options* **;**
  CUSTOMER *variable* **;**
  OUTPUT *options* **;**
  TARGET *variable* **;**

All statements are required.

The following sections describe the PROC FISM statement and then describe the other statements in alphabetical order.

PROC FISM Statement

PROC FISM *options* **;**

The PROC FISM statement invokes the procedure.

You must specify the following *option*:

DATA=*libref.data-table*
names the input data table for PROC FISM to use. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to the data. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 206.

data-table specifies the name of the input data table.

You must specify either of the following *options*:

SUPPORT=*number*
specifies the minimum level of support (minimum frequency of an item) for a rule, where *number* must be an integer greater than or equal to 1. This option overrides the specification of the PCTSUP= option.

PCTSUP=*number*

SUPPCT=*number*

SUP_PCT=*number*

PCTSUPPORT=*number*
specifies the minimum level of support for a rule as a percentage of the number of baskets in the input data table, where *number* must be a real number between 0 and 100, inclusive. This option is ignored if the SUPPORT= option is specified.

You can also specify either of the following *options*:

ITEMS=*number*
specifies the number of items in a frequent item set, where *number* must be an integer between 1 and 1,000. By default, ITEMS=2 when you specify the OUT= option in the OUTPUT statement; otherwise, ITEMS=1 by default. This option is ignored if you specify the NFIS_RANGE= option.

NFIS_RANGE=(*number***,** *number***)**
specifies the range of number of items in the frequent item sets, where *number* must be an integer between 1 and 1,000.

By default, NFIS_RANGE=(1,1000).

CUSTOMER Statement

CUSTOMER *variable* **;**

The CUSTOMER statement specifies the variable that is used to group the target *variable* into a basket. You can also specify this statement as ID *variable*.

OUTPUT Statement

OUTPUT *options* **;**

The OUTPUT statement specifies output tables that contain result of FISM procedure.

You must specify at least one of the following options:

OUT=*libref.data-table*
creates a data table that contains information about frequent item sets with their transaction counts and support.

OUTFREQ=*libref.data-table*
creates a data table that contains information about the unique frequent items with their transaction counts and support.

TARGET Statement

TARGET *variable* **;**

The TARGET statement specifies a single nominal *variable* to use as the target variable. If this *variable* is not present in the input data table, then PROC FISM exits with an error.

Details: FISM Procedure

The FISM procedure uses support input to identify the frequent item sets. By default, item sets that contain a maximum of two items are generated. You can specify the item set size in the ITEMS= option.

PROC FISM implements the FP-growth algorithm for finding frequent item sets. PROC FISM discovers the frequent item set by using a support (coverage) measure, which indicates how frequently the items appear in the database. Support of a rule for A and B is expressed as

$$A => B : P(A, B) = X/Y$$

where X is the number of transactions that contain both A and B, and Y is the total number of transactions.

FP-Growth Algorithm

Association rule mining (ARM) is an important data mining task that tries to find interesting rules from a transactional data set. Association rule mining is formally described in Agrawal, Imieliński, and Swami (1993).

Many association rule mining algorithms are based on a "support-confidence" framework, which has two major steps: First, minimum support is applied to find all frequent item sets in a data set. Second, these frequent item sets and the minimum confidence constraint are used to generate rules. Discovering all frequent item sets in a data set is difficult because the task is combinatorial.

The Apriori algorithm in Agrawal and Srikant (1994) uses a breadth-first search (BFS) strategy to count the support of item sets and uses a candidate generation function that exploits the downward closure property of support in Agrawal, Imieliński, and Swami (1993). The basic idea of the downward-closure property is that the support of an item set is less than a particular threshold if the support of any subset of this item set is less than that threshold. In this way, many small item sets are excluded during the generation-and-test process. The Apriori algorithms can reduce the search space by doing a restricted generation-and-test process. The Apriori algorithm is the one of the most efficient association rule mining algorithms.

The Apriori algorithm of Agrawal and Srikant (1994) and many similar algorithms need to scan the data set n–2 times when generating an item set that contains n items. The frequent pattern growth (FP-growth) algorithm builds a frequent-pattern tree (FP-tree) by scanning the data set twice

(Han, Pei, and Yin 2000; Han et al. 2004). Then it discovers the frequent item sets by traversing the FP-tree in a depth-first search (DFS). DFS can find all the frequent item sets without enumerating all the candidates in 2^n search space. Therefore, FP-growth algorithms are faster than BFS-based algorithms. The nature of the FP-growth algorithm is restricted-test-only without generation of the n-item set.

The FP-tree construction algorithm first scans the data set and then collects the set of frequent items, F, and their supports. Next, it sorts F in support-descending order as L, the list of frequent items. Frequent items are sorted in frequency-descending order so that there is a better chance that more prefix strings can be shared.

An FP-tree has two components: an item prefix tree and a header table of frequent items. Each node in the item prefix tree consists of three fields: item-name, count, and node-link. Here item-name registers which item this node represents, count registers the number of transactions represented by the portion of the path that reaches this node, and node-link links to the next node in the FP-tree that carries the same item name, or null if there is none. Each entry in the frequent-item header table consists of two (or three) fields: item-name, count (frequency count of this one-item set), and head of node-link.

When items are sorted in support-descending order, more frequent items are arranged closer to the top of the FP-tree and thus are more likely to be shared. This arrangement suggests that an FP-tree is usually a highly compact projection of the entire data set. However, if the data set is large, the FP-tree might not fit in memory. There are two ways to distribute the data:

- Parallel projection is done in parallel: Each observation is projected to every node. The problem is that the total size of the projected data sets could be multiple times the size of the original data set.
- Partition projection is done in sequence: an observation T is projected to the X-projected data set only if X is a frequent item in T and there is no other item after X in the list of frequent items that appears in the observation. The advantage of partition projection is that the total size of the projected data sets at each level is smaller than the original data set.

The frequent item sets are mined by traversing a node's conditional pattern base (under the condition of the node's existence) and thus are called the conditional FP-tree (CFPT). The mining function is defined as mine(CFPT | node), which is also a recursive function that uses a divide-and-conquer process.

Displayed Output

The FISM procedure does not produce any displayed output tables.

Results

The results that the FISM procedure produces are stored in tables. You can access these tables by their names, which are shown in Table 9.1.

Table 9.1 Results Tables Produced by the FISM Procedure

Table Name	Description	Statement	Option
out	Contains general information about the frequent item sets	PROC FISM	OUT=
outfreq	Contains information about the items' frequencies	PROC FISM	OUTFREQ=

Limitations

When the number of observations is greater than 10 million and the number of items in a frequent item set (as specified in the ITEMS= option in the PROC FISM statement) is considerably large (more than 10), the amount of memory required to process the analysis might be so great that the procedure stops responding.

The maximum number of frequent item sets that are generated per thread is 10 million.

Examples: FISM Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

Example 9.1: Identifying Frequent Item Sets

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

This example identifies frequent item sets in the sampsio.assocs data set. This data set contains 1,000 transactions of 1,000 customers, each of which contains customer, time, and product information.

You can load the sampsio.assocs data set into your SAS library in the following DATA step:

```
data mylib.assocs;
   set sampsio.assocs;
run;
```

These statements assume that your libref is named mylib, but you can substitute any appropriately defined libref.

The following statements run the FP-growth algorithm on the mylib.assocs data table:

```
proc fism data=mylib.assocs items=5 support=200;
 output out=mylib.out outfreq=mylib.outfreq;
 customer Customer;
 target product;
 run;
```

The DATA= option names the input data table to be analyzed. The ITEMS= option specifies the number of items in a frequent item set. The SUPPORT= option specifies the minimum level of support for a rule. The OUTPUT statement specifies output tables to contain results of the FISM procedure. The CUSTOMER statement requests that the Customer variable be used to group the target variable into baskets. The TARGET statement requests that the Product variable be used as the target variable.

References

Agrawal, R., Imieliński, T., and Swami, A. (1993). "Mining Association Rules between Sets of Items in Large Databases." In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, 207–216. New York: ACM Press. https://doi.org/10.1145/170035.170072.

Agrawal, R., and Srikant, R. (1994). "Fast Algorithms for Mining Association Rules in Large Databases." In *Proceedings of the 20th International Conference on Very Large Data Bases*, 487–499. San Francisco: Morgan Kaufmann. http://www.vldb.org/conf/1994/P487.PDF.

Han, J., Pei, J., and Yin, Y. (2000). "Mining Frequent Patterns without Candidate Generation." In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*. New York: ACM Press. https://www.cs.sfu.ca/~jpei/publications/sigmod00.pdf.

Han, J., Pei, J., Yin, Y., and Mao, R. (2004). "Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach." *Data Mining and Knowledge Discovery* 8:53–87.

Chapter 10
FOREST Procedure

Contents

Overview: FOREST Procedure . . . **215**
- PROC FOREST Features . . . 215
- Using SAS Viya Workbench . . . 215

Getting Started: FOREST Procedure . . . **216**

Syntax: FOREST Procedure . . . **220**
- PROC FOREST Statement . . . 221
- AUTOTUNE Statement . . . 228
- CODE Statement . . . 233
- CROSSVALIDATION Statement . . . 234
- FREQ Statement . . . 234
- GROW Statement . . . 235
- ID Statement . . . 236
- INPUT Statement . . . 236
- MITIGATEBIAS Statement . . . 236
- OUTPUT Statement . . . 239
- PARTITION Statement . . . 240
- SAVESTATE Statement . . . 240
- TARGET Statement . . . 241
- VIICODE Statement . . . 241
- WEIGHT Statement . . . 242

Details: FOREST Procedure . . . **242**
- Bagging the Data . . . 242
- Training a Decision Tree . . . 242
- Splitting Nominal Values . . . 243
  - Methods . . . 243
  - Risks of Many Categories . . . 244
- Loh Method . . . 244
- Predicting an Observation . . . 245
- Measuring Prediction Error . . . 245
- Handling Missing Values . . . 246
  - Strategies . . . 246
  - Specifics . . . 247
- Handling Values That Are Absent from Training Data . . . 247
- Measuring Variable Importance . . . 247
  - Residual Sum of Squares Importance Method . . . 248
  - Random Branch Assignment Importance Method . . . 250

Isolation Forests . . . 250
Quantile Regression Forest . . . 251
Hyperparameter Tuning . . . 252
k-fold Cross Validation . . . 253
TreeSHAP Values . . . 253
Bias Mitigation . . . 253
Exponentiated Gradient Reduction (EGR) Algorithm . . . 254
Displayed Output . . . 256
Model Information . . . 256
Number of Observations . . . 256
Variable Importance . . . 256
RBA Variable Importance . . . 256
Fit Statistics . . . 256
Tuner Information . . . 256
Tuner Summary . . . 256
Tuner Timing . . . 257
Best Configuration . . . 257
Tuner Results . . . 257
Cross-Validation Fit Statistics . . . 257
Evaluation History . . . 257
OutputCasTables Table . . . 257
Predicted Probability Names . . . 257
Constraint History . . . 257
Group Metrics History . . . 258
Iteration History . . . 258
Mitigation Information . . . 258
Timing . . . 258
Weight History . . . 258
ODS Table Names . . . 258
ODS Graphics . . . 260
Examples: FOREST Procedure . . . **260**
Example 10.1: Scoring New Data by Using a Previous Forest Model . . . 261
Example 10.2: Generating an Isolation Forest . . . 264
Example 10.3: Tuning a Forest Model . . . 266
Example 10.4: Mitigating Bias of a Classification Forest . . . 269
References . . . **273**

Overview: FOREST Procedure

The FOREST procedure creates a predictive model called a *forest* (which consists of several decision trees) in SAS Viya Workbench. A *predictive model* defines a relationship between input variables and a target variable. The purpose of a predictive model is to predict a target value from inputs. The FOREST procedure *trains* the model; that is, it creates the model by using training data in which the target values are known. The model can then be applied to observations in which the target is unknown. If the predictions fit the new data well, the model is said to *generalize* well. Good generalization is the primary goal for predictive tasks. A predictive model might fit the training data well but generalize poorly.

A *decision tree* is a type of predictive model that has been developed independently in the statistics and artificial intelligence communities. The FOREST procedure creates a tree recursively: The procedure chooses an input variable and uses it to create a rule to split the data into two or more subsets. The process is then repeated in each subset, and then again in each new subset, and so on until some constraint is met. In the terminology of the tree metaphor, the subsets are *nodes*, the original data table is the *root* node, and the final unpartitioned subsets are *leaves* or *terminal nodes*. A node is an *internal node* if it is not a leaf. The data in a leaf determine the estimates of the value of the target variable. These estimates are subsequently applied to predict the target of a new observation that is assigned to the leaf.

The FOREST procedure creates multiple decision trees that differ from each other in two ways: First, the training data for each tree constitute a different sample; each sample is created by sampling with replacement observations from the original training data of the forest. Second, the input variables that are considered for splitting a node are randomly selected from all available inputs. Among these randomly selected variables, the FOREST procedure chooses a single variable, which is associated the most with the target, when it forms a splitting rule.

PROC FOREST Features

The FOREST procedure creates an ensemble of decision trees to predict a single target of either interval or nominal measurement level. An input variable can have an interval or nominal measurement level.

The FOREST procedure ignores any observation from the training data that has a missing target value.

Using SAS Viya Workbench

SAS Viya Workbench has greatly simplified the SAS programming experience by offering a syntax style nearly identical to that of SAS 9 and earlier releases. However, there are minor exceptions. In the past, SAS Viya procedures required the use of a CAS engine libref. This engine served multiple purposes, but it was primarily used to upload data to the CAS server and manage the data. Its additional purposes included support of new data types and improved performance for large data sets. No CAS server is needed for SAS Viya Workbench, because all computations are done in place for you in a single location.

In the following program, the LIBNAME statement creates a mylib libref that the examples in this chapter use to refer to your current or desired work directory:

```
libname mylib ".";
```

For more information, see *SAS Viya Workbench: Getting Started*.

Getting Started: FOREST Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

A common use of forest models is to predict whether a mortgage applicant will default on a loan. The home equity data table Hmeq, which is in the Sampsio library that SAS provides, contains observations for 5,960 mortgage applicants. A variable named Bad indicates whether the applicant, after being approved for a loan, paid off or defaulted on the loan.

This example uses the Hmeq data table to build a forest model that is used to score the data and can be used to score data about new loan applicants. Table 10.1 describes the variables in Hmeq.

Table 10.1 Variables in the Home Equity (Hmeq) Data Table

Variable	Role	Level	Description
Bad	Response	Binary	1 = applicant defaulted on the loan or is seriously delinquent 0 = applicant paid off the loan
CLAge	Predictor	Interval	Age of oldest credit line in months
CLNo	Predictor	Interval	Number of credit lines
DebtInc	Predictor	Interval	Debt-to-income ratio
Delinq	Predictor	Interval	Number of delinquent credit lines
Derog	Predictor	Interval	Number of major derogatory reports
Job	Predictor	Nominal	Occupational category
Loan	Predictor	Interval	Requested loan amount
MortDue	Predictor	Interval	Amount due on mortgage
nInq	Predictor	Interval	Number of recent credit inquiries
Reason	Predictor	Binary	`'DebtCon'` = debt consolidation `'HomeImp'` = home improvement
Value	Predictor	Interval	Value of property
YoJ	Predictor	Interval	Years at present job

The following statements load the mylib.hmeq data into your SAS library. These statements assume that your SAS libref is named mylib, as in the section "Using SAS Viya Workbench" on page 215, but you can substitute any appropriately defined SAS libref.

```
/* Convert variable names to mixed case */
data mylib.hmeq;
   length Bad Loan MortDue Value 8 Reason Job $7
             YoJ Derog Delinq CLAge nInq CLNo DebtInc 8;
   set sampsio.hmeq;
```

```
run;

proc print data=mylib.hmeq(obs=10); run;
```

Output 10.1 shows the first 10 observations of mylib.hmeq.

Figure 10.1 Partial Listing of the mylib.hmeq Data

Obs	Bad	Loan	MortDue	Value	Reason	Job	YoJ	Derog	Delinq	CLAge	nInq	CLNo	DebtInc
1	1	1100	25860	39025	HomeImp	Other	10.5	0	0	94.367	1	9	.
2	1	1500	.	.			.	.	.	.	.	.	.
3	1	1800	48649	57037	HomeImp	Other	5.0	3	2	77.100	1	17	.
4	1	2000	.	62250	HomeImp	Sales	16.0	0	0	115.800	0	13	.
5	1	2000	45000	55000	HomeImp	Other	3.0	0	0	86.067	2	25	.
6	1	2200	24280	34687	HomeImp	Other	.	0	1	300.867	0	8	.
7	1	2300	28192	40150	HomeImp	Other	4.5	0	0	54.600	1	16	.
8	1	2400	50000	73395	HomeImp	ProfExe	5.0	1	0	.	1	0	.
9	1	2400	.	17180	HomeImp	Other	.	0	0	14.567	3	4	.
10	1	2500	15000	20200	HomeImp		18.0	0	0	136.067	1	19	.

PROC FOREST treats numeric variables as interval inputs unless you specify otherwise. Character variables are always treated as nominal inputs. The following statements run PROC FOREST and save the model in a table named mylib.savedModel:

```
proc forest data=mylib.hmeq outmodel=mylib.savedModel seed=12345;
   input Delinq Derog Job nInq Reason / level = nominal;
   input CLAge CLNo DebtInc Loan Mortdue Value YoJ / level = interval;
   target Bad / level = nominal;
   ods output FitStatistics=fitstats;
run;
```

No parameters are specified in the PROC FOREST statement; therefore, the procedure uses all default values. For example, the number of trees in the forest is 100, the number of bins for interval input variables is 20, and the number of variables that are examined at each node for a split is the square root of the number of input variables.

The INPUT and TARGET statements are required in order to run PROC FOREST. The INPUT statement indicates which variables to use to build the model, and the TARGET statement indicates which variable the procedure predicts.

Figure 10.2 displays the “Model Information” table. This table shows the values of the training parameters in the first six rows, in addition to some basic information about the trees in the forest.

Figure 10.2 Model Information

The FOREST Procedure

Model Information	
Number of Trees	100
Number of Variables Per Split	4
Seed	12345
Bootstrap Percentage	60
Number of Bins	50
Number of Input Variables	12
Maximum Number of Tree Nodes	329
Minimum Number of Tree Nodes	143
Maximum Number of Branches	2
Minimum Number of Branches	2
Maximum Depth	20
Minimum Depth	20
Maximum Number of Leaves	165
Minimum Number of Leaves	72
Maximum Leaf Size	2282
Minimum Leaf Size	5
OOB Misclassification Rate	0.09765101
Average Number of Leaves	122.33

Figure 10.3 displays the "Number of Observations" table, which shows how many observations were read and used. If you specify a PARTITION statement, the "Number of Observations" table also displays the number of observations that were read and used per partition.

Figure 10.3 Number of Observations

	Training
Number of Observations Read	5960
Number of Observations Used	5960

Figure 10.4 displays the estimates of variable importance. The rows in this figure are sorted by the importance measure. A conclusion from fitting the forest model to these data is that DebtInc is the most important predictor of loan default.

Figure 10.4 Variable Importance

Variable Importance			
Variable	Importance	Std Dev Importance	Relative Importance
DebtInc	384.10	63.0376	1.0000
Value	72.7549	25.4882	0.1894
Delinq	58.6063	18.1454	0.1526
CLAge	46.9144	12.0956	0.1221
Derog	39.7649	15.5401	0.1035
CLNo	31.2729	8.6660	0.0814
Loan	30.9741	12.1380	0.0806
YoJ	27.4476	10.7459	0.0715
Job	26.8259	9.2556	0.0698
nInq	26.3861	9.3798	0.0687
MortDue	26.1040	9.8348	0.0680
Reason	4.9037	4.3171	0.0128

Figure 10.5 shows the first 10 and last 10 observations of the fit statistics. When PROC FOREST runs, it computes fit statistics for a sequence of forests that have an increasing number of trees. As the number of trees increases, the fit statistics usually improve (decrease) at first and then level off and fluctuate within a small range. Forest models provide an alternative estimate of the average square error and misclassification rate. This alternative is called the *out-of-bag* (OOB) estimate. The OOB estimate is a convenient substitute for an estimate that is based on test data and is a less biased estimate of how the model will perform on future data. For more information, see the section "Bagging the Data" on page 242. The listing shows that the out-of-bag error estimate is worse (larger) than the estimate that evaluates all observations on all trees. This is common.

Figure 10.5 Fit Statistics

Fit Statistics						
Number of Trees	OOB Average Square Error	Training Average Square Error	OOB Misclassification Rate	Training Misclassification Rate	OOB Log Loss	Training Log Loss
1	0.1059	0.0788	0.1347	0.1037	1.301	0.777
2	0.0979	0.0669	0.1266	0.0871	0.915	0.290
3	0.0942	0.0627	0.1234	0.0802	0.707	0.226
4	0.0909	0.0615	0.1197	0.0817	0.586	0.212
5	0.0867	0.0595	0.1175	0.0760	0.452	0.207
6	0.0840	0.0583	0.1133	0.0768	0.403	0.203
7	0.0825	0.0572	0.1120	0.0743	0.354	0.200
8	0.0808	0.0572	0.1098	0.0730	0.326	0.200
9	0.0801	0.0569	0.1081	0.0732	0.307	0.199
10	0.0793	0.0568	0.1079	0.0750	0.297	0.198
.	.	.	.	.	.	.
.	.	.	.	.	.	.
.	.	.	.	.	.	.
91	0.0736	0.0550	0.0980	0.0742	0.249	0.193
92	0.0736	0.0549	0.0982	0.0740	0.249	0.193
93	0.0736	0.0549	0.0987	0.0740	0.249	0.193
94	0.0736	0.0550	0.0983	0.0738	0.249	0.193
95	0.0737	0.0550	0.0978	0.0747	0.249	0.193
96	0.0737	0.0550	0.0975	0.0745	0.249	0.193
97	0.0737	0.0550	0.0978	0.0743	0.249	0.194
98	0.0737	0.0550	0.0968	0.0738	0.249	0.194
99	0.0737	0.0550	0.0971	0.0737	0.249	0.193
100	0.0737	0.0550	0.0977	0.0737	0.249	0.193

Syntax: FOREST Procedure

The following statements are available in the FOREST procedure:

```
PROC FOREST <options> ;
    AUTOTUNE <options> ;
    CODE <options> ;
    CROSSVALIDATION <options> ;
    FREQ variable ;
    GROW criterion ;
    ID variables ;
    INPUT variables < / LEVEL=NOMINAL | INTERVAL > ;
    MITIGATEBIAS mitigate-options ;
    OUTPUT OUT=libref.data-table < option > ;
    PARTITION partition-option ;
    SAVESTATE RSTORE=libref.data-table ;
    TARGET variable < / LEVEL=NOMINAL | INTERVAL > ;
    VIICODE < options > ;
    WEIGHT variable ;
```

The PROC FOREST and INPUT statements are required. The TARGET statement is required if you omit the ISOLATION option in the PROC FOREST statement. The INPUT statement can appear multiple times.

PROC FOREST Statement

PROC FOREST < *options* > **;**

The PROC FOREST statement invokes the procedure. Table 10.2 summarizes the options in this statement.

Table 10.2 PROC FOREST Statement Options

Option	Description
Basic Options	
BINMETHOD=	Specifies how to bin interval inputs prior to training
DATA=	Specifies the name of the input table
INBAGFRACTION=	Specifies the fraction of the training data to use for growing each tree
INMODEL=	Specifies a saved forest model to use for scoring a new table
ISOLATION	Generates an isolation forest for anomaly detection instead of a forest for target prediction
LOH=	Specifies the number of variables to preselect using the Loh method
NOPRINT	Suppresses ODS output
NTHREADS=	Specifies the number of threads to use in the computation
NTREES=	Specifies the number of trees to grow in the forest model
NUMBIN=	Specifies the number of bins for continuous variables
OUTMODEL=	Specifies the data table to score the forest model
PRINTTARGET	Creates tables that contain information about columns in the output
QUANTILEREGRESSION	Trains a quantile regression forest and specifies the quantile values to compute during scoring
RBAIMP	Creates a variable importance table by using random branch assignment
SEED=	Specifies the random number seed to use for model building
VARS_TO_TRY=	Specifies the number of variables to examine at each node split
VII=	Calculates the importance of the specified types of variable interactions
VOTE=	Specifies the method for calculating the predicted probabilities for a nominal target
Splitting Options	
ASSIGNMISSING=	Specifies how to handle missing values in a predictor variable
MAXBRANCH=	Specifies the maximum number of splits per node
MAXDEPTH=	Specifies the maximum tree depth
MINLEAFSIZE=	Specifies the minimum number of observations per leaf
MINUSEINSEARCH=	Specifies the minimum number of observations to use with the USEINSEARCH policy for handling missing values
NOMSEARCH=	Specifies the method to use to find a split on a nominal variable

You can specify the following *options*:

ASSIGNMISSING=NONE | MACSMALL | USEINSEARCH

specifies how PROC FOREST creates a default splitting rule that is used to handle missing values and unknown levels. An unknown level is a level of a categorical predictor that does not exist in the training data but is encountered during scoring.

This option controls how missing values are used in model training, and controls the creation of the default splitting rule.

The primary splitting rule for a node is created during model training. During model scoring, observations are assigned to a node in a tree based upon the primary splitting rule if the rule's variable is not missing. If the variable is missing for the observation, then the default splitting rule is used.

The default splitting rule enables all data to be scored, even if the primary rule cannot be used on a particular observation.

You can specify one of the following values to determine the default splitting rule:

NONE excludes observations that have any missing variables from training the forest model. In the scoring phase, this default rule assigns observations that have missing values of an interval predictor variable to the branch with the smallest predictor values, and assigns observations that have unknown and missing nominal levels to the branch with the most training observations.

MACSMALL treats a missing value as a separate, legitimate value in the search for a split for the primary splitting rule. Missing values in interval inputs are treated as less than any other number. In the scoring phase, this default rule assigns missing interval inputs to the leftmost branch of the split, and unknown nominal levels to the largest branch in the split.

USEINSEARCH treats a missing value as a separate, legitimate value in the search for a split for the primary splitting rule. Missing values in interval inputs are treated as a special level that is used during the split process. In the scoring phase, this default rule assigns missing interval inputs to the branch determined during forest growing, and unknown nominal levels to the largest branch in the split.

By default, ASSIGNMISSING=USEINSEARCH.

BINMETHOD=BUCKET | QUANTILE

specifies how to bin interval input variables prior to growing the forest model. The number of bins that are created is determined by the NUMBIN= option.

You can specify one of the following values:

BUCKET bins interval input variables into fixed-width bins. The width of each bin for a particular variable is calculated by subtracting the smallest value among all observations from the largest value among all observations, and then dividing that result by the number of bins.

QUANTILE bins interval input variables into bins according to their quantile. The width of the bins for a particular variable is not fixed, but the number of observations in each bin is approximately equal.

By default, BINMETHOD=QUANTILE.

DATA=*libref.data-table*

names the input data table for PROC FOREST to use. The default is the most recently created data table. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to the data. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 215.

data-table specifies the name of the input data table.

INBAGFRACTION=*number*

BOOTSTRAP=*number*

specifies the fraction of the random bootstrap sample of the training data to be used for growing each tree in the forest, where *number* is a value between 0 and 1. Observations can be duplicated in the sample. The number of distinct observations in the sample will be less than the number of observations available for training even when *number* is 1.

By default, INBAGFRACTION=0.6. This value can be tuned with the AUTOTUNE statement.

INMODEL=< *libref.* **>***data-table*

specifies the data table that you have previously saved as a forest model by using the OUTMODEL= option in a previous run of PROC FOREST. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the input data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 215.

When you use the INMODEL= option, the OUTPUT statement is required and any other options in the PROC FOREST statement, except for NOPRINT and VOTE=, are ignored.

The *data-table* that you specify in the INMODEL= option must include the attributes that are associated with the table that the OUTMODEL= option produces, or PROC FOREST issues an error.

ISOLATION < (*suboptions*) **>**

creates an isolation forest for anomaly detection instead of creating a forest for target prediction. For more information about isolation forests, see the section "Isolation Forests" on page 250.

You can specify the following *suboptions*:

BETA=*number*

specifies the value to use as both inputs to the beta distribution in order to sample the random value for splitting interval variables. If you specify a *number* equal to 1, then PROC FOREST samples from the uniform distribution.

By default, BETA=4.

NPASSES=*number*

specifies the number of times to evaluate interval nodes. For each pass after the first, nodes whose splits create empty branches are reevaluated and recalculated.

By default, NPASSES=1.

SAMPLEN=*number*

specifies the *number* of observations, sampled without replacement, to use in each tree of the isolation forest.

By default, SAMPLEN=100.

When you specify the ISOLATION option, the GROW and TARGET statements are ignored, and the BINMETHOD=, INBAGFRACTION=, RBAIMP, VARS_TO_TRY=, and VII= options in the PROC FOREST statement are ignored. Fit statistics are not computed.

LOH=*L*

specifies a number of variables (*L*) that are preselected to consider for candidate splits for each node. The variables are selected using the Loh method.

If *L* is less than the value of the VARS_TO_TRY= option (*m*), then the variables are selected from among the *m* variables. If *L* is greater than or equal to *m*, or if no *L* is specified, then the Loh method is not used.

MAXBRANCH=*b*

specifies the maximum number of children per node in the tree. PROC FOREST tries to create this number of children unless it is impossible (for example, if a split variable does not have enough levels).

By default, MAXBRANCH=2.

MAXDEPTH=*number*

specifies the maximum depth of the tree to be grown. The number of levels in a tree is equal to the depth plus 1.

By default, MAXDEPTH=20.

This value can be tuned with the AUTOTUNE statement. In the ODS output that corresponds to tables that are generated using the AUTOTUNE statement, the term "MAXLEVEL" is used. The value of MAXLEVEL is equal to the value of the MAXDEPTH= option plus 1.

MINLEAFSIZE=*number*

LEAFSIZE=*number*

specifies the minimum *number* of observations that each child of a split must contain in the training data table in order for the split to be considered. The count of observations includes those that are excluded from training as a consequence of specifying a value less than 1 in the INBAGFRACTION= option. By default, MINLEAFSIZE=5. This value can be tuned with the AUTOTUNE statement.

MINUSEINSEARCH=*number*

specifies a threshold for using missing values in the split search when ASSIGNMISSING=USEINSEARCH. If the number of observations in which the splitting variable has missing values is greater than or equal to *number*, then PROC FOREST uses the USEINSEARCH policy to handle missing values for that variable.

By default, MINUSERINSEARCH=1.

NOMSEARCH(*suboptions*)

specifies search methods for splitting on a nominal variable. For more information, see the section "Splitting Nominal Values" on page 243. If you omit this option, the default values of the *suboptions* determine the search method. You can specify one or more of the following *suboptions*:

MAXCATEGORIES=*number*

specifies the maximum number of categories to use in a splitting rule. The input is eligible for splitting nodes in which no more than *number* categories appear. The value of *number* must be greater than 1.

By default, MAXCATEGORIES=128.

SHRINKAGE=*number*

specifies how much weight to give the average gradient when you combine it with the average gradient within a category. You use this option only when sorting the categories. The value of *number* must be nonnegative.

By default, SHRINKAGE=10.

SORT=*number*

specifies the minimum cardinality in the node of the nominal variable for using the sort method. The sort method sorts the categories and then examines binary splits that preserve the ordering. The value of *number* must be a nonnegative integer. If *number* is greater than the value of the MAXCATEGORIES=*number* option, then the variable is not used in the node.

By default, SORT=10.

NOPRINT

suppresses ODS output.

NTHREADS=*number-of-threads*

specifies the number of threads to use in the computation.

NTREES=*number*

specifies the *number* of trees to grow in the forest model.

By default, NTREES=100. This value can be tuned with the AUTOTUNE statement.

NUMBIN=*number*

NBINS=*number*

specifies the *number* of bins to use for binning the interval input variables. PROC FOREST bins continuous predictors to a fixed bin size. This option controls the number of bins and thereby also the size of the bins.

By default, NUMBIN=50. This value can be tuned with the AUTOTUNE statement.

OUTMODEL=< *libref.*>*data-table*

specifies the data table to which to save the forest model. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 215.

The data table that results from this option contains information about each node and each tree in the forest model, including the splitting variables, the child nodes, the number of observations at each

node, and the predicted response at each node. This table also has attributes attached that contain information about the input variables, the target variable, and the forest model.

PRINTTARGET

outputs tables that indicate generated columns in the OUT= table from the OUTPUT statement. For a continuous response, PROC FOREST generates an output table named PredName, which indicates the name of the predicted value column. For a categorical response, PROC FOREST generates an output table named PredIntoName, which indicates the name of the predicted value column, and also an output table named PredProbName, which indicates the names of the predicted probability columns.

QUANTILEREGRESSION < (*suboptions*) >

QUANTILEREG < (*suboptions*) >

QUANTREG < (*suboptions*) >

QRF < (*suboptions*) >

trains a quantile regression forest and specifies the quantile values to compute during scoring. A quantile regression forest stores additional information about the distribution of target values at each leaf node during training; this enables you to estimate conditional quantiles during scoring. Use the QUANTILES() suboption to specify which quantile values to compute during scoring.

This option is valid only for an interval target. If the target is nominal, PROC FOREST issues an error and stops processing. You cannot use the QUANTILEREGRESSION option with the ISOLATION option or with a MAXBRANCH= option value greater than 2. If either combination is present, the procedure issues an error and stops processing.

You can specify the following *suboptions*:

KHIGH=*number*

specifies the number of largest target values to store per leaf for accurate high quantile estimation, where *number* is an integer between 0 and 10,000 (inclusive).

By default, KHIGH=50.

KLOW=*number*

specifies the number of smallest target values to store per leaf for accurate low quantile estimation, where *number* is an integer between 0 and 10,000 (inclusive).

By default, KLOW=50.

METHOD=EXACT | FAST | HYBRID | TAIL | UNIFORM

specifies the storage method for quantile computation in quantile regression forests. You can specify one of the following methods:

EXACT stores all target values per leaf for exact quantile computation. Use this method only when exact quantiles are required and memory is not a constraint. For large data tables, this method might require significant memory and can take noticeably longer to score.

FAST stores a single random value per leaf. This method uses minimal memory and is suitable for rough median estimation.

HYBRID combines exact tail storage with compact body representation. Use the KLOW= and KHIGH= suboptions to specify how many of the smallest and

largest values to store for the tails, and use the NANCHORS= suboption to specify the number of anchor points for the body representation. The HYBRID method requires at least one tail to be nonzero. If you specify both KLOW=0 and KHIGH=0, both suboptions are reset to their default values. If a leaf contains fewer target values than the sum of the KLOW=, KHIGH=, and NANCHORS= suboption values, PROC FOREST stores all target values for that leaf.

TAIL stores extreme target values per leaf for accurate estimation of tail quantiles. Use the KLOW= suboption to specify how many of the smallest values to store, and use the KHIGH= suboption to specify how many of the largest values to store. Quantile values that are requested during scoring but fall outside the range of stored tail values cannot be estimated accurately and are set to missing in the output.

UNIFORM specifies uniform reservoir sampling to store a representative sample of target values per leaf. Use the NSAMPLES= suboption to specify how many values to store.

By default, METHOD=HYBRID.

NANCHORS=*number*

specifies the number of anchor points to use in the hybrid method to compactly represent the body of the distribution, where *number* is an integer between 2 and 39 (inclusive).

By default, NANCHORS=19.

NSAMPLES=*number*

specifies the number of random target samples to store per leaf by using reservoir sampling, where *number* is an integer between 1 and 10,000 (inclusive).

By default, NSAMPLES=100.

QUANTILES(*value-list***)**

specifies a list of quantile values to compute for quantile regression forest predictions during scoring, where *value-list* is a space-separated list of numbers between 0 and 1 (inclusive). Each value represents a percentile that is expressed as a decimal. You must specify the QUANTILE-REGRESSION option during training in order to use this suboption during scoring.

RBAIMP

creates a variable importance table by using random branch assignment (RBA). This table is created in addition to the normal variable importance table that is calculated using the residual sum of squares (RSS) error. For more information about RBA and RSS variable importance, see the section "Measuring Variable Importance" on page 247.

SEED=*number*

specifies the initial seed for random number generation that is used in both selecting out-of-bag observations for the trees and selecting the subset of variables to determine splits. The value of *number* must be an integer. If you do not specify a seed or you specify a value less than or equal to 0, the seed is generated by reading the time of day from the computer's clock.

VARS_TO_TRY=*m*

M=*m*

specifies the number of input variables to consider splitting on in a node, where *m* ranges from 1 to the number of input variables.

By default, *m* is the square root of the number of input variables. This value can be tuned with the AUTOTUNE statement.

VII=2 | 3

INTERACTIONIMP=2 | 3

calculates the variable interaction importance, which is described in the section "Variable Interaction Importance" on page 249. You can specify the following values:

2 calculates the importance of all two-way variable interactions.

3 calculates the importance of all three-way and all two-way variable interactions.

If you do not specify this option, then the variable interaction importance is not calculated.

VOTE=MAJORITY | PROBABILITY

specifies how to calculate the predicted probability of the target levels for a nominal target. The predicted level is the level that has the highest predicted probability. This option affects the scoring and fit statistics of the forest model. You can specify the following values:

MAJORITY specifies that the predicted probability of each target level is equal to the number of trees in the forest that predicted that level as the target, divided by the total number of trees in the forest.

PROBABILITY specifies that the predicted probability of each target level is equal to the probability of that level averaged over each tree in the forest.

When using the INMODEL= option, you can specify how to calculate the predicted probability of the target levels, even if the model was trained with a different value for the VOTE= option.

By default, VOTE=PROBABILITY.

AUTOTUNE Statement

AUTOTUNE < *options* > **;**

The AUTOTUNE statement searches for the best combination of values of the INBAGFRACTION=, MAXDEPTH=, MINLEAFSIZE=, NUMBIN=, NTREES=, and VARS_TO_TRY= options in the PROC FOREST statement. You cannot specify both the AUTOTUNE statement and the CROSSVALIDATION statement in the same procedure run.

Table 10.3 summarizes the *options* that you can specify in the AUTOTUNE statement. For more information about all options except the TUNINGPARAMETERS= option, see the option's description in the section "AUTOTUNE Statement" on page 6 in Chapter 2, "Shared Concepts." The TUNINGPARAMETERS= option is described following Table 10.3.

Table 10.3 AUTOTUNE Statement Options

Option	Description
APPENDLOOKUP	Specifies that the table specified in the HISTORYTABLE= option contain the rows from the table specified in the LOOKUPTABLE= option
CONSTRAINTS=	Specifies the constraints on the output metrics
EVALHISTORY=	Specifies how to report the evaluation history of the tuner
FOLDCOLUMN=	Specifies the column in the data table in which the cross validation fold value is indicated
FRACTION=	Specifies the fraction of observations to use for validation
HISTORYTABLE=	Specifies the table that contains the evaluation history
KFOLD=	Specifies the number of folds for *k*-fold cross validation
LIVEUPDATE	Specifies that the table specified in the HISTORYTABLE= option be updated at every evaluation
LOOKUPTABLE=	Specifies the table to use for evaluation lookup
MAXBAYES=	Specifies the maximum number of points in the kriging model
MAXEVALS=	Specifies the maximum number of evaluations
MAXITER=	Specifies the maximum number of iterations when SEARCHMETHOD=GA or SEARCHMETHOD=BAYESIAN
MAXTIME=	Specifies the maximum time for all iterations
MAXTRAINTIME=	Specifies the maximum time for a model training
NCONVITER=	Specifies the number of convergence iterations
NOGRIDSHUFFLE	Requests that the grid points not be shuffled
NOLOCALSEARCH	Disables local search optimization
NPARALLEL=	Specifies the number of parallel sessions
NSUBSESSIONWORKERS=	Specifies the number of workers in parallel sessions
OBJECTIVE=	Specifies the objective function
POPSIZE=	Specifies the population size when SEARCHMETHOD=GA or SEARCHMETHOD=BAYESIAN
SAMPLESIZE=	Specifies the sample size when SEARCHMETHOD=LHS or SEARCHMETHOD=RANDOM
SEARCHMETHOD=	Specifies the search method that the optimizer uses
SECONDOBJECTIVE=	Specifies the second objective to use for tuning
SELECTINITPOINT	Specifies that the tuner select the best evaluation from the lookup table
TARGETEVENT=	Specifies the target event for ROC-based calculations
TRAINFRACTION=	Specifies the fraction of observations to use for training
TUNINGPARAMETERS=	Specifies the custom tuning parameters
USEPARAMETERS=	Specifies how to handle the TUNINGPARAMETERS= option

TUNINGPARAMETERS=(*suboption* **| . . . | <** *suboption* **>)**

TUNEPARMS=(*suboption* **| . . . | <** *suboption* **>)**

specifies which parameters to tune and which ranges to tune over. If USEPARAMETERS=STANDARD, this option is ignored.

You can specify one or more of the following *suboptions*:

INBAGFRACTION (LB=*number* UB=*number* VALUES=*value-list* INIT=*number* EXCLUDE)
specifies information about the fraction of the training data to use for each bagged tree while tuning the forest model. For more information, see the INBAGFRACTION= option in the PROC FOREST statement.

You can specify the following additional suboptions:

LB=*number*
specifies the minimum fraction of training data to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=0.1.

UB=*number*
specifies the maximum fraction of training data to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=0.9.

VALUES=*value-list*
specifies a list of fractions of training data to consider during tuning, where *value-list* is a space-separated list of numbers greater than 0 and less than or equal to 1. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial fraction of training data for the tuner to use.

By default, INIT=0.6.

EXCLUDE
excludes the fraction of the training data to use for each bagged tree from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

MAXDEPTH (LB=*number* UB=*number* VALUES=*value-list* INIT=*number* EXCLUDE)
specifies information about the maximum depth to which to grow the trees in the forest. For more information, see the MAXDEPTH= option in the PROC FOREST statement.

You can specify the following additional suboptions:

LB=*number*
specifies a lower bound on the maximum depth to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=1.

UB=*number*
specifies an upper bound on the maximum depth to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=29.

VALUES=*value-list*
specifies a list of values to consider for the maximum depth of the trees in the forest, where *value-list* is a space-separated list of numbers. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial maximum depth of trees in the forest.

By default, INIT=20.

EXCLUDE
excludes maximum depth from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

MINLEAFSIZE (LB=*number* **UB=** *number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**
specifies information about tuning the leaf size option for training the decision tree. This option is not tuned by default. If it is specified without any suboptions, then tuning uses the following values: 1, 5, 10, 20, 40, 80, 160, 320. For more information, see the MINLEAFSIZE= option in the PROC FOREST statement.

You can specify the following additional suboptions:

LB=*number*
specifies the minimum leaf size value to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=1.

UB=*number*
specifies the maximum leaf size value to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=320.

VALUES=*value-list*
specifies a list of leaf size values to consider during tuning, where *value-list* is a space-separated list of positive integers. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial leaf size for the tuner to use.

By default, INIT=5.

EXCLUDE
excludes the leaf size from the tuning process.

NUMBIN (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**
specifies information about the number of bins in which to bin the interval inputs while tuning the decision tree. For more information, see the NUMBIN= option in the PROC FOREST statement.

You can specify the following additional suboptions:

LB=*number*
specifies the minimum number of bins to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=20.

UB=*number*
specifies the maximum number of bins to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=100.

VALUES=*value-list*
specifies a list of numbers of bins to consider during tuning, where *value-list* is a space-separated list of positive integers. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial number of bins for the tuner to use.

By default, INIT=50.

EXCLUDE
excludes the number of bins from the tuning process.

NTREES (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**
specifies information about the number of trees in the forest to use for tuning the forest model. For more information, see the NTREES= option in the PROC FOREST statement.

You can specify the following additional suboptions:

LB=*number*
specifies the minimum number of trees to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=20.

UB=*number*
specifies the maximum number of trees to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=150.

VALUES=*value-list*
specifies a list of numbers of trees to consider during tuning, where *value-list* is a space-separated list of positive integers. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial number of trees for the tuner to use.

By default, INIT=100.

EXCLUDE
excludes the number of trees from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

VARS_TO_TRY (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**
specifies information about the number of variables to consider at each split during tree growth while tuning the forest model. For more information, see the VARS_TO_TRY= option in the PROC FOREST statement.

You can specify the following additional suboptions:

LB=*number*
specifies the minimum number of variables to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=1.

UB=*number*
specifies the maximum number of variables to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=min(n,100), where n is the total number of input variables.

VALUES=*value-list*
specifies a list of numbers of variables to consider during tuning, where *value-list* is a space-separated list of positive integers. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial number of variables for the tuner to use.

By default, INIT is the square root of the number of the input variables, rounded up.

EXCLUDE
excludes the number of variables from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

CODE Statement

CODE < *options* > **;**

The CODE statement writes SAS DATA step code for computing predicted values of the fitted model to a file. To score new data, you can then include the file in a DATA step.

Table 10.4 summarizes the *options* available in the CODE statement.

Table 10.4 CODE Statement Options

Option	Description
COMMENT	Adds comments to the generated code
FILE=	Names the file in which to save the generated code
FORMATWIDTH=	Specifies the numeric format width for the regression coefficients
INDENTSIZE=	Specifies the number of spaces to indent the generated code
LABELID=	Specifies a number used to construct names and labels
LINESIZE=	Specifies the line size for the generated code
NOTRIM	Compares formatted values, including blank padding

For more information about the syntax of the CODE statement, see the section "CODE Statement" on page 20 in Chapter 2, "Shared Concepts."

CROSSVALIDATION Statement

CROSSVALIDATION < *options* > **;**

The CROSSVALIDATION statement performs *k*-fold cross validation to find the average estimated validation error. You cannot specify this statement if you specify the PARTITION statement.

You can specify the following *options*:

KFOLD=*number*
specifies the number of partition folds in the cross validation process, where *number* is between 2 and 20, inclusive. By default, KFOLD=5.

NOPARALLEL
specifies that *k*-fold cross validation not be run in parallel. By default, the process runs in parallel.

FREQ Statement

FREQ *variable* **;**

The *variable* in the FREQ statement identifies a numeric variable in the input data table that contains the frequency of occurrence of each observation. PROC FOREST treats each observation as if it appears f times, where f is the value of the FREQ *variable* for the observation. If f is not an integer, it is truncated to an integer. If f is less than 1 or missing, the observation is not used in the analysis. When you omit the FREQ statement, each observation is assigned a frequency of 1.

GROW Statement

GROW *criterion* **;**

The GROW statement specifies the *criterion* by which to split a parent node into child nodes. As it grows the tree, PROC FOREST calculates the specified *criterion* for each predictor variable and then splits on the predictor variable that optimizes the specified *criterion*.

For categorical responses, the available *criteria* are CHAID, CHISQUARE, ENTROPY, GINI, and IGR; the default is IGR. For continuous responses, the available *criteria* are CHAID, FTEST, and RSS; the default is RSS.

For either categorical or continuous responses, you can specify the following *criterion*:

CHAID

for categorical predictor variables, CHAID uses the value (as specified in the ALPHA= option) of a chi-square statistic (for a classification tree) or an F statistic (for a regression tree) to merge similar levels of the predictor variable until the number of children in the proposed split reaches the number that you specify in the MAXBRANCH= option in the PROC FOREST statement. The p-values for the final split determine the variable on which to split.

For continuous predictor variables, CHAID chooses the best single split until the number of children in the proposed split reaches the value that you specify in the MAXBRANCH= option in the PROC FOREST statement.

For categorical responses only, you can specify the following *criteria*:

CHISQUARE

uses a chi-square statistic to split each variable and then uses the p-values that correspond to the resulting splits to determine the splitting variable.

ENTROPY

GAIN

uses the gain in information (decrease in entropy) to split each variable and then to determine the split.

GINI

uses the decrease in the Gini index to split each variable and then to determine the split.

IGR

uses the entropy metric to split each variable and then uses the information gain ratio to determine the split.

The default *criterion* for categorical responses is IGR.

For continuous responses only, you can specify the following *criteria*:

FTEST

uses an F statistic to split each variable and then uses the resulting p-value to determine the split variable.

RSS

VARIANCE

uses the change in response variance to split each variable and then to determine the split.

The default *criterion* for continuous responses is RSS.

ID Statement

ID *variables* **;**

The ID statement lists one or more *variables* that are to be copied from the input data table to the output data tables that are specified in the OUT= option in the OUTPUT statement and the RSTORE= option in the SAVESTATE statement.

INPUT Statement

INPUT *variables* < / **LEVEL=INTERVAL | NOMINAL** > **;**

The INPUT statement names input *variables* that share a common *option*. You can specify the INPUT statement multiple times.

You can specify the following option after a slash:

LEVEL=INTERVAL | NOMINAL

specifies the level of measurement of the *variables*. You can specify the following values:

INTERVAL specifies that the level of measurement of the *variables* is interval.

NOMINAL specifies that the level of measurement of the *variables* is nominal.

By default, LEVEL=INTERVAL for numeric variables and LEVEL=NOMINAL for categorical variables.

MITIGATEBIAS Statement

MITIGATEBIAS SENSITIVEVAR=*variable* **PREDICTEDVARS=(***variable-names***) PREDICTEDEVENTS=***"event-list"* < *options* > **;**

The MITIGATEBIAS statement enables PROC FOREST to iteratively train a model while minimizing the bias metric that you specify. To use this statement, you must also specify the SAVESTATE statement and a categorical response (target) variable. You cannot also specify the AUTOTUNE statement or the CROSSVALIDATION statement.

You must specify the following options:

PREDICTEDEVENTS="*event-list***"**
PEVENTS="*event-list***"**
specifies the events that correspond to each variable in the PREDICTEDVARS= option. You must specify the PREDICTEDEVENTS= and PREDICTEDVARS= options with the same number of items in the same order. Items in the *event-list* must be separated by the delimiter that you specify in the DELIMETER= option.

PREDICTEDVARS=(*variable-names***)**
PVARS=(*variable-names***)**
specifies the names of the variables that contain the posterior probability for each level in model prediction that corresponds to the response (target) variable. You must specify the PREDICTEDVARS= and PREDICTEDEVENTS= options with the same number of items in the same order. Duplicate variables are not allowed.

SENSITIVEVAR=*variable*
specifies the sensitive variable to use in order to reduce the value of the bias measurement that you specify in the BIASMETRIC= option.

You can also specify the following *options*:

BIASMETRIC=*bias-metric*
specifies the type of bias measurement. You can specify the following *bias-metrics*:

DEMOGRAPHICPARITY specifies demographic parity as the bias measurement.

EQUALIZEDODDS specifies equalized odds as the bias measurement.

EQUALOPPORTUNITY specifies equal opportunity as the bias measurement.

PREDICTIVEPARITY specifies predictive parity as the bias measurement.

By default, BIASMETRIC=DEMOGRAPHICPARITY.

BOUND=*number*
specifies the bound value for the exponentiated gradient reduction algorithm.

By default, BOUND=100.

DELIMITER="*character***"**
DLM="*character***"**
specifies the delimiter to be used to separate events that you specify in the PREDICTEDEVENTS= option. A delimiter is used when event names contain embedded spaces (such as “Fuel Oil”) or special characters (such as “;” or “*”). You must use quotation marks around *character*. You can specify only one delimiter; combinations of delimiters are not supported. The valid values of the DELIMITER= option are “ ” (space), “;” (semicolon), “*” (asterisk), “.” (period), and “,” (comma).

By default, DELIMITER=“ ” (space).

If you specify a delimiter other than “ ” (space), PROC FOREST treats any spaces (including empty spaces) as valid characters. You are responsible for mapping events correctly in order to obtain correct results.

LEARNINGRATE=*number*
specifies the step size to use in updating the exponentiated gradient reduction algorithm.

By default, LEARNINGRATE=0.01.

LOGLEVEL=0 | 1 | 2
specifies the level of log information to print.

0 prints only warnings and errors.

1 prints some notes.

2 prints notes for each iteration.

By default, LOGLEVEL=1.

MAXITER=*number*
specifies the maximum number of iterations to run the exponentiated gradient reduction algorithm.

By default, MAXITER=10.

SEED=*number*
specifies the seed for the pseudorandom number generator. By default, the random number stream is based on the time of day from the computer's clock. A 0 or negative seed value also results in random number streams that are based on the time of day from the computer's clock. If you want a reproducible random number sequence between runs, specify a value greater than 0.

By default, SEED=0.

TARGETEVENT="*event***"**
specifies the formatted value of the response (target) variable that represents the event of interest.

By default, *event* is the first level that you specify in the PREDICTEDEVENTS= option.

TOLERANCE=*number*
specifies the parity constraint violation tolerance. If you set *number* to 0, PROC FOREST trains for the maximum number of iterations, which you specify in the MAXITER= option.

By default, TOLERANCE=0.005.

TUNEBOUND
specifies that the bound value must be tuned.

OUTPUT Statement

OUTPUT OUT=*libref.data-table* < *option* > **;**

The OUTPUT statement creates an output data table that contains the results of PROC FOREST.

You must specify the following option:

OUT=*libref.data-table*
names the output data table for PROC FOREST to use. You must specify this option before any other options. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to where the data table is to be stored. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 215.

data-table specifies the name of the output data table.

You can also specify the following *options*:

COPYVAR=*variable*
COPYVARS=(*variables***)**
lists one or more *variables* from the input data table to be transferred to the output data table.

ROLE< =*name* **>**
generates a numeric variable that indicates the role played by each observation in fitting the model. By default, the variable is named _ROLE_. You can add an optional *=name* to change the name of this generated variable. For each observation, the interpretation of this variable is shown in Table 10.5.

Table 10.5 Role Interpretation

Value	Observation Role
0	Not used
1	Training
2	Validation
3	Testing

If you do not partition the input data by using a PARTITION statement, then the value of the role variable is 1 for all observations.

PARTITION Statement

PARTITION *partition-option* **;**

The PARTITION statement specifies how observations in the input data set are logically partitioned into disjoint subsets for model training, validation, and testing. For more information, see the section "Using Validation and Test Data" on page 23 in Chapter 2, "Shared Concepts." Either you can designate a variable in the input data table and a set of formatted values of that variable to determine the role of each observation, or you can specify proportions to use for randomly assigning observations to each role.

You must specify exactly one of the following *partition-options*:

FRACTION(< TEST=*fraction* **> < VALIDATE=***fraction* **> < SEED=***number* **>)**
randomly assigns specified proportions of the observations in the input data table to the roles. You specify the proportions for testing and validation by using the TEST= and VALIDATE= suboptions. If you specify both the TEST= and VALIDATE= suboptions, then the sum of the specified fractions must be less than 1 and the remaining fraction is the proportion of the observations assigned to the training role. The SEED= option specifies an integer that is used to start the pseudorandom number generator for random partitioning of data for training, testing, and validation. If you do not specify SEED=*number* or if *number* is less than or equal to 0, the seed is generated by reading the time of day from the computer's clock.

ROLE=*variable* **(< TEST='***value***' > < TRAIN='***value***' > < VALIDATE='***value***' >)**
ROLEVAR=*variable* **(< TEST='***value***' > < TRAIN='***value***' > < VALIDATE='***value***' >)**
names the *variable* in the input data table whose values are used to assign roles to each observation. This *variable* cannot also appear as an analysis variable in other statements or options. The TEST=, TRAIN=, and VALIDATE= suboptions specify the formatted values of this variable that are used to assign observation roles. If you do not specify the TRAIN= suboption, then all observations whose role is not determined by the TEST= or VALIDATE= suboption are assigned to the training role.

SAVESTATE Statement

SAVESTATE RSTORE=*libref.data-table* **;**

The SAVESTATE statement creates an analytic store for the model and saves it as a binary object in a data table. You can use the analytic store in the ASTORE procedure to score new data. For more information, see Chapter 4, "ASTORE Procedure."

You must specify the following option:

RSTORE=*libref.data-table*
specifies a data table in which to save the analytic store for the model. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 215.

TARGET Statement

TARGET *variable* < / **LEVEL=NOMINAL | INTERVAL** > **;**

The TARGET statement names the *variable* whose values PROC FOREST tries to predict.

You can specify the following option:

LEVEL=NOMINAL | INTERVAL
specifies the level of measurement. You can specify the following values:

NOMINAL specifies that the level of measurement of the variables is nominal.

INTERVAL specifies that the level of measurement of the variables is interval.

By default, LEVEL=INTERVAL for numeric variables and LEVEL=NOMINAL for categorical variables.

VIICODE Statement

VIICODE < *options* > **;**

The VIICODE statement writes SAS DATA step code to a file or to a catalog entry. The SAS DATA step code creates new variables on the basis of the detected variable interactions.

You can specify the following *options* in addition to all options in the CODE statement. For more information about the syntax of the CODE statement, see the section "CODE Statement" on page 20 in Chapter 2, "Shared Concepts."

ADD
requests that the newly created variables be of the form $V + W$.

LIMIT=*number*
specifies the maximum *number* of new variables to create. By default, LIMIT=200.

MISS
requests that the generated code handle missing values.

MULTIPLY
requests that the newly created variables be of the form $V \times W$.

SUBTRACT
requests that the newly created variables be of the form $V - W$.

THRESHOLD=*number*
requests that interactions with an importance less than *number* times the maximum interaction importance be ignored, where *number* must be between 0 and 1. By default, THRESHOLD=0.0001.

If the VII= option is not specified in the PROC FOREST statement, then the VIICODE statement is ignored. For more information about variable interaction importance, see the section "Variable Interaction Importance" on page 249.

WEIGHT Statement

WEIGHT *variable* **;**

The *variable* in the WEIGHT statement is used as a weight to perform a weighted analysis of the data. Observations that have nonpositive or missing weights are not included in the analysis. The values of the WEIGHT variable can be nonintegral and are not truncated. When you omit the WEIGHT statement, all observations that are used in the analysis are assigned a weight of 1.

Details: FOREST Procedure

Bagging the Data

A decision tree in a forest trains on new training data that are derived from the original training data presented to the FOREST procedure. Using different data to train different trees reduces the correlation of the predictions of the trees, which in turn should improve the predictions of the forest.

The FOREST procedure samples the original data with replacement to create the training data for an individual tree. The convention of sampling with replacement originated with Leo Breiman's bagging algorithm (Breiman 1996, 2001). The word *bagging* stems from "bootstrap aggregating," where "bootstrap" refers to a process that uses sampling with replacement. Breiman refers to the observations that are excluded from the sample as *out-of-bag* (OOB) observations. Therefore, observations in the training sample are called the *bagged* observations, and the training data for a specific decision tree are called the *bagged data*.

The INBAGFRACTION= option in the PROC FOREST statement specifies the number of observations to sample with replacement into the bagged data. The bagging process takes into account the frequency variable that you specify in the FREQ statement.

Estimating the goodness of fit of the model by using the training data is usually too optimistic; the fit of the model to new data is usually worse than the fit to the training data. Estimating the goodness of fit by using the out-of-bag data is usually too pessimistic at first. With enough trees, the out-of-bag estimates are an unbiased estimate of the generalization fit.

Training a Decision Tree

The FOREST procedure trains a decision tree by splitting the bagged data, then splitting each of the resulting segments, and so on recursively until some constraint is met.

Splitting involves the following subtasks:

1. selecting candidate inputs
2. computing the association of each input with the target
3. searching for the best split that uses the most highly associated inputs

PROC FOREST randomly selects m candidate input variables independently in every node, where m is the value of the VARS_TO_TRY= option in the PROC FOREST statement. If you specify L as the value of the LOH= option and $L < m$, then PROC FOREST chooses the best L input variables from the m variables according to the criterion described in section "Loh Method" on page 244. A split search is performed on all L or m variables, and the best rule is kept to split the node.

The reason for searching fewer input variables for a splitting rule instead of searching all inputs and choosing the best split is to improve prediction on new data. An input that offers more splitting possibilities provides the search routine more chances to find a spurious split. Loh and Shih (1997) demonstrate the bias towards spurious splits that result. They also demonstrate that preselecting the input variable and then searching only on that one input reduces the bias. You can choose to preselect a number of input variables by using the LOH= option.

The split search seeks to maximize the reduction in the gain for a nominal target and the reduction in variance of an interval target.

Splitting Nominal Values

Methods

PROC FOREST evaluates a nominal variable in a node in one of three ways:

- enumerate: evaluate all possible splits
- sort: evaluate splits that preserve a sort order of the values
- reject: reject the variable without considering any splits

The number of categories K that are present in a node of an input variable X determines the choice. A different choice might be used for a different variable in the same node, or for the same variable in a different node because K might be different in different nodes. Let s denote the value of the SORT= suboption of the NOMSEARCH option, and let c denote the value of the MAXCATEGORIES= suboption. The following rules determine which choice PROC FOREST uses:

- Enumerate when $K \leq \min(15, s, c)$.
- Sort when $\min(s, 16) \leq K < c$.
- Reject the variable when $c \leq K$.

The enumerate choice examines all possible splits. The number of splits grows exponentially with the number of categories K.

The sort choice first sorts the categories and then finds the best split that preserves the sort. The sort is on z_k, which is defined as

$$z_k = \frac{\Sigma_{i \text{ in } k}\, g_i + r\bar{g}}{n_k + r}$$

where

- k is a category
- Σ sums over observations in category k
- g_i is either the interval target value or the majority target class indicator
- r is the NOMSEARCH(SHRINKAGE=) suboption value
- $\bar{g}$ is the average of g_i across all categories
- n_k is the number of observations in category k

For a binary or nominal target, g_i is 1 if observation i is of the majority class in the node, and 0 otherwise.

Risks of Many Categories

Algorithms that search for a splitting rule preferentially select nominal inputs that contain lots of categories because of the large number of ways the categories can be combined to match noise in the training data. Irrelevant nominal inputs commonly supplant predictive inputs, creating spurious splits that can degrade accuracy on new data.

Memory depletion is another risk, especially in the output model table. The branch assignment of every category is stored. The OUTMODEL= table contains a row for every node in the model and a column for every category assigned to that node. The table contains enough columns to accommodate the node with the most category assignments, even if only one node requires it. A single nominal input that contains lots of categories can create a model so large as to exhaust memory. The MAXCATEGORIES=*number* suboption prevents splitting a node with a nominal input that has more than *number* categories in the node. An input with more than *number* categories in one node and fewer than *number* categories in a second node is eligible for splitting the second node but not the first.

Scoring can take noticeably longer when a splitting rules include many categories.

Loh Method

Specify the LOH= option to use ideas developed by Loh in a series of papers (Loh and Shih 1997; Loh 2002, 2009).

This method selects the number of variables that have the smallest p-value of a chi-square test of association in a contingency table. These variables are selected from the VARS_TO_TRY=*m* randomly selected variables that are chosen for a single decision tree in the forest.

Let Y and X denote the target variable and input variable, respectively. Let Y_i and X_i denote their values in observation i. If Y is categorical, let J denote the number of values. Similarly, if X is categorical, let K denote the number of values.

If both Y and X are categorical, then form the $J \times K$ contingency table of the frequencies of the observations and compute the p-value. If Y has an interval measurement level, then note whether Y_i is greater than or less than the average of Y, $\bar{Y}$, in the node, and then form the $2 \times K$ table of frequencies and compute the p-value.

If X has an interval measurement level, then let

$$K = \begin{cases} 3 & \text{if } N < 20J \\ 4 & \text{otherwise} \end{cases}$$

where N is the number of observations in the calculations and $J = 2$ if Y has an interval measurement level. If $K = 3$, assign X_i to a table column that is defined by the following boundary points:

$$\begin{aligned}\xi_1 &= \bar{X} - \sqrt{3}\hat{\sigma}/3 \\ \xi_2 &= \bar{X} + \sqrt{3}\hat{\sigma}/3\end{aligned}$$

Otherwise, use the boundary points

$$\begin{aligned}\xi_1 &= \bar{X} - \sqrt{3}\hat{\sigma}/2 \\ \xi_2 &= \bar{X} \\ \xi_3 &= \bar{X} + \sqrt{3}\hat{\sigma}/2\end{aligned}$$

where $\bar{X}$ denotes the average value of X and

$$\hat{\sigma}^2 = \frac{\sum_i (X_i - \bar{X})^2}{N}$$

Predicting an Observation

To predict an observation, the FOREST procedure first assigns the observation to a single leaf in each decision tree in the forest, then uses that leaf to make a prediction based on the tree that contains the leaf, and finally simply averages the predictions over the trees. For an interval target, the prediction in a leaf equals the average of the target values among the bagged training observations in that leaf. For a nominal target, the posterior probability of a target category equals the proportion of that category among the bagged training observations in that leaf. The predicted nominal target category is the category that has the largest posterior probability. In case of a tie, the first category that occurs in the training data is the prediction.

Measuring Prediction Error

The FOREST procedure computes the average square error measure of prediction error. For a nominal target, PROC FOREST also computes the misclassification rate and the log-loss.

The average square error for an interval target, the average square error for a nominal target, the misclassification rate, and the log-loss are defined, respectively, as

$$\mathrm{ASE_{int}} = \sum_{i=1}^{N} \frac{(y_i - \hat{y}_i)^2}{N}$$

$$\mathrm{ASE_{cat}} = \sum_{i=1}^{N} \sum_{j=1}^{J} \frac{(\delta_{ij} - \hat{p}_{ij})^2}{JN}$$

$$\mathrm{Misc} = \sum_{i=1}^{N} \frac{1(y_i \neq \hat{y}_i)}{N}$$

$$\mathrm{LogLoss} = -\sum_{i=1}^{N} \sum_{j=1}^{J} \frac{\delta_{ij} \log(\ddot{p}_{ij})}{N}$$

where $\hat{y}_i$ is the target prediction of observation i, δ_{ij} equals 1 if the nominal target value j occurs in observation i or 0 of it does not, $\hat{p}_{ij}$ is the predicted probability of nominal target value j for observation i, N is the number of observations, J is the number of nominal target values (classes), and $\ddot{p}_{ij}$ is $\hat{p}_{ij}$ truncated away from 0 and 1:

$$\ddot{p_{ij}} = \max(\min(\hat{p_{ij}}, 1 - 10^{-10}), 10^{-10})$$

The definitions are valid whether $\hat{y}_i$ is the usual model prediction or the out-of-bag prediction. The ASE_{int} that is based on the usual model predictions of the original training data is usually optimistic and smaller than what its value will be on future data.

Handling Missing Values

Strategies

Tree-based models use observations that have missing input values. The FOREST procedure offers the following strategies for handling missing values:

- The simple strategy is to regard a missing value as a special nonmissing value. For a nominal input, a missing value simply constitutes a new categorical value. For an input whose values are ordered, each missing value constitutes a special value that is assigned a place in the ordering that yields the best split. That place is usually different in different nodes of the tree.

  This strategy is beneficial when missing values are predictive of certain target values. For example, people who have large incomes might be more reluctant to disclose their income than people who have ordinary incomes. If income were predictive of a target, then a missing income value would be predictive of the target and the missing values would be regarded as a special large-income value. The strategy seems harmless when the distribution of missing values is uncorrelated with the target because no choice of branch for the missing values would help predict the target.

  A linear regression could use the same strategy by adding binary indicator variables to designate whether a value is missing. Alternatively, and much more commonly, a linear regression could simply remove observations in which any input is missing. Let p denote the probability that a variable value is missing, and let v denote the number of input variables. The probability that an observation has one or more missing values is $1 - (1 - p)^v$ (assuming missingness is independent and identically distributed among the inputs). If $p = 0.1$ and $v = 10$, then 65% of the observations would have missing values and would be removed from linear regression.

- The alternative strategy for decision trees is to exclude from the search algorithm any observations that have a missing value in the single input variable that defines the splitting rule. If $p = 0.1$ and $v = 10$, then only 10% instead of 65% of the observations are excluded. Although this compares favorably with common linear regression, using observations that have missing values might still be better.

Specifics

If the value of a target variable is missing, the observation is excluded from training and from evaluating the model. If the value of an input variable is missing, PROC FOREST uses the missing value as a legitimate value when ASSIGNMISSING=USEINSEARCH (the default value) and the number of observations in which the splitting variable has missing values is at least as large as the value of the MINUSEINSEARCH= option. When ASSIGNMISSING=USEINSEARCH and the number of observations in which the splitting value has missing values is less than the value of the MINUSEINSEARCH= option, the splitting rule assigns observations that have missing values to the largest branch.

If you specify ASSIGNMISSING=NONE, then PROC FOREST ignores training observations in which the input variables have missing values. When observations that have missing values are scored, if ASSIGNMISSING=NONE was used during model training, then observations that have missing values are scored using ASSIGNMISSING=MACSMALL as the default rule.

Handling Values That Are Absent from Training Data

A splitting rule that uses a categorical variable might not recognize all possible values of the variable because some categories might not exist in the training data. Splitting rules assign unseen categorical values to the branch that has the most in-bag training observations.

Measuring Variable Importance

The importance of a variable is the contribution it makes to the success of the model. For a predictive model, success means good prediction. Often the prediction relies mainly on a few variables. A good measure of importance reveals those variables. The better the prediction, the more closely the model represents reality and the more plausible it is that the important variables represent the true cause of prediction. Some people prefer a simple model so that they can understand it. However, a simple model usually relinquishes details of reality. Sometimes it is better to first find a good model and then ask which variables are important than to first ask which model is good for variable importance and then train that model.

Van der Laan (2006) asks whether a predictive model is appropriate at all. He believes that if variable importance is your goal, then you should predict importance directly instead of fitting a model. If your goal is to select suspicious genes for further study in a laboratory or to find variables in an industrial process that might influence the quality of the product, then his argument is persuasive. However, the purpose of many predictive models is to make predictions. In these cases, gaining insight into causes can be useful.

Variable importance is also useful for selecting variables for a subsequent model. The comparative importance between the selected variables does not matter. Researchers often seek speed and simplicity from the first model and seek accuracy from the subsequent model. Despite this tendency, a forest is often more useful than a simpler regression as a first model when you want interactions because variables contribute to the forest model through interactions.

Several authors have demonstrated that using a forest to first select variables and then using only those variables in a subsequent forest produces a final forest that predicts the target better than only training a forest without the variable selection.

The FOREST procedure implements two methods for computing variable importance, which are described in the following subsections. By default, the variable importance is calculated by using the change in the

residual sum of square errors. You can request that the FOREST procedure to also calculate the variable importance by random branch assignment (RBA) by specifying the RBAIMP option.

Residual Sum of Squares Importance Method

The residual sum of squares (RSS) for regression trees is defined as

$$\mathrm{RSS} = \sum_{\lambda} \sum_{i \in \lambda} \left(y_i - \hat{y}_{\lambda}^{T} \right)^2$$

where

- i is an observation on leaf λ
- y_i is the predicted value of the response variable of observation i
- $\hat{y}_{\lambda}^{T}$ is the actual value of the response variable on leaf λ

The residual sum of squares (RSS) for classification trees is defined as

$$\mathrm{RSS} = \sum_{\lambda} \sum_{\Phi} N_{\Phi}^{\lambda} \left[\sum_{\tau \neq \Phi} \left(P_{\tau}^{\lambda} \right)^2 + \left(1 - P_{\Phi}^{\lambda} \right)^2 \right]$$

where

- Φ is the actual response level
- N_{Φ}^{λ} is the number of observations on leaf λ that have response level Φ
- P_{τ}^{λ} is the posterior probability for the response level τ on leaf λ
- P_{Φ}^{λ} is the posterior probability for the actual response level Φ on leaf λ

For a single tree in the forest, the RSS-based metric measures variable importance based on the change in RSS when a split is found at a node. The change for variable v is

$$\Delta_{d}^{v} = \mathrm{RSS}_{d} - \sum_{i} \mathrm{RSS}_{i}^{d}$$

where

- d denotes the node
- i denotes the index of a child that this node includes
- RSS_{d} is the RSS if the node is treated as a leaf
- RSS_{i}^{d} is the RSS of the node after it has been split

If the change in RSS is negative (which is possible when you use the validation set), then the change is set to 0.

The RSS-based importance for a single variable, v, in a single tree is then defined as

$$\sum_{d=1}^{D} \Delta_d^v$$

where D is the total number of nodes in which v was used as the splitting variable.

The RSS variable importance for the forest is the average of the RSS variable importance across all trees in the forest.

Variable Interaction Importance

In some cases, interactions of variables are of more interest than a single variable. When you specify the VII= option in the PROC FOREST statement, the procedure computes variable interaction importance as follows.

The two-way interaction importance for variables v and w is

$$\sum_{d,e} \frac{\Delta_d^v + \Delta_e^w}{\Delta_d^v}$$

where

- d and e denote nodes, where node d is the parent of node e
- v and w are variables, where v is the splitting variable for node d and w is the splitting variable for node e
- This summation is across all parent-child node combinations with respect to the variables v and w.

The three-way interaction importance for variables v, w, and x is

$$\sum_{d,e,f} \frac{\Delta_d^v + \Delta_e^w + \Delta_f^x}{\Delta_d^v + \Delta_e^w}$$

where

- d, e, and f denote nodes, where node d is the parent of node e, which is the parent of node f
- v, w, and x are variables, where v is the splitting variable for node d, w is the splitting variable for node e, and x is the splitting variable for node f
- This summation is across all parent-child-grandchild node combinations with respect to the variables v, w, and x.

When comparing variable importance values, you should compare a two-way interaction only with another two-way interaction, or a three-way interaction only with another three-way interaction. Comparing a two-way interaction to a three-way interaction is not meaningful, because the values for the importance are not on the same scale.

In addition to calculating the variable interaction importance, you can also generate SAS DATA step code to create interaction variables and save the code to a file by using the VIICODE statement. When you run the generated SAS DATA step code on the original data or on new data, interaction variables are created.

Created interaction variables are of the form $W + V$, $W - V$, or $W \times V$, where the values used in the calculations come from the variables whose interactions are determined to be important. The values used are as follows:

- For continuous variables, the raw value of the variable is used.
- For categorical variables, the proportion of observations in the training set that is assigned to a specific branch is used.

Random Branch Assignment Importance Method

The random branch assignment (RBA) method computes the importance of an input variable v by comparing how well the data fit the predictions before and after they are modified. Neville and Tan (2014) motivate and introduce the RBA method of variable importance. To modify the predictions, the FOREST procedure replaces all splitting rules that use variable v by a rule that randomly assigns an observation to a branch. The probability of assigning an observation to a branch is proportional to the number of observations that are assigned to the branch in the current data. The current data are the training data when RBA is computed during training. Otherwise, the current data are those being scored on an existing model.

The RBA importance can be expressed mathematically as

$$I_{\mathrm{RBA}}(v) \propto \sum_{i=1}^{n} \mathrm{Loss}(y_i, \ddot{y}_i) - \sum_{i=1}^{n} \mathrm{Loss}(y_i, \hat{y}_i)$$

where $\ddot{y}_i$ is the modified prediction for observation i and $\hat{y}_i$ is the standard prediction.

For an interval target, PROC FOREST computes the RBA importance of squared error loss. For a nominal target, PROC FOREST uses the misclassification rate as the loss function.

Isolation Forests

An isolation forest (Liu, Ting, and Zhou (2008)) is a specially constructed forest that is used for anomaly detection instead of target prediction. When the FOREST procedure creates an isolation forest, it outputs anomaly scores in the scored data table that is specified in the OUTPUT statement.

For each split in an isolation forest, one input variable is randomly chosen. If the variable is an interval variable, then it is split at a random value between the maximum and minimum values of the observations in that node. The random value is sampled from a $\mathrm{Beta}(\beta, \beta)$ distribution. You can specify the value of β in the BETA= suboption of the ISOLATION option in the PROC FOREST statement. If you specify a value of 1, the procedure samples from the uniform distribution instead. If the variable is nominal, then each level of the variable is assigned to a random branch. When the forest is constructed this way, anomalous observations are likely to have a shorter path from the root node to the leaf node than nonanomalous observations have.

The anomaly score, $s(x)$, of observation x is calculated as

$$s(x) = 2^{-h(x)}$$

where $h(x)$ is the average, over all trees, of the length of the path from the root node to the leaf node that contains observation x, divided by the average length of all paths across all trees.

The anomaly score is always between 0 and 1, where values closer to 1 indicate a higher chance of the observation being an anomaly.

For better performance in terms of speed and memory, PROC FOREST determines the split variables and branch assignments all at once. This behavior has the potential drawback that some nodes of the forest might have splits that do not assign any data to one of the branches. This can produce smaller trees in the isolation forest. To mitigate this problem, you can use the NPASSES= suboption of the ISOLATION option to request that PROC FOREST reevaluate nodes that are split using an interval variable and have no data assigned to a branch. The number of times that the procedure reevaluates nodes is equal to the NPASSES= suboption value minus 1.

Quantile Regression Forest

A quantile regression forest extends the standard random forest to estimate conditional quantiles of the target variable, rather than only the conditional mean. During training, PROC FOREST stores additional information about the distribution of target values at each leaf node. During scoring, this stored information is used to estimate the requested quantiles.

The QUANTILEREGRESSION option in the PROC FOREST statement specifies that the procedure train a quantile regression forest and specifies the quantile values to compute during scoring. The QUANTILES() suboption of the QUANTILEREGRESSION option specifies which quantile values to compute during scoring.

Quantile regression forests require an interval target. If the target is nominal, the procedure issues an error and stops processing.

The METHOD= suboption specifies how to store the target value distribution at each leaf. Different methods offer different trade-offs between memory usage and accuracy:

- The EXACT method stores all target values per leaf for exact quantile computation, but it requires the most memory and can take noticeably longer to score. For large data tables, this method might exhaust available memory.
- The FAST method stores a single random value per leaf and uses minimal memory. It is suitable for rough median estimation.
- The HYBRID method combines exact tail storage with a compact body representation by using anchor points, providing a good balance between accuracy and memory usage. This is the default method. It requires at least one tail to be nonzero; if you set both the KLOW= and KHIGH= suboptions to 0, both are reset to their default values.
- The TAIL method stores extreme target values per leaf to ensure accurate estimation of tail quantiles. Quantiles that fall outside the effective range of stored values are set to missing.
- The UNIFORM method uses reservoir sampling to store a representative sample of target values per leaf.

You cannot use the QUANTILEREGRESSION option with the ISOLATION option or with a MAXBRANCH= option value greater than 2. If either combination is present, PROC FOREST issues an error and stops processing.

The number of target values available for quantile estimation varies by leaf node. The MAXDEPTH= option affects this number. Increasing its value generally produces more leaf nodes with fewer training observations per leaf.

For the HYBRID method, if a leaf contains fewer target values than the sum of the KLOW=, KHIGH=, and NANCHORS= suboption values, PROC FOREST stores all target values for that leaf. Because the leaf already contains few target values, the resulting storage for that leaf is small and does not contribute significantly to memory use.

When you select a quantile storage method and its suboption values, be sure to consider the expected number of training observations per leaf. Larger values of the KLOW=, KHIGH=, and NANCHORS= suboptions enable you to retain more information about the target distribution and can improve quantile estimation accuracy, but they also require additional memory and can increase scoring time. Smaller values reduce both memory use and scoring time, but they can reduce the accuracy of quantile estimates at the tails of the distribution.

When you score a quantile regression forest by using the ASTORE procedure, you can specify the quantile values to compute by using the QUANTILES option in the SETOPTION statement.

Hyperparameter Tuning

The options that control the training algorithm are often called hyperparameters to distinguish them from the parameters of the trained model. The main parameters in tree-based models are the splitting rules. Optimal hyperparameter settings depend on the data and are never known for certain. Nevertheless, here are some recommendations:

BINMETHOD=QUANTILE Using quantile bins instead of bins of equal width often results in a better fit, especially when some inputs have outliers.

NUMBIN= Using more bins sometimes improves and rarely degrades accuracy. However, training takes longer with more bins, often much longer with many more bins.

For more information about hyperparameter tuning, see the section "Hyperparameter Tuning" on page 24 in Chapter 2, "Shared Concepts."

You can use the AUTOTUNE statement to tune the following options in the PROC FOREST statement:

- MAXDEPTH= option for the maximum depth of trees to grow in the forest
- NTREES= option for the number of trees to grow in the forest
- INBAGFRACTION= option for the bootstrap sample size for building each tree in the forest
- VARS_TO_TRY= option for the number of variables to randomly select at each node split for each tree in the forest

k-fold Cross Validation

The CROSSVALIDATION statement performs *k*-fold cross validation to assess the accuracy of a model. During cross validation, all data are divided into *k* subsets (folds), where *k* is the value of the KFOLD= option. For each fold, a new model is trained on the (*k*–1) folds and then validated using the selected (holdout) fold. The assessment metrics are then averaged across all the holdout folds. The CROSSVALIDATION statement creates a table that has *k*+1 rows. The first *k* rows contain the assessment metrics for each holdout fold, and the last row contains the average across all the holdout folds. Which metrics the table includes depends on whether the target is nominal or interval. For nominal targets, the table includes the values of square error, mean consequential error, and multiclass log loss. For interval targets, the table includes the values of square error, absolute error, and square logarithmic error.

TreeSHAP Values

TreeSHAP values are used to explain the contribution of each variable to the prediction of a single observation. The TreeSHAP algorithm is used by the Shapley value approximation method for tree-based models that was introduced by Lundberg et al. (2020). Compared to the algorithms that other model-agnostic Shapley value approximation methods use, the TreeSHAP algorithm is faster because it uses the specific structures of tree-based models.

When you use PROC FOREST to train a model, you can save the trained model in an analytic store by using the SAVESTATE statement. You can then use the ASTORE procedure to calculate the TreeSHAP values when you score new data by using this saved analytic store. To enable the calculation of TreeSHAP values in PROC ASTORE, you must set the TREESHAP option value to 1 in the SETOPTION statement. See Chapter 4, "ASTORE Procedure," for more information about the SETOPTION statement.

NOTE: Currently, TreeSHAP values are available only for models that have a maximum of two splits per node. Additionally, the TreeSHAP algorithm is available only if you use the average probability as the voting strategy. By default, PROC ASTORE uses the voting strategy that you specify at training. However, you can change the voting strategy in PROC ASTORE to the average probability by setting the VOTETYPE option value to 1 in the SETOPTION statement. Isolation forests do not support TreeSHAP value calculations.

Bias Mitigation

With the widespread use of AI and machine learning models in decision-making processes, there is concern about the social impacts of these models. Bias can be embedded in both the training data and the trained model, either of which can lead to unfair or biased predictions for certain groups or populations. To reduce bias in the data and models, researchers have developed three main categories of bias mitigation methods:

- Preprocessing methods, which transform the data prior to model training. These methods try to eliminate the correlations between the input features and the sensitive feature while retaining as much information from the input data as possible.
- In-process methods, which consider fairness constraints during the model training process. These methods actively adjust the model parameters during training to create a model that produces fair predictions and classifications.

- Postprocessing methods, which do not directly alter the input data or the model training process. These methods adjust the predictive outputs from the learned models to compensate for the bias in the model.

You can reduce bias in the trained model by specifying the MITIGATEBIAS statement. This statement uses the exponentiated gradient reduction (EGR) algorithm, which is an in-process method of mitigating bias.

Exponentiated Gradient Reduction (EGR) Algorithm

The exponentiated gradient reduction (EGR) algorithm was introduced at ICML 2018 (Agarwal et al. 2018). As an in-process algorithm, it considers fairness constraints during the model training process. The EGR algorithm has three main advantages. First, it is model-agnostic, which means that you can apply it to a wide variety of machine learning models. The EGR algorithm works like a wrapper that is built around existing machine learning models. The wrapper iteratively reweights and possibly relabels the data, and then it trains a new classifier model on the basis of reweighted data in each iteration. By doing so, it reduces the complex optimization problem of training a machine learning model with fairness constraints to a series of standard model training problems without the fairness constraints. Second, the EGR algorithm supports various fairness measurements, such as demographic parity, equal opportunity, and equalized odds. Finally, the algorithm has strong theoretical support in terms of its optimization trade-off between fairness and accuracy. In theory, the EGR algorithm returns solutions that satisfy all the fairness constraints and also achieve the optimal error rate of the best fair solution.

Training a fair machine learning model is in general a nonlinear programming problem. In nonlinear optimization tasks, it is not uncommon to get suboptimal solutions because the constraints are infeasible or because the algorithm is stuck in a local minimum. As with many other machine learning algorithms, the effectiveness of the EGR algorithm depends on many factors, such as the quality of the input data, the selection of fairness constraints, the quality of the machine learning model that it wraps around, and the values of the hyperparameters.

PROC FOREST supports bias mitigation for multilevel nominal target variables and multilevel nominal sensitive variables. Note that when you include a large number of levels in the sensitive variable, the EGR algorithm searches more complex optimization scenarios; this can result in suboptimal solutions. Therefore, the solutions from this algorithm might not always be desirable when the number of levels in the sensitive variable is high. You should review and validate your sensitive variable, training models, and fairness constraints before applying the EGR algorithm.

Fairness Constraints

PROC FOREST supports the following fairness measurements: demographic parity, predictive parity, equal opportunity, and equalized odds. The goal of the EGR algorithm is to bring the fairness metrics of groups as close together as possible, as well as to convert fair machine learning to a sequence of standard cost-sensitive machine learning problems whose solutions yield a randomized classifier with the lowest error rate, subject to the desired constraints. As a result, the EGR algorithm iteratively trains the machine learning models by using different weights to reduce the fairness disparities without significantly increasing the loss of the final machine learning model. For example, when there are two protected groups in the sensitive variable, the fairness constraints that the EGR algorithm requires for demographic parity are as follows:

$$\mu_1^t - \mu_0^t \leq c$$
$$\mu_1^t - \mu_0^t \geq -c$$
$$\mu_2^t - \mu_0^t \leq c$$
$$\mu_2^t - \mu_0^t \geq -c$$

where μ_1^t is the demographic parity for the first protected group during iteration t; μ_2^t is the demographic parity for the second protected group during iteration t; μ_0^t is the demographic parity for the entire population during iteration t; and c is the predefined violation tolerance threshold.

Exponentiated Gradient Reduction Algorithm Workflow

Two key hyperparameters in the EGR algorithm are specified by the BOUND= option (denoted by B) and the LEARNINGRATE= option (denoted by lr); both are available in the MITIGATEBIAS statement. To derive the fair machine learning algorithm, the fairness constraints (see the section "Fairness Constraints" on page 254) are added to the loss function of the cost-sensitive machine learning model by introducing the Lagrange multipliers $\lambda_{j,t}$, where j is the index in the constraint set. The multipliers $\lambda_{j,t}$ are updated on the basis of the degree of constraint violations $\gamma_{j,t}$, B, and lr. Then $\lambda_{j,t}$ is used to derive the new weights for each observation in the input data set for the next round of training. Note that the values of B and lr play important roles in the performance of the EGR algorithm, based on experiments. The next section discusses how to come up with appropriate values for B and lr.

Bound Value Recommendation

Because the EGR algorithm is designed to strike a balance between model accuracy and fairness constraint violations, it does not perform well unless you specify appropriate values for the BOUND= and LEARNINGRATE= options in the MITIGATEBIAS statement. The BOUND= option is important for the sample reweighting and relabeling process. It controls how heavily the EGR algorithm penalizes fairness constraint violations, and therefore it determines the trade-off between the fairness constraints and the model accuracy. When the bound is too small, there is not enough momentum to drive the model close enough to the saddle points in the solution sets. In this case, the bias metric values of the different subgroups do not approach each other.

The LEARNINGRATE= option controls how quickly the algorithm reduces fairness constraint violations. If the learning rate is too small, the algorithm converges very slowly, resulting in a long training process; if the learning rate is too large, the algorithm becomes unstable and might not converge. In general, the learning rate and the bound value are coupled, and changing one of these values changes the optimal value of the other.

The MITIGATEBIAS statement offers a built-in bound value recommendation function to automatically search for a good candidate bound value when you request it. To eliminate the coupling effect of the bound and the learning rate, PROC FOREST fixes the initial learning rate at 0.01 during the tuning process. The procedure runs the bound value recommendation function if you specify the TUNEBOUND option. This function finds the best bound value through multiple iterations by analyzing the convergence patterns of the fairness metrics. PROC FOREST first runs the EGR algorithm with some initial bound value for a few iterations, while monitoring the convergence pattern for signs of stagnation or overshooting to determine whether the bound value is too small or too large. It then adjusts the bound value until it finds an appropriate value or reaches the maximum number of iterations that are allowed. After the recommendation process, PROC FOREST applies the recommended bound value to the EGR algorithm to train a final model. Note that because of the nonlinear nature of the optimization process, the bound value recommendation is also subject to the problem of local minima. This recommendation should be used as a general guideline for setting the bound value; you might find a better solution by further fine-tuning.

Displayed Output

The FOREST procedure displays the parameters that are used to train the model, the fit statistics of the trained model, and other information. The output is organized into various tables, which are discussed here in order of their appearance.

Model Information

The "Model Information" table contains the settings of the training parameters. This table also contains some basic information about the trees in the resulting forest. This table is produced by default.

Number of Observations

The "Number of Observations" table contains the number of observations that are read from the input data table and the number of observations that are used in the analysis. When you specify the PARTITION statement, the table also indicates the number of observations that are used in each partition. This table is produced by default.

Variable Importance

The "Variable Importance" table displays variable importance based on residual sum of square errors, which is explained in the section "Measuring Variable Importance" on page 247. This table is produced by default.

RBA Variable Importance

The "RBA Variable Importance" table displays variable importance based on the random branch assignment (RBA) method, which is explained in the section "Random Branch Assignment Importance Method" on page 250. This table is produced by the RBAIMP option in the PROC FOREST statement.

Fit Statistics

The "Fit Statistics" table contains statistics that measure the model's goodness of fit. The fit of the model to the data improves as the number of trees in the forest increases. Successive rows in the table contain fit statistics for a forest that has more trees. Fit statistics are described in the section "Measuring Prediction Error" on page 245. This table is produced by default.

Tuner Information

The "Tuner Information" table displays the setup values that the tuner uses. This table is produced by the AUTOTUNE statement.

Tuner Summary

The "Tuner Summary" table displays statistics about the tuning process. This table is produced by the AUTOTUNE statement.

Tuner Timing

The "Tuner Timing" table displays the total time spent on different tasks during tuning. This table is produced by the AUTOTUNE statement.

Best Configuration

The "Best Configuration" table displays the hyperparameters and objective function values for the best configuration. This table is produced by the AUTOTUNE statement.

Tuner Results

The "Tuner Results" table displays the values of the hyperparameters, the objective function for the default configuration (Iteration 0), and up to 10 of the best configurations that are found. This table is produced by the AUTOTUNE statement.

Cross-Validation Fit Statistics

The "Cross-Validation Fit Statistics" table contains the per fold and average assessment metrics of *k*-fold cross validation.

Evaluation History

The "Evaluation History" tables displays the values of the hyperparameters and the objective function for all configurations. This table is produced by the AUTOTUNE statement, either by default or when EVALHISTORY=ALL.

OutputCasTables Table

The OutputCasTables table is a special table that has information about each table that is created. The information for each table consists of the table name, the library in which the table resides, and the number of columns and rows in the table.

Predicted Probability Names

The "Predicted Probability Names" table indicates the names of the created variables in either the score code or the data table that is specified in the OUT= option in the OUTPUT statement. This table is produced when you specify the PRINTTARGET option in the PROC FOREST statement.

Constraint History

The "Constraint History" table lists the values of the various constraints at each iteration of the EGR algorithm. This table is produced by the MITIGATEBIAS statement.

Group Metrics History

The "Group Metrics History" table displays the metrics for accuracy and fairness for each subpopulation over each iteration of the EGR algorithm. This table is produced by the MITIGATEBIAS statement.

Iteration History

The "Iteration History" table displays the values of accuracy and bias metric at each iteration of the EGR algorithm. This table is produced by the MITIGATEBIAS statement.

Mitigation Information

The "Mitigation Information" table displays information about the options that you use for bias mitigation. This table is produced by the MITIGATEBIAS statement.

Timing

The "Timing" table displays the breakdown of how long bias mitigation spent on the various tasks of training models. This table is produced by the MITIGATEBIAS statement.

Weight History

The "Weight History" table displays the weights that are assigned by the EGR algorithm for each level of the response (target) variable and for each subpopulation. This table is produced by the MITIGATEBIAS statement.

ODS Table Names

Each table created by the FOREST procedure has a name associated with it, and you must use this name to refer to the table when you use ODS statements. The names of each table and a short description of the contents are listed in Table 10.6.

Table 10.6 ODS Tables Produced by PROC FOREST

Table Name	Description	Statement	Option
BestConfiguration	Hyperparameters and objective function values for the best configuration	AUTOTUNE	Default
ConstraintHistory	Values of the various constraints at each iteration of the EGR algorithm	MITIGATEBIAS	Default
CrossValidateMLFitStat	Per fold and average assessment metrics of *k*-fold cross validation	CROSSVALIDATION	Default
EvaluationHistory	Values of the hyperparameters and the objective function for all configurations	AUTOTUNE	Default / EVALHISTORY=ALL

Table 10.6 *continued*

Table Name	Description	Statement	Option
FitStatistics	Fit statistics from the model	PROC FOREST	Default
GroupMetricsHistory	Metrics for accuracy and fairness for each subpopulation over each iteration of the EGR algorithm	MITIGATEBIAS	Default
IterationHistory	Values of the model accuracy and bias metric over each iteration of the EGR algorithm	MITIGATEBIAS	Default
MitigationInformation	Information about the options that are used for bias mitigation	MITIGATEBIAS	Default
ModelInfo	Model information	PROC FOREST	Default
Nobs	Number of observations	PROC FOREST	Default
OutputCasTables	See the section "OutputCasTables Table" on page 257	PROC FOREST / OUTPUT	OUTMODEL= / Default
PredName	Predicted name information for interval targets	PROC FOREST	PRINTTARGET
PredProbName	Predicted probability name information for nominal targets	PROC FOREST	PRINTTARGET
PredIntoName	Predicted name information for nominal targets	PROC FOREST	PRINTTARGET
RBAImportance	Random branch assignment variable importance	PROC FOREST	RBAIMP
Timing	Breakdown of how long bias mitigation spent on the various tasks of training models	MITIGATEBIAS	Default
TunerInfo	Setup values used by the tuner	AUTOTUNE	Default
TunerResults	Values of the hyperparameters, the objective function for the default configuration (Iteration 0), and up to 10 best found configurations	AUTOTUNE	Default
TunerSummary	Statistics about the tuning process	AUTOTUNE	Default

Table 10.6 *continued*

Table Name	Description	Statement	Option
TunerTiming	Total time spent on different tasks while tuning	AUTOTUNE	Default
VariableImportance	Residual sum of squares variable importance	PROC FOREST	Default
WeightHistory	Weights that are assigned by the EGR algorithm for each level of the response (target) variable and for each subpopulation	MITIGATEBIAS	Default

ODS Graphics

You can refer to every graph that is produced through ODS Graphics by name. The name of the graph that PROC FOREST generates is listed in Table 10.7.

Table 10.7 Graph Produced by PROC FOREST

ODS Graph Name	Plot Description	Statement	Option
IterationHistoryPlot	Iteration history of the model accuracy and bias metric	MITIGATEBIAS	Default

Examples: FOREST Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

Example 10.1: Scoring New Data by Using a Previous Forest Model

This example illustrates how you can use the OUTMODEL= option to save a model table, and later use the model table to score a data table. It uses the JunkMail data set in the Sashelp library.

The JunkMail data set comes from a study that classifies whether an email is junk email (coded as 1) or not (coded as 0). The data set contains 4,601 observations with 59 variables. The response variable is a binary indicator of whether an email is considered spam or not. There are 57 predictor variables that record the frequencies of some common words and characters and the lengths of uninterrupted sequences of capital letters in emails.

You can load the Sashelp.JunkMail data set into your SAS library in the following DATA step:

```
data mylib.junkmail;
   set sashelp.junkmail;
run;
```

These statements assume that your SAS libref is named mylib, as in the section "Using SAS Viya Workbench" on page 215, but you can substitute any appropriately defined SAS libref.

The following statements train a forest model and score the training data table. The OUTPUT statement scores the training data and saves the results to a new table named *fit_at_runtime*.

```
proc forest data=mylib.junkmail outmodel=mylib.forest_model seed=12345;
   input Address Addresses All Bracket Business CS CapAvg CapLong
         CapTotal Conference Credit Data Direct Dollar Edu Email
         Exclamation Font Free George HP HPL Internet Lab Labs
         Mail Make Meeting Money Order Original Our Over PM Paren
         Parts People Pound Project RE Receive Remove Semicolon
         Table Technology Telnet Will You Your _000 _85 _415 _650
         _857 _1999 _3D / level = interval;
   target class /level=nominal;
   output out=mylib.score_at_runtime;
   ods output FitStatistics=fit_at_runtime;
run;
```

The preceding statements produce the table shown in Output 10.1.1. The table shows the training and out-of-bag statistics.

Output 10.1.1 Fit Statistics: Fit at Run Time

Fit Statistics						
Number of Trees	OOB Average Square Error	Training Average Square Error	OOB Misclassification Rate	Training Misclassification Rate	OOB Log Loss	Training Log Loss
1	0.0796	0.0651	0.0994	0.0841	0.818	0.516
2	0.0780	0.0543	0.0981	0.0682	0.692	0.236
3	0.0726	0.0491	0.0924	0.0613	0.524	0.189
4	0.0680	0.0469	0.0878	0.0604	0.412	0.172
5	0.0649	0.0457	0.0839	0.0593	0.358	0.165
6	0.0618	0.0442	0.0777	0.0526	0.293	0.161
7	0.0607	0.0441	0.0766	0.0537	0.273	0.162
8	0.0598	0.0435	0.0755	0.0526	0.252	0.161
9	0.0592	0.0431	0.0744	0.0517	0.252	0.161
10	0.0583	0.0426	0.0719	0.0517	0.250	0.159
.	.	.	.	.	.	.
.	.	.	.	.	.	.
.	.	.	.	.	.	.
91	0.0498	0.0378	0.0591	0.0448	0.186	0.151
92	0.0498	0.0378	0.0593	0.0454	0.186	0.151
93	0.0498	0.0378	0.0600	0.0448	0.186	0.151
94	0.0498	0.0378	0.0598	0.0448	0.186	0.150
95	0.0498	0.0378	0.0591	0.0448	0.186	0.151
96	0.0498	0.0378	0.0598	0.0446	0.186	0.151
97	0.0498	0.0379	0.0598	0.0452	0.186	0.151
98	0.0498	0.0379	0.0591	0.0452	0.186	0.151
99	0.0498	0.0379	0.0589	0.0452	0.186	0.151
100	0.0499	0.0379	0.0591	0.0452	0.186	0.151

The following statements use a previously saved model to score new data:

```
proc forest data=mylib.junkmail inmodel=mylib.forest_model;
     output out=mylib.score_later;
     ods output FitStatistics=fit_later;
run;
```

When you specify the INMODEL= option to use a previously created forest model, the fit statistics table no longer shows the out-of-bag statistics, because you are scoring new data. In this example, the scored data are the same as the training data, so you can see that the statistics in Output 10.1.2 match those previously seen in Output 10.1.1.

Output 10.1.2 Fit Statistics: Fit Later

Fit Statistics			
Number of Trees	Average Square Error	Misclassification Rate	Log Loss
1	0.0651	0.0841	0.516
2	0.0543	0.0682	0.236
3	0.0491	0.0613	0.189
4	0.0469	0.0604	0.172
5	0.0457	0.0593	0.165
6	0.0442	0.0526	0.161
7	0.0441	0.0537	0.162
8	0.0435	0.0526	0.161
9	0.0431	0.0517	0.161
10	0.0426	0.0517	0.159
.	.	.	.
.	.	.	.
.	.	.	.
91	0.0378	0.0448	0.151
92	0.0378	0.0454	0.151
93	0.0378	0.0448	0.151
94	0.0378	0.0448	0.150
95	0.0378	0.0448	0.151
96	0.0378	0.0446	0.151
97	0.0379	0.0452	0.151
98	0.0379	0.0452	0.151
99	0.0379	0.0452	0.151
100	0.0379	0.0452	0.151

This example demonstrates that the FOREST procedure can score an input data table by using a previously saved forest model, which was saved using the OUTMODEL= option in a previous procedure run. If you want to properly score a new data table, you must not modify the mylib.forest_model table, because doing so could invalidate the constructed forest model. As with any scoring of new data, the variables that are used in the model creation must be present in order for you to score a new table.

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

Example 10.2: Generating an Isolation Forest

This example illustrates how to create an isolation forest and generate an anomaly score for each observation. The score estimates how atypical the observation is.

The following DATA step generates a data table and loads the table into your SAS library. The table consists of 456 observations in 13 clumps, which are arranged in a spiral that is defined by the two input variables, x1 and x2. The farther the clump is along the spiral curve from the center, the larger the clump. The data also contain a target variable (y) and an identification variable (id).

```
data mylib.spiral;
      keep x1 x2 y id;
      nNow = 200;
      sdNow = 0.2;
      four_pi = 4 * constant("pi");
      call streaminit(3331333);
      y  = 0;
      id = 0;
      do theta = 0 to 12;
          r      = 1.0 - theta / four_pi;
          mu1    = cos(theta);
          mu2    = sin(theta);
          ss     = r*r / (mu1*mu1 + mu2*mu2);
          s      = sqrt(ss);
          mu1    = s*mu1;
          mu2    = s*mu2;
          nNow   = nNow  * 0.7;
          sdNow  = sdNow * 0.75;
          do i = 1 to nNow;
             x1 = rand('normal', mu1, sdNow);
             x2 = rand('normal', mu2, sdNow);
             y  = 1 - y;
             id = id + 1;
             output;
          end;
      end;
run;
```

These statements assume that your SAS libref is named mylib, as in the section "Using SAS Viya Workbench" on page 215, but you can substitute any appropriately defined SAS libref.

The ISOLATION option in the PROC FOREST statement of the following code requests that the procedure create an isolation forest. The OUTPUT statement scores the training data and saves the results to a new table named *score*, which contains the anomaly score and the predictions for each observation. The predictions are unreliable because the trees are generally shallow and are generated by using random splits.

The following code runs the FOREST procedure with the ISOLATION option to generate an isolation forest. The results are saved in the table mylib.score, and all of the input variables are copied to the output table.

```
proc forest data=mylib.spiral isolation seed=12345;
   input x1 x2 /level=interval;
   id id;
   output out=mylib.score copyvars=(_ALL_);
```

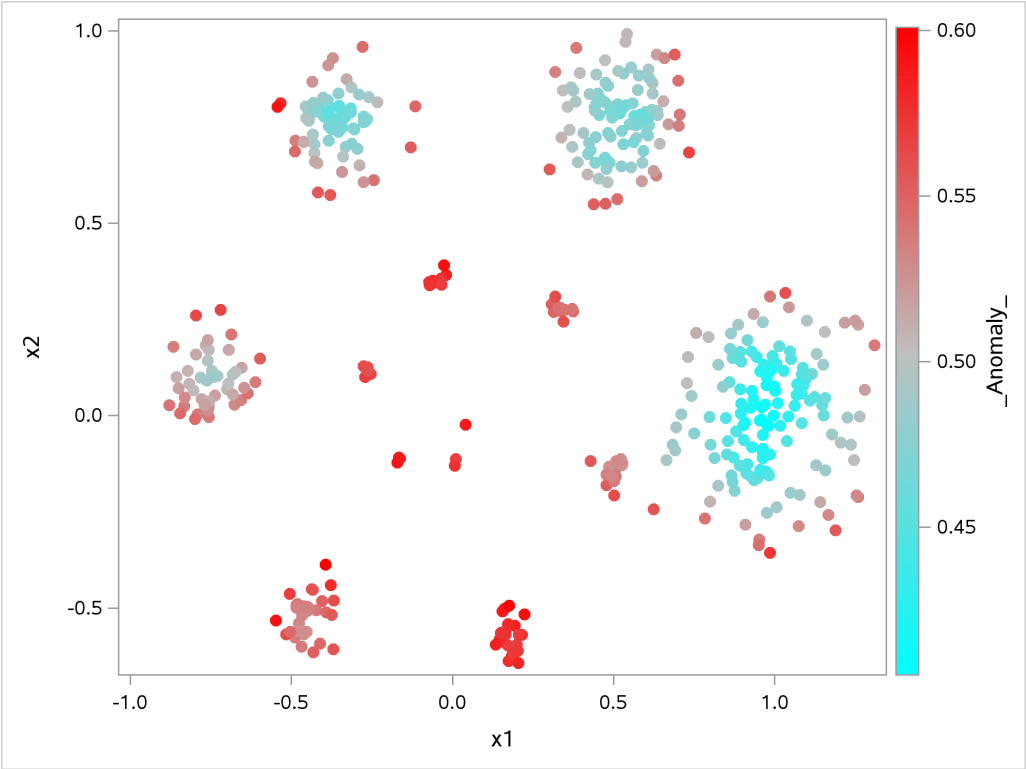
```
run;

proc sort data=mylib.score out=score;
by id;
run;
```

The following code plots the data with a color ramp from cyan to red, based on the anomaly score. The most anomalous observations are colored red.

```
proc template;
    define statgraph anomalyPlot;
        begingraph;
            layout overlay;
            scatterplot y=x2 x=x1 /
               name='color'
               markerattrs=(symbol=circlefilled)
               colormodel=(cyan ligr red)
               colorresponse=_Anomaly_;
               continuouslegend 'color'/ title='_Anomaly_';
            endlayout;
        endgraph;
    end;
run;

proc sgrender data=score template=anomalyPlot;
run;
```

Output 10.2.1 shows that the most anomalous points are either near the exterior boundary of the data or in a small clump in the center.

Output 10.2.1 Spiral Data Colored by Anomaly Score



Example 10.3: Tuning a Forest Model

This example illustrates how you can use the AUTOTUNE statement in the FOREST procedure to tune a forest model. It uses the JunkMail data set in the Sashelp library.

The JunkMail data set comes from a study that classifies whether an email is junk email (coded as 1) or not (coded as 0). The data set contains 4,601 observations with 59 variables. The response variable is a binary indicator of whether an email is considered spam or not. There are 57 predictor variables that record the frequencies of some common words and characters and the lengths of uninterrupted sequences of capital letters in emails.

You can load the Sashelp.JunkMail data set into your SAS library by naming your SAS libref in the first statement of the following DATA step:

```
data mylib.junkmail;
   set sashelp.junkmail;
run;
```

These statements assume that your SAS libref is named mylib, but you can substitute any appropriately defined libref.

The following statements use the AUTOTUNE statement in PROC FOREST to run a tuning process on a forest model:

```
ods graphics on;

proc forest data=mylib.junkmail outmodel=mylib.forest_model seed=12345;
   input Address Addresses All Bracket Business CS CapAvg CapLong
         CapTotal Conference Credit Data Direct Dollar Edu Email
         Exclamation Font Free George HP HPL Internet Lab Labs
         Mail Make Meeting Money Order Original Our Over PM Paren
         Parts People Pound Project RE Receive Remove Semicolon
         Table Technology Telnet Will You Your _000 _85 _415 _650
         _857 _1999 _3D / level = interval;
   target class /level=nominal;
   autotune
      /*
      You do not need to specify any tuning parameters for the default
      tuning process. If you want to make adjustments to the default
      tuning process, uncomment the following block of code and change
      any of the tuning parameters' attributes.

      tuningParameters=(
         INBAGFRACTION (lb=0.1 ub=0.9  init=0.6 )
         MAXDEPTH      (lb=2   ub=30   init=21  )
         MINLEAFSIZE   (values=1 5 10 20 40 80 160 320 init=5 exclude)
         NTREES        (lb=20  ub=150  init=100 )
         NUMBIN        (lb=20  ub=100  init=50  )
         VARS_TO_TRY   (lb=1   ub=56   init=7   )
      )
      */
   ;
   /* Remove this line to see all results */
   ods select BestConfiguration EvaluationHistoryPlot IterationHistoryPlot;
run;
```

The preceding statements produce the table and the plots shown in Output 10.3.1 through Output 10.3.3. The table in Output 10.3.1 displays the evaluation number, the values of the tuning parameters, and the error metric value for the best decision tree model that the tuner found. Note that the ODS SELECT statement limits the displayed results to a single table and two plots. You can remove this statement to display all tables and plots. For the full list of ODS tables that PROC FOREST produces, see Table 10.6.

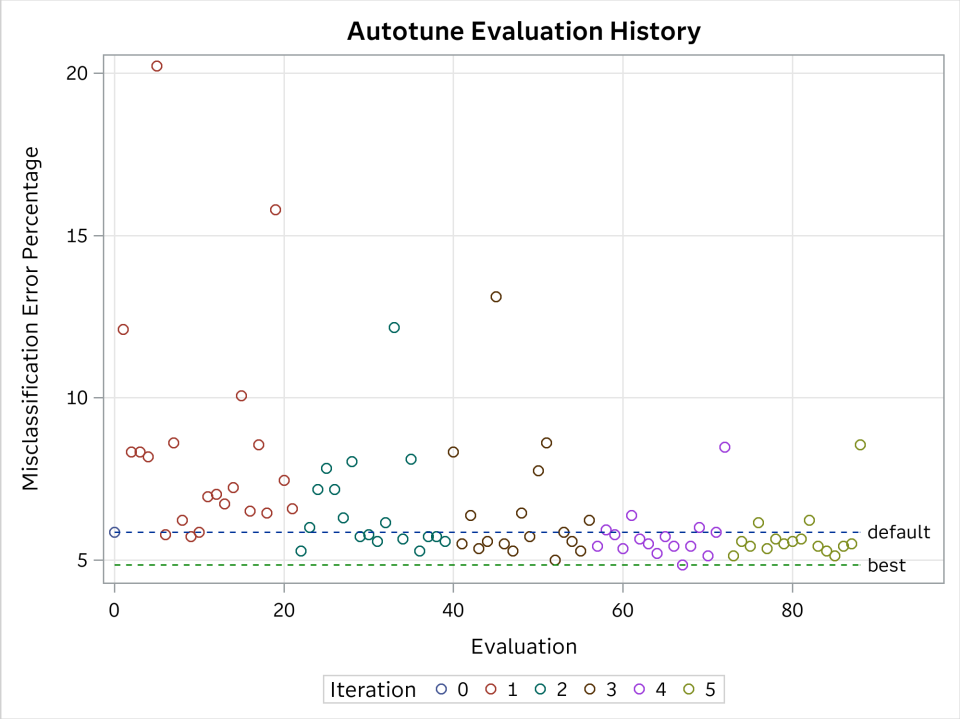
Output 10.3.1 Best Configuration Table

The FOREST Procedure

Best Configuration	
Evaluation	67
Number of Trees	114
Number of Variables to Try	7
Bootstrap	0.9
Maximum Tree Levels	30
Number of Bins	78
Misclassification Error Percentage	4.86

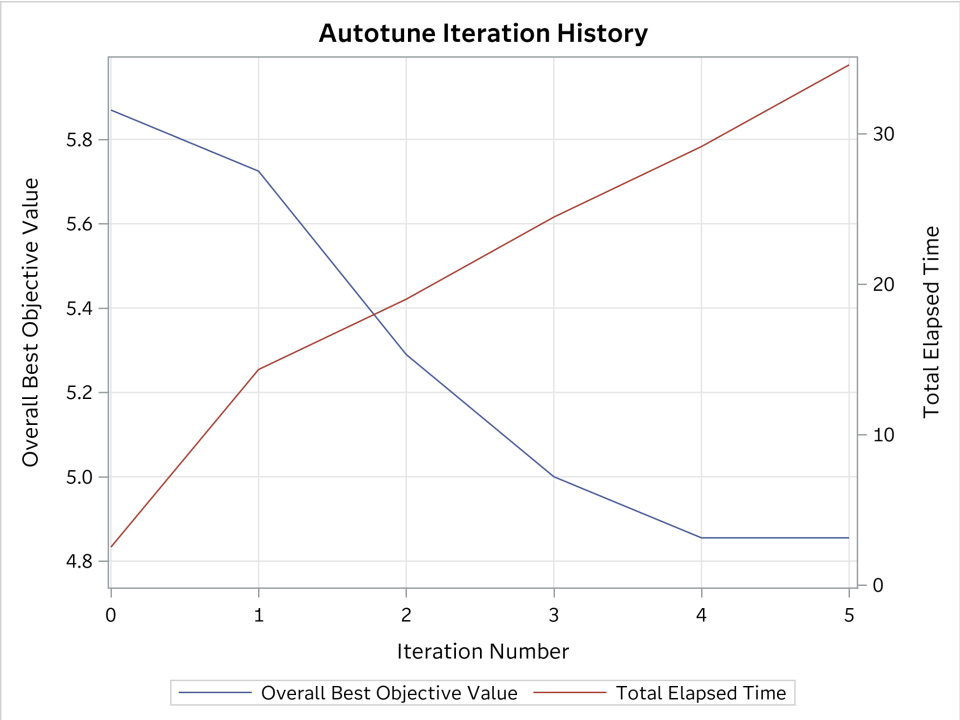
Output 10.3.2 displays a scatter plot of all configurations that the tuner tried. The objective values are shown on the Y axis, and the evaluation numbers are shown on the X axis.

Output 10.3.2 Evaluation History Plot



The plot in Output 10.3.3 displays how the best found objective value and the elapsed time changed with each iteration of the tuner.

Output 10.3.3 Iteration History Plot



NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

Example 10.4: Mitigating Bias of a Classification Forest

This example uses the FOREST procedure to create a forest model while minimizing the demographic parity for the Job variable. The response variable for the forest model is Bad, a classification variable that has two values (0 for payment of loan and 1 for default). The other variables are predictor variables for the model.

The following statements load the data table mylib.hmeq into your SAS library. For this example, the statements assume that your SAS library is named mylib, but you can substitute any appropriately defined SAS library.

```
data mylib.hmeq;
   set sampsio.hmeq;
run;
```

The following statements use PROC FOREST to create a forest model without minimizing the demographic parity:

```
proc forest data=mylib.hmeq nTrees=20;
   input CLAge CLNo DebtInc Derog
         Loan MortDue Value / level=interval;
   input Reason / level=nominal;
   target Bad / level=nominal;
```

```
   output out=mylib.scored copyvars=(_ALL_);
   saveState rstore=mylib.model;
run;
```

The NTREES= option specifies that there are 20 trees to grow in the forest. The two INPUT statements specify CLAge, CLNo, DebtInc, Derog, Loan, MortDue, Value, and Reason as the input variables. The TARGET statement specifies Bad as the target variable. The OUTPUT statement generates the data table mylib.scored, which contains the predicted mortgage default from the model for each observation and contains all variables from the input data table as copy variables. The SAVESTATE statement creates an analytic store for the model and saves it as a binary object in the data table mylib.model.

Output 10.4.1 displays the fit statistics for the forest model.

Output 10.4.1 Fit Statistics Table

The FOREST Procedure

Fit Statistics						
Number of Trees	OOB Average Square Error	Training Average Square Error	OOB Misclassification Rate	Training Misclassification Rate	OOB Log Loss	Training Log Loss
1	0.1126	0.0876	0.150	0.1191	1.190	0.740
2	0.1039	0.0753	0.143	0.1052	0.838	0.315
3	0.0977	0.0710	0.135	0.0965	0.710	0.262
4	0.0941	0.0693	0.128	0.0945	0.609	0.245
5	0.0914	0.0674	0.124	0.0913	0.533	0.227
6	0.0907	0.0669	0.123	0.0928	0.453	0.228
7	0.0894	0.0667	0.122	0.0921	0.397	0.227
8	0.0883	0.0662	0.122	0.0894	0.349	0.226
9	0.0874	0.0655	0.121	0.0878	0.340	0.223
10	0.0870	0.0654	0.122	0.0889	0.328	0.222
11	0.0863	0.0650	0.120	0.0871	0.323	0.221
12	0.0861	0.0645	0.120	0.0869	0.308	0.220
13	0.0858	0.0641	0.120	0.0849	0.304	0.219
14	0.0853	0.0639	0.120	0.0847	0.299	0.218
15	0.0849	0.0633	0.120	0.0844	0.297	0.217
16	0.0848	0.0631	0.119	0.0837	0.297	0.216
17	0.0847	0.0631	0.121	0.0844	0.296	0.217
18	0.0848	0.0633	0.121	0.0854	0.296	0.217
19	0.0845	0.0631	0.119	0.0832	0.296	0.217
20	0.0844	0.0628	0.118	0.0829	0.295	0.216

After you train a classification forest model, you might want to view the demographic parity of the trained model. You can do this by running the ASSESSBIAS procedure. For more information about PROC ASSESSBIAS, see Chapter 3, "ASSESSBIAS Procedure."

The following statements use PROC ASSESSBIAS to assess the bias measurements of the model:

```
proc assessbias data=mylib.scored;
   target bad / event="1" level=nominal;
   var P_Bad1;
   fitstat pvar=P_BAD0 / pevent="0";
   sensitiveVar Job;
```

```
run;
```

Output 10.4.2 displays the bias metric values of the model. PROC ASSESSBIAS also produces other output not shown here.

Output 10.4.2 Bias Metrics Table

The ASSESSBIAS Procedure

Bias Metrics					
Bias Statistic	**Bias Statistic Label**	**Bias Statistic Value**	**Base Level**	**Compare Level**	**Note**
DemographicParity	Demographic Parity (Statistical Parity)	0.1842	Self		
PredictiveParity	Predictive Parity	0.1185	Self		
EqualAccuracy	Equal Accuracy	0.0895	Office	Sales	
EqualizedOdds	Equalized Odds	0.1788	Self	Sales	The maximum TPR difference is greater than the maximum FPR difference.
EqualOpportunity	Equal Opportunity	0.1788	Self	Sales	

In order to use the MITIGATEBIAS statement to reduce the demographic parity, you must specify the levels of the response (target) variable and the corresponding predicted variables. You can use the DESCRIBE statement in the ASTORE procedure to obtain this information. The following statements use PROC ASTORE to describe the model:

```
proc astore;
   describe rstore=mylib.model;
run;
```

Output 10.4.3 lists the output variables that the model creates. The predicted variables are P_BAD1 and P_BAD0. Their corresponding response (target) levels are 1 and 0, respectively.

Output 10.4.3 Output Variables Table

The ASTORE Procedure

Output Variables			
Name	**Length**	**Type**	**Label**
P_BAD1	8	Num	Predicted: BAD=1
P_BAD0	8	Num	Predicted: BAD=0
I_BAD	12	Character	Into: BAD
WARN	4	Character	Warnings

The following statements use PROC FOREST to create a forest model, while minimizing the demographic parity for Job by using the MITIGATEBIAS statement:

```
ods graphics on;

proc forest data=mylib.hmeq nTrees=20;
   input CLAge CLNo DebtInc Derog
         Loan MortDue Value / level=interval;
   input Reason / level=nominal;
   target Bad / level=nominal;
```

```
    output out=mylib.scored;
    saveState rstore=mylib.model;
    mitigateBias pevents="1 0"
                 pvars=(P_BAD1 P_BAD0)
                 seed=12345
                 sensitivevar=Job
                 targetEvent="1";
run;
```

The PEVENTS= option lists the levels of the response (target) variable Bad. The predicted variables correspond to the levels that are listed in the PVARS= option. The PVARS= option lists the predicted variables that contain the posterior probability for each level in model prediction that corresponds to the response (target) variable Bad. The SEED= option specifies the seed for the pseudorandom number generator to be 12345. The SENSITIVEVAR= option names the variable Job as the sensitive variable for the purpose of reducing the value of the bias measurement. The TARGETEVENT= option specifies the value 1 as the event of interest.

Output 10.4.4 displays the information about the options that are used for bias mitigation. Output 10.4.5 displays the model accuracy and bias metric values over each iteration of the EGR algorithm. Output 10.4.6 shows the iteration history, which is a graphical representation of the iteration history table. The graph displays the bias metric and the misclassification rate for each iteration. PROC FOREST selects the iteration that has the lowest bias metric. In this case, the seventh iteration is selected; the graph indicates this selection by using the legend and a vertical line. The MITIGATEBIAS statement also creates other tables, which are not shown here but are listed in the section "ODS Table Names" on page 258.

Output 10.4.4 Mitigation Information Table

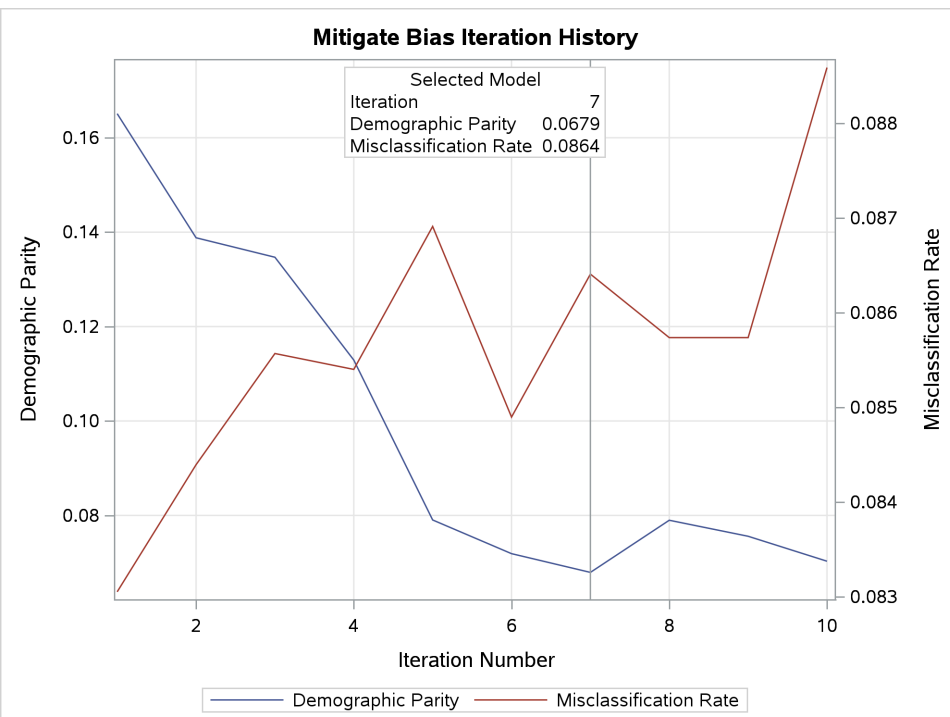
The FOREST Procedure

Mitigation Information		
Description	**Character Value**	**Value**
Bias Metric	DEMOGRAPHICPARITY	.
Sensitive Variable	JOB	.
Response	BAD	.
Bound Value	100	100
Learning Rate	0.01	0.01
Maximum Iteration	10	10
Tolerance	0.005	0.005
Event Level	1	.

Output 10.4.5 Iteration History Table

EGR Iteration History			
Iteration	Demographic Parity (Statistical Parity)	Misclassification Rate	Multi-Class Log Loss
1	0.165097	0.0831	0.2149
2	0.138814	0.0844	0.2167
3	0.134688	0.0856	0.2221
4	0.112907	0.0854	0.2246
5	0.079036	0.0869	0.2273
6	0.071900	0.0849	0.2257
7	0.067930	0.0864	0.2241
8	0.078976	0.0857	0.2262
9	0.075583	0.0857	0.2262
10	0.070299	0.0886	0.2257

Output 10.4.6 Iteration History Graph



References

Agarwal, A., Beygelzimer, A., Dudik, M., Langford, J., and Wallach, H. (2018). "A Reductions Approach to Fair Classification." In *Proceedings of the 35th International Conference on Machine Learning*, 60–69. PMLR.

Breiman, L. (1996). "Bagging Predictors." *Machine Learning* 24:123–140.

Breiman, L. (2001). "Random Forests." *Machine Learning* 45:5–32.

Liu, F. T., Ting, K. M., and Zhou, Z.-H. (2008). "Isolation Forest." In *2008 IEEE 8th International Conference on Data Mining (ICDM)*. Piscataway, NJ: Institute of Electrical and Electronics Engineers.

Loh, W.-Y. (2002). "Regression Trees with Unbiased Variable Selection and Interaction Detection." *Statistica Sinica* 12:361–386.

Loh, W.-Y. (2009). "Improving the Precision of Classification Trees." *Annals of Applied Statistics* 3:1710–1737.

Loh, W.-Y., and Shih, Y.-S. (1997). "Split Selection Methods for Classification Trees." *Statistica Sinica* 7:815–840.

Lundberg, S. M., Erion, G., Chen, H., DeGrave, A., Prutkin, J. M., Nair, B., Katz, R., Himmelfarb, J., et al. (2020). "From Local Explanations to Global Understanding with Explainable AI for Trees." *Nature Machine Intelligence* 2:56–67.

Neville, P. G., and Tan, P.-Y. (2014). "A Forest Measure of Variable Importance Resistant to Correlations." In *Proceedings of the 2014 Joint Statistical Meetings*. Alexandria, VA: American Statistical Association.

Van der Laan, M. J. (2006). "Statistical Inference for Variable Importance." *International Journal of Biostatistics* 2:1–31. Article 2.

Chapter 11
FPCA Procedure

Contents

Overview: FPCA Procedure **276**
- PROC FPCA Features 276
- Using SAS Viya Workbench 276

Getting Started: FPCA Procedure **277**

Syntax: FPCA Procedure **278**
- PROC FPCA Statement 278
- DISPLAY Statement 280
- DISPLAYOUT Statement 281
- ID Statement 282
- INPUT Statement 282
- OUTPUT Statement 282
- SAVESTATE Statement 283

Details: FPCA Procedure **283**
- FPCA Training 283
- Output Data Tables 286

Example: FPCA Procedure **286**
- Example 11.1: Simulated Functional Data 286
- Example 11.2: Berkeley Growth Study Data 289

References **290**

Overview: FPCA Procedure

The FPCA procedure performs functional principal component analysis (FPCA) on dense, regularly spaced functional data. This technique is widely used to analyze variations in functional data, reduce dimensionality, and extract meaningful patterns. PROC FPCA decomposes functions into orthonormal basis functions (eigenfunctions) and associated scores, providing a compact representation of the data.

FPCA is useful in a variety of applications, including signal processing, climate analysis, and biomedical studies. The primary outputs of this type of analysis include eigenvalues, eigenvectors (eigenfunctions), and functional principal component scores, which help analysts summarize and interpret the functional data structure.

PROC FPCA Features

PROC FPCA has the following features:

- is multithreaded during the FPCA training and scoring phases
- supports large-scale dense and regularly sampled functional data
- enables you to estimate the principal components from functional curves
- enables you to compute functional principal component (FPC) scores for each curve
- enables you to export the basis functions (eigenfunctions), and scores for downstream modeling
- enables you to perform the analysis on selected curves of the functional data

Using SAS Viya Workbench

SAS Viya Workbench has greatly simplified the SAS programming experience by offering a syntax style nearly identical to that of SAS 9 and earlier releases. However, there are minor exceptions. In the past, SAS Viya procedures required the use of a CAS engine libref. This engine served multiple purposes, but it was primarily used to upload data to the CAS server and manage the data. Its additional purposes included support of new data types and improved performance for large data sets. No CAS server is needed for SAS Viya Workbench, because all computations are done in place for you in a single location.

In the following program, the LIBNAME statement creates a mylib libref that the examples in this chapter use to refer to your current or desired work directory:

```
libname mylib ".";
```

For more information, see *SAS Viya Workbench: Getting Started*.

Getting Started: FPCA Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

This example shows how to use the FPCA procedure to perform functional principal component analysis on a simulated functional data set. Each curve in the data is generated as a linear combination of two orthonormal eigenfunctions (a cosine and a sine wave) plus a nonlinear mean function. The following statements use the IML procedure to generate the simulated data:

```
proc iml;
   call randseed(233);
   N = 50;         /* Number of subjects */
   M = 100;        /* Number of time points */
   s = do(0, 10, (10 - 0) / (M - 1));

   start meanFunct(s);
      return (s + 10 # exp(-(s - 5)##2));
   finish;
   start eigFunct1(s);
      return (cos(2 * s * constant('PI') / 10) / sqrt(5));
   finish;
   start eigFunct2(s);
      return (-sin(2 * s * constant('PI') / 10) / sqrt(5));
   finish;

   Ksi = j(N, 2);
   call randgen(Ksi, "Normal");
   Ksi = (Ksi - mean(Ksi)) / std(Ksi);   /* standardize */
   Ksi = Ksi * diag({5, 2});

   eig1 = eigFunct1(s);
   eig2 = eigFunct2(s);
   meanVec = meanFunct(s);
   y = Ksi * (eig1 // eig2) + meanVec;

   s_col = s`;
   s_y = s_col || y`;
   y_colnames = "y1":"y50";
   colnames = {"_TIMEPOINTS_"} || y_colnames;

   create work.fpca_train_data from s_y[colname=colnames];
   append from s_y;
   close work.fpca_train_data;
quit;
```

The data table fpca_train_data contains 100 rows (one per time point) and 51 columns: the first column contains the time points, and the remaining columns represent the 50 functional observations.

The following DATA step creates the input data table mylib.fpca_train_data. These statements assume that your SAS library is named mylib, but you can substitute any appropriately defined SAS library.

```
data mylib.fpca_train_data;
   set fpca_train_data;
run;
```

The following statements use PROC FPCA to extract the dominant functional components from the data. In this example, you use 50 bins to smooth the data, and two principal components are retained.

```
proc fpca data=mylib.fpca_train_data nbins=50;
   input y1-y50;
   output out=mylib.scores npc=2;
   savestate rstore=mylib.state;
run;
```

The NPC=2 option retains two functional principal components. The mylib.scores output table contains the scores for each function, and the mylib.state table contains the analytic store that you can use to score new functional data.

These results provide a low-dimensional representation of the original functional data that you can use in a downstream task such as clustering, anomaly detection, forecasting, or visualization.

Syntax: FPCA Procedure

The following statements are available in the FPCA procedure:

PROC FPCA < *options* > **;**
  DISPLAY < *table-list* > < */options* > **;**
  DISPLAYOUT *table-spec-list* < */options* > **;**
  ID *variables* **;**
  INPUT *variables* **;**
  OUTPUT OUT=*libref.data-table* < *option* > **;**
  SAVESTATE RSTORE=*libref.data-table* **;**

The PROC FPCA and INPUT statements are required. The following sections describe the PROC FPCA statement and then describe the other statements in alphabetical order.

PROC FPCA Statement

PROC FPCA < *options* > **;**

The PROC FPCA statement invokes the procedure. Table 11.1 summarizes the *options* available in the PROC FPCA statement.

Table 11.1 PROC FPCA Statement Options

Option	Description
Input Data Table Option	
DATA=	Specifies the input data table

Table 11.1 *continued*

Option	Description
FPCA Options	
NBINS=	Specifies the number of bins to smooth the functional curve
NTHREADS=	Specifies the number of threads to use on each computation node
RANKTHRESHOLD=	Specifies the epsilon value to determine the rank of the covariance matrix
Output Table Options	
EIGENVAL=	Specifies the name of the eigenvalue output table
EIGENVEC=	Specifies the name of the eigenfunction output table

You can specify the following *options*:

DATA=*libref.data-table*

names the input data table for PROC FPCA to use. The default is the most recently created data table. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to the data. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 276.

data-table specifies the name of the input data table.

EIGENVAL=*libref.data-table*

specifies the name of the output table that contains the eigenvalues. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 276.

EIGENVEC=*libref.data-table*

specifies the name of the output table that contains the eigenfunctions (principal component functions). *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 276.

NBINS

specifies the number of bins used to use for smoothing the functional curves. By default, PROC FPCA automatically and dynamically determines the number of bins.

NTHREADS=*number-of-threads*

specifies the number of threads to use on each computation node. The default value is either 16 or the number of threads available per node, whichever is less.

RANKTHRESHOLD=*number*
specifies the value of epsilon for determining the rank of the covariance matrix. The default *number* is 10^{-8}.

DISPLAY Statement

DISPLAY < *table-list* > < / *options* > **;**

The DISPLAY statement enables you to specify a list of display tables to display or exclude. This statement is similar to the ODS SELECT, ODS EXCLUDE, and ODS TRACE statements. However, the DISPLAY statement can improve performance when a large number of tables could be generated (such as in BY-group processing). The DISPLAY statement produces only the requested tables, but ODS statements generate all the tables and then keep the requested tables.

If you use both DISPLAY and ODS statements together, the DISPLAY statement takes precedence over the ODS statements. Note that the ODS EXCLUDE statement processes tables after they have been filtered by the DISPLAY statement. In some cases, it might appear that the ODS EXCLUDE statement is taking precedence because it can further filter the tables. For more information about ODS, see *SAS Output Delivery System: Procedures Guide*.

You can specify the *table-list* as a list of table names, paths, partial pathnames, and regular expressions.

The table names that you can specify are listed in the section "Displayed Output" on page 285. A path is a table name that is prefixed with dot-separated grouping information. For example, a SelectionSummary table that a procedure produces during a selection routine might have the path *Bygroup1.Summary.SelectionSummary*. A partial pathname does not include all groups; for example, *SelectionSummary* and *Summary.SelectionSummary* are partial pathnames for *Bygroup1.Summary.SelectionSummary*.

When you specify a table name or partial pathname, all display tables whose paths end in the specified name are selected for display or exclusion. For example, both *SelectionSummary* and *Summary.SelectionSummary* select *Bygroup1.Summary.SelectionSummary*.

A regular expression is enclosed in forward slashes (/). For example, specifying "/tions/" selects all pathnames that contain the substring "tions"; in particular, the *Bygroup1.Summary.SelectionSummary* table is selected. Specifying "!/tions/" selects all pathnames that do not contain the substring "tions"; in particular, the *Bygroup1.Summary.SelectionSummary* table is not selected.

You can specify the following *options* after a slash (/):

CASESENSITIVE
performs a case-sensitive comparison of table names in the *table-list* to display table names when tables are subsetted for display. To preserve case, you must enclose table names in the *table-list* in quotation marks.

EXCLUDE
displays all display tables except those that you specify in the *table-list*.

EXCLUDEALL
suppresses display of all tables. This option takes precedence over the other options.

TRACE
displays the display table names, labels, and paths.

DISPLAYOUT Statement

DISPLAYOUT *table-spec-list* < / *options* > **;**

The DISPLAYOUT statement enables you to create output tables from your displayed output. This statement is similar to the ODS OUTPUT statement. For more information about ODS, see *SAS Output Delivery System: Procedures Guide*.

The *table-spec-list* specifies a list of output tables to create. Each entry in the list has either a *key*=*value* format or a *key* format:

key=*value* specifies *key* as the ODS table name, path, or partial pathname, and specifies *value* as the output table name.

key specifies *key* as the ODS table name and also as the output table name.

The ODS table names that you can specify are listed in the section "Displayed Output" on page 285. You cannot specify the ODS table named OutputCasTables in the *table-spec-list*.

Table names and partial pathnames are discussed under the DISPLAY statement. The DISPLAYOUT statement does not support regular expressions.

You can specify the following *options* after a slash (/):

INCLUDEALL
creates output tables for all display tables. The name of the created output table is the same as the name of the corresponding display table. If you specify this option, the *table-spec-list* specification is ignored.

NOREPLACE
does not replace any existing output table of the same name.

ID Statement

ID *variables* **;**

The ID statement lists one or more *variables* whose values are used to identify observations at different time points.

INPUT Statement

INPUT *variables* **;**

The INPUT statement specifies the names of *variables* to consider in the procedure run. PROC FPCA accepts numeric variables as input and ignores nonnumeric variables. This statement is required.

OUTPUT Statement

OUTPUT OUT=*libref.data-table* < *option* > **;**

The OUTPUT statement creates an output data table to contain the results of the procedure run.

You must specify the following option:

OUT=*libref.data-table*
names the output data table for PROC FPCA to use. You must specify this option before any other options. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to where the data table is to be stored. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 276.

data-table specifies the name of the output data table.

You can also specify the following *option*:

NPC=*number*
specifies the number of principal components to display.

SAVESTATE Statement

SAVESTATE RSTORE=*libref.data-table* **;**

The SAVESTATE statement creates an analytic store for the functional principal component analysis model and saves it as a binary object in a data table. You can use the analytic store in the ASTORE procedure to score new functional data. For more information, see Chapter 4, "ASTORE Procedure."

You must specify the following option:

RSTORE=*libref.data-table*
specifies the name of the table in which to save the analytic store. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 276.

Scoring new data from the saved analytic store enables you to project the input functional data onto the functional principal components. You can use the FPCASCORE procedure to perform this scoring.

Details: FPCA Procedure

Functional principal component analysis (FPCA) is a technique for analyzing functional data, in which each observation is a function or curve rather than a traditional vector. Functional data commonly occur in contexts such as growth curves, temperature profiles over time, and spectrometric analysis. FPCA aims to capture the main modes of variation within these curves, enabling dimensionality reduction and interpretation.

FPCA Training

This section describes the FPCA methods that you can implement in the FPCA procedure, specifically for dense, regularly spaced functional data sets.

FPCA Training Methodology

You apply FPCA to functional data that are represented as the continuous functions $X_i(t)$ as observed over a domain T. Each $X_i(t)$ represents an observation that is recorded across T (such as time); it captures variability in a population of curves. FPCA has the following five steps.

Step 1: Binning and Smoothing

Functional data often have high variability, which can obscure underlying patterns. As a preprocessing step, you apply binning in order to group nearby observations within specific intervals (or bins) and then average them, smoothing the curves and reducing noise. This step makes FPCA estimates more stable by reducing random variability.

Step 2: Calculating the Mean Function

After smoothing, you calculate the mean function $\mu(t)$ of the data as follows to represent the average trend across all observed functions:

$$\mu(t) = \frac{1}{n}\sum_{i=1}^{n} X_i(t)$$

This mean function is subtracted from each individual observation to center the data in the functional space.

Step 3: Estimating the Covariance Function

To understand the dependency structure within the data, you estimate the covariance function $C(s,t)$ as follows:

$$C(s,t) = \frac{1}{n}\sum_{i=1}^{n}(X_i(s) - \mu(s))(X_i(t) - \mu(t))$$

where $s, t \in T$. This covariance function provides a measure of similarity between observations across different points in the domain.

Step 4: Performing Eigenfunction Decomposition

You obtain the eigenfunctions $\phi_k(t)$ and the corresponding eigenvalues λ_k by solving the following integral equation:

$$\int_T C(s,t)\phi_k(t)\,dt = \lambda_k \phi_k(s)$$

The eigenfunctions $\{\phi_k(t)\}$ form the principal components in the functional space, capturing the main directions of variability. The eigenvalues $\{\lambda_k\}$ quantify the amount of variance that each component explains.

Step 5: Calculating Principal Component Scores

For each function $X_i(t)$, you calculate the principal component scores α_{ik} to quantify the contribution of each eigenfunction $\phi_k(t)$ to $X_i(t)$. These scores are computed as follows:

$$\alpha_{ik} = \int_T (X_i(t) - \mu(t))\phi_k(t)\,dt$$

where α_{ik} represents the score of the ith function on the kth principal component. These scores indicate how much each principal component contributes to representing $X_i(t)$. You can then approximate the function $X_i(t)$ as

$$X_i(t) \approx \mu(t) + \sum_{k=1}^{K} \alpha_{ik}\phi_k(t)$$

The number of components K is chosen to balance the computation complexity and the variance explained by the first K eigenfunctions.

Displayed Output

PROC FPCA displays various output tables that are related to input data, model parameters, and results. The following sections describe these tables in the order of their appearance.

Model Information

The “Model Information” table provides the following essential details about the FPCA model:

- total number of curves that were analyzed in the FPCA model
- number of bins that were applied during data smoothing and binning
- threshold that was used to identify significant eigenvalues for component selection
- total number of principal components that were generated in the analysis
- estimated variance of measurement error for the observed data

Number of Observations

The “Number of Observations” table summarizes the data points that were processed in the FPCA. It displays the total number of time points that were read from the data set and the number of time points that were included in the FPCA model after any filtering or preprocessing.

Descriptive Statistics

The “Descriptive Statistics” table displays the mean and standard deviation of each variable in the data set, allowing for an overview of the central tendency and dispersion of each input variable.

Task Timing

The “Task Timing” table displays the time in seconds and the percentage of total time that each step in the FPCA process required, which can include the following:

- time to bin and smooth the data
- time to compute the covariance matrix
- time for eigenfunction decomposition of the covariance matrix
- time to compute the FPCA scores for each observation
- time for additional processes that are not covered by the previous items in this list
- total time to complete the FPCA training process

Output Data Tables

PROC FPCA creates output tables for eigenvalues (if you specify the EIGENVAL= option in the PROC FPCA statement) and eigenfunctions (if you specify the EIGENVEC= option in the same statement). Table 11.2 provides details about these data tables.

Table 11.2 Output Data Tables Produced by PROC FPCA

Table Name	Description	Dimensions
Eigenval	Eigenvalues of the covariance operator	Number of eigenvalues × 1
Eigenvec	Eigenfunctions of the covariance operator	Number of bins × number of eigenfunctions
Out	Projection of training functional curves onto the functional principal components	Number of curves × number of eigenvalues

Example: FPCA Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

Example 11.1: Simulated Functional Data

This example demonstrates how to use the FPCA procedure on simulated functional data that have a known low-dimensional structure. The data consist of 50 functional observations; each observation is recorded at 100 equally spaced time points in the interval [0, 10]. Each curve is generated as a linear combination of two orthonormal basis functions, $\phi_1(t) = \cos(2\pi t/10)/\sqrt{5}$ and $\phi_2(t) = -\sin(2\pi t/10)/\sqrt{5}$, and class-specific mean functions and random coefficients are drawn from a standardized Gaussian distribution that is scaled by eigenvalues. The first 25 curves belong to class 0 and use a mean function whose peak is centered at $t = 5$. The remaining 25 curves belong to class 1 and have a mean peak around $t = 7$.

The following IML procedure statements generate the simulated data set func_curves. Each column (y1 to y50) corresponds to a functional observation, and each row represents a time point. The variable _TIMEPOINTS_ stores the common time grid across all observations.

```
proc iml;
   call randseed(123);
   N = 50;     /* number of curves */
   M = 100;    /* number of time points */
   s = do(0, 10, 10 / (M - 1));
```

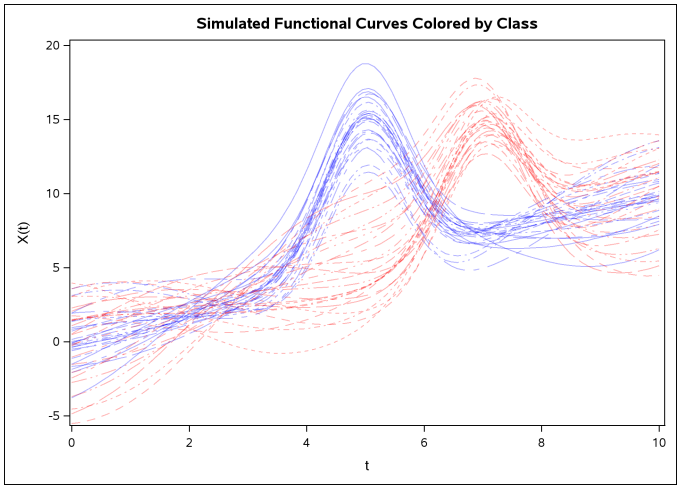
```
   /* Class-specific mean functions */
   start meanFunct_class0(s);
      return (s + 10 # exp(-(s - 5)##2));
   finish;
   start meanFunct_class1(s);
      return (s + 8 # exp(-(s - 7)##2));
   finish;

   /* Shared eigenfunctions */
   start eigFunct1(s);
      return (cos(2 * s * constant('PI') / 10) / sqrt(5));
   finish;
   start eigFunct2(s);
      return (-sin(2 * s * constant('PI') / 10) / sqrt(5));
   finish;

   /* Generate scores */
   Ksi = j(N, 2);
   call randgen(Ksi, "Normal");
   Ksi = (Ksi - Ksi[:,]) / sqrt(var(Ksi)[,]);
   Ksi = Ksi * diag({5, 2});

   /* Evaluate basis and means */
   eig1 = eigFunct1(s);
   eig2 = eigFunct2(s);
   basis = eig1 // eig2;
   meanVec0 = meanFunct_class0(s);
   meanVec1 = meanFunct_class1(s);

   /* Build curves */
   y = j(N, M, .);
   do i = 1 to N;
      if i <= N/2 then
         y[i,] = Ksi[i,] * basis + meanVec0;
      else
         y[i,] = Ksi[i,] * basis + meanVec1;
   end;

   /* Export wide-format data */
   s_y = s` || y`;
   y_colnames = "y1":"y50";
   colnames = {"_TIMEPOINTS_"} || y_colnames;

   create func_curves from s_y[colname=colnames];
   append from s_y;
   close func_curves;
quit;
```

After generating the data, you can use the following statements to convert the data to long format and visualize the curves by class. The results are shown in Output 11.1.1.

```
/* Transpose to long format */
proc transpose data=func_curves out=func_long(rename=(col1=value)) name=id;
   by _TIMEPOINTS_;
```

```
run;

/* Create attribute map to color curves by class */
data attrmap;
   length id value $10 linecolor $8;
   do i = 1 to 50;
      value = "y" || strip(put(i, best.));
      id = "idmap";
      if i <= 25 then linecolor = "blue";
      else linecolor = "red";
      output;
   end;
run;

/* Attach attrid to plot data */
data func_long_class;
   set func_long;
   attrid = "idmap";
run;

/* Sort data */
proc sort data=func_long_class;
   by id _TIMEPOINTS_;
run;

/* Plot curves colored by class */
ods graphics / reset width=7in height=5in;

proc sgplot data=func_long_class dattrmap=attrmap;
   series x=_TIMEPOINTS_ y=value / group=id attrid=idmap
          lineattrs=(thickness=1) transparency=0.7;
   xaxis label="t";
   yaxis label="X(t)";
   title "Simulated Functional Curves Colored by Class";
run;
```

Output 11.1.1 Plot of Functional Curves by Mean Curve Class



Output 11.1.1 displays all 50 functional curves over the common time domain. Each curve is plotted as a line, and the coloring reflects class membership: curves from class 0 (observations y1 to y25) are shown in blue, and curves from class 1 (y26 to y50) are shown in red. This color coding makes it easier to visually assess differences between the two classes. In this example, the two classes exhibit different average behaviors over time, as defined by their respective mean functions. The plot highlights these differences, providing useful information about the functional structure and demonstrating the value of using FPCA for dimension reduction.

You can use the following DATA step to load the generated func_curves data set into your SAS library and by using your libref:

```
data mylib.func_curves;
   set func_curves;
run;
```

Then you fit a functional principal component model by using the following PROC FPCA statement:

```
proc fpca data=mylib.func_curves nBins=50;
   input y1-y50;
   savestate rstore=mylib.fpca_model;
run;
```

The SAVESTATE statement saves the trained FPCA model to an analytic store, enabling you to efficiently reuse it for future scoring. A downstream example that shows how to score new observations and visualize class separation by using FPCA scores is provided in the documentation of the FPCASCORE procedure.

Example 11.2: Berkeley Growth Study Data

This example applies the FPCA procedure to real-world data. The data come from the Berkeley Growth Study, which took the height measurements of 93 children as part of a longitudinal study of children's physical development that was conducted in California. Each observation in the growth_curves data set corresponds to a time point (in years), and each column from y1 to y93 represents an individual child's growth trajectory over 31 time points, as stored in the variable _TIMEPOINTS_. The data set is publicly available as part of the fda R package for functional data analysis and can also be accessed through the scikit-fda Python package. The data that this example uses are derived from Tuddenham and Snyder (1954).[1]

This following DATA step creates a data table named growth_curves that contains the Berkeley Growth Study functional data, where each row represents a time point and each column (y1 to y93) represents a subject's measurement at that time point. The _TIMEPOINTS_ variable stores the time value for each row, and the data are entered directly by using the DATALINES statement.

[1]Disclaimer: SAS may reference other websites or content or resources for use at Customer's sole discretion. SAS has no control over any websites or resources that are provided by companies or persons other than SAS. Customer acknowledges and agrees that SAS is not responsible for the availability or use of any such external sites or resources, and does not endorse any advertising, products, or other materials on or available from such websites or resources. Customer acknowledges and agrees that SAS is not liable for any loss or damage that may be incurred by Customer or its end users as a result of the availability or use of those external sites or resources, or as a result of any reliance placed by Customer or its end users on the completeness, accuracy, or existence of any advertising, products, or other materials on, or available from, such websites or resources.

```
   data growth_curves;
input _TIMEPOINTS_ y1-y93;
datalines;
1.0 81.3 76.2 76.8 74.1 74.2 76.8 72.4 73.8 75.4
78.8 76.9 81.6 78.0 76.4 76.4 76.2 75.0 79.7 70.0

   ... more lines ...
```

You can load the growth_curves data table into your SAS library and by using your libref in the first statement of the following DATA step.

```
   data mylib.growth;
   set growth_curves;
   run;
```

You use the following PROC FPCA statements to extract the dominant modes of variation in the growth curves by using 10 smoothing bins:

```
proc fpca data=mylib.growth nBins=10;
   input y1-y93;
   output out=mylib.score_fpca npc=4;
   savestate rstore=mylib.state_fpca;
run;
```

The OUTPUT statement generates the scores of each child's growth curve on the first four functional principal components. The SAVESTATE statement creates an analytic store that you can use later with the FPCASCORE procedure to score new observations by using the same FPCA model.

A downstream analysis task, such as visualizing the first two principal components by gender, is demonstrated in the documentation of PROC FPCASCORE. For more information, see Chapter 12, "FPCASCORE Procedure."

References

Tuddenham, R. D., and Snyder, M. M. (1954). "Physical Growth of California Boys and Girls from Birth to Eighteen Years." *Publications in Child Development* 1:183–364.

Chapter 12
FPCASCORE Procedure

Contents

Overview: FPCASCORE Procedure **291**
PROC FPCASCORE Features 292
Using SAS Viya Workbench 292
Getting Started: FPCASCORE Procedure **292**
Syntax: FPCASCORE Procedure **294**
PROC FPCASCORE Statement 295
DISPLAY Statement 295
DISPLAYOUT Statement 296
ID Statement 297
INPUT Statement 297
OUTPUT Statement 297
Details: FPCASCORE Procedure **298**
Scoring Methodology 298
Output Data Tables 299
Example: FPCASCORE Procedure **300**
Example 12.1: Scoring Simulated Functional Data with a Saved FPCA Model 300
Example 12.2: Scoring Growth Data with a Saved FPCA Model 302
References **306**

Overview: FPCASCORE Procedure

The FPCASCORE procedure applies a previously trained functional principal component analysis (FPCA) model to new functional data in order to compute functional principal component scores. You would typically use this procedure to score or project new curves onto the principal components that are derived from a training data set by the FPCA procedure.

PROC FPCASCORE requires a saved analytic store (created by the SAVESTATE statement in PROC FPCA), which contains the estimated mean function and eigenfunctions from the training data. The procedure computes projection scores for the input data by using these stored components.

This scoring capability is useful in tasks such as anomaly detection, functional classification, and forecasting, or in any application that benefits from dimensionality reduction of functional data.

PROC FPCASCORE Features

PROC FPCASCORE has the following features:

- is multithreaded when calculating FPCA score for new functional curves
- supports large-scale dense and regularly sampled functional data for scoring
- enables you to compute functional principal component (FPC) scores for new functional curves
- enables you to perform the analysis on selected curves in the new functional data

Using SAS Viya Workbench

SAS Viya Workbench has greatly simplified the SAS programming experience by offering a syntax style nearly identical to that of SAS 9 and earlier releases. However, there are minor exceptions. In the past, SAS Viya procedures required the use of a CAS engine libref. This engine served multiple purposes, but it was primarily used to upload data to the CAS server and manage the data. Its additional purposes included support of new data types and improved performance for large data sets. No CAS server is needed for SAS Viya Workbench, because all computations are done in place for you in a single location.

In the following program, the LIBNAME statement creates a mylib libref that the examples in this chapter use to refer to your current or desired work directory:

```
libname mylib ".";
```

For more information, see *SAS Viya Workbench: Getting Started*.

Getting Started: FPCASCORE Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

This example demonstrates how to use the FPCASCORE procedure to score new functional observations by using a model that is saved by the FPCA procedure. The following statements generate two simulated data sets by using the IML procedure:

```
proc iml;
   call randseed(233);
   N = 50;       /* Number of training curves */
   N_new = 50;   /* Number of new curves to score */
   M = 100;      /* Number of time points */
   s = do(0, 10, (10 - 0) / (M - 1));

   start meanFunct(s);
```

```
      return (s + 10 # exp(-(s - 5)##2));
   finish;
   start eigFunct1(s);
      return (cos(2 * s * constant('PI') / 10) / sqrt(5));
   finish;
   start eigFunct2(s);
      return (-sin(2 * s * constant('PI') / 10) / sqrt(5));
   finish;

   /* Simulate training data */
   Ksi_train = j(N, 2);
   call randgen(Ksi_train, "Normal");
   Ksi_train = Ksi_train * diag({5, 2});

   eig1 = eigFunct1(s);
   eig2 = eigFunct2(s);
   meanVec = meanFunct(s);
   y_train = Ksi_train * (eig1 // eig2) + meanVec;

   /* Simulate new data */
   Ksi_new = j(N_new, 2);
   call randgen(Ksi_new, "Normal");
   Ksi_new = Ksi_new * diag({5, 2});
   y_new = Ksi_new * (eig1 // eig2) + meanVec;

   s_col = s`;
   s_ytrain = s_col || y_train`;
   s_ynew = s_col || y_new`;

   ytrain_colnames = "y1":"y50";
   ynew_colnames = "y1":"y50";
   colnames_train = {"_TIMEPOINTS_"} || ytrain_colnames;
   colnames_new = {"_TIMEPOINTS_"} || ynew_colnames;

   create fpca_train_data from s_ytrain[colname=colnames_train];
   append from s_ytrain;
   close fpca_train_data;

   create fpca_new_data from s_ynew[colname=colnames_new];
   append from s_ynew;
   close fpca_new_data;
quit;
```

The fpca_train_data and fpca_new_data tables contain functional observations that have 100 time points. The first column contains the time points, and the remaining columns contain the observed values for each function.

To use these data sets with PROC FPCA and PROC FPCASCORE, you submit the following statements. These statements assume that your SAS library is named mylib, but you can substitute any appropriately defined SAS library.

```
data mylib.fpca_train_data;
   set fpca_train_data;
run;

data mylib.fpca_new_data;
   set fpca_new_data;
run;
```

The following PROC FPCA call trains a model on the training data and saves the model in an analytic store:

```
proc fpca data=mylib.fpca_train_data nbins=50;
   input y1-y50;
   output out=mylib.fpca_scores npc=2;
   savestate rstore=mylib.state_fpca;
run;
```

The following statements use PROC FPCASCORE to compute functional principal component scores for the new data:

```
proc fpcascore data=mylib.fpca_new_data model=mylib.state_fpca;
   input y1-y50;
   output out=mylib.fpca_scored_new npc=2;
run;
```

The mylib.fpca_scored_new output table contains the first two functional principal component scores for the new curves; the scores are based on the mean function and eigenfunctions from the saved FPCA model. These scores provide a reduced-dimension representation of the new functional observations that you can use in a downstream analysis task such as classification, anomaly detection, or visualization.

Syntax: FPCASCORE Procedure

The following statements are available in the FPCASCORE procedure:

PROC FPCASCORE DATA=*libref.data-table* **MODEL=***libref.data-table* < *option* > **;**
  DISPLAY < *table-list* > < *options* > **;**
  DISPLAYOUT *table-spec-list* < *options* > **;**
  INPUT *variables* **;**
  OUTPUT OUT=*libref.data-table* < *option* > **;**

The PROC FPCASCORE and INPUT statements are required. The following sections describe the PROC FPCASCORE statement and then describe the other statements in alphabetical order.

PROC FPCASCORE Statement

PROC FPCASCORE DATA=*libref.data-table* **MODEL=***libref.data-table* < *option* > **;**

The PROC FPCASCORE statement invokes the procedure. You specify the input data table and the analytic store from a previously trained FPCA model.

You must specify the following options:

DATA=*libref.data-table*
names the input data table for PROC FPCASCORE to use. The default is the most recently created data table. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to the data. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 292.

data-table specifies the name of the input data table.

MODEL=*libref.data-table*
specifies the analytic store model that contains the mean function and eigenfunctions. You create this model by using the SAVESTATE statement in the FPCA procedure. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 292.

You can also specify the following *option*:

NTHREADS=*number-of-threads*
specifies the number of threads to use on each computation node for scoring new functional curves. The default value is either 16 or the number of threads available per node, whichever is less.

DISPLAY Statement

DISPLAY < *table-list* > < / *options* > **;**

The DISPLAY statement enables you to specify a list of display tables to display or exclude. This statement is similar to the ODS SELECT, ODS EXCLUDE, and ODS TRACE statements. However, the DISPLAY statement can improve performance when a large number of tables could be generated (such as in BY-group processing). The DISPLAY statement produces only the requested tables, but ODS statements generate all the tables and then keep the requested tables.

If you use both DISPLAY and ODS statements together, the DISPLAY statement takes precedence over the ODS statements. Note that the ODS EXCLUDE statement processes tables after they have been filtered by the DISPLAY statement. In some cases, it might appear that the ODS EXCLUDE statement is taking precedence because it can further filter the tables. For more information about ODS, see *SAS Output Delivery System: Procedures Guide*.

You can specify the *table-list* as a list of table names, paths, partial pathnames, and regular expressions.

The table names that you can specify are listed in the section "Displayed Output" on page 299. A path is a table name that is prefixed with dot-separated grouping information. For example, a SelectionSummary table that a procedure produces during a selection routine might have the path *Bygroup1.Summary.SelectionSummary*. A partial pathname does not include all groups; for example, *SelectionSummary* and *Summary.SelectionSummary* are partial pathnames for *Bygroup1.Summary.SelectionSummary*.

When you specify a table name or partial pathname, all display tables whose paths end in the specified name are selected for display or exclusion. For example, both *SelectionSummary* and *Summary.SelectionSummary* select *Bygroup1.Summary.SelectionSummary*.

A regular expression is enclosed in forward slashes (/). For example, specifying "/tions/" selects all pathnames that contain the substring "tions"; in particular, the *Bygroup1.Summary.SelectionSummary* table is selected. Specifying "!/tions/" selects all pathnames that do not contain the substring "tions"; in particular, the *Bygroup1.Summary.SelectionSummary* table is not selected.

You can specify the following *options* after a slash (/):

CASESENSITIVE
performs a case-sensitive comparison of table names in the *table-list* to display table names when tables are subsetted for display. To preserve case, you must enclose table names in the *table-list* in quotation marks.

EXCLUDE
displays all display tables except those that you specify in the *table-list*.

EXCLUDEALL
suppresses display of all tables. This option takes precedence over the other options.

TRACE
displays the display table names, labels, and paths.

DISPLAYOUT Statement

DISPLAYOUT *table-spec-list* < / *options* > **;**

The DISPLAYOUT statement enables you to create output tables from your displayed output. This statement is similar to the ODS OUTPUT statement. For more information about ODS, see *SAS Output Delivery System: Procedures Guide*.

The *table-spec-list* specifies a list of output tables to create. Each entry in the list has either a *key*=*value* format or a *key* format:

key=*value* specifies *key* as the ODS table name, path, or partial pathname, and specifies *value* as the output table name.

key specifies *key* as the ODS table name and also as the output table name.

The ODS table names that you can specify are listed in the section "Displayed Output" on page 299. You cannot specify the ODS table named OutputCasTables in the *table-spec-list*.

Table names and partial pathnames are discussed under the DISPLAY statement. The DISPLAYOUT statement does not support regular expressions.

You can specify the following *options* after a slash (/):

INCLUDEALL
creates output tables for all display tables. The name of the created output table is the same as the name of the corresponding display table. If you specify this option, the *table-spec-list* specification is ignored.

NOREPLACE
does not replace any existing output table of the same name.

ID Statement

ID *variables* **;**

The ID statement lists one or more *variables* whose values are used to identify observations at different time points.

INPUT Statement

INPUT *variables* **;**

The INPUT statement specifies the names of *variables* that represent the functional data in the input table.

OUTPUT Statement

OUTPUT OUT=*libref.data-table* < *option* > **;**

The OUTPUT statement creates an output data table to contain the results of the procedure run.

You must specify the following option:

OUT=*libref.data-table*
names the output data table for PROC FPCASCORE to use. You must specify this option before any other options. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to where the data table is to be stored. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 292.

data-table specifies the name of the output data table.

You can also specify the following *option*:

NPC=*number*
specifies the number of principal components to display.

Details: FPCASCORE Procedure

The FPCASCORE procedure scores new functional data by projecting them onto the functional principal components that are estimated by the FPCA procedure. Each new observation is treated as a function and evaluated against the eigenfunctions that are computed during training. The results are principal component scores that summarize the main features of the input curves in a lower-dimensional representation.

This scoring capability is essential for tasks such as real-time monitoring, functional regression, and anomaly detection, in which new data must be interpreted consistently by using a previously learned FPCA model.

Scoring Methodology

This section describes how PROC FPCASCORE applies a previously trained FPCA model to new functional data.

PROC FPCASCORE Methodology

To score a new functional curve $X_i(t)$, the procedure performs the following three steps.

Step 1: Binning and Smoothing

The new functional observations are first preprocessed by using the same binning and smoothing scheme that the original FPCA training step uses. This ensures that the structure of the new data matches the structure of the training data.

Step 2: Subtracting the Mean Function

The mean function $\mu(t)$ that is estimated during training is subtracted from each new curve as follows:

$$X_i^*(t) = X_i(t) - \mu(t)$$

This step centers the new data in the same functional space that is used for training.

Step 3: Calculating Scores

Each centered curve $X_i^*(t)$ is projected onto the previously learned eigenfunctions $\phi_k(t)$ to calculate the principal component scores as follows:

$$\alpha_{ik} = \int_T X_i^*(t)\,\phi_k(t)\,dt$$

These scores quantify how the new function aligns with each of the k functional principal components and provide a compact representation to use for downstream tasks.

Displayed Output

PROC FPCASCORE produces two output tables that contain information about the scoring process and results.

Number of Observations

This table shows the total number of time points that were read from the input data set and the number of time points that were retained after preprocessing.

Task Timing

The "Task Timing" table reports the duration and relative percentage of time that the procedure spent on various phases of the scoring process, such as the following:

- binning and smoothing the input data
- applying the mean function and computing scores
- other internal computations
- total time to complete the scoring process

Output Data Tables

PROC FPCASCORE produces a scored output data table (if you specify the OUT= option in the OUTPUT statement). This table contains the principal component scores for each scored curve. Table 12.1 provides details.

Table 12.1 Output Table Produced by PROC FPCASCORE

Table Name	Description	Dimensions
Out	Principal component scores for new functional curves	Number of curves × number of principal components specified in NPC= option

Example: FPCASCORE Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

Example 12.1: Scoring Simulated Functional Data with a Saved FPCA Model

This example demonstrates how to use the FPCASCORE procedure to score new functional data by using a previously trained FPCA model. The data consist of 50 simulated functional curves that are evaluated at 100 equally spaced time points over the interval [0, 10]. The curves are split evenly between two classes that have different mean functions: class 0 corresponds to curves y1 through y25, and class 1 corresponds to curves y26 through y50. Class 0 curves exhibit a symmetric peak that is centered at $t = 5$, and class 1 curves have a smaller peak that is shifted toward $t = 7$. Random variation is added along two orthonormal eigenfunctions.

The following IML procedure statements generate the simulated data set func_curves. Each column (y1 to y50) corresponds to a curve, and each row corresponds to a time point.

```
proc iml;
   call randseed(123);
   N = 50;          /* Total number of subjects */
   M = 100;         /* Number of time points */
   s = do(0, 10, (10 - 0) / (M - 1));  /* Time grid */

   /* Define two class-specific mean functions */
   start meanFunct_class0(s);
      return (s + 10 # exp(-(s - 5)##2));
   finish;

   start meanFunct_class1(s);
      return (s + 8 # exp(-(s - 7)##2));   /* Slightly shifted & smaller bump */
   finish;

   /* Define two eigenfunctions */
   start eigFunct1(s);
      return (cos(2 * s * constant('PI') / 10) / sqrt(5));
   finish;

   start eigFunct2(s);
      return (-sin(2 * s * constant('PI') / 10) / sqrt(5));
   finish;

   /* Simulate normalized FPCA scores */
   Ksi = j(N, 2);
   call randgen(Ksi, "Normal");
   Ksi = (Ksi - mean(Ksi)) / std(Ksi);   /* standardize */
   Ksi = Ksi * diag({5, 2});

   /* Evaluate eigenfunctions and class-specific means */
```

```
   eig1 = eigFunct1(s);
   eig2 = eigFunct2(s);
   meanVec0 = meanFunct_class0(s);  /* Class 0 */
   meanVec1 = meanFunct_class1(s);  /* Class 1 */

   /* Allocate curve matrix */
   y = j(N, M, .);

   /* Fill curves: first half = class 0, second half = class 1 */
   do i = 1 to N;
      if i <= N/2 then
         y[i,] = Ksi[i,] * (eig1 // eig2) + meanVec0;
      else
         y[i,] = Ksi[i,] * (eig1 // eig2) + meanVec1;
   end;

   /* Prepare output: time points + y1 to y50 */
   s_col = s`;
   s_y = s_col || y`;  /* Transpose so each y is a column */
   y_colnames = "y1":"y50";
   colnames = {"_TIMEPOINTS_"} || y_colnames;

   create work.func_curves from s_y[colname=colnames];
   append from s_y;
   close work.func_curves;
quit;
```

You can use the following DATA step to load the generated func_curves data set into your SAS library and by using your libref:

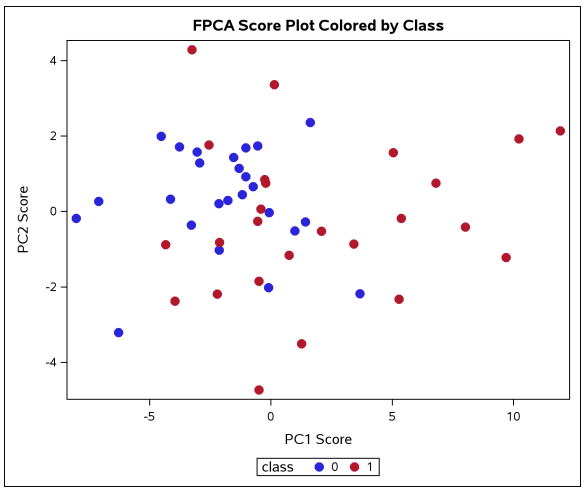
```
   data mylib.func_curves;
      set func_curves;
   run;
```

You can then use the FPCA procedure as follows to train a functional principal component model and save it to an analytic store. Then you apply PROC FPCASCORE as shown to score the same data by using the saved model and extract the first two principal component scores for visualization. The results are displayed in Output 12.1.1.

```
proc fpca data=mylib.func_curves nBins=50;
   input y1-y50;
   savestate rstore=mylib.fpca_model;
run;

proc fpcascore data=mylib.func_curves model=mylib.fpca_model;
   input y1-y50;
   output out=mylib.fpca_scores npc=2;
run;

/* Assign class label by observation order */
data fpca_scores_labeled;
   set mylib.fpca_scores;
   if _N_ <= 25 then class = 0;
```

```
   else class = 1;
run;

/* Plot PC1 vs PC2 colored by class */
ods graphics / reset width=6in height=5in;

proc sgplot data=fpca_scores_labeled;
   scatter x=_PCS_1 y=_PCS_2 / group=class
           markerattrs=(symbol=CircleFilled size=9);
   xaxis label="PC1 Score";
   yaxis label="PC2 Score";
   title "FPCA Score Plot Colored by Class";
run;
```

Output 12.1.1 First Two FPCA Scores Colored by Class



The plot in Output 12.1.1 shows that the first two functional principal component scores mostly separate the two simulated classes, showing distinct clustering along with some overlap. This demonstrates that FPCA effectively captures the dominant variations that are introduced by the class-specific mean structure in the functional data.

Example 12.2: Scoring Growth Data with a Saved FPCA Model

This example demonstrates how to use the FPCASCORE procedure to score new functional data by using a previously trained FPCA model. The data come from the Berkeley Growth Study (Tuddenham and Snyder 1954), which includes height trajectories of 93 children that were measured over 31 time points.[1]

[1]Disclaimer: SAS may reference other websites or content or resources for use at Customer's sole discretion. SAS has no control over any websites or resources that are provided by companies or persons other than SAS. Customer acknowledges and agrees that SAS is not responsible for the availability or use of any such external sites or resources, and does not endorse any advertising, products, or other materials on or available from such websites or resources. Customer acknowledges and agrees that SAS is not liable for any loss or damage that may be incurred by Customer or its end users as a result of the availability or use of those external sites or resources, or as a result of any reliance placed by Customer or its end users on the completeness, accuracy, or existence of any advertising, products, or other materials on, or available from, such websites or resources.

The following DATA step creates the data table growth_curve. Each of the columns y1 through y93 corresponds to a child's growth curve, and the column _TIMEPOINTS_ contains the time points.

```
   data growth_curves;
input _TIMEPOINTS_ y1-y93;
datalines;
1.0 81.3 76.2 76.8 74.1 74.2 76.8 72.4 73.8 75.4
78.8 76.9 81.6 78.0 76.4 76.4 76.2 75.0 79.7 70.0

   ... more lines ...
```

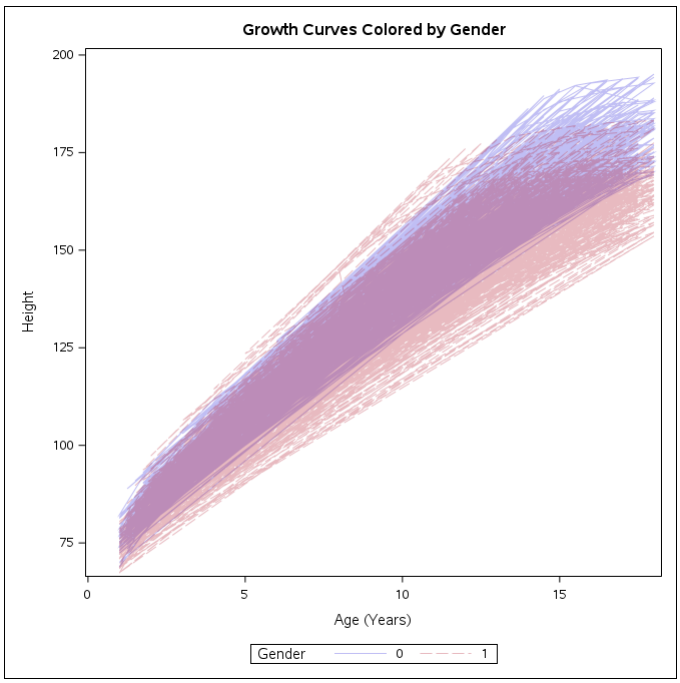
The following DATA step creates the data table growth_labels, which contains the gender ID for each curve—0 for males and 1 for females:

```
   data growth_labels;
infile datalines dsd;
input ID :$5. Gender;
datalines;
y1,0

   ... more lines ...
```

To visualize the raw functional data, the data are first transposed to long format and then merged with gender information, as shown in the following statements. The results are displayed in Output 12.2.1, which plots the height trajectories for all 93 children; the curves are colored by gender.

```
/* Convert to long format and merge with gender */
proc transpose data=growth_curves out=growth_long(rename=(col1=value)) name=id;
   by _TIMEPOINTS_;
run;

proc sql;
   create table growth_long_gender as
   select a._TIMEPOINTS_, a.id, a.value, b.gender
   from growth_long as a
        left join growth_labels as b
        on a.id = b.id;
quit;

/* Plot all curves colored by gender */
ods graphics / reset width=7in height=5in imagename="growth_curves_by_gender"
imagefmt=png;

proc sgplot data=growth_long_gender;
   series x=_TIMEPOINTS_ y=value / group=gender lineattrs=(thickness=1)
   transparency=0.7;
   xaxis label="Age (Years)";
   yaxis label="Height (cm)";
   title "Growth Curves Colored by Gender";
run;
ods listing close;
```

Output 12.2.1 Plot of Raw Functional Gowth Curves



The plot in Output 12.2.1 shows that boys generally exhibit smoother and taller growth curves than girls, although you can see considerable overlap among the trajectories.

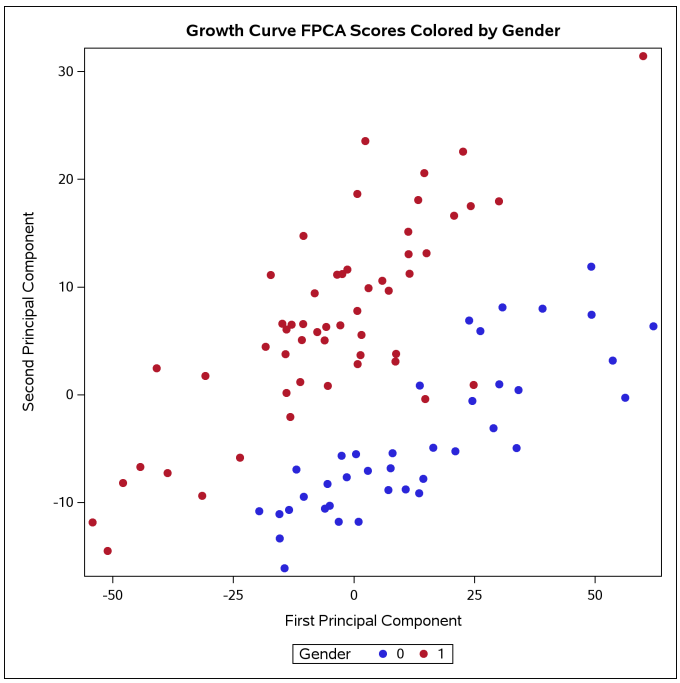
The following statements perform functional principal component analysis on the growth data by using the FPCA procedure, extract the first two principal component scores by using PROC FPCASCORE, and visualize these scores in a scatter plot that is colored by gender. Output 12.2.2 shows the results.

You start by loading the growth_curves and growth_label data into your SAS library and by using your libref in the first statement of the following DATA step. These statements assume that your SAS library is named mylib, but you can substitute any appropriately defined CAS engine libref.

```
data mylib.growth;
set growth_curves;
run;

data mylib.labels;
set growth_labels;
run;


/* Train FPCA model */
proc fpca data=mylib.growth nBins=10;
   input y1-y93;
   savestate rstore=mylib.state_fpca;
run;

/* Score same data using the analytic store */
proc fpcascore data=mylib.growth model=mylib.state_fpca;
   input y1-y93;
```

```
   output out=mylib.score_new npc=4;
run;

/* Assign child ID and merge with gender */
data score_fpca_id;
   set mylib.score_new;
   length id $4;
   id = cats("y", _n_);
run;

proc sql;
   create table score_gender as
   select a._PCS_1, a._PCS_2, a.id, b.gender
   from score_fpca_id as a
        left join growth_labels as b
        on a.id = b.id;
quit;

/* Plot FPCA scores colored by gender */
ods graphics / imagename="growth_fpca_scores" imagefmt=png;

proc sgplot data=score_gender;
   scatter x=_PCS_1 y=_PCS_2 / group=gender markerattrs=(symbol=circlefilled);
   xaxis label="First Principal Component";
   yaxis label="Second Principal Component";
   title "Growth Curve FPCA Scores Colored by Gender";
run;
ods listing close;
```

Output 12.2.2 Plot of First Two FPCA Scores by Gender



The FPCA scores in the final scatter plot, shown in Output 12.2.2, reveal clear clustering of individuals by

gender. This suggests that the dominant growth variations that FPCA captures are influenced by gender-based biological patterns.

References

Tuddenham, R. D., and Snyder, M. M. (1954). "Physical Growth of California Boys and Girls from Birth to Eighteen Years." *Publications in Child Development* 1:183–364.

Chapter 13
GMM Procedure

Contents

Overview: GMM Procedure . . . **308**
- GMM Procedure Features . . . 308
- GMM Procedure Results . . . 308
- Using SAS Viya Workbench . . . 309

Getting Started: GMM Procedure . . . **309**

Syntax: GMM Procedure . . . **313**
- PROC GMM Statement . . . 313
- DISPLAY Statement . . . 315
- DISPLAYOUT Statement . . . 316
- INPUT Statement . . . 317
- SAVESTATE Statement . . . 317
- SCORE Statement . . . 317

Details: GMM Procedure . . . **318**
- Gaussian Mixture Model . . . 319
- Dirichlet Process . . . 319
- Missing Values . . . 320
- Random Seed . . . 320
- Variational Bayesian Method . . . 320
- Covariance Matrix Type . . . 321
- Score Output . . . 321
- ClusterSumOut Output . . . 321
- ClusterCovOut Output . . . 322
- Displayed Output . . . 322
  - Number of Observations . . . 322
  - Descriptive Statistics . . . 322
  - Model Information . . . 322
  - Cluster Information . . . 323
- ODS Table Names . . . 323

Examples: GMM Procedure . . . **323**
- Example 13.1: Performing Cluster Analysis . . . 323
- Example 13.2: Performing Cluster Analysis and Storing the Model . . . 327

References . . . **329**

Overview: GMM Procedure

The GMM procedure performs clustering—a common step in data exploration—on quantitative data in SAS Viya Workbench.

You can use this procedure to read and write data in highly multithreaded form, as well as to perform clustering and scoring by making full use of multicore computers.

PROC GMM performs cluster analysis by using the Gaussian mixture model (GMM), which is a probabilistic model that assumes that all the data points are generated from a mixture of Gaussian distributions. This model can be regarded as generalizing k-means clustering to incorporate information about the covariance structure of the data as well as the centers of the latent Gaussians.

GMM Procedure Features

The GMM procedure has the following basic features:

- can execute clustering in highly multithreaded form.
- performs soft clustering, which provides not only the predicted cluster score but also the probability distribution over the clusters for each observation, as a benefit of the probabilistic modeling. For more information, see the section "Gaussian Mixture Model" on page 319.
- learns the best number of clusters during the clustering process, which is supported by the Dirichlet process. This is a nonparametric Bayesian machine learning method. For more information, see the section "Dirichlet Process" on page 319.
- uses a parallel variational Bayes (VB) method as the model inference method. VB is an inference method for probabilistic models that approximates the (intractable) posterior distribution, then iteratively updates the model parameters until it reaches convergence. For more information, see the section "Variational Bayesian Method" on page 320.

GMM Procedure Results

The GMM procedure produces results of the cluster analysis in four data tables:

- The CLUSTERCOVOUT= data table is produced by the PROC GMM statement. This data table contains the Gaussian covariance matrix of each cluster. It can be used to examine the shape of each cluster in the input data space. For more information, see the section "ClusterCovOut Output" on page 322.
- The CLUSTERSUMOUT= data table is produced by the PROC GMM statement. This data table contains the Gaussian center of each cluster. It also contains the number of observations in each cluster and the neighbor cluster of each cluster. For more information, see the section "ClusterSumOut Output" on page 321.

- The OUT= data table is produced by the SCORE statement. This data table shows the soft-clustering results that contain the cluster membership and the probability distribution over all clusters for each observation in the input data table. When the probability distribution over all clusters for each observation is given, PROC GMM shows not only the predicted cluster scores but also how it makes the predictions. For more information, see the section "Score Output" on page 321.
- The RSTORE= data table is produced by the SAVESTATE statement. This data table saves the predicted Gaussian mixture model that PROC GMM generates so that the model can be used for later data clustering. For more information, see the section "SAVESTATE Statement" on page 317.

Using SAS Viya Workbench

SAS Viya Workbench has greatly simplified the SAS programming experience by offering a syntax style nearly identical to that of SAS 9 and earlier releases. However, there are minor exceptions. In the past, SAS Viya procedures required the use of a CAS engine libref. This engine served multiple purposes, but it was primarily used to upload data to the CAS server and manage the data. Its additional purposes included support of new data types and improved performance for large data sets. No CAS server is needed for SAS Viya Workbench, because all computations are done in place for you in a single location.

In the following program, the LIBNAME statement creates a mylib libref that the examples in this chapter use to refer to your current or desired work directory:

```
libname mylib ".";
```

For more information, see *SAS Viya Workbench: Getting Started*.

Getting Started: GMM Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

This example shows how to use the GMM procedure to do soft clustering for observations in an input table.

Suppose you want to group the observations in the input table mylib.triangle, in which the variables are raw measures on interval scales.

The following DATA step creates the input data table mylib.triangle in your SAS library. This data table contains three variables: x and y are the input variables, and c is the cluster index.

```
data mylib.triangle;
   title 'Using PROC GMM for Soft Clustering';
   drop n;
   do n=1 to 100;
      x=rannor(123);
      y=rannor(123);
      c=1;
      output;
```

```
   end;
   do n=1 to 100;
      x=-3/2+rannor(123);
      y=sqrt(27)/2+rannor(123);
      c=2;
      output;
   end;
   do n=1 to 100;
      x=3/2+rannor(123);
      y=sqrt(27)/2+rannor(123);
      c=3;
      output;
   end;
run;
```

These statements assume that your SAS libref is named mylib, as in the section "Using SAS Viya Workbench" on page 309, but you can substitute any appropriately defined SAS libref.

The following statements run PROC GMM and output the results to ODS tables and output tables:

```
proc gmm
   data=mylib.triangle
   seed=123
   nThreads=32
   maxClusters=100
   alpha=1
   inference=VB (maxVbIter=100 covariance=diagonal threshold=0.01)
   clusterSumOut=mylib.clustersum
   clusterCovOut=mylib.clustercov;
   input x y;
   score out=mylib.score copyvars=(x y c);
   ods select nObs descStats modelInfo;
run;
```

Figure 13.1 shows the number of observations read and number of observations used. Because the mylib.triangle data table contains no missing values, the number of observations read and number of observations used are equal in this example. When the input data table contains missing values, PROC GMM ignores the observations (rows) with missing values in the input data table and does not use them in the analysis.

Figure 13.1 Number of Observations

Using PROC GMM for Soft Clustering

The GMM Procedure

Number of Observations Read	300
Number of Observations Used	300

Figure 13.2 shows statistics for each variable in the INPUT statement, including the mean, standard deviation, minimum, and maximum of each input variable.

Figure 13.2 Input Variable Statistics

Input Variable Statistics				
Variable	Mean	Standard Deviation	Min	Max
x	-0.037144	1.591540	-4.123222	3.955673
y	1.691436	1.620588	-3.078746	5.400477

Figure 13.3 shows the values of the parameters that are used in the clustering. The number of clusters is estimated using the Dirichlet process in a nonparametric way. In this example, three clusters are correctly detected. Figure 13.3 also shows the parameter values for the other options.

Figure 13.3 Model Information

Model Information	
Clustering Algorithm	Gaussian Mixture Model
Inference Method	Variational Bayesian
Random Number Seed	123
Clusters Maximum	100
Dirichlet Process Mass	1
Covariance Matrix Type	Diagonal
VB Iterations	66
VB Threshold	0.01
VB Converge	Yes
Clusters Discovered	3

The following statements extract the first 10 observations from the output score table and display them in the table in Figure 13.4. This figure provides not only the predicted cluster score but also the probability distribution over all clusters for each observation. As a result, you can get a sense of how the GMM procedure makes the cluster score predictions. To be specific, in Figure 13.4 PROC GMM is less confident in its prediction for the first observation than in its prediction for the others. Also note that the cluster indices in scored output are different from the ones in the input variable. For instance, cluster index 2 in _PREDICTED_CLUSTER_ matches to 1 in input c.

```
proc print noobs data=mylib.score(obs=10);
run;
```

Figure 13.4 Cluster Score Table

Using PROC GMM for Soft Clustering

x	y	c	_CLUSTER_1_	_CLUSTER_2_	_CLUSTER_3_	_PREDICTED_CLUSTER_
-0.32659	1.54244	1	0.17924	0.60946	0.21129	2
0.25903	-0.55583	1	0.00085	0.99797	0.00118	2
0.77872	-0.65520	1	0.00135	0.99834	0.00031	2
-0.02210	-0.77265	1	0.00027	0.99832	0.00141	2
0.72423	1.26331	1	0.37093	0.61718	0.01188	2
0.11544	-0.39933	1	0.00119	0.99659	0.00222	2
1.04173	-1.85990	1	0.00002	0.99996	0.00002	2
-0.63786	-0.39249	1	0.00046	0.98795	0.01159	2
-0.22034	-1.05458	1	0.00007	0.99869	0.00124	2
1.67034	-0.47339	1	0.01251	0.98743	0.00007	2

The following statements produce the cluster summary table in Figure 13.5:

```
proc print noobs data=mylib.clustersum;
run;
```

Figure 13.5 Cluster Summary Table

Using PROC GMM for Soft Clustering

_CLUSTER_ID_	_SIZE_	_NEIGHBOR_	x	y
1	105	2	1.39723	2.74285
2	111	3	-0.06628	0.04046
3	84	2	-1.82057	2.51631

In Figure 13.5 the column _SIZE_ shows the number of observations in each cluster. The column _NEIGHBOR_ shows the cluster closest to each cluster. Figure 13.5 also shows the Gaussian mean for each cluster.

The following statements produce the cluster covariance table in Figure 13.6:

```
proc print noobs data=mylib.clustercov;
run;
```

Figure 13.6 Cluster Covariance Table

Using PROC GMM for Soft Clustering

_CLUSTER_ID_	x	y
1	1.08201	0.00000
1	0.00000	0.81649
2	0.78780	0.00000
2	0.00000	1.10321
3	0.80359	0.00000
3	0.00000	1.19961

Figure 13.6 shows the Gaussian covariance matrices of the Gaussian mixture.

Syntax: GMM Procedure

The following statements are available in the GMM procedure:

```
PROC GMM <options> ;
    DISPLAY <table-list> </options> ;
    DISPLAYOUT <table-spec-list> </options> ;
    INPUT variables <INTERVAL> ;
    SCORE OUT=libref.data-table <options> ;
    SAVESTATE RSTORE=libref.data-table ;
```

The PROC GMM statement and an INPUT statement are required. The PROC GMM statement takes only interval input variables.

The following sections describe the PROC GMM statement and then describe the other statements in alphabetical order.

PROC GMM Statement

```
PROC GMM <options> ;
```

The PROC GMM statement invokes the procedure. Table 13.1 summarizes the *options* available in the PROC GMM statement.

Table 13.1 PROC GMM Statement Options

Option	Description
Input and Output Data Set Option	
DATA=	Specifies the input data table
Performance Option	
NTHREADS=	Specifies the number of threads to use for the computation
Clustering Options	
ALPHA=	Specifies the mass parameter of the Dirichlet process
CLUSTERCOVOUT=	Specifies the output data table to contain the Gaussian covariance matrices
CLUSTERSUMOUT=	Specifies the output data table to contain the cluster summary
INFERENCE=	Specifies the inference method for the Gaussian mixture model
MAXCLUSTERS=	Specifies the maximum number of possible clusters
SEED=	Specifies the seed to use for pseudorandom number generation

You can specify the following *options*:

ALPHA=*number*

specifies the mass parameter of the Dirichlet process. The value of *number* must be a positive real number. When you specify a larger *number*, the Dirichlet process tends to discover more clusters in the input data. By default, ALPHA=1.

CLUSTERCOVOUT=*libref.data-table*

creates the output data table that contains the summary of the clustering results. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 309.

This data table includes the covariance matrices of the Gaussian distributions for the clusters that PROC GMM discovers. The table contains the cluster ID (_CLUSTER_ID_) and the covariance matrices of the Gaussian distributions of the discovered clusters, which consist of the nonconstant variables specified in the INPUT statement. For more information, see the section "ClusterCovOut Output" on page 322.

CLUSTERSUMOUT=*libref.data-table*

creates the output data table that contains the summary of the clustering results. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 309.

The summary consists of the following items: the cluster ID (_CLUSTER_ID_), the numbers of observations in the clusters (_SIZE_), the clusters closest to the clusters (_NEIGHBOR_), and the cluster centroids (the means of the Gaussian distributions, which consist of the nonconstant variables specified in the INPUT statement). For more information, see the section "ClusterSumOut Output" on page 321.

DATA=*libref.data-table*

names the input data table for PROC GMM to use. The default is the most recently created data table. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to the data. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 309.

data-table specifies the name of the input data table.

INFERENCE=VB < *suboptions* >

performs the inference by using the variational Bayesian (VB) algorithm; for more information see the section "Variational Bayesian Method" on page 320.

You can specify one or more of the following *suboptions*:

COVARIANCE=*keyword* specifies the type of covariance matrices in the Gaussian mixture model. For more information, see the section "Covariance Matrix Type" on page 321. You can specify the following *keywords*:

DIAGONAL specifies the covariance matrices of the Gaussian distributions as diagonal matrices.

FULL specifies the covariance matrices of the Gaussian distributions as full matrices.

By default, COVARIANCE=DIAGONAL.

MAXVBITER=*number* specifies the maximum number of VB iterations. The VB iterations stop either when the number of iterations reaches the *number* or when the change in the Kullback-Leibler distance between the approximated distribution q and the true posterior distribution p between two consecutive VB iterations is lower than the threshold specified by THRESHOLD=*number*. When the VB iterations stop with the threshold criterion satisfied, the VB reaches its convergence. By default, MAXVBITER=100.

THRESHOLD=*number* specifies the threshold to define the convergence of the VB algorithm. By default, THRESHOLD=0.01.

MAXCLUSTERS=*number*
specifies the maximum number of clusters that PROC GMM searches to find the best number of clusters by using the Dirichlet process. By default, MAXCLUSTERS=100.

NTHREADS=*number-of-threads*
specifies the number of threads to use for the computation. The default value is the number of CPUs available in the machine.

SEED=*number*
specifies an integer to use to start the pseudorandom number generator. If you do not specify a seed or if you specify a value less than or equal to zero, the seed is generated by reading the time of day from the computer's clock. For more information, see the section "Random Seed" on page 320.

DISPLAY Statement

DISPLAY < *table-list* > < / *options* > **;**

The DISPLAY statement enables you to specify a list of display tables to display or exclude. This statement is similar to the ODS SELECT, ODS EXCLUDE, and ODS TRACE statements. However, the DISPLAY statement can improve performance when a large number of tables could be generated (such as in BY-group processing). The DISPLAY statement produces only the requested tables, but ODS statements generate all the tables and then keep the requested tables.

If you use both DISPLAY and ODS statements together, the DISPLAY statement takes precedence over the ODS statements. Note that the ODS EXCLUDE statement processes tables after they have been filtered by the DISPLAY statement. In some cases, it might appear that the ODS EXCLUDE statement is taking precedence because it can further filter the tables. For more information about ODS, see *SAS Output Delivery System: Procedures Guide*.

You can specify the *table-list* as a list of table names, paths, partial pathnames, and regular expressions.

The table names that you can specify are listed in the section "ODS Table Names" on page 323. A path is a table name that is prefixed with dot-separated grouping information. For example, a

SelectionSummary table that a procedure produces during a selection routine might have the path *Bygroup1.Summary.SelectionSummary*. A partial pathname does not include all groups; for example, *SelectionSummary* and *Summary.SelectionSummary* are partial pathnames for *Bygroup1.Summary.SelectionSummary*.

When you specify a table name or partial pathname, all display tables whose paths end in the specified name are selected for display or exclusion. For example, both *SelectionSummary* and *Summary.SelectionSummary* select *Bygroup1.Summary.SelectionSummary*.

A regular expression is enclosed in forward slashes (/). For example, specifying "/tions/" selects all pathnames that contain the substring "tions"; in particular, the *Bygroup1.Summary.SelectionSummary* table is selected. Specifying "!/tions/" selects all pathnames that do not contain the substring "tions"; in particular, the *Bygroup1.Summary.SelectionSummary* table is not selected.

You can specify the following *options* after a slash (/):

CASESENSITIVE
performs a case-sensitive comparison of table names in the *table-list* to display table names when tables are subsetted for display. To preserve case, you must enclose table names in the *table-list* in quotation marks.

EXCLUDE
displays all display tables except those that you specify in the *table-list*.

EXCLUDEALL
suppresses display of all tables. This option takes precedence over the other options.

TRACE
displays the display table names, labels, and paths.

DISPLAYOUT Statement

DISPLAYOUT *table-spec-list* < / *options* > **;**

The DISPLAYOUT statement enables you to create output tables from your displayed output. This statement is similar to the ODS OUTPUT statement. For more information about ODS, see *SAS Output Delivery System: Procedures Guide*.

The *table-spec-list* specifies a list of output tables to create. Each entry in the list has either a *key*=*value* format or a *key* format:

key=*value* specifies *key* as the ODS table name, path, or partial pathname, and specifies *value* as the output table name.

key specifies *key* as the ODS table name and also as the output table name.

The ODS table names that you can specify are listed in the section "ODS Table Names" on page 323. You cannot specify the ODS table named OutputCasTables in the *table-spec-list*.

Table names and partial pathnames are discussed under the DISPLAY statement. The DISPLAYOUT statement does not support regular expressions.

You can specify the following *options* after a slash (/):

INCLUDEALL
creates output tables for all display tables. The name of the created output table is the same as the name of the corresponding display table. If you specify this option, the *table-spec-list* specification is ignored.

NOREPLACE
does not replace any existing output table of the same name.

INPUT Statement

INPUT *variables* < **INTERVAL** > **;**

The INPUT statement specifies the names of the *variables* to use in clustering. The GMM procedure uses only interval input variables for its clustering.

Any input interval variable that has a standard deviation of zero indicates a constant variable (column) in the input data table. Such constant variables do not contribute to the clustering and are ignored.

SAVESTATE Statement

SAVESTATE RSTORE=*libref.data-table* **;**

The SAVESTATE statement creates an analytic store for the model and saves it as a binary object in a data table. You can then use the analytic store in the ASTORE procedure to score new data. For more information, see Chapter 4, "ASTORE Procedure."

You must specify the following option:

RSTORE=*libref.data-table*
specifies a data table in which to save the analytic store for the model. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 309.

SCORE Statement

SCORE OUT=*libref.data-table* < *option* > **;**

OUTPUT OUT=*libref.data-table* < *option* > **;**

The SCORE statement writes the cluster membership information of each observation to the output data table that is specified in the OUT= option. This information includes the variables that are specified in the COPYVARS= option, the predicted cluster scores, and the probability distributions over the discovered clusters. The probability value can be interpreted as the probability of that observation being a member of that cluster. This information enables you to use the GMM procedure as a soft-clustering technique. The details are described in the section "Score Output" on page 321.

You must specify the following option:

OUT=*libref.data-table*

names the output data table for PROC GMM to use. You must specify this option before any other options. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to where the data table is to be stored. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 309.

data-table specifies the name of the output data table.

The output data table contains the scored data. When you specify this option, the table contains all variables that are specified in the COPYVARS= option, followed by the probability distribution over the discovered clusters, and then the predicted cluster score.

You can also specify the following *option*:

COPYVAR=*variable*

COPYVARS=(*variables***)**

lists one or more variables to transfer from the input data table to the scored output data table, provided that the output data table produces one or more records per input observation. By default, the GMM procedure does not transfer any variables from the input data table to the output data table. The COPYVARS= option accepts numeric and character variables. You can also use COPYVARS=(_ALL_) to include all the input variables.

Details: GMM Procedure

The GMM procedure performs soft clustering by using the Gaussian mixture model on the data in the input data table. The number of clusters among the input data is inferred by using the Dirichlet process in a nonparametric style. PROC GMM computes the following statistics as the model output:

- the input variable statistics, including the mean, standard deviation, minimum, and maximum
- a summary of the clustering results, including the number of clusters, the number of observations in the clusters, and the neighbor clusters
- the means, which indicate the centers, and the covariances, which indicate the shape of the discovered clusters
- the predicted cluster scores and probability distributions over discovered clusters for the observations

You can use these statistics to analyze the clustering results.

Gaussian Mixture Model

The Gaussian mixture model is a probabilistic model that assumes that all the data points are generated from a mixture of Gaussian distributions; see Rasmussen (1999). The Gaussian mixture model can be regarded as generalizing the k-means clustering to incorporate information about the covariance structure of the data as well as the centers of the latent Gaussians. In probabilistic modeling, the probability distribution over all the discovered clusters is inferred for each observation. Thus the Gaussian mixture model provides soft clustering, which is different from the hard clustering that is provided by k-means, hierarchical, and density-based spatial clustering.

In the GMM procedure, the Gaussian mixture model is Bayesian: the parameters in the Gaussian mixture model (the means and covariances of the Gaussian distributions and the proportion of the Gaussian mixture) are generated along with their prior distributions; conditioned on these parameters, the observations are generated along with the likelihood distributions; and the parameters in the model are estimated by their posterior distributions. To be specific, the prior and likelihood distributions are listed as follows, where μ is the Gaussian mean; Σ is the Gaussian covariance matrix; π is the proportion of the Gaussian mixture; K is the number of clusters discovered; N is the total number of observations in the input data table; and α_0 and H_0 are the mass and base measure, respectively, of the Dirichlet process H:

- Priors:

  $\mu_k \sim N(\mu_0, \Sigma_0), \quad k = 1, 2, \ldots, K$
  $\Sigma_k^{-1} \sim \mathrm{Wishart}(k_0 \Sigma_0^{-1}, m_0), \quad k = 1, 2, \ldots, K$
  $\pi \sim \mathrm{DP}(\alpha_0, H_0)$

- Likelihood:

  $x_i \sim \sum_{k=1}^{K} \pi_k N(\mu_k, \Sigma_k), \quad i = 1, 2, \ldots, N$

Dirichlet Process

The GMM procedure uses the Dirichlet process in the Gaussian mixture model to find the best number of clusters among the input data. The nonparametric Bayes approach uses stochastic processes instead of probability distributions in probabilistic models to achieve better model flexibility and adaptivity. In PROC GMM, the Dirichlet process uses the Gaussian mixture model, with its ability to add new clusters and remove existing clusters during the clustering process, thus finding the best number of clusters adaptively.

In PROC GMM, the Dirichlet process serves as the prior for the proportion of the Gaussian mixture. The likelihood is simply a multinomial distribution over the clusters. To be specific, the prior and posterior for the proportion of the Gaussian mixture are listed as follows, where K is the discovered number of clusters; $n_1, n_2, \ldots, n_K$ is the number of observations in the K clusters; $N = \sum_k^K n_k$ is the total number of observations in the input data table; and α_0 and H_0 are the mass and base measure, respectively, of the Dirichlet process:

- Priors:

  $H \sim \mathrm{DP}(\alpha_0, H_0)$

- Posterior:
  $H|n_1, n_2, \ldots, n_K \sim \mathrm{DP}(\alpha_0 + N, \frac{\alpha_0}{\alpha_0+N} H_0 + \sum_k \frac{n_k}{\alpha_0+N})$

Missing Values

The GMM procedure does not perform missing data imputation. If the input data table contains any missing data, PROC GMM simply ignores the observations (rows) that contain missing values. You can use any data preprocessing steps, such as the VARIMPUTE procedure, to impute the missing data before you use the input data table in the GMM procedure.

Random Seed

You can specify the random seed by using the SEED= option in the PROC GMM statement. The seed value is used in the random number generator, which initializes the cluster assignment to start the iterations of the variational Bayesian algorithm. If a random seed is not specified or if the value is invalid (zero or negative), then the GMM procedure generates a random number by using the time of day from the computer's clock. Because the random seed determines the initial state of the cluster assignment, different seeds can produce different clustering results.

Variational Bayesian Method

The GMM procedure uses a parallel variational Bayesian (VB) method of model inference. Variational Bayesian methods are a family of techniques for approximating the intractable integrals that arise in the posterior distributions of Bayesian models with computable alternative distributions. The VB algorithm that is applied in PROC GMM is the mean-field VB, where the approximation criterion is to minimize the Kullback-Leibler distance to the true model posterior distribution. To be specific, the mean-field principle is as follows, where p is the true posterior distribution and q is the mean-field approximated distribution:

$$q = \mathrm{argmin}_q KL(q||p) = \mathrm{argmin}_q \int q \log(\frac{q}{p})$$

In PROC GMM, the variational Bayesian method can produce nondeterministic clustering or density estimation results. Outcomes depend on data partitioning across threads or workers, as well as on the order of processing within minibatches. Changing the number of threads or workers can therefore yield meaningfully different posterior approximations and clustering even with the same seed.

For a general introduction to VB inference, see Bishop (2006). For a specific VB inference for the Gaussian mixture model, see Blei and Jordan (2006).

Covariance Matrix Type

The VB inference that PROC GMM uses provides a choice of two covariance matrix types for the Gaussian distributions in the Gaussian mixture model: diagonal and full.

The full type of covariance matrix represents a complete form of the Gaussian covariance matrices. The model's time complexity is quadratic with respect to the number of variables that the model uses.

The diagonal type of covariance matrix represents a diagonal form of the Gaussian covariance matrices. The model's time complexity is linear with respect to the number of variables that the model uses. In some cases, when the input variables are independent of each other, the diagonal type can yield good results and a faster processing time than the full type.

Score Output

The GMM procedure writes the cluster membership information, including both the predicted cluster scores and the probability distributions over the clusters, to the output table that is specified in the OUT= option in the SCORE statement. This information also includes the variables that are specified in the COPYVARS= option and the following variables:

- _CLUSTER_1_ to _CLUSTER_K_ (the probability distribution over all the K clusters)
- _PREDICTED_CLUSTER_ (the predicted cluster ID)

ClusterSumOut Output

The GMM procedure presents the summary of the clustering results in the table that is specified in the CLUSTERSUMOUT= option in the PROC GMM statement. In this table, each row contains the following results for a cluster: the mean of the Gaussian distribution of the cluster, the number of observations in the cluster, and the neighbor cluster whose Gaussian mean is closest to the Gaussian mean of the cluster.

This table contains all the variables that the Gaussian mixture model uses to indicate the Gaussian means (the constant variables with zero standard deviation are excluded) and the following variables to indicate the statistics of the clusters:

- _CLUSTER_ID_ (the cluster ID)
- _SIZE_ (the number of observations in the cluster specified by _CLUSTER_ID_)
- _NEIGHBOR_ (the cluster closest to the cluster specified by _CLUSTER_ID_)

ClusterCovOut Output

The GMM procedure presents the covariance matrices of the Gaussian distributions in the table that is specified in the CLUSTERCOVOUT= option in the PROC GMM statement. The covariance matrix of a cluster indicates the shape of the cluster; this contrasts with other clustering algorithms, such as the *k*-means clustering algorithm, where the only statistics that the clustering algorithm knows are the distances of the observations to the centroids of their clusters. You can further see the clustering advantage of using the GMM procedure in Example 13.1.

This table contains all the variables used that the Gaussian mixture model uses to indicate the Gaussian covariances (the constant variables with a standard deviation of zero are excluded) and the _CLUSTER_ID_ variable, which identifies the rows with the same cluster ID that form the covariance matrix of a cluster.

Displayed Output

The GMM procedure displays various tables for use in cluster analysis. The following sections describe the output tables in the order of their appearance when the related options are specified.

Number of Observations

The "Number of Observations" table displays the number of observations that are read from the input data table and the number of observations that are used in the clustering. The Number of Observations Read column displays the count of raw data that are read from the input data table, and the Number of Observations Used column displays the count of data that are actually used in the clustering. The count in the Number of Observations Used column can be smaller if you exclude the observations (rows) that contain missing values in the input data table.

Descriptive Statistics

The "Descriptive Statistics" table displays the mean, standard deviation, minimum, and maximum for each interval variable that is specified in the INPUT statement. The calculation of these statistics is based on all observations (rows) that do not contain missing values in the input data table.

Any input variable whose standard deviation is zero is a constant variable (column) in the input data table. Such constant variables do not contribute to the clustering, and they are not used in the Gaussian mixture model.

Model Information

The "Model Information" table provides basic information about the parameters that are used in the Gaussian mixture model. This information includes the clustering algorithm, inference method, random number seed, maximum number of clusters, Dirichlet process mass, covariance matrix type, variational Bayesian (VB) iterations, VB threshold, VB convergence, and number of clusters discovered.

Cluster Information

The ClusterInfo table provides basic information about the clustering results. This information includes the cluster weights, cluster means, and cluster variances (the diagonal elements of the cluster covariances).

ODS Table Names

Each ODS table that the GMM procedure creates has a name associated with it. You must use this name to refer to the table when you use ODS statements. These names are listed in Table 13.2.

Table 13.2 ODS Tables Produced by PROC GMM

Table Name	Description	Statement	Option
ClusterInfo	Cluster information	PROC GMM	Default output
DescStats	Descriptive statistics for interval input variables	PROC GMM	Default output
ModelInfo	Model information	PROC GMM	Default output
NObs	Number of observations	PROC GMM	Default output

Examples: GMM Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

Example 13.1: Performing Cluster Analysis

This example uses the MouseEars data table that is generated by the following SAS code to demonstrate how to use the GMM procedure to perform cluster analysis. There are three clusters in the MouseEars data: a bigger cluster of the "head," with a radius of 0.9, and two smaller clusters of "ears," with a radius of 0.5.

These data are a simple example of clusters of varying shapes, but they are sufficient to demonstrate the advantage of the GMM procedure over the *k*-means clustering algorithm. For this example, because PROC GMM is equipped with the covariances of Gaussian distributions, it performs well in separating the three varying clusters. By contrast, the *k*-means algorithm cuts parts of the head cluster and assigns those parts to the two ear clusters.

The following statements generate the MouseEars data table, which has three Gaussian clusters (one face and two ears):

```
data mylib.mickey;
   title "Using PROC GMM for Varying Clusters";
   drop n;
   seed=12345;
   do n=1 to 200;
      x=2.5+rannor(seed)*0.9;
      y=rannor(seed)*0.9;
      c=1;
      output;
   end;
   do n=1 to 50;
      x=rannor(seed)*0.5;
      y=2.5+rannor(seed)*0.5;
      c=2;
      output;
   end;
   do n=1 to 50;
      x=5+rannor(seed)*0.5;
      y=2.5+rannor(seed)*0.5;
      c=2;
      output;
   end;
run;
```

These statements assume that your SAS library is named mylib, but you can substitute any appropriately defined SAS library.

The following statements run the GMM procedure and send the results to output tables and ODS tables. In this example, COVARIANCE= FULL is specified to capture the possible variations between the clusters. Also, the MAXVBITER= option is specified using a large value and the THRESHOLD= option is specified using a small value to allow a sufficient VB convergence.

```
proc gmm
   data=mylib.mickey
   seed=12345
   nThreads=32
   maxClusters=100
   alpha=1
   inference=VB (maxVbIter=1000 covariance=FULL threshold=0.001)
   clusterSumOut=mylib.clustersum
   clusterCovOut=mylib.clustercov;
   savestate rstore=mylib.gmmAstore;
   input x y;
   score out=mylib.score copyvars=(x y c);
   ods select nObs descStats modelInfo clusterInfo;
run;
```

The GMM procedure generates four ODS tables, which are shown in Output 13.1.1 through Output 13.1.4.

Output 13.1.1 Number of Observations

Using PROC GMM for Varying Clusters

The GMM Procedure

Number of Observations Read	300
Number of Observations Used	300

Output 13.1.2 Descriptive Statistics

Input Variable Statistics

Variable	Mean	Standard Deviation	Min	Max
x	2.460096	1.601650	-0.901323	5.715064
y	0.792385	1.459254	-2.635585	3.899748

Output 13.1.3 Model Information

Model Information

Clustering Algorithm	Gaussian Mixture Model
Inference Method	Variational Bayesian
Random Number Seed	12345
Clusters Maximum	100
Dirichlet Process Mass	1
Covariance Matrix Type	Full
VB Iterations	57
VB Threshold	0.001
VB Converge	Yes
Clusters Discovered	3

Output 13.1.4 Cluster Information

Cluster Information

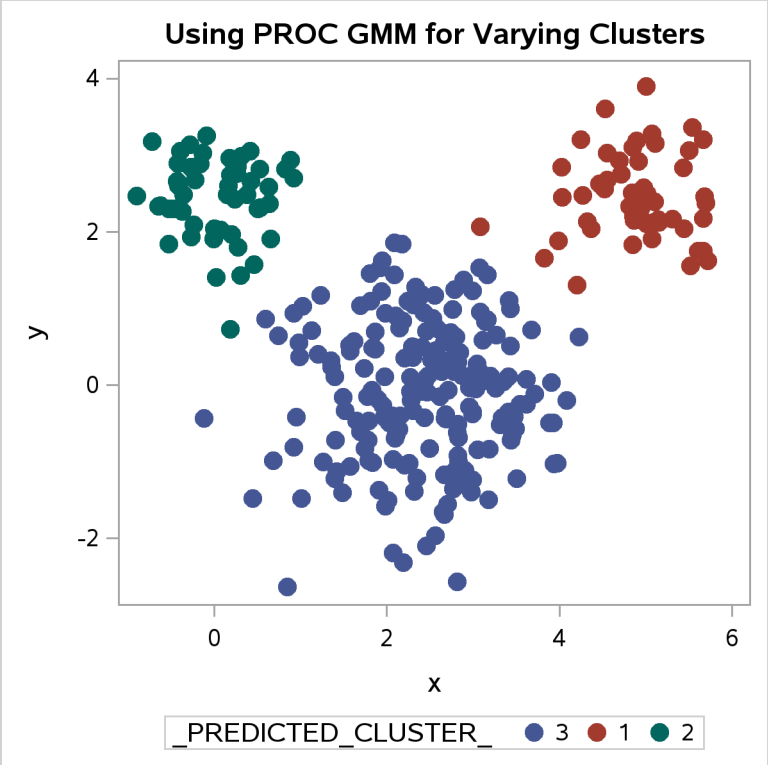
Cluster ID	Cluster Weight	x Mean	y Mean	x Variance	y Variance
1	0.1847208924	4.7923728373	2.3847821927	0.8368114503	0.4856846864
2	0.1704912308	0.0876735822	2.415116746	0.2476087198	0.4147477139
3	0.6447878769	2.4196158379	-0.105014816	0.666632776	0.8276400992

In this example, the GMM procedure generates the cluster score table mylib.score; For more information, see the section "Score Output" on page 321. The following PROC SGPLOT statements produce the scatter plot of mylib.score. PROC GMM performs well in separating the head cluster from the two ear clusters, as shown in Output 13.1.5. Only cluster scores are shown in Output 13.1.5.

```
ods graphics / reset width=4in height=4in imagename='Clusters';
proc sgplot data=mylib.score;
   scatter x=x y=y / group=_predicted_cluster_
```

```
   markerattrs=(symbol=CircleFilled size=9);
run;
```

Output 13.1.5 Cluster Scores



The GMM procedure generates the cluster summary table mylib.clusterSum. The following statements print this table, which is shown in Output 13.1.6. For more information, see the section "ClusterSumOut Output" on page 321. In this table, you can see that PROC GMM performs well in assigning the observations to the correct clusters and also finds the correct cluster centers.

```
proc print noobs data=mylib.clustersum;
run;
```

Output 13.1.6 Cluster Summary Table

Using PROC GMM for Varying Clusters

_CLUSTER_ID_	_SIZE_	_NEIGHBOR_	x	y
1	54	3	4.79237	2.38478
2	51	3	0.08767	2.41512
3	195	2	2.41962	-0.10501

The GMM procedure also generates the cluster covariance table mylib.clusterCov. The following statements print this table, which is shown in Output 13.1.7. For more information, see the section "ClusterCovOut Output" on page 322. In this table, you can see that PROC GMM successfully captures the variation in the shapes of the clusters.

```
proc print noobs data=mylib.clustercov;
run;
```

Output 13.1.7 Cluster Covariance Table

Using PROC GMM for Varying Clusters

_CLUSTER_ID_	x	y
1	0.83681	0.29120
1	0.29120	0.48568
2	0.24761	-0.02656
2	-0.02656	0.41475
3	0.66663	0.00302
3	0.00302	0.82764

The GMM procedure also generates the analytic store table by using the SAVESTATE statement. The analytic store table saves the Gaussian mixture model for later use after PROC GMM finishes running on the MouseEars data, as shown in Example 13.2.

Example 13.2: Performing Cluster Analysis and Storing the Model

This example uses the Fish data set in the Sashelp library to demonstrate how to use the GMM procedure to perform cluster analysis and how to use the analytic store to save the model and use the saved model for future clustering.

The Fish data set contains 159 observations and seven variables. Among these variables, Species contains different fish species and the other six numeric variables contain statistics of the fish: height, weight, width, and three measures of length.

Because the statistics of fish in different species overlap, the clusters that PROC GMM discovers do not exactly match the fish species in the Species variable. However, PROC GMM provides a good analysis of the inhomogeneity in the data. Besides, this example shows you how to save the Gaussian mixture model in the analytic store so that you can apply this saved model to other data.

The following DATA step divides the mylib.Fish data into training and testing data tables. PROC GMM uses the first 150 observations in the mylib.Fish data for clustering and saves the model, and it uses the saved model and the last 9 observations directly for cluster score prediction.

```
data mylib.fish_train;
   title "Using PROC GMM for Cluster Prediction";
   set sashelp.fish (obs=150);
run;
data mylib.fish_test;
   set sashelp.fish (firstobs=151);
run;
```

These statements assume that your SAS library is named mylib, but you can substitute any appropriately defined SAS library.

The following statements run PROC GMM on the training data and save the trained model in the table mylib.astore:

```
proc gmm
   data=mylib.fish_train
   nThreads=32
   seed=1234567890
   maxClusters=100
   alpha=1
   inference=VB (maxVbIter=1000 covariance=DIAGONAL threshold=0.001)
   clusterSumOut=mylib.clustersum
   clusterCovOut=mylib.clustercov;
   input _NUMERIC_;
   score out=mylib.score copyvars=(_ALL_);
   ods select nObs descStats modelInfo;
   savestate rstore=mylib.astore(replace=yes);
run;
```

The following statements run the ASTORE procedure with the saved model on the testing data, and output the cluster scores in the mylib.newscore table:

```
proc astore;
   score data=mylib.fish_test out=mylib.newscore
   copyvars=(_ALL_) rstore=mylib.astore;
run;
```

The following statements use PROC PRINT to display the predictions about the testing data in the mylib.newscore table, as shown in Output 13.2.1. In this table, you can see that all nine fish that belong to the smelt species are assigned (very confidently) to the correct cluster, just as they are in the Fish training data.

```
proc print noobs data=mylib.newscore (obs=9);
run;
```

Output 13.2.1 Cluster Scores on Testing Data

Using PROC GMM for Cluster Prediction

_CLUSTER_1_	_CLUSTER_2_	_CLUSTER_3_	_CLUSTER_4_	_CLUSTER_5_	_CLUSTER_6_	_CLUSTER_7_
1.1102E-12	1.1102E-12	1.1102E-12	1.1102E-12	1.00000	1.1102E-12	1.1102E-12
1.1102E-12	1.1102E-12	1.1102E-12	1.1102E-12	1.00000	1.1102E-12	1.1102E-12
1.1102E-12	1.1102E-12	1.1102E-12	1.1102E-12	1.00000	1.1102E-12	1.1102E-12
1.1102E-12	1.1102E-12	1.1102E-12	1.1102E-12	1.00000	1.1102E-12	1.1102E-12
1.1102E-12	1.1102E-12	1.1102E-12	1.1102E-12	1.00000	1.1102E-12	1.1102E-12
1.1102E-12	1.1102E-12	1.1102E-12	1.1102E-12	1.00000	1.1102E-12	1.1102E-12
1.1102E-12	1.1102E-12	1.1102E-12	1.1102E-12	1.00000	1.1102E-12	1.1102E-12
1.1102E-12	1.1102E-12	1.1102E-12	1.1102E-12	1.00000	1.1102E-12	1.1102E-12
1.1102E-12	1.1102E-12	1.1102E-12	1.1102E-12	1.00000	1.1102E-12	1.1102E-12

_CLUSTER_8_	_PREDICTED_CLUSTER_	Species	Weight	Length1	Length2	Length3	Height	Width
1.1102E-12	5	Smelt	8.7	10.8	11.3	12.6	1.9782	1.2852
1.1102E-12	5	Smelt	10.0	11.3	11.8	13.1	2.2139	1.2838
1.1102E-12	5	Smelt	9.9	11.3	11.8	13.1	2.2139	1.1659
1.1102E-12	5	Smelt	9.8	11.4	12.0	13.2	2.2044	1.1484
1.1102E-12	5	Smelt	12.2	11.5	12.2	13.4	2.0904	1.3936
1.1102E-12	5	Smelt	13.4	11.7	12.4	13.5	2.4300	1.2690
1.1102E-12	5	Smelt	12.2	12.1	13.0	13.8	2.2770	1.2558
1.21655E-8	5	Smelt	19.7	13.2	14.3	15.2	2.8728	2.0672
3.32857E-8	5	Smelt	19.9	13.8	15.0	16.2	2.9322	1.8792

References

Bishop, C. M. (2006). *Pattern Recognition and Machine Learning*. Information Science and Statistics. New York: Springer.

Blei, D. M., and Jordan, M. I. (2006). "Variational Inference for Dirichlet Process Mixtures." *Bayesian Analysis* 1:121–144.

Rasmussen, C. E. (1999). "The Infinite Gaussian Mixture Model." In *NIPS '99: Proceedings of the 12th International Conference on Neural Information Processing Systems*, 554–560. Cambridge, MA: MIT Press.

Chapter 14
GPCLASS Procedure

Contents

Overview: GPCLASS Procedure . . . **332**
- PROC GPCLASS Features . . . 332
- Using SAS Viya Workbench . . . 333

Getting Started: GPCLASS Procedure . . . **333**

Syntax: GPCLASS Procedure . . . **337**
- PROC GPCLASS Statement . . . 337
- AUTOTUNE Statement . . . 339
- DISPLAY Statement . . . 343
- DISPLAYOUT Statement . . . 344
- INFERENCE Statement . . . 345
- INPUT Statement . . . 345
- KERNEL Statement . . . 345
- OUTPUT Statement . . . 346
- SAVESTATE Statement . . . 347
- TARGET Statement . . . 347

Details: GPCLASS Procedure . . . **347**
- Gaussian Process . . . 347
- Laplacian Approximation . . . 348
- Inference . . . 348
- Link Function . . . 348
- Kernel . . . 349
- Missing Values . . . 349
- Random Seed . . . 349
- Displayed Output . . . 350
  - Number of Observations . . . 350
  - Descriptive Statistics . . . 350
  - Model Information . . . 350
  - Iteration Statistics . . . 350
  - Predicted Variable Names . . . 350
- Table Output . . . 351
  - Classification Score Table . . . 351
  - Analytic Store Table . . . 351
- ODS Table Names . . . 351

Examples: GPCLASS Procedure . . . **352**
- Example 14.1: Performing Classification on Iris Data . . . 352
- Example 14.2: Tuning the Hyperparameters of a Gaussian Process Classification Model 355

References . . . **358**

Overview: GPCLASS Procedure

The GPCLASS procedure performs classification—a common step in data exploration—on quantitative data in SAS Viya Workbench.

This procedure performs binary classification by using Gaussian process classification (GPC), which is a method of probabilistic modeling that assumes that an underlying function is generated by a Gaussian process (GP) that determines the class labels of the data. As a nonparametric method, GPC allows the model parameters to adapt to the input data. To be specific, the Gaussian process adaptively learns the parameters of the latent function that lies in its corresponding reproducing kernel Hilbert space (RKHS), which is defined by the kernel of the Gaussian process, thus enabling good prediction performance.

PROC GPCLASS Features

Gaussian process classification is a method of probabilistic modeling that assumes that all the data points are generated by a latent function plus some Gaussian noise. The Gaussian process is used as the prior to learn this underlying function. In probabilistic modeling, the probability distribution over the two classes is inferred for each observation. Thus Gaussian process classification provides not only the predicted class labels but also the probabilities that correspond to the class labels; this is referred to as soft classification. It is different from the hard classification that is provided by support vector machines (SVM), decision trees, and logistic regression, where only the predicted class labels are provided.

In PROC GPCLASS, the Gaussian process model is Bayesian: the parameters in the Gaussian process's kernel are generated along with their prior distributions; conditioned on these parameters, the observations are generated along with the likelihood distributions; and the parameters in the model are estimated by their posterior distributions. To be specific, the prior and likelihood distributions are listed as follows, where $f(x)$ is the latent function; GP is the Gaussian process; K is the Gaussian process kernel; x is the observation, and y is the corresponding class label; L is the likelihood function, which maps the values of the latent function to binary labels; and N is the total number of observations in the input data table:

- Priors:
  $f(x) \sim \mathrm{GP}(0, K)$

- Likelihood:
  $p(y_i | f(x_i)) = L(y_i f(x_i)), \quad i = 1, 2, \ldots, N$

PROC GPCLASS has the following basic features:

- performs soft classification, which provides not only class labels but also the probabilities over the two classes for the input data, as a benefit of the probabilistic modeling
- learns the best parameters during the classification process, which is supported by the Gaussian process. This is a nonparametric Bayesian machine learning method. For more information, see the section "Gaussian Process" on page 347.

- uses a Laplacian approximation (LA) method as the model inference method. LA is an inference method for probabilistic models that approximates the (intractable) posterior distribution by using a Gaussian distribution. It uses Newton iterations to update the model parameters until it reaches convergence. For more information, see the section "Laplacian Approximation" on page 348.

Using SAS Viya Workbench

SAS Viya Workbench has greatly simplified the SAS programming experience by offering a syntax style nearly identical to that of SAS 9 and earlier releases. However, there are minor exceptions. In the past, SAS Viya procedures required the use of a CAS engine libref. This engine served multiple purposes, but it was primarily used to upload data to the CAS server and manage the data. Its additional purposes included support of new data types and improved performance for large data sets. No CAS server is needed for SAS Viya Workbench, because all computations are done in place for you in a single location.

In the following program, the LIBNAME statement creates a mylib libref that the examples in this chapter use to refer to your current or desired work directory:

```
libname mylib ".";
```

For more information, see *SAS Viya Workbench: Getting Started*.

Getting Started: GPCLASS Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

This example uses the Toy data set to demonstrate how to use the GPCLASS procedure to perform classification, and then how to use the analytic store to save the trained model and use the saved model for future classification.

The Toy data set contains 40 observations and three variables. The variable label contains the two binary classes 1 and 0, and the other two numeric variables contain statistics about the location.

The following DATA steps load the Toy data set into your session and then divide the data into the training data table, the testing data table, and another testing data table for future use with the saved model. PROC GPCLASS then uses the first 30 observations in data for training and the next 10 observations for testing. Because the ASTORE procedure is specified in this example, the trained model is saved for future prediction.

```
data toy;
   input id x y label;
   datalines;
   1 0.7562437251693241 -0.4449246304395207 1
   2 1.6425910060420805 -0.10293958159084055 1
   3 0.5036628219528629 -0.3756019302569773 1
   4 1.9356458876375808 0.1508435933753537 1
   5 -0.48929352371915874 0.9004153510746639 0
   6 1.9612377920931525 0.3688561398322367 1
```

```
   7 -0.4457973476327784 0.923703991351446 0
   8 0.897645351230446 0.13625291562699787 0
   9 1.0428050419382913 0.2538945316161072 0
   10 1.7841025147461618 -0.03250971343343456 1
   11 0.6525149362580007 -0.5043431552576471 1
   12 1.8526407982796789 0.13718466658924755 1
   13 0.7736555917316961 -0.48367703386369615 1
   14 -0.9816672335041904 0.33290929557635607 0
   15 0.24612298119546702 -0.1868663956140018 1
   16 1.99621877108224 0.3245177413955755 1
   17 -0.04791750586561572 0.2659542532295433 1
   18 2.110087224576905 0.5250574025280107 1
   19 -0.9602167257824953 0.3787188727510752 0
   20 -0.6841219033906107 0.7364079763792437 0
   21 -0.9920556836181335 0.20730041712702757 0
   22 2.0187106442510587 0.329144855526437 1
   23 -0.9363848220016563 0.5820604427678648 0
   24 0.3176242827173276 -0.23130675115061858 1
   25 -0.08766070923125874 0.9228708936342618 0
   26 0.9503505174789424 0.22930242945171128 0
   27 1.9137065930212924 0.04727138786088236 1
   28 -0.5740857513898902 0.7860024177122371 0
   29 1.7311039665579235 -0.15822368653296057 1
   30 0.15601420930527587 -0.012971094201072592 1
   31 0.03440178911461349 0.35665294277687765 1
   32 1.734352820043718 -0.16711805469096147 1
   33 0.9743086385469273 -0.03306556437399488 0
   34 1.3646032525150116 -0.33201208895864687 1
   35 -1.000588826462477 0.057717691210846855 0
   36 0.0937572740953208 -0.0973344952572724 1
   37 -0.2791878004436257 0.9850672774312587 0
   38 -0.9994507007532529 0.33048792628967455 0
   39 0.9990673318735382 -0.505133345971595 1
   40 0.8997014125100633 0.5246863891252541 0
;

data mylib.toy_train;
   set toy (obs=30);
run;

data mylib.toy_test;
   set toy(firstobs=31 obs=35);
run;

data mylib.toy_test1;
   set toy(firstobs=36);
run;
```

These statements assume that your libref is named mylib, but you can substitute any appropriately named libref.

The following statements use PROC GPCLASS to perform classification on the training data:

```
proc gpclass
   data=mylib.toy_train
   testdata=mylib.toy_test
   seed=12345
   nThreads=32
   link=Logit;
   input x y;
   target label/level=nominal;
   kernel gaussian(sigma=1);
   inference LA(maxIter=1000 threshold=0.001);
   savestate rstore=mylib.astore;
   output out=mylib.score copyvars=(x y label);
   display nObs descStats modelInfo iterStats;
run;
```

This PROC GPCLASS generates four ODS tables, which are shown in Figure 14.1 through Figure 14.4.

Figure 14.1 Number of Observations

The GPCLASS Procedure

Number of Training Observations Read	30
Number of Training Observations Used	30
Number of Testing Observations Read	5
Number of Testing Observations Used	5

Figure 14.2 Descriptive Statistics

Input Variable Statistics

Variable	Mean	Standard Deviation	Min	Max
x	0.634783	1.079817	-0.992056	2.110087
y	0.200177	0.412637	-0.504343	0.923704

Figure 14.3 Model Information

Model Information

Classification Algorithm	Gaussian Process Classification
Inference Method	Laplacian Approximation
Random Number Seed	12345
Kernel Type	Gaussian Kernel
Kernel Sigma	1
Link Type	Logit Likelihood
LA Iterations	4
LA Threshold	0.001
LA Converge	Yes

Figure 14.4 Iteration Statistics

Iteration History	
Iteration	Log Likelihood
1	-16.197960
2	-15.870310
3	-15.828555
4	-15.827863

In this example, PROC GPCLASS generates the classification score table mylib.score, which contains the predicted class labels and probabilities. The following statements print the table shown in Figure 14.5:

```
proc print noobs data=mylib.score;
run;
```

Figure 14.5 Classification Score Table

x	y	label	P_label0	P_label1	I_label	P_VAR_
0.03440	0.35665	1	0.57263	0.42737	0	0.37342
1.73435	-0.16712	1	0.21402	0.78598	1	0.38873
0.97431	-0.03307	0	0.29224	0.70776	1	0.33329
1.36460	-0.33201	1	0.22919	0.77081	1	0.40313
-1.00059	0.05772	0	0.67936	0.32064	0	0.52435

PROC GPCLASS also generates the analytic store table by using the SAVESTATE statement. This table saves the trained model and uses it for testing, as shown in the following code:

```
proc astore;
   score data=mylib.toy_test1 out=mylib.gpcnewscore
   rstore=mylib.astore;
run;
```

The following statements print the prediction for the testing data in the output table mylib.gpcnewscore, which is shown in Figure 14.6:

```
proc print noobs data=mylib.gpcnewscore;
run;
```

Figure 14.6 Classification Scores for Testing Data

P_label0	P_label1	I_label	P_VAR_
0.41729	0.58271	1	0.37123
0.73734	0.26266	0	0.46789
0.73010	0.26990	0	0.45856
0.24254	0.75746	1	0.42171
0.43640	0.56360	1	0.46223

Syntax: GPCLASS Procedure

The following statements are available in the GPCLASS procedure:

PROC GPCLASS < *options* > **;**
  AUTOTUNE < *options* > **;**
  DISPLAY < *table-list* > < */options* > **;**
  DISPLAYOUT *table-spec-list* < */options* > **;**
  INFERENCE *inference-algorithm* < *options* > **;**
  INPUT *variables* < **LEVEL=INTERVAL** > **;**
  KERNEL *kernel-type* < *options* > **;**
  OUTPUT OUT=*libref.data-table* < *options* > **;**
  SAVESTATE RSTORE=*libref.data-table* < *option* > **;**
  TARGET *variables* < **LEVEL=NOMINAL** > **;**

The PROC GPCLASS statement, an INPUT statement, and a TARGET statement are required. You can specify multiple INPUT statements. The GPCLASS procedure takes only interval input variables.

The following sections describe the PROC GPCLASS statement and then describe the other statements in alphabetical order.

PROC GPCLASS Statement

PROC GPCLASS < *options* > **;**

The PROC GPCLASS statement invokes the procedure. Table 14.1 summarizes the *options* available in this statement.

Table 14.1 PROC GPCLASS Statement Options

Option	Description
Input and Output Data Set Options	
DATA=	Specifies the input training data table
ID=	Specifies the IDs of observations in the data table
TESTDATA=	Specifies the input testing data table
Gaussian Process Option	
LINK=	Specifies the likelihood function of the Gaussian process
Performance Options	
NTHREADS=	Specifies the number of threads for the computation
SEED=	Specifies the seed for the computation

You can specify the following *options*:

DATA=*libref.data-table*

names the input data table for PROC GPCLASS to use. The default is the most recently created data table. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to the data. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 333.

data-table specifies the name of the input data table.

ID=*variable*

specifies the variable that contains IDs of the observations in the input data table. This option specifies a column in the input data table to contain the input observations IDs.

LINK=*keyword*

specifies the link function to be used in the Gaussian process classification. In this method of classification, the link function is a sigmoid type of function that maps from the latent Gaussian process to a real number between 0 and 1 (that is, the predicted class probability). PROC GPCLASS currently supports only the logit function. For more information, see the section "Link Function" on page 348.

NTHREADS=*number-of-threads*

specifies the number of threads to use in the computation. The default value is the number of CPUs available in the machine.

SEED=*number*

specifies an integer to be used to start the pseudorandom number generator. If you do not specify a seed or if you specify a value less than or equal to 0, the seed is generated by reading the time of day from the computer's clock.

TESTDATA=*libref.data-table*

names the input data table for PROC GPCLASS to use for testing.

As a supervised machine learning procedure, PROC GPCLASS takes its input data in two parts: the training data table as specified by the DATA= option and the test testing data table as specified by the TESTDATA= option. The procedure first performs the training task of the Gaussian process classification, and then it performs the testing process in which the predicted class labels and probabilities are generated from the data in the testing data table.

AUTOTUNE Statement

AUTOTUNE < *options* > **;**

The AUTOTUNE statement searches for the best combination of some option values in the PROC GPCLASS, KERNEL, and INFERENCE statement.

Table 14.2 summarizes the *options* that you can specify in the AUTOTUNE statement. For more information about all options except the TUNINGPARAMETERS= option, see the option's description in the section "AUTOTUNE Statement" on page 6 in Chapter 2, "Shared Concepts." The TUNINGPARAMETERS= option is described following Table 14.2.

Table 14.2 AUTOTUNE Statement Options

Option	Description
APPENDLOOKUP	Specifies that the table specified in the HISTORYTABLE= option contain the rows from the table specified in the LOOKUPTABLE= option
CONSTRAINTS=	Specifies the constraints on the output metrics
EVALHISTORY=	Specifies how to report the evaluation history of the tuner
FOLDCOLUMN=	Specifies the column in the data table in which the cross validation fold value is indicated
FRACTION=	Specifies the fraction of observations to use for validation
HISTORYTABLE=	Specifies the table that contains the evaluation history
KFOLD=	Specifies the number of folds for *k*-fold cross validation
LIVEUPDATE	Specifies that the table specified in the HISTORYTABLE= option be updated at every evaluation
LOOKUPTABLE=	Specifies the table to use for evaluation lookup
MAXBAYES=	Specifies the maximum number of points in the kriging model
MAXEVALS=	Specifies the maximum number of evaluations
MAXITER=	Specifies the maximum number of iterations when SEARCHMETHOD=GA or SEARCHMETHOD=BAYESIAN
MAXTIME=	Specifies the maximum time for all iterations
MAXTRAINTIME=	Specifies the maximum time for a model training
NCONVITER=	Specifies the number of convergence iterations
NOGRIDSHUFFLE	Requests that the grid points not be shuffled
NOLOCALSEARCH	Disables local search optimization
NPARALLEL=	Specifies the number of parallel sessions
NSUBSESSIONWORKERS=	Specifies the number of workers in parallel sessions
OBJECTIVE=	Specifies the objective function
POPSIZE=	Specifies the population size when SEARCHMETHOD=GA or SEARCHMETHOD=BAYESIAN
SAMPLESIZE=	Specifies the sample size when SEARCHMETHOD=LHS or SEARCHMETHOD=RANDOM
SEARCHMETHOD=	Specifies the search method that the optimizer uses
SECONDOBJECTIVE=	Specifies the second objective to use for tuning
SELECTINITPOINT	Specifies that the tuner select the best evaluation from the lookup table

Table 14.2 *continued*

Option	Description
TARGETEVENT=	Specifies the target event for ROC-based calculations
TRAINFRACTION=	Specifies the fraction of observations to use for training
TUNINGPARAMETERS=	Specifies the custom tuning parameters
USEPARAMETERS=	Specifies how to handle the TUNINGPARAMETERS= option

TUNINGPARAMETERS=(*suboption* **| . . . | <** *suboption* **>)**

TUNEPARMS=(*suboption* **| . . . | <** *suboption* **>)**

specifies which parameters to tune and which ranges to tune over. If USEPARAMETERS=STANDARD, this option is ignored. You can specify one or more of the following *suboptions*:

CONSTANT (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**

specifies the constant to be used in the linear kernel.

You can specify the following additional *suboptions*:

LB=*number*

specifies the minimum kernel constant to use in the tuning process, where *number* is a nonnegative double. If you specify this *suboption*, you cannot specify the VALUES= *suboption*.

By default, LB=0.

UB=*number*

specifies the maximum kernel constant to use in the tuning process, where *number* is a nonnegative double. If you specify this *suboption*, you cannot specify the VALUES= *suboption*.

By default, UB=100.

VALUES=*value-list*

specifies a list of kernel constants to use in the tuning process, where *value-list* is a space-separated list of nonnegative doubles. If you specify this *suboption*, you cannot specify either the LB= or UB= *suboption*.

INIT=*number*

specifies the initial kernel constant in the tuning process.

By default, INIT=0.

EXCLUDE

excludes the kernel constant from the tuning process. If you specify this *suboption*, any specified LB=, UB=, VALUES=, and INIT= *suboptions* are ignored.

KERNEL (VALUES=*value-list* **INIT=***value* **EXCLUDE)**
specifies the kernel type to be used in the Gaussian process. For more information about the available kernel types, see the section "KERNEL Statement" on page 345. You can specify the following additional *suboptions*:

VALUES=*value-list*
specifies a list of kernel types to use in the tuning process, where *value-list* is a space-separated list of GAUSSIAN and LINEAR.

INIT=GAUSSIAN | LINEAR
specifies the initial kernel type in the tuning process.

By default, INIT=GAUSSIAN.

EXCLUDE
excludes the kernel type from the tuning process. If you specify this *suboption*, the VALUES= and INIT= *suboptions* are ignored.

MAXLAITER (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**
specifies the maximum number of the iterations for the Laplacian approximation. You can specify the following additional *suboptions*:

LB=*number*
specifies the minimum number of the iterations to use in the tuning process, where *number* is a nonnegative integer. If you specify this *suboption*, you cannot also specify the VALUES= *suboption*.

By default, LB=10.

UB=*number*
specifies the maximum number of the iterations to use in the tuning process, where *number* is a nonnegative integer. If you specify this *suboption*, you cannot specify the VALUES= *suboption*.

By default, UB=200.

VALUES=*value-list*
specifies a list of numbers of the iterations to use in the tuning process, where *value-list* is a space-separated list of nonnegative integers. If you specify this *suboption*, you cannot specify either the LB= or UB= *suboption*.

INIT=*number*
specifies the initial number of the iterations in the tuning process.

By default, INIT=100.

EXCLUDE
excludes the number of the iterations from the tuning process. If you specify this *suboption*, any specified LB=, UB=, VALUES=, and INIT= *suboptions* are ignored.

SIGMA (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**
specifies the bandwidth to be used in the Gaussian kernel. You can specify the following additional *suboptions*:

LB=*number*
specifies the minimum bandwidth to use in the tuning process, where *number* is a nonnegative integer. If you specify this *suboption*, you cannot specify the VALUES= *suboption*.

By default, LB=0.01.

UB=*number*
specifies the maximum bandwidth to use in the tuning process, where *number* is a nonnegative double. If you specify this *suboption*, you cannot specify the VALUES= *suboption*.

By default, UB=100.

VALUES=*value-list*
specifies a list of bandwidths to use in the tuning process, where *value-list* is a space-separated list of nonnegative doubles. If you specify this *suboption*, you cannot specify either the LB= or UB= *suboption*.

INIT=*number*
specifies the initial bandwidth in the tuning process.

By default, INIT=1.

EXCLUDE
excludes the bandwidth from the tuning process. If you specify this *suboption*, any specified LB=, UB=, VALUES=, and INIT= *suboptions* are ignored.

THRESHOLD (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**
specifies the minimum change of the loss function between consecutive Newton iterations. You can specify the following additional *suboptions*:

LB=*number*
specifies the minimum threshold to use in the tuning process, where *number* is a nonnegative double. If you specify this *suboption*, you cannot specify the VALUES= *suboption*.

By default, LB=1E–11.

UB=*number*
specifies the maximum threshold to use in the tuning process, where *number* is a nonnegative double. If you specify this *suboption*, you cannot specify the VALUES= *suboption*.

By default, UB=1.

VALUES=*value-list*
specifies a list of thresholds to use in the tuning process, where *value-list* is a space-separated list of nonnegative doubles. If you specify this *suboption*, you cannot specify either the LB= or UB= *suboption*.

INIT=*number*
specifies the initial threshold in the tuning process.

By default, INIT=1E–10.

EXCLUDE
excludes the threshold from the tuning process. If you specify this *suboption*, any specified LB=, UB=, VALUES=, and INIT= *suboptions* are ignored.

DISPLAY Statement

DISPLAY < *table-list* > < / *options* > **;**

The DISPLAY statement enables you to specify a list of display tables to display or exclude. This statement is similar to the ODS SELECT, ODS EXCLUDE, and ODS TRACE statements. However, the DISPLAY statement can improve performance when a large number of tables could be generated (such as in BY-group processing). The DISPLAY statement produces only the requested tables, but ODS statements generate all the tables and then keep the requested tables.

If you use both DISPLAY and ODS statements together, the DISPLAY statement takes precedence over the ODS statements. Note that the ODS EXCLUDE statement processes tables after they have been filtered by the DISPLAY statement. In some cases, it might appear that the ODS EXCLUDE statement is taking precedence because it can further filter the tables. For more information about ODS, see *SAS Output Delivery System: Procedures Guide*.

You can specify the *table-list* as a list of table names, paths, partial pathnames, and regular expressions.

The table names that you can specify are listed in the section "ODS Table Names" on page 351. A path is a table name that is prefixed with dot-separated grouping information. For example, a SelectionSummary table that a procedure produces during a selection routine might have the path *Bygroup1.Summary.SelectionSummary*. A partial pathname does not include all groups; for example, *SelectionSummary* and *Summary.SelectionSummary* are partial pathnames for *Bygroup1.Summary.SelectionSummary*.

When you specify a table name or partial pathname, all display tables whose paths end in the specified name are selected for display or exclusion. For example, both *SelectionSummary* and *Summary.SelectionSummary* select *Bygroup1.Summary.SelectionSummary*.

A regular expression is enclosed in forward slashes (/). For example, specifying "/tions/" selects all pathnames that contain the substring "tions"; in particular, the *Bygroup1.Summary.SelectionSummary* table is selected. Specifying "!/tions/" selects all pathnames that do not contain the substring "tions"; in particular, the *Bygroup1.Summary.SelectionSummary* table is not selected.

You can specify the following *options* after a slash (/):

CASESENSITIVE
performs a case-sensitive comparison of table names in the *table-list* to display table names when tables are subsetted for display. To preserve case, you must enclose table names in the *table-list* in quotation marks.

EXCLUDE
displays all display tables except those that you specify in the *table-list*.

EXCLUDEALL
suppresses display of all tables. This option takes precedence over the other options.

TRACE
displays the display table names, labels, and paths.

DISPLAYOUT Statement

DISPLAYOUT *table-spec-list* < / *options* > **;**

The DISPLAYOUT statement enables you to create output tables from your displayed output. This statement is similar to the ODS OUTPUT statement. For more information about ODS, see *SAS Output Delivery System: Procedures Guide*.

The *table-spec-list* specifies a list of output tables to create. Each entry in the list has either a *key*=*value* format or a *key* format:

key=*value* specifies *key* as the ODS table name, path, or partial pathname, and specifies *value* as the output table name.

key specifies *key* as the ODS table name and also as the output table name.

The ODS table names that you can specify are listed in the section "ODS Table Names" on page 351. You cannot specify the ODS table named OutputCasTables in the *table-spec-list*.

Table names and partial pathnames are discussed under the DISPLAY statement. The DISPLAYOUT statement does not support regular expressions.

You can specify the following *options* after a slash (/):

INCLUDEALL
creates output tables for all display tables. The name of the created output table is the same as the name of the corresponding display table. If you specify this option, the *table-spec-list* specification is ignored.

NOREPLACE
does not replace any existing output table of the same name.

INFERENCE Statement

INFERENCE LA< *option* > **;**

The INFERENCE statement specifies the inference-related parameters to use in the Gaussian process classification.

You must use the following keyword to specify the algorithm for the inference:

LA uses Laplacian approximation.

You can use the following *options* to specify the features of the kernel:

MAXITER=*number*
specifies the maximum number of Newton iterations.

THRESHOLD=*number*
specifies the minimum change of the loss function between consecutive Newton iterations.

For more information, see the section "Inference" on page 348.

INPUT Statement

INPUT *variables* < **LEVEL=INTERVAL** > **;**

The INPUT statement specifies the names of the *variables* to use in the classification. You can specify multiple INPUT statements, and one INPUT statement can include the LEVEL=INTERVAL option for interval input variables. For classification, only interval variables are accepted.

KERNEL Statement

KERNEL GAUSSIAN | LINEAR < *option* > **;**

The KERNEL statement specifies the kernel-related parameters to use in the Gaussian process.

You can use the following keywords to specify the kernel type:

GAUSSIAN uses a Gaussian kernel.

LINEAR uses a linear kernel.

You can use the following *options* to specify the kernel features:

CONSTANT=*number*
specifies the constant in the linear kernel function.

SIGMA=*number*
specifies the bandwidth in the Gaussian kernel function.

For more information, see the section "Kernel" on page 349.

OUTPUT Statement

OUTPUT OUT=*libref.data-table* < *option* > **;**

The OUTPUT statement writes the classification score of each observation to the output data table that you specify in the OUT= option. This information includes the variables that you specify in the COPYVARS= option and four new variables: _I_TARGET_ (the predicted class label of the target variable), _P_TARGETCLASS0_(the predicted class probability for class 0), _P_TARGETCLASS1_(the predicted class probability for class 1), and _P_VAR_ (the variance value of the prediction). For more information, see the section "Table Output" on page 351.

You must specify the following option:

OUT=*libref.data-table*
names the output data table for PROC GPCLASS to use. You must specify this option before any other options. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to where the data table is to be stored. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 333.

data-table specifies the name of the output data table.

The output data table contains the scored data. When you specify this option, all variables that you specify in the COPYVARS= option are added to the specified output data table.

You can also specify the following *option*:

COPYVAR=*variable*
COPYVARS=(*variables***)**
lists one or more variables from the input data table that are transferred to the scored output data table, provided that the output data table produces one or more records per input observation. By default, PROC GPCLASS does not transfer any variables from the input data table to the output data table. The COPYVARS= option accepts numeric and character variables. You can also use the COPYVARS=(_ALL_) option to include all the input variables.

SAVESTATE Statement

SAVESTATE RSTORE=*libref.data-table* **;**

The SAVESTATE statement creates an analytic store for the model and saves it as a binary object in a data table. You can then use the analytic store in the ASTORE procedure to score new data. The analytic store inherits the same parameters that the SAVESTATE statement uses.

You must specify the following option:

RSTORE=*libref.data-table*
specifies a data table in which to save the analytic store for the model. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 333.

TARGET Statement

TARGET *variable* < **LEVEL=NOMINAL** > **;**

The TARGET statement specifies the name of the *variable* to be used as the classification target. You can use one TARGET statement that includes the LEVEL=NOMINAL option for the nominal target variable. Only one nominal variable is accepted as the target.

Details: GPCLASS Procedure

Gaussian Process

Gaussian process classification is a nonparametric Bayesian method that is used for classification tasks. Its nonparametric feature is represented by the Gaussian process that is applied in the model, which can be regarded as a stochastic process that is generalized from the Gaussian distribution; see Rasmussen (1999). There are two parameters for a Gaussian process: the mean function and the kernel. The kernel determines the reproducing kernel Hilbert space (RKHS) where the Gaussian process lives. The Gaussian process searches over this RKHS to find the best function for the classification task, using an adaptive approach based on the input data.

Laplacian Approximation

PROC GPCLASS uses a Laplacian approximation (LA) method of model inference. Laplacian approximation methods are a family of techniques for approximating the intractable integrals that arise in the posterior distributions of Bayesian models that have computable alternative distributions, in the same way as the variational Bayesian (VB) inferences; see Blei and Jordan (2006). Unlike the VB algorithm, the LA algorithm uses a Gaussian distribution to imitate the true model posterior distribution. In particular, it uses Newton iterations to find the mean and covariance of the approximated Gaussian distribution. The Newton iteration is similar to the gradient descent method, which is widely used in deep learning, but different in that the second-order information is incorporated to consider the curvature of the loss function.

Inference

The INFERENCE statement specifies the inference method that PROC GPCLASS uses. Currently, PROC GPCLASS supports only LA inference.

The MAXITER= option specifies the maximum number of LA iterations. The iterations stop in either of two cases: when the number of iterations reaches the specified number, or when the change in the logarithm of the marginal likelihood of the model between two consecutive LA iterations is lower than the threshold that you specify in the THRESHOLD= option.

The THRESHOLD= option specifies the threshold that defines the convergence of the LA iterations. When the iterations stop after the threshold criterion is satisfied, the LA algorithm reaches convergence.

Link Function

The LINK= option specifies the link (likelihood) function that the procedure uses. For Gaussian process classification, the likelihood function is a sigmoid function that maps the latent function to a probability. Currently, PROC GPCLASS supports only the logit function:

$$f(x) = \frac{1}{1 + \exp(-x)}$$

Kernel

The LA inference that PROC GPCLASS applies uses either of two kernel types for the Gaussian process in Gaussian process classification: Gaussian or linear. The kernel type is specified by the KERNEL statement.

The Gaussian kernel type represents a covariance matrix that is generated by a squared exponential function with specified input data, as shown in the following equation, where σ is the Gaussian kernel bandwidth parameter that you specify in the SIGMA= option. You can use the Gaussian kernel to separate two classes that are not linearly separable. When you specify a small SIGMA= option value, the Gaussian kernel can capture a curvy boundary between the two classes.

$$k(x, x') = \exp(\frac{||x - x'||^2}{2\sigma^2})$$

The linear kernel type represents a covariance matrix that is generated using a linear function, as shown in the following equation, where c is the constant in the linear function that is specified by the CONSTANT= option. When the classes in the data are linearly separable, the linear type can yield good results and a faster processing time than the Gaussian type, where the CONSTANT= option captures the bias in the input data.

$$k(x, x') = < x, x' > + c$$

Missing Values

PROC GPCLASS does not perform missing data imputation. If the input data table contains any missing data, the procedure simply ignores the observations (rows) that contain missing values. You can use any data preprocessing steps, such as those in the VARIMPUTE procedure, to impute the missing data before you use the input data table in PROC GPCLASS.

Random Seed

You can specify the random seed by using the SEED= option in the PROC GPCLASS statement. The SEED= option value is used in the random number generator, which initializes the cluster assignment to start the iterations of the variational Bayesian algorithm. If a random seed is not specified or if the seed value is invalid (zero or negative), then PROC GPCLASS generates a random number by using the time of day from the computer's clock.

Displayed Output

PROC GPCLASS displays the following tables for use in cluster analysis when the related parameters are specified.

Number of Observations

The "Number of Observations" table displays the number of observations that are read from the input data table and the number of observations that are used in the classification. The Number of Observations Read column displays the count of raw data that are read from the input data table, and the Number of Observations Used column displays the count of data that are actually used in the classification. The count in the Number of Observations Used column can be smaller if you exclude the observations (rows) in the input data table that contain missing values.

Descriptive Statistics

The "Descriptive Statistics" table displays the mean, standard deviation, minimum, and maximum for each interval variable that is specified in the INPUT statement. The calculation of these statistics is based on all observations (rows) that do not contain missing values in the input data table.

Any input variable whose standard deviation is zero is a constant variable (column) in the input data table. Such constant variables do not contribute to the classification, and they are not used in the Gaussian mixture model.

Model Information

The "Model Information" table provides basic information about the parameters that are used in the Gaussian mixture model. This information includes the classification algorithm, inference method, random number seed, kernel type, kernel sigma/constant, link type, number of Laplacian (LA) iterations, LA threshold, and LA convergence.

Iteration Statistics

The "Iteration Statistics" table provides basic information about the logarithm of the marginal likelihood that is returned by each Newton iteration. This information includes the number of the Newton iteration and the logarithm of the marginal likelihood of each iteration.

Predicted Variable Names

The "Predicted Variable Names" table provides the predicted variable names in the output score table. This information includes the variable names of the predicted class label, probabilities of the two classes, and variance of the prediction.

Table Output

PROC GPCLASS displays results of the classification in the following output data tables.

Classification Score Table

The classification score table is specified in the OUTPUT statement. This table shows the soft-classification results that contain the class labels and the probability distribution over the two classes for each observation in the input data table and the variance in making the predictions. When the probability distributions over the two classes for all observations are given, PROC GPCLASS not only predicts the classification scores but also provides a sense of how it makes the predictions. The classification score table includes the variables that are specified in the COPYVARS= option and the variables shown in the "Predicted Variable Names" table.

Analytic Store Table

The analytic store table is specified in the SAVESTATE statement. This data table contains the saved Gaussian process classification model that PROC GPCLASS generates. This saved Gaussian process classification model keeps the model status information, such as the latent function, after the classification is finished. You can use this model, which is saved in the analytic store table, directly on the data on which you want to perform classification analysis.

ODS Table Names

The results that PROC GPCLASS produces are stored in tables. You can access these tables by their names, which are shown in Table 14.6. You use these table names when you specify the DISPLAY statement.

Table 14.3 ODS Tables Produced by PROC GPCLASS

Table Name	Description	Statement	Option
BestConfiguration	Hyperparameters and objective function values for the best configuration	AUTOTUNE	Default
DescStats	Descriptive statistics for interval input variables	INPUT	Default
EvaluationHistory	Values of the hyperparameters and the objective function for all configurations	AUTOTUNE	Default

Table 14.3 *continued*

Table Name	Description	Statement	Option
HyperparameterImportance	Relative importance of hyperparameters	AUTOTUNE	Default
IterationHistory	Tuner iteration history	AUTOTUNE	Default
IterStats	Iteration statistics	PROC GPCLASS	Default
ModelInfo	Model information	PROC GPCLASS	Default
NObs	Number of observations	PROC GPCLASS	Default
PredVarName	Predicted variables names	PROC GPCLASS	Default
TunerCasOutputTables	Tables created during tuning	AUTOTUNE	Default
TunerInfo	Setup values that are used by the tuner	AUTOTUNE	Default
TunerResults	Values of the hyperparameters, the objective function for the default configuration (iteration 0), and up to 10 best configurations found	AUTOTUNE	Default
TunerSummary	Statistics about the tuning process	AUTOTUNE	Default
TunerTiming	Total time spent on different tasks during tuning	AUTOTUNE	Default

Examples: GPCLASS Procedure

Example 14.1: Performing Classification on Iris Data

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

This example uses the Iris data table to demonstrate how to use the GPCLASS procedure to perform classification. There are three clusters in the Iris data. First, the observations are separated into two classes: rows of the species *Setosa* are labeled as class 1, and rows of the species *Versicolour* and *Virginica* are labeled as class 0. Then the data are split into subsets for training and testing. The following DATA steps generate the Iris data:

```
data iris;
   set sashelp.iris;
run;

data iris_binary_class;
   set iris;
   if Species = 'Setosa' then label=1;
   else label=0;
run;
```

```
data mylib.iris_train;
   set iris_binary_class(obs=140);
run;

data mylib.iris_test;
   set iris_binary_class(firstobs=141 obs=150);
run;
```

These statements assume that your libref is named mylib, but you can substitute any appropriately defined libref.

The following statements run PROC GPCLASS and output the results to data tables and ODS tables. In this example, a linear kernel is used to separate the two classes.

```
proc gpclass
   data=mylib.iris_train
   testdata=mylib.iris_test
   seed=12345
   nThreads=32
   link=Logit;
   input SepalLength SepalWidth PetalLength PetalWidth;
   target label/level=nominal;
   kernel linear(constant=0);
   inference LA(maxIter=100 threshold=0.001);
   output out=mylib.iris_score copyvars=(_all_);
   display nObs descStats modelInfo iterStats;
run;
```

PROC GPCLASS generates four ODS tables, which are shown in Output 14.1.1 through Output 14.1.4.

Output 14.1.1 Number of Observations

The GPCLASS Procedure

Number of Training Observations Read	140
Number of Training Observations Used	140
Number of Testing Observations Read	10
Number of Testing Observations Used	10

Output 14.1.2 Descriptive Statistics

Input Variable Statistics

Variable	Mean	Standard Deviation	Min	Max
SepalLength	57.978571	8.304864	43.000000	79.000000
SepalWidth	30.635714	4.415816	20.000000	44.000000
PetalLength	36.328571	17.589010	10.000000	69.000000
PetalWidth	11.314286	7.410944	1.000000	25.000000

Output 14.1.3 Model Information

Model Information	
Classification Algorithm	Gaussian Process Classification
Inference Method	Laplacian Approximation
Random Number Seed	12345
Kernel Type	Linear Kernel
Kernel Constant	0
Link Type	Logit Likelihood
LA Iterations	12
LA Threshold	0.001
LA Converge	Yes

Output 14.1.4 Iteration Statistics

Iteration History	
Iteration	Log Likelihood
1	-39.057368
2	-23.031626
3	-15.809021
4	-11.866309
5	-9.316223
6	-7.455245
7	-6.042881
8	-5.046018
9	-4.481966
10	-4.290494
11	-4.265787
12	-4.265268

In this example, PROC GPCLASS generates the classification score table mylib.iris_score, which lists the predicted class labels and probabilities. The following statements print the table shown in Output 14.1.5:

```
proc print noobs data=mylib.iris_score;
run;
```

Output 14.1.5 Classification Score Table

Species	SepalLength	SepalWidth	PetalLength	PetalWidth	label	P_label0	P_label1	I_label	P_VAR_
Virginica	67	25	58	18	0	0.92653	0.07347	0	151.660
Virginica	65	30	58	22	0	0.91497	0.08503	0	162.480
Virginica	69	31	54	21	0	0.91079	0.08921	0	116.686
Virginica	72	36	61	25	0	0.89332	0.10668	0	189.129
Virginica	65	32	51	20	0	0.89193	0.10807	0	109.226
Virginica	64	27	53	19	0	0.92327	0.07673	0	115.094
Virginica	68	30	55	21	0	0.91675	0.08325	0	124.591
Virginica	57	25	50	20	0	0.91966	0.08034	0	121.267
Virginica	58	28	51	24	0	0.89070	0.10930	0	167.544
Virginica	63	33	60	25	0	0.88784	0.11216	0	231.991

From the score table, you can see that by using a simple linear kernel, PROC GPCLASS produces a perfect prediction of good confidence, with the predicted probabilities close to 1.

Example 14.2: Tuning the Hyperparameters of a Gaussian Process Classification Model

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

This example uses the Toy data set to demonstrate how to use the AUTOTUNE statement in the GPCLASS procedure to automatically tune the hyperparameters of a Gaussian process classification model.

The Toy data set contains 40 observations and three variables. The variable label contains the two binary classes 1 and 0, and the other two numeric variables contain statistics about the location.

The following DATA steps load the Toy data set into your session and then divide the data into the training data table and the testing data table. PROC GPCLASS then uses the first data table for training and the second data table for testing. These statements assume that your libref is named mylib, but you can substitute any appropriately named libref.

```
data toy;
   input id x y label;
   datalines;
   1 0.7562437251693241 -0.4449246304395207 1
   2 1.6425910060420805 -0.10293958159084055 1
   3 0.5036628219528629 -0.3756019302569773 1
   4 1.9356458876375808 0.1508435933753537 1
   5 -0.48929352371915874 0.9004153510746639 0
   6 1.9612377920931525 0.3688561398322367 1
   7 -0.4457973476327784 0.923703991351446 0
   8 0.897645351230446 0.13625291562699787 0
   9 1.0428050419382913 0.2538945316161072 0
   10 1.7841025147461618 -0.03250971343343456 1
   11 0.6525149362580007 -0.5043431552576471 1
   12 1.8526407982796789 0.13718466658924755 1
   13 0.7736555917316961 -0.48367703386369615 1
   14 -0.9816672335041904 0.33290929557635607 0
   15 0.24612298119546702 -0.1868663956140018 1
   16 1.99621877108224 0.3245177413955755 1
   17 -0.04791750586561572 0.2659542532295433 1
   18 2.110087224576905 0.5250574025280107 1
   19 -0.9602167257824953 0.3787188727510752 0
   20 -0.6841219033906107 0.7364079763792437 0
   21 -0.9920556836181335 0.20730041712702757 0
   22 2.0187106442510587 0.329144855526437 1
   23 -0.9363848220016563 0.5820604427678648 0
   24 0.3176242827173276 -0.23130675115061858 1
   25 -0.08766070923125874 0.9228708936342618 0
   26 0.9503505174789424 0.22930242945171128 0
   27 1.9137065930212924 0.04727138786088236 1
```

```
   28 -0.5740857513898902 0.7860024177122371 0
   29 1.7311039665579235 -0.15822368653296057 1
   30 0.15601420930527587 -0.012971094201072592 1
   31 0.03440178911461349 0.35665294277687765 1
   32 1.734352820043718 -0.16711805469096147 1
   33 0.9743086385469273 -0.03306556437399488 0
   34 1.3646032525150116 -0.33201208895864687 1
   35 -1.000588826462477 0.057717691210846855 0
   36 0.0937572740953208 -0.0973344952572724 1
   37 -0.2791878004436257 0.9850672774312587 0
   38 -0.9994507007532529 0.33048792628967455 0
   39 0.9990673318735382 -0.505133345971595 1
   40 0.8997014125100633 0.5246863891252541 0
;

data mylib.toy_train;
   set toy (obs=30);
run;

data mylib.toy_test;
   set toy(firstobs=31);
run;
```

The following statements use PROC GPCLASS with the AUTOTUNE statement to automatically tune the hyperparameters of a Gaussian process classification model:

```
proc gpclass
   data=mylib.toy_train
   testdata=mylib.toy_test
   seed=12345
   nThreads=32
   link=Logit;
   input x y;
   target label/level=nominal;
   autotune
      /* Tuning Parameters
      You do not need to specify any tuning parameters for the default
      tuning process. If you want to make adjustments to the default
      tuning process, uncomment the following block of code and change
      any of the tuning parameters' attributes.

      tuningparameters=(
         KERNEL    ( values=LINEAR GAUSSIAN init=GAUSSIAN )
         SIGMA     ( lb=0.01  ub=100  init=1.0 )
         CONSTANT  ( lb=0.0   ub=100  init=0.0 )
         MAXLAITER ( lb=10    ub=200  init=100 )
         THRESHOLD ( lb=1E-11 ub=1.0  init=1E-10 )
      )
      */
   ;
   /* Remove this line to see all results */
   ods select BestConfiguration EvaluationHistoryPlot IterationHistoryPlot;
run;
```

The preceding statements produce the table and the plots shown in Output 14.2.1 through Output 14.2.3. The table in Output 14.2.1 displays the evaluation number, the values of the tuning parameters, and the error metric value for the best Gaussian process classification model that the tuner found. Note that the ODS SELECT statement limits the displayed results to a single table and two plots. You can remove this statement to display all tables and plots. For the full list of ODS tables that PROC GPCLASS produces, see Table 14.3.

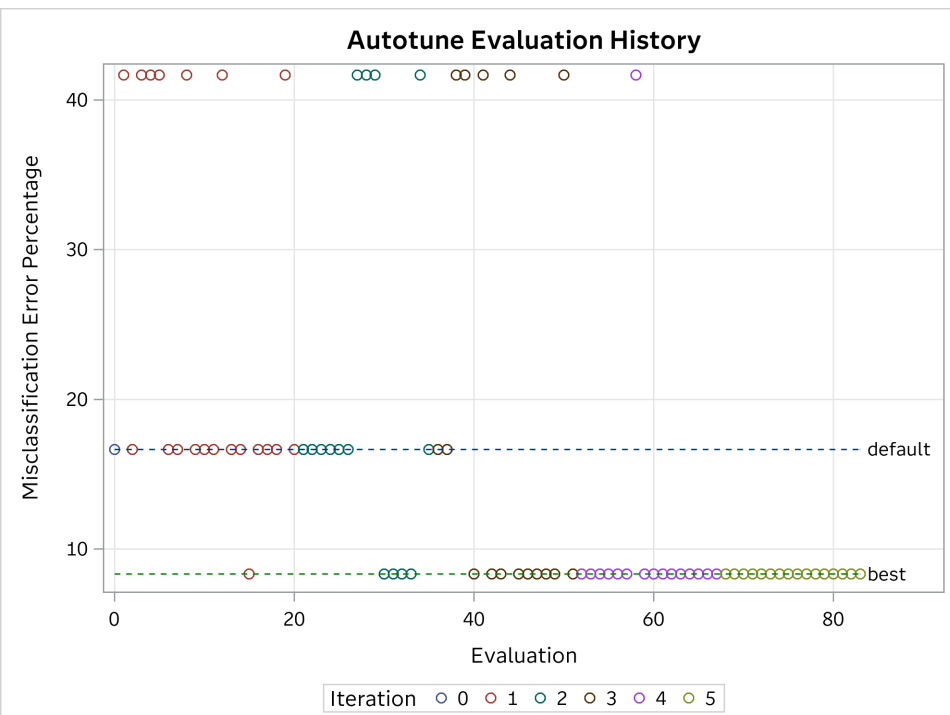
Output 14.2.1 Best Configuration Table

The GPCLASS Procedure

Best Configuration	
Evaluation	15
Kernel	LINEAR
Linear Kernel Constant	100
Maximum Laplacian Approximation Iterations	105
Laplacian Approximation Threshold	0.5
Misclassification Error Percentage	8.33

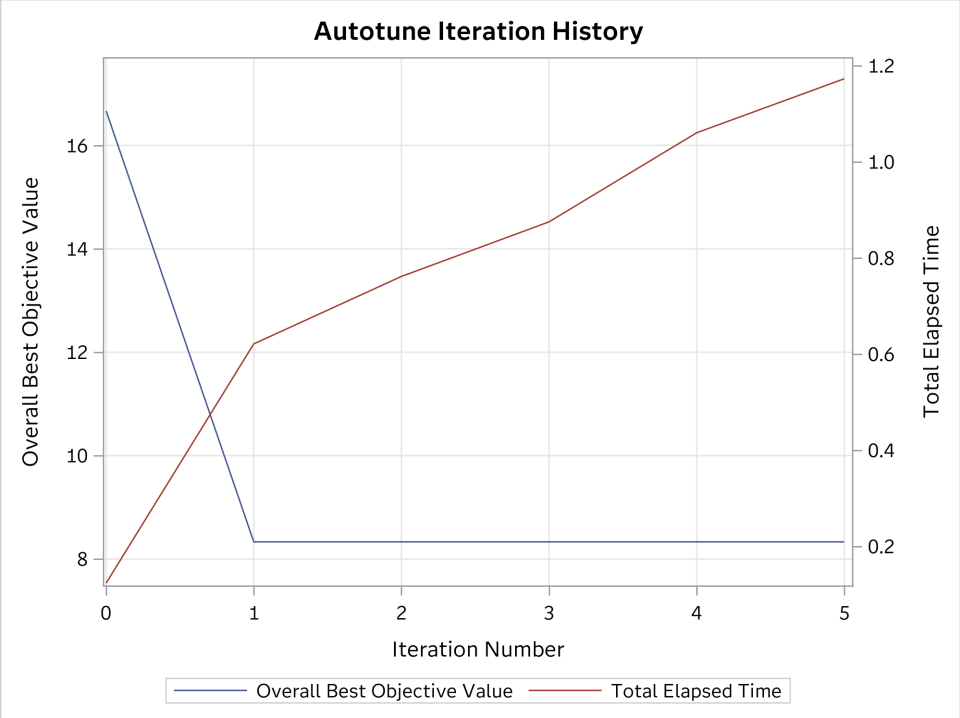
Output 14.2.2 displays a scatter plot of all configurations that the tuner tried. The objective values are shown on the Y axis, and the evaluation numbers are shown on the X axis.

Output 14.2.2 Evaluation History Plot



The plot in Output 14.2.3 displays how the best found objective value and the elapsed time changed with each iteration of the tuner.

Output 14.2.3 Iteration History Plot



References

Blei, D. M., and Jordan, M. I. (2006). "Variational Inference for Dirichlet Process Mixtures." *Bayesian Analysis* 1:121–144.

Rasmussen, C. E. (1999). "The Infinite Gaussian Mixture Model." In *NIPS '99: Proceedings of the 12th International Conference on Neural Information Processing Systems*, 554–560. Cambridge, MA: MIT Press.

Chapter 15
GPREG Procedure

Contents

Overview: GPREG Procedure . . . **360**
- PROC GPREG Features . . . 360
- Using SAS Viya Workbench . . . 361

Getting Started: GPREG Procedure . . . **361**

Syntax: GPREG Procedure . . . **364**
- PROC GPREG Statement . . . 365
- AUTOTUNE Statement . . . 366
- DISPLAY Statement . . . 373
- DISPLAYOUT Statement . . . 374
- INPUT Statement . . . 375
- KERNEL Statement . . . 375
- OPTIMIZATION Statement . . . 376
- OUTPUT Statement . . . 379
- PARTITION Statement . . . 380
- SAVESTATE Statement . . . 380
- TARGET Statement . . . 381

Details: GPREG Procedure . . . **381**
- Gaussian Process Regression . . . 381
- Kernel Types . . . 382
- Optimization . . . 383
- Missing Values . . . 383
- Random Seed . . . 384
- Displayed Output . . . 384
  - Model Information . . . 384
  - Number of Observations . . . 384
  - Parameter Estimates . . . 384
  - Variable Statistics . . . 384
- Table Output . . . 385
  - Output . . . 385
  - Inducing Points Table . . . 385
  - Variational Covariance Table . . . 385
  - Analytic Store Table . . . 385
- ODS Table Names . . . 385

Examples: GPREG Procedure . . . **386**
- Example 15.1: Performing Regression on Data about Students . . . 387
- Example 15.2: Tuning Hyperparameters of a Gaussian Process Regression Model . . . 389

References . . . **390**

Overview: GPREG Procedure

The GPREG procedure performs Gaussian process regression (GPR) for very large data sets in SAS Viya Workbench.

GPR assumes that the predictions are based on a latent function that is a posterior distribution of a Gaussian process prior and a Gaussian noise likelihood. This latent function is adaptively learned on the basis of the observed data input in the reproducing kernel Hilbert space (RKHS), which is defined by the kernel of the Gaussian process. Thus it provides good regression performance.

This procedure uses stochastic variational inference for Gaussian process models, which can process large data sets fast (Hensman, Fusi, and Lawrence 2013), and where the Gaussian processes can be variationally decomposed to depend on a set of relevant inducing variables that factorize the model in the manner necessary to perform variational inference fast. This enables you to apply Gaussian process models to data sets that contain millions of data points.

PROC GPREG Features

GPR models are nonparametric kernel-based probabilistic models. The exact inference method for Gaussian process models involves decomposing the kernel matrices with the time complexity $O(n^3)$, where n is the number of observations, and is thus not suitable for very large data sets. PROC GPREG implements the stochastic variational inference (SVI), which reduces the overall time complexity to $O(nm^2)$, where m is the number of inducing points; this approach is scalable to big data.

PROC GPREG applies the SVI to a Gaussian process regression model by using the inducing points u as a set of variables. The goal of introducing inducing points is to reduce the effective number of input data points to the Gaussian process from n to m, where $m << n$. The m points are called the inducing points. The inducing points of the approximate GP can be viewed as a set of m latent variables that are denoted as u. They are used as the catalysts for the computation of the inference of the model. For more information, see the section "Gaussian Process Regression" on page 381.

PROC GPREG has the following features:

- performs Gaussian process regression, which provides not only the predicted target score but also the probability distribution over the target for each observation, as a benefit of the probabilistic modeling
- uses a stochastic variational inference (SVI) method. This method maximizes the lower bound of the posterior distribution, and iteratively updates the model parameters until the model reaches convergence.

Using SAS Viya Workbench

SAS Viya Workbench has greatly simplified the SAS programming experience by offering a syntax style nearly identical to that of SAS 9 and earlier releases. However, there are minor exceptions. In the past, SAS Viya procedures required the use of a CAS engine libref. This engine served multiple purposes, but it was primarily used to upload data to the CAS server and manage the data. Its additional purposes included support of new data types and improved performance for large data sets. No CAS server is needed for SAS Viya Workbench, because all computations are done in place for you in a single location.

In the following program, the LIBNAME statement creates a mylib libref that the examples in this chapter use to refer to your current or desired work directory:

```
libname mylib ".";
```

For more information, see *SAS Viya Workbench: Getting Started*.

Getting Started: GPREG Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

This example uses the Sinewave data set to demonstrate how you can use the GPREG procedure to perform Gaussian process regression analysis and save the trained model in an analytic store. The Sinewave data set contains 1,000 noisy samples from a two-dimensional sine wave function; it has two variables, x and y.

The following DATA steps generate the Sinewave data set and upload it to your session:

```
data Sinewave;
   keep x y;
   call streaminit(123);       /* set random number seed */
   do i = 1 to 1000;
      u = rand("Uniform");
      x = u;
      y = sin(6*x)+rand("Normal", 0, 0.01);
      output;
   end;
run;

data mylib.sinewave;
   set Sinewave;
run;
```

The second DATA step assumes that your libref is named mylib, but you can substitute any appropriately named libref.

The following statements use PROC GPREG to perform Gaussian process regression on the data and save the data in an analytic store:

```
proc gpreg
   data=mylib.sinewave
   seed=12345
   nThreads=32
   nInducingPoints=10
   fixInducingPoints
   outInducingPoints=mylib.gpreg_ip
   outVariationalCov=mylib.gpreg_cov;
   target y/level=interval;
   input x;
   partition fraction(valid=0 test=0.2 seed=1235);
   kernel rbf(
      autoRelevanceDetermination
      fixKernelParmFirstIter
      jitterMaxIters=10
      );
   optimization adam(
      regl1=0.5
      maxiters=500
      learningRate=0.05
      momentum=0.8
      adaptiveRate
      adaptiveDecay=0.9
      printfreq=200
   );
    savestate rstore=mylib.astore;
    output out=mylib.score copyvars=(x y);
    display nObs varStats modelInfo ParameterEstimates;
run;
```

These statements assume that your libref is named mylib, but you can substitute any appropriately defined libref.

PROC GPREG generates four ODS tables, which are shown in Figure 15.1 through Figure 15.4.

Figure 15.1 Number of Observations

The GPREG Procedure

Number of Observations	
Number of Observations Read	1000
Number of Observations Used	1000
Number of Observations Used for Training	815
Number of Observations Used for Testing	185

Figure 15.2 Variable Statistics

The GPREG Procedure

Input Variable Statistics		
Variable	Mean	Standard Deviation
x	0.511767	0.284632
y	-0.020674	0.724311

Figure 15.3 Model Information

Model Information	
Seed for Random Initialization for Clustering and Optimization Parameters	12345
Number of inducing points	10
Kernel Type	RBF
Automatic Relevance Determination	TRUE
Fix Kernel Parameters in the First Iteration	FALSE
Fix Inducing Points in Optimization	TRUE
Use Simple Initialization	TRUE
Maximum Number of Iterations for Jitter Cholesky Decomposition	10

Figure 15.4 Number of Observations

Parameter Estimates		
Parameter Type	Parameter Name	Parameter Value
Kernel	Variance	323.367996
Kernel	Inverse square of length scale for variable x	735.930478
Noise	Variance	328.349708

In this example, PROC GPREG generates the score output table mylib.score. The following statements print the first 15 rows of the table, which is shown in Figure 15.5. The value 1 in the _ROLE_ column means that the observation has a training role, and the value 3 in the _ROLE_ column means that the observation has a testing role.

```
proc print noobs data=mylib.score (obs=15);
run;
```

Figure 15.5 Score Output Table

x	y	P_y	P_yMEAN	P_ySTD	_WARN_	_ROLE_
0.58170	-0.35197	-0.019638	-0.019638	13.6534		1
0.36159	0.81021	-0.020507	-0.020507	13.0249		1
0.33759	0.88297	0.002292	0.002292	13.8129		1
0.97680	-0.41130	-0.020674	-0.020674	13.0249		1
0.75099	-0.96933	-0.021611	-0.021611	13.0845		1
0.81052	-0.99303	-0.020674	-0.020674	13.0249		1
0.01800	0.10657	-0.018723	-0.018723	14.1779		1
0.10503	0.59752	-0.015572	-0.015572	13.1166		1
0.07730	0.43982	-0.034881	-0.034881	15.1158		1
0.73482	-0.94985	-0.026007	-0.026007	14.8356		1
0.92549	-0.65055	-0.021292	-0.021292	13.0015		1
0.92680	-0.65987	-0.021171	-0.021171	13.0097		3
0.45990	0.38082	-0.015619	-0.015619	13.0897		1
0.99078	-0.33322	-0.020674	-0.020674	13.0249		1
0.62949	-0.60994	-0.020674	-0.020674	13.0249		1

Syntax: GPREG Procedure

The following statements are available in the GPREG procedure:

PROC GPREG < *options* > **;**
  AUTOTUNE < *options* > **;**
  DISPLAY < *table-list* > < */options* > **;**
  DISPLAYOUT *table-spec-list* < */options* > **;**
  INPUT *variables* < **LEVEL=INTERVAL** > **;**
  KERNEL *kernel-type* < *options* > **;**
  OPTIMIZATION < *optimization-algorithm* > < *options* > **;**
  OUTPUT OUT=*libref.data-table* < *options* > **;**
  PARTITION < *partition-options* > **;**
  SAVESTATE RSTORE=*libref.data-table* **;**
  TARGET *variables* < **LEVEL=INTERVAL** > **;**

The PROC GPREG statement, an INPUT statement, and a TARGET statement are required. You can specify multiple INPUT statements.

The following sections describe the PROC GPREG statement and then describe the other statements in alphabetical order.

PROC GPREG Statement

PROC GPREG < *options* > **;**

The PROC GPREG statement invokes the procedure. Table 15.1 summarizes the *options* available in this statement.

Table 15.1 PROC GPREG Statement Options

Option	Description
Input and Output Data Set Option	
DATA=	Specifies the input data table
Sparse Gaussian Process Options	
FIXINDUCINGPOINTS	Specifies that the inducing points are to be fixed during optimization
NINDUCINGPOINTS=	Specifies the number of inducing points
Regression Output Options	
OUTINDUCINGPOINTS=	Outputs the inducing points in the regression
OUTVARIATIONALCOV=	Outputs the covariance of the Gaussian process
Performance Options	
NTHREADS=	Specifies the seed to start the random number generator
SEED=	Specifies seed for the computation

You can specify the following *options*:

DATA=*libref.data-table*

names the input data table for PROC GPREG to use. The default is the most recently created data table. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to the data. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 361.

data-table specifies the name of the input data table.

FIXINDUCINGPOINTS

specifies that the inducing points in the sparse Gaussian process are to be fixed or adjusted during the optimization. If you omit this option, PROC GPREG adjusts the locations of the inducing points during the iterations of the model inference.

NINDUCINGPOINTS=*number*

specifies the number of inducing points to use in the sparse Gaussian process. The *number* must be a positive integer. When you specify a larger *number*, the PROC GPREG provides a more precise prediction results for the regression, but at a slower processing speed. When you specify a smaller *number*, the result are less accurate but the processing is faster.

By default, NINDUCINGPOINTS=100.

NTHREADS=*number-of-threads*
specifies the number of threads to use in the computation. The default value is the number of CPUs available in the machine.

OUTINDUCINGPOINTS=*libref.data-table*
creates the data table that contains the inducing points to use in the sparse Gaussian process. If you also specify the FIXINDUCINGPOINTS option, this table contains the cluster centroids of the *k*-means used as the initial inducing points. Otherwise, the table contains the locations of the inducing points of the last iteration of the model inference. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 361.

OUTVARIATIONALCOV=*libref.data-table*
creates the data table that contains the covariance to use in the sparse Gaussian process. The table contains the covariance of the Gaussian process used in the last iteration of the model inference. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 361.

SEED=*number*
specifies an integer to be used to start the pseudorandom number generator. If you do not specify a seed or if you specify a value less than or equal to 0, the seed is generated by reading the time of day from the computer's clock.

AUTOTUNE Statement

AUTOTUNE < *options* > **;**

The AUTOTUNE statement searches for the best combination of some option values in the PROC GPREG, KERNEL, and OPTIMIZATION statement.

Table 15.2 summarizes the *options* that you can specify in the AUTOTUNE statement. For more information about all options except the TUNINGPARAMETERS= option, see the option's description in the section "AUTOTUNE Statement" on page 6 in Chapter 2, "Shared Concepts." The TUNINGPARAMETERS= option is described following Table 15.2.

Table 15.2 AUTOTUNE Statement Options

Option	Description
APPENDLOOKUP	Specifies that the table specified in the HISTORYTABLE= option contain the rows from the table specified in the LOOKUPTABLE= option
CONSTRAINTS=	Specifies the constraints on the output metrics
EVALHISTORY=	Specifies how to report the evaluation history of the tuner
FOLDCOLUMN=	Specifies the column in the data table in which the cross validation fold value is indicated
FRACTION=	Specifies the fraction of observations to use for validation
HISTORYTABLE=	Specifies the table that contains the evaluation history

Table 15.2 *continued*

Option	Description
KFOLD=	Specifies the number of folds for *k*-fold cross validation
LIVEUPDATE	Specifies that the table specified in the HISTORYTABLE= option be updated at every evaluation
LOOKUPTABLE=	Specifies the table to use for evaluation lookup
MAXBAYES=	Specifies the maximum number of points in the kriging model
MAXEVALS=	Specifies the maximum number of evaluations
MAXITER=	Specifies the maximum number of iterations when SEARCHMETHOD=GA or SEARCHMETHOD=BAYESIAN
MAXTIME=	Specifies the maximum time for all iterations
MAXTRAINTIME=	Specifies the maximum time for a model training
NCONVITER=	Specifies the number of convergence iterations
NOGRIDSHUFFLE	Requests that the grid points not be shuffled
NOLOCALSEARCH	Disables local search optimization
NPARALLEL=	Specifies the number of parallel sessions
NSUBSESSIONWORKERS=	Specifies the number of workers in parallel sessions
OBJECTIVE=	Specifies the objective function
POPSIZE=	Specifies the population size when SEARCHMETHOD=GA or SEARCHMETHOD=BAYESIAN
SAMPLESIZE=	Specifies the sample size when SEARCHMETHOD=LHS or SEARCHMETHOD=RANDOM
SEARCHMETHOD=	Specifies the search method that the optimizer uses
SECONDOBJECTIVE=	Specifies the second objective to use for tuning
SELECTINITPOINT	Specifies that the tuner select the best evaluation from the lookup table
TARGETEVENT=	Specifies the target event for ROC-based calculations
TRAINFRACTION=	Specifies the fraction of observations to use for training
TUNINGPARAMETERS=	Specifies the custom tuning parameters
USEPARAMETERS=	Specifies how to handle the TUNINGPARAMETERS= option

TUNINGPARAMETERS=(*suboption* **| . . . | <** *suboption* **>)**

TUNEPARMS=(*suboption* **| . . . | <** *suboption* **>)**

specifies which parameters to tune and which ranges to tune over. If USEPARAMETERS=STANDARD, this option is ignored.

You can specify one or more of the following *suboptions*:

ALGORITHM (VALUES=*value-list* **INIT=***value* **EXCLUDE)**

specifies the optimization algorithm. You can specify the following additional *suboptions*:

VALUES=*value-list*

specifies a list of algorithm types to use in the tuning process, where *value-list* is a space-separated list of ADAM and SGD.

INIT=ADAM|SGD
specifies the initial algorithm type in the tuning process.

By default, INIT=SGD.

EXCLUDE
excludes the algorithm type from the tuning process. If you specify this suboption, the VALUES= and INIT= suboptions are ignored.

ARD (VALUES=*value-list* **INIT=***value* **EXCLUDE)**
uses automatic relevance determination in the kernel function. You can specify the following additional *suboptions*:

VALUES=*value-list*
specifies a list of values to use in the tuning process, where *value-list* is a space-separated list of FALSE and TRUE.

INIT=FALSE|TRUE
specifies the initial automatic relevance determination value in the tuning process.

By default, INIT=FALSE.

EXCLUDE
excludes the automatic relevance determination from the tuning process. If you specify this suboption, the VALUES= and INIT= suboptions are ignored.

JITTERMAXITERS (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**
specifies the maximum number of iterations for jitter Cholesky decomposition.

You can specify the following additional *suboptions*:

LB=*number*
specifies the minimum jitter iterations to use in the tuning process, where *number* is a nonnegative integer. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=5.

UB=*number*
specifies the maximum jitter iterations to use in the tuning process, where *number* is a nonnegative integer. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=15.

VALUES=*value-list*
specifies a list of jitter iterations to use in the tuning process, where *value-list* is a space-separated list of nonnegative integers. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial jitter iterations in the tuning process.

By default, INIT=10.

EXCLUDE
excludes the jitter iterations from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

KERNEL (VALUES=*value-list* **INIT=***value* **EXCLUDE)**
specifies the kernel type to be used in the Gaussian process. For more information about the available kernel types, see the section "KERNEL Statement" on page 375. You can specify the following additional *suboptions*:

VALUES=*value-list*
specifies a list of kernel types to use in the tuning process, where *value-list* is a space-separated list of LINEAR, MATERN32, MATERN52, PERIODIC, and RBF.

INIT=LINEAR|MATERN32|MATERN52|PERIODIC|RBF
specifies the initial kernel type in the tuning process.

By default, INIT=RBF.

EXCLUDE
excludes the kernel from the tuning process. If you specify this suboption, the VALUES= and INIT= suboptions are ignored.

LEARNINGRATE (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**
specifies the learning rate parameter that the optimizaton algorithm uses.

You can specify the following additional *suboptions*:

LB=*number*
specifies the minimum learning rate to use in the tuning process, where *number* is a nonnegative double. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=1E-7.

UB=*number*
specifies the maximum learning rate to use in the tuning process, where *number* is a nonnegative double. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=2.

VALUES=*value-list*
specifies a list of learning rate to use in the tuning process, where *value-list* is a space-separated list of nonnegative doubles. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial learning rate in the tuning process.

By default, INIT=1E-3.

EXCLUDE
excludes the learning rate from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

MINIBATCHSIZE (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**
specifies the value for size of the minibatch in the gradient descent.

You can specify the following additional *suboptions*:

LB=*number*
specifies the minimum minibatch size to use in the tuning process, where *number* is a nonnegative integer. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=1.

UB=*number*
specifies the maximum minibatch size to use in the tuning process, where *number* is a nonnegative integer. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=64.

VALUES=*value-list*
specifies a list of minibatch sizes to use in the tuning process, where *value-list* is a space-separated list of nonnegative integers. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial minibatch size in the tuning process.

By default, INIT=1.

EXCLUDE
excludes the minibatch size from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

MOMENTUM (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**
specifies the value for momentum in the optimization.

You can specify the following additional *suboptions*:

LB=*number*
specifies the minimum momentum to use in the tuning process, where *number* is a nonnegative double. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=0.

UB=*number*
specifies the maximum momentum to use in the tuning process, where *number* is a nonnegative double. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=0.99.

VALUES=*value-list*
specifies a list of momentum to use in the tuning process, where *value-list* is a space-separated list of nonnegative doubles. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the momentum rate in the tuning process.

By default, INIT=0.1.

EXCLUDE
excludes the momentum from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

NINDUCINGPOINTS (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**
specifies the number of inducing points in the Gaussian process.

You can specify the following additional *suboptions*:

LB=*number*
specifies the minimum number of inducing points to use in the tuning process, where *number* is a nonnegative integer. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=25.

UB=*number*
specifies the maximum number of inducing points to use in the tuning process, where *number* is a nonnegative integer. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=150.

VALUES=*value-list*
specifies a list of number of inducing points to use in the tuning process, where *value-list* is a space-separated list of nonnegative integers. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial number of inducing points in the tuning process.

By default, INIT=100.

EXCLUDE
excludes the number of inducing points from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

TUNEADAPTIVEDECAY (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**

specifies the rate at which the second moment of the gradient is decayed. You can specify the following additional *suboptions*:

LB=*number*
specifies the minimum adaptive decay to use in the tuning process, where *number* is a nonnegative double. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=0.

UB=*number*
specifies the maximum adaptive decay to use in the tuning process, where *number* is a nonnegative double. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=0.99.

VALUES=*value-list*
specifies a list of adaptive decays to use in the tuning process, where *value-list* is a space-separated list of nonnegative doubles. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial adaptive decay in the tuning process.

By default, INIT=0.95.

EXCLUDE
excludes the adaptive decay from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

TUNEADAPTIVERATE (VALUES=*value-list* **INIT=***value* **EXCLUDE)**
uses an adaptive learning rate in the gradient descent. You can specify the following additional *suboptions*:

VALUES=*value-list*
specifies a list of values to use in the tuning process, where *value-list* is a space-separated list of the values FALSE and TRUE.

INIT=FALSE|TRUE
specifies the initial adaptive rate value in the tuning process.

By default, INIT=FALSE.

EXCLUDE
excludes the adaptive rate from the tuning process. If you specify this suboption, the VALUES= and INIT= suboptions are ignored.

DISPLAY Statement

DISPLAY < *table-list* > < / *options* > **;**

The DISPLAY statement enables you to specify a list of display tables to display or exclude. This statement is similar to the ODS SELECT, ODS EXCLUDE, and ODS TRACE statements. However, the DISPLAY statement can improve performance when a large number of tables could be generated (such as in BY-group processing). The DISPLAY statement produces only the requested tables, but ODS statements generate all the tables and then keep the requested tables.

If you use both DISPLAY and ODS statements together, the DISPLAY statement takes precedence over the ODS statements. Note that the ODS EXCLUDE statement processes tables after they have been filtered by the DISPLAY statement. In some cases, it might appear that the ODS EXCLUDE statement is taking precedence because it can further filter the tables. For more information about ODS, see *SAS Output Delivery System: Procedures Guide*.

You can specify the *table-list* as a list of table names, paths, partial pathnames, and regular expressions.

The table names that you can specify are listed in the section "ODS Table Names" on page 385. A path is a table name that is prefixed with dot-separated grouping information. For example, a SelectionSummary table that a procedure produces during a selection routine might have the path *Bygroup1.Summary.SelectionSummary*. A partial pathname does not include all groups; for example, *SelectionSummary* and *Summary.SelectionSummary* are partial pathnames for *Bygroup1.Summary.SelectionSummary*.

When you specify a table name or partial pathname, all display tables whose paths end in the specified name are selected for display or exclusion. For example, both *SelectionSummary* and *Summary.SelectionSummary* select *Bygroup1.Summary.SelectionSummary*.

A regular expression is enclosed in forward slashes (/). For example, specifying "/tions/" selects all pathnames that contain the substring "tions"; in particular, the *Bygroup1.Summary.SelectionSummary* table is selected. Specifying "!/tions/" selects all pathnames that do not contain the substring "tions"; in particular, the *Bygroup1.Summary.SelectionSummary* table is not selected.

You can specify the following *options* after a slash (/):

CASESENSITIVE
performs a case-sensitive comparison of table names in the *table-list* to display table names when tables are subsetted for display. To preserve case, you must enclose table names in the *table-list* in quotation marks.

EXCLUDE
displays all display tables except those that you specify in the *table-list*.

EXCLUDEALL
suppresses display of all tables. This option takes precedence over the other options.

TRACE
displays the display table names, labels, and paths.

DISPLAYOUT Statement

DISPLAYOUT *table-spec-list* < / *options* > **;**

The DISPLAYOUT statement enables you to create output tables from your displayed output. This statement is similar to the ODS OUTPUT statement. For more information about ODS, see *SAS Output Delivery System: Procedures Guide*.

The *table-spec-list* specifies a list of output tables to create. Each entry in the list has either a *key*=*value* format or a *key* format:

key=*value* specifies *key* as the ODS table name, path, or partial pathname, and specifies *value* as the output table name.

key specifies *key* as the ODS table name and also as the output table name.

The ODS table names that you can specify are listed in the section "ODS Table Names" on page 385. You cannot specify the ODS table named OutputCasTables in the *table-spec-list*.

Table names and partial pathnames are discussed under the DISPLAY statement. The DISPLAYOUT statement does not support regular expressions.

You can specify the following *options* after a slash (/):

INCLUDEALL
creates output tables for all display tables. The name of the created output table is the same as the name of the corresponding display table. If you specify this option, the *table-spec-list* specification is ignored.

NOREPLACE
does not replace any existing output table of the same name.

INPUT Statement

INPUT *variables* < **LEVEL= INTERVAL** > **;**

The INPUT statement specifies the names of the *variables* to be used in clustering. You can specify multiple INPUT statements, and you can use the LEVEL=INTERVAL option for interval input variables. For regression, only interval variables are accepted.

KERNEL Statement

KERNEL *kernel-type* < *option* > **;**

The KERNEL statement specifies the kernel-related parameters to be used in the Gaussian process.

You can specify the following *kernel-types*:

LINEAR	specifies a linear kernel.
MATERN32	specifies a Matern 3/2 kernel.
MATERN52	specifies a Matern 5/2 kernel.
PERIODIC	specifies a periodic kernel.
RBF	specifies a radial-basis function kernel.

You can use the following *options* to specify the features of the kernel:

AUTORELEVANCEDETERMINATION
uses automatic relevance determination in the kernel function.

FIXKERNELPARMFIRSTITER
sets kernel parameters as fixed values in the first iteration.

JITTERMAXITERS=*number*
specifies the maximum number of iterations for jitter Cholesky decomposition.

For more information, see the section "Kernel Types" on page 382.

OPTIMIZATION Statement

OPTIMIZATION < *optimization-algorithm* >< *options* > **;**

The optimization statement specifies options for the optimization method that you use to train your model. When you are training your model, the objective function to be minimized is

$$f(w) = \frac{1}{n}\sum_{i=0}^{n} L(w; x_i, y_i) + R(w)$$

where $L(w; x_i, y_i)$ is the loss associated with observation i having data x_i and correct classification y_i, and $R(w)$ is a regularization term defined by

$$R(w) = \lambda_1 \|w\|_1 + \frac{\lambda_2}{2}\|w\|_2^2$$

You can specify the following *options*:

ALGORITHM=ADAM < *sgd-options* > | **SGD** < *sgd-options* >

ALG=ADAM < *sgd-options* > | **SGD** < *sgd-options* >

specifies the *optimization-algorithm* to use during training. You can specify one of the four following algorithms:

ADAM < *sgd-options* > specifies the adaptive moments (Adam) algorithm, which is one of the variations of the stochastic gradient descent algorithm. It keeps track of the decaying averages of the past gradients and past squared gradients.

You can specify these additional *sgd-options*:

ANNEALINGRATE=*number*

specifies the annealing parameter, β. Annealing is a way to automatically reduce the learning rate as the algorithm progresses, producing smaller steps as the algorithm approaches a solution. Effectively, it replaces the learning rate parameter, η, with

$$\eta' = \frac{\eta}{1 + \beta t}$$

where t is the number of iterations that the algorithm has performed.

By default, ANNEALINGRATE=1.0E–6. The *number* must be a nonnegative double.

COMMFREQ=*number*

specifies the number of minibatches that each computational thread processes before weights are synchronized across all threads.

LEARNINGRATE=*number*

specifies the learning rate parameter, η, for the algorithm. New iterates for the algorithm are found by using

$$w_{k+1} = w_k - \frac{\eta}{\|I_k\|}\sum_{(x_i, y_i)\in I_k} \nabla L(w_k; x_i, y_i)$$

where w_k is the current weight vector, w_{k+1} is the new weight vector, I_k is the minibatch used during iteration k, and $L(w_k; x_i, y_i)$ is the loss associated with the ith observation.

If you see a huge objective value from the algorithm, especially for a small data set, it is likely that the learning rate is set too high.

By default, LEARNINGRATE=0.001. The *number* must be a nonnegative double.

MINIBATCHSIZE=*number*
specifies the size of the minibatches to use in the algorithm.

By default, MINIBATCHSIZE=10.

MOMENTUM=*number*
specifies the value for momentum. The *number* must be greater than or equal to 0 and less than or equal to 1.

By default, MOMENTUM=0.

SEED=*number*
specifies the seed for random access of observations on each thread for the algorithm. If *number* is less than or equal to 0, a random seed is generated by reading the time of day from the computer's clock.

USELOCKING
specifies that computational threads share a common weight vector and update weight vector without race conditions. If you omit this option, computational threads update a single weight vector simultaneously. This causes intentional race conditions and nondeterministic behavior but increases performance significantly.

SGD < *sgd-options* > specifies the plain stochastic gradient descent (SGD) algorithm.

You can specify the following *sgd-options*:

ANNEALINGRATE=*number*
specifies the annealing parameter, β. Annealing is a way to automatically reduce the learning rate as the algorithm progresses, producing smaller steps as the algorithm approaches a solution. Effectively, it replaces the learning rate parameter, η, with

$$\eta' = \frac{\eta}{1 + \beta t}$$

where t is the number of iterations that the algorithm has performed.

By default, ANNEALINGRATE=1.0E–6. The *number* must be a nonnegative double.

COMMFREQ=*number*
specifies the number of minibatches that each computational thread processes before weights are synchronized across all threads.

LEARNINGRATE=*number*

specifies the learning rate parameter, η, for the algorithm. New iterates for the algorithm are found by using

$$w_{k+1} = w_k - \frac{\eta}{\|I_k\|} \sum_{(x_i, y_i) \in I_k} \nabla L(w_k; x_i, y_i)$$

where w_k is the current weight vector, w_{k+1} is the new weight vector, I_k is the minibatch used during iteration k, and $L(w_k; x_i, y_i)$ is the loss associated with the ith observation.

If you see a huge objective value from the algorithm, especially for a small data set, it is likely that the learning rate is set too high.

By default, LEARNINGRATE=0.001. The *number* must be a nonnegative double.

MINIBATCHSIZE=*number*

specifies the size of the minibatches to use in the algorithm.

By default, MINIBATCHSIZE=10.

MOMENTUM=*number*

specifies the value for momentum. The *number* must be greater than or equal to 0 and less than or equal to 1.

By default, MOMENTUM=0.

SEED=*number*

specifies the seed for random access of observations on each thread for the algorithm. If *number* is less than or equal to 0 or not specified, a random seed is generated by reading the time of day from the computer's clock.

USELOCKING

specifies that computational threads share a common weight vector and update the weight vector without race conditions. If you omit this option, computational threads update a single weight vector simultaneously. This causes intentional race conditions and nondeterministic behavior but increases performance significantly.

By default, ALGORITHM=ADAM.

MAXITER=*number*

specifies the iteration budget for training. When ALGORITHM=SGD or ADAM, *number* specifies the desired number of training epochs.

By default, MAXITER=250.

MAXTIME=*number*

specifies the maximum time (in seconds) allowed for optimization, where *number* is greater than or equal to 1. When this value is reached, the optimization terminates the search and returns results. When MAXTIME=0, no maximum time is set.

By default, MAXTIME=0.

REGL1=*number*
specifies the L1 regularization parameter λ_1 for the model loss function. The *number* must be nonnegative. Note that this value is autotuned when you specify the AUTOTUNE statement.

By default, REGL1=0.

REGL2=*number*
specifies the L2 regularization parameter λ_2. The *number* must be nonnegative. Note that this value is autotuned when you specify the AUTOTUNE statement.

By default, REGL2=0.

OUTPUT Statement

OUTPUT OUT=*libref.data-table* < *option* > **;**

The OUTPUT statement writes the regression score of each observation to the output data table that you specify in the OUT= option. This information includes the variables that you specify in the COPYVARS= option and additional variables, as described in the section "Table Output" on page 385.

You must specify the following option:

OUT=*libref.data-table*
names the output data table for PROC GPREG to use. You must specify this option before any other options. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to where the data table is to be stored. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 361.

data-table specifies the name of the output data table.

The output data table contains the scored data. When you specify this option, all variables that are specified in the COPYVARS= option are added to the output data table that is specified in this option.

You can also specify the following *option*:

COPYVAR=*variable*
COPYVARS=(*variables***)**
lists one or more variables from the input data table that are transferred to the scored output data table, provided that the output data table produces one or more records per input observation. By default, PROC GPREG does not transfer any variables from the input data table to the output data table. The COPYVARS= option accepts numeric and character variables. You can also use the COPYVARS=(_ALL_) option to include all the input variables.

PARTITION Statement

PARTITION *partition-option* **;**

The PARTITION statement specifies how observations in the input data set are logically partitioned into disjoint subsets for model training, validation, and testing. For more information, see the section "Using Validation and Test Data" on page 23 in Chapter 2, "Shared Concepts." Either you can designate a variable in the input data table and a set of formatted values of that variable to determine the role of each observation, or you can specify proportions to use for randomly assigning observations to each role.

You must specify exactly one of the following *partition-options*:

FRACTION(< TEST=*fraction* **> < VALIDATE=***fraction* **> < SEED=***number* **>)**
randomly assigns specified proportions of the observations in the input data table to the roles. You specify the proportions for testing and validation by using the TEST= and VALIDATE= suboptions. If you specify both the TEST= and VALIDATE= suboptions, then the sum of the specified fractions must be less than 1 and the remaining fraction is the proportion of the observations assigned to the training role. The SEED= option specifies an integer that is used to start the pseudorandom number generator for random partitioning of data for training, testing, and validation. If you do not specify SEED=*number* or if *number* is less than or equal to 0, the seed is generated by reading the time of day from the computer's clock.

ROLE=*variable* **(< TEST='***value***' > < TRAIN='***value***' > < VALIDATE='***value***' >)**
ROLEVAR=*variable* **(< TEST='***value***' > < TRAIN='***value***' > < VALIDATE='***value***' >)**
names the *variable* in the input data table whose values are used to assign roles to each observation. This *variable* cannot also appear as an analysis variable in other statements or options. The TEST=, TRAIN=, and VALIDATE= suboptions specify the formatted values of this variable that are used to assign observation roles. If you do not specify the TRAIN= suboption, then all observations whose role is not determined by the TEST= or VALIDATE= suboption are assigned to the training role.

SAVESTATE Statement

SAVESTATE RSTORE=*libref.data-table* **;**

The SAVESTATE statement creates an analytic store for the model and saves it as a binary object in a data table. You can then use the analytic store in the ASTORE procedure to score new data. The analytic store inherits the same parameters that PROC GPREG uses.

You must specify the following option:

RSTORE=*libref.data-table*
specifies a data table in which to save the analytic store for the model. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 361.

TARGET Statement

TARGET *variable* < **LEVEL= INTERVAL** > **;**

The TARGET statement specifies the name of the *variable* to be used as the regression target. You can use one TARGET statement that includes the LEVEL=INTERVAL option for the interval target variable. Only one interval variable is accepted as the target.

Details: GPREG Procedure

Gaussian Process Regression

Gaussian process regression (GPR) models are nonparametric kernel-based probabilistic models. The exact method is not suitable for very large data sets because the time complexity is $O(n^3)$, where n is the number of observations. The GPREG procedure implements the stochastic variational Gaussian process regression (SVGPR) model, which is scalable to big data.

The SVGPR model applies stochastic variational inference (SVI) to a Gaussian process regression model by using the inducing points u as a set of variables. The goal of introducing inducing points is to reduce the effective number of input data points to the Gaussian process from n to m, where $m << n$. The m points are called the inducing points. The inducing points of the approximate Gaussian process can be viewed as a set of m latent variables that are denoted as u. They are the values of the Gaussian process at the inducing input locations z. A variational distribution $q(u)$ is then used to provide a lower bound for the marginal likelihood $p(\mathbf{y} \mid \mathbf{X})$. The optimal distribution is Gaussian, and it is parameterized as $q(\mathbf{u}) = \mathcal{N}(\mathbf{u} \mid \mathbf{m}, \mathbf{S})$. The lower bound L of the quantity $p(\mathbf{y} \mid \mathbf{X})$ becomes

$$L = \sum_{i=1}^{n} (\log N(y_i | \mathbf{k}_i^{\mathsf{T}} \mathbf{K}_{mm}^{-1} \mathbf{m}, \beta^{-1}) - \frac{1}{2}\beta \tilde{k}_{i,i} - \frac{1}{2}\mathrm{tr}(\mathbf{S}\Lambda_i)) - KL(q(\mathbf{u}) \parallel p(\mathbf{u}))$$

where $\mathbf{K}_{mm}$ is the covariance (kernel) function that is evaluated between all the pairs of inducing points; β is the precision of independent and identically distributed (iid) Gaussian observational noise; $\tilde{\mathbf{K}} = \mathbf{K}_{nn} - \mathbf{K}_{nm}\mathbf{K}_{mm}^{-1}\mathbf{K}_{mn}$, where $\mathbf{K}_{mn}$ is the covariance function between all inducing points and training points; $\mathbf{k}_i$ is a vector of the ith column of $\mathbf{K}_{mn}$; $\tilde{k}_{i,i}$ is the ith diagonal element of $\tilde{\mathbf{K}}$; and $\Lambda_i = \beta \mathbf{K}_{mm}^{-1} \mathbf{k}_i \mathbf{k}_i^{\mathsf{T}} \mathbf{K}_{mm}^{-1}$.

Notice that L is in a summation form across all the data points. Hence, you can apply the stochastic gradient descent methods to find the optimal solution of the distribution $q(\mathbf{u})$, noise precision β, and kernel parameters (and the inducing points if you omit the FIXINDUCINGPOINTS option that minimizes the $-L$).

Here are some implementation details that are worth mentioning:

- k-means clustering is used to find the initial inducing points.
- Ls, which is the Cholesky decomposition (lower-triangular representation) of the covariance matrix S, is updated using the stochastic gradient descent methods, and $Ls * Ls^{\mathsf{T}}$ is used to restore S, which guarantees the semi-positive definiteness of S.

- Jitter Cholesky decomposition is used for kernel matrices inversion to handle the singular matrix case. This technique attempts a Cholesky decomposition on the specified matrix; if the matrix is not positive definite, the function adds "jitter" and tries again. Thereafter, the amount of jitter is multiplied by 10 each time it is added again.
- The procedure always optimizes parameters in the variational distribution and in the kernel functions. In addition, you can request that inducing points be optimized by omitting the FIXINDUCINGPOINTS option.

For more information about the SVGPR model, see Hensman, Fusi, and Lawrence (2013).

Kernel Types

PROC GPREG supports the following five kernel (covariance) function types for Gaussian distributions in the Gaussian process regression model:

- Periodic kernel function, which is defined as

  $K(\mathbf{x}, \mathbf{y}) = \sigma^2 \exp(-2d^2)$

  where σ^2 is the variance.

  If you specify the AUTORELEVANCEDETERMINATION option, d is defined as

  $d = \sqrt{\sum_{j=1}^{D} \frac{\sin^2(\pi \frac{(x_j - y_j)}{p_j})}{l_j^2}}$

  where l_j is the length scale of dimension j and p_j is the period of dimension j.

  If you omit the AUTORELEVANCEDETERMINATION option, d is defined as

  $d = \sqrt{\sum_{j=1}^{D} \frac{\sin^2(\pi \frac{(x_j - y_j)}{p})}{l^2}}$

  where l is the length scale and p is the period.

- Linear kernel function:

  If you specify the AUTORELEVANCEDETERMINATION option, the function is defined as

  $K(\mathbf{x}, \mathbf{y}) = \sum_{j=1}^{D} x_j y_j \sigma_j^2$

  where σ_j^2 is the variance of dimension j.

  If you omit the AUTORELEVANCEDETERMINATION option, the function is defined as

  $K(\mathbf{x}, \mathbf{y}) = \sum_{j=1}^{D} x_j y_j \sigma^2$

  where σ^2 is the variance.

- Radial basis function (RBF) kernel function, which is defined as

  $K(\mathbf{x}, \mathbf{y}) = \sigma^2 \exp(-\frac{1}{2} d^2)$

  where σ^2 is the variance.

- Matern 3/2 kernel function, which is defined as
  $K(\mathbf{x}, \mathbf{y}) = \sigma^2(1 + \sqrt{3}d)\exp(-\sqrt{3}d)$
  where σ^2 is the variance.
- Matern 5/2 kernel function, which is defined as
  $K(\mathbf{x}, \mathbf{y}) = \sigma^2(1 + \sqrt{5}d + 5d^2/3)\exp(-\sqrt{5}d)$
  where σ^2 is the variance.

For the RBF, Matern 3/2, and Matern 5/2 kernels:

If you specify the AUTORELEVANCEDETERMINATION option, d is defined as

$$d = \sqrt{\sum_{j=1}^{D} \frac{(x_j - y_j)^2}{l_j^2}}$$

where l_j is the length scale of dimension j.

If you omit the AUTORELEVANCEDETERMINATION option, d is defined as

$$d = \sqrt{\sum_{j=1}^{D} \frac{(x_j - y_j)^2}{l^2}}$$

where l is the length scale.

Optimization

PROC GPREG supports the stochastic gradient descent (SGD) and Adam optimizations and uses options that are specified in the OPTIMIZATION statement. For the widely used SGD optimization (which is the default), the procedure supports momentum, adaptive learning rate, and adaptive decay. These options correspond to the momentum and ADADELTA variant of SGD. The validation error for one iteration is the previous iteration's loss. For more information about Adam optimization, see Kingma and Ba (2015). For more information about ADADELTA, see Zeiler (2012).

Missing Values

PROC GPREG does not perform missing data imputation. If the input data table contains any missing data, the procedure simply ignores the observations (rows) that contain missing values. You can use any data preprocessing steps (such as those in the VARIMPUTE procedure) to impute the missing data before you use the input data table in PROC GPREG.

Random Seed

You can specify the random seed by using the SEED= option in the PROC GPREG statement. The seed value is used in the random number generator, which is used both in initializing weights for optimization and in initializing the cluster assignments in clustering to find inducing points. If a random seed is not specified or if the value is invalid (zero or negative), then PROC GPREG generates a random number by using the time of day from the computer's clock.

Displayed Output

PROC GPREG displays the following output tables for use in Gaussian process regression analysis.

Model Information

The "Model Information" table provides information about the parameters that are used in the Gaussian process regression model. This information includes parameter values for seed, number of inducing points, kernel type, automatic relevance determination, whether to fix kernel parameters in the first iteration, whether to fix inducing points in optimization, whether to use simple initialization, and the maximum number of iterations for jitter Cholesky decomposition.

Number of Observations

The "Number of Observations" table displays the information about the number of observations in the input data. The Number of Observations Read column displays the count of raw data that are read from the input data table, and the Number of Observations Used column displays the count of data that are actually used in the Gaussian process regression. The count in the Number of Observations Used column can be smaller if there are observations (rows) in the input data table that have missing values.

Parameter Estimates

The "Parameter Estimates" table provides the parameter estimates for kernel parameters and the noise parameter that the Gaussian process regression uses.

Variable Statistics

The "Variable Statistics" table displays the mean and standard deviation for each interval variable that is specified in the INPUT statement and also displays the interval variable that is specified in the TARGET statement. The calculation of these statistics is based on all observations (rows) that are used for training and do not contain missing values in the input data table.

Table Output

PROC GPREG produces results of the Gaussian process regression analysis in the following output data tables.

Output

The score output table is specified in the OUTPUT statement. This table shows the target variable's posterior distribution (a Gaussian distribution that is specified by its mean and standard deviation) for each observation in the input data table. Because a Gaussian process regression model is probabilistic, it is possible to use the trained model to compute the prediction intervals. The score output table includes the variables that are specified in the COPYVARS= option, the predicted target variable's mean, the predicted target variable's standard deviation, the warning flag _WARN_, and the partition role variable _ROLE_ if partitioning of data is specified in the OPTIMIZATION statement.

Inducing Points Table

The inducing points table is specified in the OUTINDUCINGPOINTS= option in the PROC GPREG statement. This table contains inducing input points that are found by k-means clustering (and updated in the optimization phase if you omit the FIXINDUCINGPOINTS option) and the target variable's mean and standard deviation at each inducing point.

Variational Covariance Table

The variational covariance table is specified in the OUTVARIATIONALCOV= option in the PROC GPREG statement. This table contains the estimated variational Gaussian covariance matrix of the inducing points. It is a square symmetric matrix whose dimension is the value of the NINDUCINGPOINTS= option in the PROC GPREG statement and that is stored in one column.

Analytic Store Table

The analytic store table is specified in the SAVESTATE statement. This data table contains the saved Gaussian process regression model that PROC GPREG generates. You can use this analytic store table to score the data on which you want to perform Gaussian process regression analysis.

ODS Table Names

The results that PROC GPREG produces are stored in tables. You can access these tables by their names, which are shown in Table 15.3. You use these table names when you specify the DISPLAY statement.

Table 15.3 ODS Tables Produced by PROC GPREG

Table Name	Description	Statement	Option
BestConfiguration	Hyperparameters and objective function values for the best configuration	AUTOTUNE	Default
EvaluationHistory	Values of the hyperparameters and the objective function for all configurations	AUTOTUNE	Default
HyperparameterImportance	Relative importance of hyperparameters	AUTOTUNE	Default
IterationHistory	Tuner iteration history	AUTOTUNE	Default
ModelInfo	Model information	PROC GPREG	Default
NObs	Number of observations	PROC GPREG	Default
ParameterEstimates	Parameter estimates	PROC GPREG	Default
TunerCasOutputTables	Tables created during tuning	AUTOTUNE	Default
TunerInfo	Setup values that are used by the tuner	AUTOTUNE	Default
TunerResults	Values of the hyperparameters, the objective function for the default configuration (iteration 0), and up to 10 best configurations found	AUTOTUNE	Default
TunerSummary	Statistics about the tuning process	AUTOTUNE	Default
TunerTiming	Total time spent on different tasks during tuning	AUTOTUNE	Default
VarStats	Variable statistics	INPUT and TARGET	Default

Examples: GPREG Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

Example 15.1: Performing Regression on Data about Students

This example uses the Class data table that is generated by the following SAS code to demonstrate how to use the GPREG procedure to perform regression. The Class data contains the weight and height data collected from a class of 19 students. PROC GPREG predicts the weight of each student in the class on the basis of the student's height.

```
data mylib.class;
   set sashelp.class;
run;
```

These statements assume that your libref is named mylib, but you can substitute any appropriately defined libref.

The following statements run PROC GPREG and output the results to data tables and ODS tables. In this example, the number of inducing points is specified as 3, a Gaussian kernel is used in the Gaussian process regression, and the model optimization applies the Adam algorithm.

```
proc gpreg
   data=mylib.class
   seed=12345
   nThreads=32
   nInducingPoints=3
   fixInducingPoints
   outInducingPoints=mylib.gpreg_ip
   outVariationalCov=mylib.gpreg_cov;
   target Weight / level=interval;
   input  Height;
   partition fraction(valid=0 test=0.4 seed=1235);
   kernel rbf(
      autoRelevanceDetermination
      fixKernelParmFirstIter
      jitterMaxIters=10
   );
   optimization adam(
      regl1=0.5
      maxiters=500
      learningRate=0.05
      momentum=0.8
      adaptiveRate
      adaptiveDecay=0.9
      printfreq=200
   );
   savestate rstore=mylib.astore;
   output out=mylib.score copyvars=(_all_);
   display nObs varStats modelInfo ParameterEstimates;
run;
```

PROC GPREG generates four ODS tables, which are shown in Output 15.1.1 through Output 15.1.4.

Output 15.1.1 Number of Observations

The GPREG Procedure

Number of Observations	
Number of Observations Read	19
Number of Observations Used	19
Number of Observations Used for Training	14
Number of Observations Used for Testing	5

Output 15.1.2 Variable Statistics

The GPREG Procedure

Input Variable Statistics		
Variable	**Mean**	**Standard Deviation**
Height	62.642857	4.066886
Weight	102.178571	16.833583

Output 15.1.3 Model Information

Model Information	
Seed for Random Initialization for Clustering and Optimization Parameters	12345
Number of inducing points	3
Kernel Type	RBF
Automatic Relevance Determination	TRUE
Fix Kernel Parameters in the First Iteration	FALSE
Fix Inducing Points in Optimization	TRUE
Use Simple Initialization	TRUE
Maximum Number of Iterations for Jitter Cholesky Decomposition	10

Output 15.1.4 Number of Observations

Parameter Estimates		
Parameter Type	**Parameter Name**	**Parameter Value**
Kernel	Variance	0.261791
Kernel	Inverse square of length scale for variable Height	2.917941
Noise	Variance	1.347042

PROC GPREG generates the predicted heights of the students in the testing data and stores in the output table mylib.score. The following statements print the first five rows of the table shown in Output 15.1.5. The value 1 in the _ROLE_ column means that the observation has a training role, and the value 3 in the _ROLE_ column means that the observation has a testing role.

```
proc print noobs data=mylib.score (obs=5);
run;
```

Output 15.1.5 Score Output Table

Name	Sex	Age	Height	Weight	P_Weight	P_WeightMEAN	P_WeightSTD	_WARN_	_ROLE_
Alfred	M	14	69.0	112.5	102.736	102.736	7.91010		1
Alice	F	13	56.5	84.0	101.375	101.375	7.58433		1
Barbara	F	13	65.3	98.0	103.210	103.210	7.55483		1
Carol	F	14	62.8	102.5	102.934	102.934	7.98559		1
Henry	M	14	63.5	102.5	103.108	103.108	7.71592		3

Example 15.2: Tuning Hyperparameters of a Gaussian Process Regression Model

This example uses the Class data table to demonstrate how to use the AUTOTUNE statement in the GPREG procedure to automatically tune the hyperparameters of a Gaussian process regression model. The Class data table, which is generated by the following SAS code, contains weight and height data that were collected from a class of 19 students. PROC GPREG predicts the weight of each student in the class on the basis of the student's height.

```
data mylib.class;
   set sashelp.class;
run;
```

These statements assume that your libref is named mylib, but you can substitute any appropriately defined libref.

The following statements use PROC GPREG with the AUTOTUNE statement to automatically tune the hyperparameters of a Gaussian process regression model:

```
proc gpreg
   data=mylib.class
   seed=12345
   nThreads=32
   nInducingPoints=3
   fixInducingPoints
   outInducingPoints=mylib.gpreg_ip
   outVariationalCov=mylib.gpreg_cov;
   target Weight / level=interval;
   input  Height;
   autotune
      /* Tuning Parameters
      You do not need to specify any tuning parameters for the default
      tuning process. If you want to make adjustments to the default
      tuning process, uncomment the following block of code and change
      any of the tuning parameters' attributes.

      tuningparameters=(
         ALGORITHM         ( values=SGD ADAM init=SGD )
```

```
            ARD                ( values=FALSE TRUE init=FALSE )
            JITTERMAXITERS     ( lb=5    ub=15   init=10   )
            KERNEL             ( values=LINEAR MATERN32 MATERN52 PERIODIC RBF init=RBF )
            LEARNINGRATE       ( lb=1E-7 ub=2.0  init=1E-3 )
            MINIBATCHSIZE      ( values=1 2 4 8 16 32 64 init=1 exclude )
            MOMENTUM           ( lb=0.0  ub=0.99 init=0.1  )
            NINDUCINGPOINTS    ( lb=25   ub=150  init=100  )
            TUNEADAPTIVEDECAY  ( lb=0.0  ub=0.99 init=0.95 )
            TUNEADAPTIVERATE   ( values=FALSE TRUE init=FALSE exclude )
         )
         */
      ;
      ods select BestConfiguration; /* Remove to see all results tables */
   run;
```

These statements produce the table shown in Output 15.2.1. The table displays the evaluation number, the values of the tuning parameters, and the error metric value for the best Gaussian process regression model that the tuner found. Note that the ODS SELECT statement limits the displayed results to a single table. You can remove this statement to see all results tables. For the full list of ODS tables that PROC GPREG produces, see Table 15.3.

Output 15.2.1 Best Configuration Table

The GPREG Procedure

Best Configuration	
Evaluation	88
Automatic Relevance Determination	TRUE
Kernel	LINEAR
Maximum Number of Iterations for Jitter Cholesky Decomposition	7
Number of Inducing Points	93
Algorithm	ADAM
Momentum	0.87117201
Learning Rate	0.07608437
Adaptive Decay	0.45358469
Root Average Square Error	21.62438784

References

Hensman, J., Fusi, N., and Lawrence, N. D. (2013). "Gaussian Processes for Big Data." In *Proceedings of the Twenty-Ninth Conference on Uncertainty in Artificial Intelligence*. Arlington, VA: AUAI. http://www.auai.org/uai2013/prints/papers/244.pdf.

Kingma, D. P., and Ba, J. (2015). "Adam: A Method for Stochastic Optimization." In *Proceedings of the 3rd International Conference for Learning Representations*. Baixas, France: ISCA. https://arxiv.org/abs/1412.6980.

Zeiler, M. D. (2012). "ADADELTA: An Adaptive Learning Rate Method." https://arxiv.org/pdf/1212.5701.pdf.

Chapter 16
GRADBOOST Procedure

Contents

Overview: GRADBOOST Procedure . . . **393**
- Using SAS Viya Workbench . . . 393

Getting Started: GRADBOOST Procedure . . . **394**

Syntax: GRADBOOST Procedure . . . **398**
- PROC GRADBOOST Statement . . . 398
- AUTOTUNE Statement . . . 405
- CODE Statement . . . 413
- CROSSVALIDATION Statement . . . 413
- FREQ Statement . . . 414
- ID Statement . . . 414
- INPUT Statement . . . 414
- MITIGATEBIAS Statement . . . 415
- OUTPUT Statement . . . 417
- PARTITION Statement . . . 418
- SAVESTATE Statement . . . 418
- TARGET Statement . . . 419
- TRANSFERLEARN Statement . . . 419
- VIICODE Statement . . . 420
- WEIGHT Statement . . . 420

Details: GRADBOOST Procedure . . . **421**
- Subsampling the Data . . . 421
- Training a Decision Tree . . . 421
- Splitting Nominal Values . . . 422
  - Methods . . . 422
  - Risks of Many Categories . . . 423
- Imposing Monotonicity Constraints . . . 423
- Boosting . . . 423
  - Early Stopping . . . 424
- Measuring Prediction Error . . . 425
- Handling Missing Values . . . 425
  - Strategies . . . 425
  - Specifics . . . 426
- Measuring Variable Importance . . . 426
  - Residual Sum of Squares Importance Method . . . 427
  - Random Branch Assignment Importance Method . . . 429
- Transfer Learning . . . 429

Hyperparameter Tuning 430
k-fold Cross Validation 431
TreeSHAP Values 431
Bias Mitigation 432
Exponentiated Gradient Reduction (EGR) Algorithm 432
Displayed Output 434
Model Information 434
Number of Observations 434
Variable Importance 434
RBA Variable Importance 434
Fit Statistics 435
Tuner Information 435
Tuner Summary 435
Tuner Timing 435
Best Configuration 435
Tuner Results 435
Evaluation History 435
Cross-Validation Fit Statistics 435
OutputCasTables Table 436
Predicted Probability Names 436
Constraint History 436
Group Metrics History 436
Iteration History 436
Mitigation Information 436
Timing 436
Weight History 436
ODS Table Names 437
ODS Graphics 438
Examples: GRADBOOST Procedure **439**
Example 16.1: Scoring New Data by Using a Previous Boosting Model 439
Example 16.2: Transfer Learning 441
Example 16.3: Tuning a Boosting Model 444
Example 16.4: Mitigating Bias of a Boosting Tree Model 447
References **451**

Overview: GRADBOOST Procedure

The GRADBOOST procedure creates a predictive model called a gradient boosting model in SAS Viya Workbench. A gradient boosting model consists of multiple decision trees. A *predictive model* defines a relationship between input variables and a target variable. The purpose of a predictive model is to predict a target value from inputs. The GRADBOOST procedure creates the model by using *training data* in which the target values are known. The model can then be applied to observations in which the target is unknown. If the predictions fit the new data well, the model is said to *generalize* well. Good generalization is the primary goal of predictive tasks. A predictive model might fit the training data well but generalize poorly.

A *decision tree* is a type of predictive model that has been developed independently in the statistics and artificial intelligence communities. Based on the boosting method in Hastie, Tibshirani, and Friedman (2001) and Friedman (2001), the GRADBOOST procedure creates a predictive model by fitting a set of additive trees.

For more information about training a gradient boosting model, see the section "Boosting" on page 423.

Using SAS Viya Workbench

SAS Viya Workbench has greatly simplified the SAS programming experience by offering a syntax style nearly identical to that of SAS 9 and earlier releases. However, there are minor exceptions. In the past, SAS Viya procedures required the use of a CAS engine libref. This engine served multiple purposes, but it was primarily used to upload data to the CAS server and manage the data. Its additional purposes included support of new data types and improved performance for large data sets. No CAS server is needed for SAS Viya Workbench, because all computations are done in place for you in a single location.

In the following program, the LIBNAME statement creates a mylib libref that the examples in this chapter use to refer to your current or desired work directory:

```
libname mylib ".";
```

For more information, see *SAS Viya Workbench: Getting Started*.

Getting Started: GRADBOOST Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

A common use of gradient boosting models is to predict whether a mortgage applicant will default on a loan. The home equity data table Hmeq, which is in the Sampsio library that SAS provides, contains observations for 5,960 mortgage applicants. A variable named Bad indicates whether the applicant, after being approved for a loan, paid off or defaulted on the loan.

This example uses the Hmeq data table to build a gradient boosting model that is used to score the data and can be used to score data about new loan applicants. Table 16.1 describes the variables in Hmeq.

Table 16.1 Variables in the Home Equity (Hmeq) Data Table

Variable	Role	Level	Description
Bad	Response	Binary	1 = applicant defaulted on the loan or is seriously delinquent 0 = applicant paid off the loan
CLAge	Predictor	Interval	Age of oldest credit line in months
CLNo	Predictor	Interval	Number of credit lines
DebtInc	Predictor	Interval	Debt-to-income ratio
Delinq	Predictor	Interval	Number of delinquent credit lines
Derog	Predictor	Interval	Number of major derogatory reports
Job	Predictor	Nominal	Occupational category
Loan	Predictor	Interval	Requested loan amount
MortDue	Predictor	Interval	Amount due on mortgage
nInq	Predictor	Interval	Number of recent credit inquiries
Reason	Predictor	Binary	`'DebtCon'` = debt consolidation `'HomeImp'` = home improvement
Value	Predictor	Interval	Value of property
YoJ	Predictor	Interval	Years at present job

The following statements load the mylib.hmeq data into your SAS library. For this example, the statements assume that your SAS libref is named mylib, but you can substitute any appropriately defined SAS libref.

```
data mylib.hmeq;
   length Bad Loan MortDue Value 8 Reason Job $7
              YoJ Derog Delinq CLAge nInq CLNo DebtInc 8;
   set sampsio.hmeq;
run;

proc print data=mylib.hmeq(obs=10); run;
```

Output 16.1 shows the first 10 observations of mylib.hmeq.

Figure 16.1 Partial Listing of the mylib.hmeq Data

Obs	Bad	Loan	MortDue	Value	Reason	Job	YoJ	Derog	Delinq	CLAge	nInq	CLNo	DebtInc
1	1	1100	25860	39025	HomeImp	Other	10.5	0	0	94.367	1	9	.
2	1	1500	.	.			.	.	.	.	.	.	.
3	1	1800	48649	57037	HomeImp	Other	5.0	3	2	77.100	1	17	.
4	1	2000	.	62250	HomeImp	Sales	16.0	0	0	115.800	0	13	.
5	1	2000	45000	55000	HomeImp	Other	3.0	0	0	86.067	2	25	.
6	1	2200	24280	34687	HomeImp	Other	.	0	1	300.867	0	8	.
7	1	2300	28192	40150	HomeImp	Other	4.5	0	0	54.600	1	16	.
8	1	2400	50000	73395	HomeImp	ProfExe	5.0	1	0	.	1	0	.
9	1	2400	.	17180	HomeImp	Other	.	0	0	14.567	3	4	.
10	1	2500	15000	20200	HomeImp		18.0	0	0	136.067	1	19	.

PROC GRADBOOST treats numeric variables as interval inputs unless you specify otherwise. Character variables are always treated as nominal inputs. The following statements run PROC GRADBOOST and save the model in a table named mylib.savedModel:

```
proc gradboost data=mylib.hmeq outmodel=mylib.savedModel seed=12345;
   input Delinq Derog Job nInq Reason / level = nominal;
   input CLAge CLNo DebtInc Loan Mortdue Value YoJ / level = interval;
   target Bad / level = nominal;
   ods output FitStatistics=fitstats;
run;
```

No parameters are specified in the PROC GRADBOOST statement; therefore, the procedure uses all default values. For example, the number of trees in the boosting model is 100, and the number of bins for interval input variables is 20.

The INPUT and TARGET statements are required in order to run PROC GRADBOOST. The INPUT statement indicates which variables to use to build the model, and the TARGET statement indicates which variable the procedure predicts.

Figure 16.2 displays the “Model Information” table. This table shows the values of the training parameters in the first six rows, in addition to some basic information about the trees in the boosting model.

Figure 16.2 Model Information

The GRADBOOST Procedure

Model Information	
Number of Trees	100
Learning Rate	0.1
Subsampling Rate	0.5
Number of Variables Per Split	12
Number of Bins	50
Number of Input Variables	12
Maximum Number of Tree Nodes	31
Minimum Number of Tree Nodes	17
Maximum Number of Branches	2
Minimum Number of Branches	2
Maximum Depth	4
Minimum Depth	4
Maximum Number of Leaves	16
Minimum Number of Leaves	9
Maximum Leaf Size	2603
Minimum Leaf Size	5
Seed	12345
Lasso (L1) penalty	0
Ridge (L2) penalty	1
Actual Number of Trees	100
Average Number of Leaves	14.17

Figure 16.3 displays the "Number of Observations" table, which shows how many observations were read and used. If you specify a PARTITION statement, the "Number of Observations" table also displays the number of observations that were read and used per partition.

Figure 16.3 Number of Observations

	Training
Number of Observations Read	5960
Number of Observations Used	5960

Figure 16.4 displays the estimates of variable importance. The rows in this figure are sorted by the importance measure. A conclusion from fitting the boosting model to these data is that DebtInc is the most important predictor of loan default.

Figure 16.4 Variable Importance

Variable Importance			
Variable	Importance	Std Dev Importance	Relative Importance
DebtInc	27.9044	77.6122	1.0000
Delinq	5.7250	7.3421	0.2052
Value	5.5142	5.9790	0.1976
CLAge	5.3850	5.9920	0.1930
Derog	4.0895	4.9382	0.1466
CLNo	3.3499	3.1560	0.1201
Job	3.0641	2.6463	0.1098
YoJ	2.8628	2.9581	0.1026
Loan	2.7176	2.9825	0.0974
MortDue	2.5979	2.7725	0.0931
nInq	1.9070	2.4852	0.0683
Reason	0.3963	1.0004	0.0142

Figure 16.5 shows the first 10 and last 10 observations of the fit statistics. PROC GRADBOOST computes fit statistics on a per-tree basis. As the number of trees increases, the fit statistics usually improve (decrease) at first and then level off and fluctuate within a small range.

Figure 16.5 Fit Statistics

Fit Statistics			
Number of Trees	Training Average Square Error	Training Misclassification Rate	Training Log Loss
1	0.1458	0.1995	0.459
2	0.1352	0.1995	0.431
3	0.1258	0.1995	0.407
4	0.1186	0.1995	0.388
5	0.1125	0.1973	0.372
6	0.1076	0.1827	0.360
7	0.1032	0.1567	0.348
8	0.0995	0.1361	0.338
9	0.0961	0.1250	0.329
10	0.0930	0.1190	0.320
.	.	.	.
.	.	.	.
.	.	.	.
91	0.0481	0.0649	0.172
92	0.0477	0.0646	0.170
93	0.0474	0.0648	0.170
94	0.0472	0.0636	0.169
95	0.0469	0.0644	0.168
96	0.0467	0.0629	0.167
97	0.0463	0.0626	0.166
98	0.0460	0.0616	0.165
99	0.0457	0.0611	0.164
100	0.0456	0.0597	0.164

Syntax: GRADBOOST Procedure

The following statements are available in the GRADBOOST procedure:

PROC GRADBOOST < *options* > ;
  AUTOTUNE < *options* > ;
  CODE < *options* > ;
  CROSSVALIDATION < *options* > ;
  FREQ *variable* ;
  ID *variables* ;
  INPUT *variables* < / *options* > ;
  MITIGATEBIAS *mitigate-options* ;
  OUTPUT OUT=*libref.data-table* < *option* > ;
  PARTITION *partition-option* ;
  SAVESTATE RSTORE=*libref.data-table* ;
  TARGET *variable* < / **LEVEL=NOMINAL** | **INTERVAL** > ;
  TRANSFERLEARN *variable* < / *options* > ;
  VIICODE < *options* > ;
  WEIGHT *variable* ;

The PROC GRADBOOST, INPUT, and TARGET statements are required. The INPUT statement can appear multiple times.

The rest of this section provides detailed syntax information about each of the preceding statements, beginning with the PROC GRADBOOST statement. The remaining statements are described in alphabetical order.

PROC GRADBOOST Statement

PROC GRADBOOST < *options* > ;

The PROC GRADBOOST statement invokes the procedure. Table 16.2 summarizes the options in this statement.

Table 16.2 PROC GRADBOOST Statement Options

Option	Description
Basic Options	
ADDTREES	Adds trees to an already generated boosting model, which is specified in the INMODEL= option
BINMETHOD=	Specifies how to bin interval inputs prior to training
DISTRIBUTION=	Specifies the distribution of the objective function
EARLYSTOP=	Specifies how to perform early stopping during model training
INMODEL=	Specifies a saved gradient boosting model to use for scoring a new table
LASSO=	Specifies the L1 norm regularization parameter
LEARNINGRATE=	Specifies the learning rate for each tree
NOPRINT	Suppresses ODS output

Table 16.2 *continued*

Option	Description
NTHREADS=	Specifies the number of threads to use in the computation
NTREES=	Specifies the number of trees to grow in the boosting model
NUMBIN=	Specifies the number of bins for continuous variables
OFFSET=	Specifies an offset variable to use with DISTRIBUTION=POISSON or TWEEDIE
OUTMODEL=	Specifies the data table to which the gradient boosting model is to be saved
PRINTTARGET	Creates tables that contain information about columns in the output
RBAIMP	Creates a variable importance table by using random branch assignment
RIDGE=	Specifies the L2 norm regularization parameter
SAMPLINGRATE=	Specifies the fraction of the training data to use for growing each tree
SEED=	Specifies the random number seed to use for model building
VARS_TO_TRY=	Specifies the number of variables to examine at each node split
VII=	Calculates the importance of the specified types of variable interactions
Splitting Options	
ASSIGNMISSING=	Specifies how to handle missing values in a predictor variable
MAXBRANCH=	Specifies the maximum number of splits per node
MAXDEPTH=	Specifies the maximum tree depth
MINLEAFSIZE=	Specifies the minimum number of observations per leaf
MINUSEINSEARCH=	Specifies the minimum number of observations to use with the USEINSEARCH policy for handling missing values
NOMSEARCH=	Specifies the method to use to find a split on a nominal variable

You also specify the following *options*:

ADDTREES

adds trees to an already generated gradient boosting model, which is specified in the INMODEL= option. The resulting model will consist of all the trees in the original model, plus a number of new trees, where the number of new trees is specified in the NTREES= option. When you update a model, you can use the original training data or you can use new data. If you use new data, then the new data must have all the variables that the original training data have.

ASSIGNMISSING=NONE | MACSMALL | USEINSEARCH

specifies how to handle missing values during training and creates a splitting rule to handle missing values and unknown levels during scoring. An unknown level is a level of a categorical predictor variable that does not exist in the training data but is encountered during scoring.

During model training, PROC GRADBOOST searches for the best splitting rule for each node, as described in the section "Training a Decision Tree" on page 421. During model scoring, observations are assigned to a node in a tree based on the best splitting rule if that rule's variable is not missing. If the variable is missing for the observation, then the default splitting that is specified by this option is

used. The default splitting rule enables all data to be scored, even if the best splitting rule cannot be used on a particular observation.

You can specify one of the following values:

NONE during training, excludes observations that have any missing variables from the model. In the scoring phase, this default rule assigns observations that have missing values of an interval predictor variable to the branch with the smallest predictor values, and assigns observations that have unknown and missing nominal levels to the branch with the most training observations.

MACSMALL during training, treats a missing value as a separate, legitimate value in the search for a split for the primary splitting rule. Missing values in interval predictor variables are treated as less than any other number. In the scoring phase, this default rule assigns observations that have missing values of interval predictor variables to the leftmost branch of the split, and assigns observations that have unknown nominal levels to the largest branch in the split.

USEINSEARCH during training, treats a missing value as a separate, legitimate value in the search for a split for the primary splitting rule. Missing values in interval predictor variables are treated as a special level that is used during the split process. In the scoring phase, this default rule assigns observations that have missing values of interval predictor variables to the branch determined during the model growing, and assigns observations that have unknown nominal levels to the largest branch in the split.

By default, ASSIGNMISSING=USEINSEARCH.

BINMETHOD=BUCKET | QUANTILE

specifies how to bin interval input variables prior to growing the gradient boosting model. The number of bins that are created is determined by the NUMBIN= option.

You can specify one of the following values:

BUCKET bins interval input variables into fixed-width bins. The width of each bin for a particular variable is calculated by subtracting the smallest value among all observations from the largest value among all observations, and then dividing that result by the number of bins.

QUANTILE bins interval input variables into bins according to their quantile. The width of the bins for a particular variable is not fixed, but the number of observations in each bin is approximately equal.

By default, BINMETHOD=QUANTILE.

DATA=*libref.data-table*

names the input data table for PROC GRADBOOST to use. The default is the most recently created data table. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to the data. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 393.

data-table specifies the name of the input data table.

DISTRIBUTION=BINARY | GAUSSIAN | MULTINOMIAL | POISSON | TWEEDIE(POWER=*p*)
specifies the distribution of the objective function. The default distribution is GAUSSIAN, BINARY, or MULTINOMIAL for an interval, binary, or nominal target, respectively. The POISSON distribution is appropriate for count data. The TWEEDIE distribution is useful for modeling total losses in insurance. The default value of POWER=p is 1.5, and $1 \leq p \leq 2$.

EARLYSTOP(*suboption* . . .)
specifies options for stopping the gradient boosting model training early. For more information, see the section "Early Stopping" on page 424. Two approaches to early stopping are available: *stagnation* and *threshold*. You can specify one or more of the following *suboptions*:

METRIC=MCR | LOGLOSS
specifies whether to use the misclassification rate or the log-loss prediction error in early stopping for a binary or nominal target.

By default, METRIC=MCR. Average square error is always used for an interval target.

MINIMUM=NO | YES
specifies whether the stagnation approach should count iterations starting from the iteration that has the smallest error.

By default, MINIMUM=NO.

STAGNATION=*number*
specifies the *number* of iterations in the gradient boosting model to consider for early stopping, where *number* must be a nonnegative integer. If STAGNATION=0, then the stagnation approach is not used. If you want to use autotuning by specifying the AUTOTUNE statement, but you do not want autotuning to use early stopping, then specify STAGNATION=0.

By default, STAGNATION=0. When you specify the AUTOTUNE statement, STAGNATION=4 by default.

THRESHOLD=τ
specifies the threshold value that stops training when the error equals or exceeds it.

By default, THRESHOLD=0 and the threshold approach to early stopping is not used.

THRESHOLDITER=*T*
specifies the minimum number of training iterations to run before the threshold approach is invoked.

TOLERANCE=*number*
specifies the *number* to be used as the tolerance for the stagnation approach to early stopping, where *number* must be nonnegative.

By default, TOLERANCE=0.

INMODEL=*libref.data-table*
specifies the data table that you previously saved as a gradient boosting model by using the OUTMODEL= option in a previous run of PROC GRADBOOST. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the input data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 393.

You can specify the INMODEL= option for the following purposes:

- To score new data. In this case, both the DATA= option and the OUTPUT statement are required, and any other options, except for NOPRINT, are ignored.
- To add trees to an existing gradient boosting model. In this case, the GRADBOOST procedure uses the originally saved model and adds new trees to it by using the same or new data. If new data are used, then they must contain all the variables of the original data. When you use the INMODEL= option to add trees to an existing gradient boosting model, you must also specify the ADDTREE option. In this case, all other training options are also honored.

The data table used in the INMODEL= option must include the attributes that are associated with the table produced by the OUTMODEL= option, or PROC GRADBOOST issues an error.

LASSO=*number*

L1=*number*

specifies the L1 norm regularization parameter, where *number* must be nonnegative.

By default, LASSO=0. This value can be tuned with the AUTOTUNE statement.

LEARNINGRATE=*number*

specifies the learning rate for the gradient boosting algorithm, where *number* must be between 0 and 1, inclusive.

By default, LEARNINGRATE=0.1. This value can be tuned with the AUTOTUNE statement.

MAXBRANCH=*b*

specifies the maximum number of children per node in the tree. PROC GRADBOOST tries to create this number of children unless it is impossible (for example, if a split variable does not have enough levels).

By default, MAXBRANCH=2.

MAXDEPTH=*number*

specifies the maximum depth of the tree to be grown. The number of levels in a tree is equal to the depth plus 1.

By default, MAXDEPTH=4.

This value can be tuned with the AUTOTUNE statement. In the ODS output that corresponds to tables that are generated using the AUTOTUNE statement, the term “MAXLEVEL” is used. The value of MAXLEVEL is equal to the value of the MAXDEPTH= option plus 1.

MINLEAFSIZE=*number*

LEAFSIZE=*number*

specifies the minimum *number* of observations that each child of a split must contain in the training data table in order for the split to be considered. The count of observations includes those that are excluded from training as a consequence of specifying a value less than one in the SAMPLINGRATE= option.

By default, MINLEAFSIZE=5. This value can be tuned with the AUTOTUNE statement.

MINUSEINSEARCH=*number*

specifies a threshold for using missing values in the split search when ASSIGNMISSING=USEINSEARCH. If the number of observations in which the splitting variable has missing values is greater than or equal to *number*, then PROC GRADBOOST uses the USEINSEARCH policy to handle missing values for that variable.

By default, MINUSEINSEARCH=1.

NOMSEARCH(*suboption* **. . .)**

specifies search methods for splitting on a nominal variable. For more information, see the section "Splitting Nominal Values" on page 422. If you do not specify this option, the default values of the *suboptions* determine the search method. You can specify one or more of the following *suboptions*:

MAXCATEGORIES=*number*

specifies the maximum number of categories to use in a splitting rule. The input is eligible for splitting nodes in which no more than *number* categories appear. The value of *number* must be greater than 1.

By default, MAXCATEGORIES=128.

SHRINKAGE=*number*

specifies how much weight to give the average gradient when you combine it with the average gradient within a category. You use this option only when sorting the categories. The value of *number* must be nonnegative.

By default, SHRINKAGE=10.

SORT=*number*

specifies the minimum cardinality in the node of the nominal variable for using the sort method. The sort method sorts the categories and then examines binary splits that preserve the ordering. The value of *number* must be a nonnegative integer. If *number* is greater than the value of the MAXCATEGORIES=*number* option, then the variable is not used in the node.

By default, SORT=10.

NOPRINT

suppresses ODS output.

NTHREADS=*number*

The gradient boosting algorithm automatically determines the default value on the basis of the properties of the training data, model, and hardware that you use. If you specify this option, the specified *number* of threads is used if it is less than or equal to the maximum number of threads that Viya allows. If the *number* of threads is larger than the maximum number of threads that Viya allows, the maximum number allowed is used instead.

NTREES=*number*

specifies the *number* of trees to grow in the gradient boosting model.

By default, NTREES=100. This value can be tuned with the AUTOTUNE statement.

NUMBIN=*number*

NBINS=*number*

specifies the *number* of bins to use for binning the interval input variables. PROC GRADBOOST bins continuous predictors to a fixed bin size. This option controls the number of bins and thereby also the size of the bins.

By default, NUMBIN=50.

OFFSET=*variable*

specifies an offset variable to be used with DISTRIBUTION=POISSON or TWEEDIE.

OUTMODEL=< *libref.***>***data-table*

specifies the data table to which to save the gradient boosting model. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 393.

The data table that results from this option contains information about each node and each tree in the gradient boosting model, including the splitting variables, the child nodes, the number of observations at each node, and the predicted response at each node. This table also has attributes attached that contain information about the input variables, the target variable, and the gradient boosting model.

PRINTTARGET

outputs tables that indicate generated columns in the OUT= table from the OUTPUT statement. For a continuous response, PROC GRADBOOST generates an output table named PredName, which indicates the name of the predicted value column. For a categorical response, PROC GRADBOOST generates an output table named PredIntoName, which indicates the name of the predicted value column, and also an output table named PredProbName, which indicates the names of the predicted probability columns.

RBAIMP

creates a variable importance table by using random branch assignment (RBA). This table is created in addition to the normal variable importance table that is calculated using the residual sum of squares (RSS) error. For more information about RBA and RSS variable importance, see the section "Measuring Variable Importance" on page 426.

RIDGE=*number*

L2=*number*

specifies the L2 norm regularization parameter on prediction. The value of *number* must be nonnegative.

By default, RIDGE=1. This value can be tuned with the AUTOTUNE statement.

SAMPLINGRATE=*number*

specifies the fraction of the training data to be used for growing each tree in the boosting model.

By default, SAMPLINGRATE=0.5. This value can be tuned with the AUTOTUNE statement.

SEED=*number*

specifies the initial seed for random number generation for model building. The value of *number* must be an integer. If you do not specify a seed or you specify a value less than or equal to 0, the seed is generated by reading the time of day from the computer's clock.

VARS_TO_TRY=*m*

M=*m*

specifies the number of input variables to consider splitting on in a node, where *m* ranges from 1 to the number of input variables.

By default, *m* is the number of input variables. This value can be tuned with the AUTOTUNE statement.

VII=2 | 3

INTERACTIONIMP=2 | 3

calculates the variable interaction importance, which is described in the section "Interaction Importance" on page 428. You can specify the following values:

2 calculates the importance of all two-way variable interactions.

3 calculates the importance of all three-way and all two-way variable interactions.

If you do not specify this option, then the variable interaction importance is not calculated.

AUTOTUNE Statement

AUTOTUNE < *options* > **;**

The AUTOTUNE statement searches for the best combination of values of the LASSO=, LEARNINGRATE=, MAXDEPTH=, MINLEAFSIZE=, NTREES=, NUMBIN=, RIDGE=, SAMPLINGRATE=, and VARS_TO_TRY= options in the PROC GRADBOOST statement. You cannot specify both the AUTOTUNE statement and the CROSSVALIDATION statement in the same procedure run.

Table 16.3 summarizes the *options* that you can specify in the AUTOTUNE statement. For more information about all options except the TUNINGPARAMETERS= option, see the option's description in the section "AUTOTUNE Statement" on page 6 in Chapter 2, "Shared Concepts." The TUNINGPARAMETERS= option is described following Table 16.3.

Table 16.3 AUTOTUNE Statement Options

Option	Description
APPENDLOOKUP	Specifies that the table specified in the HISTORYTABLE= option contain the rows from the table specified in the LOOKUPTABLE= option
CONSTRAINTS=	Specifies the constraints on the output metrics
EVALHISTORY=	Specifies how to report the evaluation history of the tuner
FOLDCOLUMN=	Specifies the column in the data table in which the cross validation fold value is indicated
FRACTION=	Specifies the fraction of observations to use for validation

Table 16.3 *continued*

Option	Description
HISTORYTABLE=	Specifies the table that contains the evaluation history
KFOLD=	Specifies the number of folds for *k*-fold cross validation
LIVEUPDATE	Specifies that the table specified in the HISTORYTABLE= option be updated at every evaluation
LOOKUPTABLE=	Specifies the table to use for evaluation lookup
MAXBAYES=	Specifies the maximum number of points in the kriging model
MAXEVALS=	Specifies the maximum number of evaluations
MAXITER=	Specifies the maximum number of iterations when SEARCHMETHOD=GA or SEARCHMETHOD=BAYESIAN
MAXTIME=	Specifies the maximum time for all iterations
MAXTRAINTIME=	Specifies the maximum time for a model training
NCONVITER=	Specifies the number of convergence iterations
NOGRIDSHUFFLE	Requests that the grid points not be shuffled
NOLOCALSEARCH	Disables local search optimization
NPARALLEL=	Specifies the number of parallel sessions
NSUBSESSIONWORKERS=	Specifies the number of workers in parallel sessions
OBJECTIVE=	Specifies the objective function
POPSIZE=	Specifies the population size when SEARCHMETHOD=GA or SEARCHMETHOD=BAYESIAN
SAMPLESIZE=	Specifies the sample size when SEARCHMETHOD=LHS or SEARCHMETHOD=RANDOM
SEARCHMETHOD=	Specifies the search method that the optimizer uses
SECONDOBJECTIVE=	Specifies the second objective to use for tuning
SELECTINITPOINT	Specifies that the tuner select the best evaluation from the lookup table
TARGETEVENT=	Specifies the target event for ROC-based calculations
TRAINFRACTION=	Specifies the fraction of observations to use for training
TUNINGPARAMETERS=	Specifies the custom tuning parameters
USEPARAMETERS=	Specifies how to handle the TUNINGPARAMETERS= option

TUNINGPARAMETERS=(*suboption* | **...** | **<** *suboption* **>)**

TUNEPARMS=(*suboption* | **...** | **<** *suboption* **>)**

specifies which parameters to tune and which ranges to tune over. If USEPARAMETERS=STANDARD, this option is ignored.

You can specify one or more of the following *suboptions*:

LASSO (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**

specifies information about the L1 regularization to use for tuning the gradient boosting model. For more information, see the LASSO= option in the PROC GRADBOOST statement.

You can specify the following additional suboptions:

LB=*number*
specifies the minimum L1 regularization value to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=0.

UB=*number*
specifies the maximum L1 regularization value to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=10.

VALUES=*value-list*
specifies a list of L1 regularization values to consider during tuning, where *value-list* is a space-separated list of numbers greater than or equal to 0. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial L1 regularization value for the tuner to use.

By default, INIT=0.

EXCLUDE
excludes L1 regularization from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

LEARNINGRATE (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**
specifies information about the learning rate to use for tuning the gradient boosting model. For more information, see the LEARNINGRATE= option in the PROC GRADBOOST statement.

You can specify the following additional suboptions:

LB=*number*
specifies the minimum learning rate to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=0.01.

UB=*number*
specifies the maximum learning rate to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=1.

VALUES=*value-list*
specifies a list of learning rates to consider during tuning, where *value-list* is a space-separated list of numbers greater than 0 and less than or equal to 1. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial learning rate for the tuner to use.

By default, INIT=0.1.

EXCLUDE
excludes the learning rate from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

MAXDEPTH (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**
specifies information about the maximum depth of trees to use for tuning the gradient boosting model. For more information, see the MAXDEPTH= option in the PROC GRADBOOST statement.

You can specify the following additional suboptions:

LB=*number*
specifies the minimum value of the MAXDEPTH= option in the PROC GRADBOOST statement to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=1.

UB=*number*
specifies the maximum value of the MAXDEPTH= option in the PROC GRADBOOST statement to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=6.

VALUES=*value-list*
specifies a list of values of the MAXDEPTH= option in the PROC GRADBOOST statement to consider during tuning, where *value-list* is a space-separated list of positive integers. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial value of the MAXDEPTH= option in the PROC GRADBOOST statement for the tuner to use.

By default, INIT=4.

MINLEAFSIZE (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**
specifies information about tuning the leaf size option for training the decision tree. This option is not tuned by default. If it is specified without any suboptions, then tuning uses the following values: 1, 5, 10, 20, 40, 80, 160, 320. For more information, see the MINLEAFSIZE= option in the PROC GRADBOOST statement.

You can specify the following additional suboptions:

LB=*number*
specifies the minimum leaf size value to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=1.

UB=*number*
specifies the maximum leaf size value to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=320.

VALUES=*value-list*
specifies a list of leaf size values to consider during tuning, where *value-list* is a space-separated list of positive integers. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial leaf size for the tuner to use.

By default, INIT=5.

EXCLUDE
excludes the leaf size from the tuning process.

EXCLUDE
excludes the MAXDEPTH= option from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

NTREES (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**
specifies information about the number of trees to use for tuning the gradient boosting model. For more information, see the NTREES= option in the PROC GRADBOOST statement.

You can specify the following additional suboptions:

LB=*number*
specifies the minimum number of trees to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=20.

UB=*number*
specifies the maximum number of trees to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=150.

VALUES=*value-list*
specifies a list of numbers of trees to consider during tuning, where *value-list* is a space-separated list of positive integers. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial number of trees for the tuner to use.

By default, INIT=100.

EXCLUDE
excludes the number of trees from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

NUMBIN (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**
specifies information about the number of bins to consider during the tuning of the options for PROC GRADBOOST. For more information, see the NUMBIN= option in the PROC GRADBOOST statement.

You can specify the following additional suboptions:

LB=*number*
specifies the minimum number of bins to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=20.

UB=*number*
specifies the maximum number of bins to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=100.

VALUES=*value-list*
specifies a list of the numbers of bins to consider during tuning, where *value-list* is a space-separated list of positive integers. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial number of bins for the tuner to use.

By default, INIT=50.

EXCLUDE
excludes the number of bins from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

RIDGE (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**
specifies information about the L2 regularization to use for tuning the gradient boosting model. For more information, see the RIDGE= option in the PROC GRADBOOST statement.

You can specify the following additional suboptions:

LB=*number*
specifies the minimum L2 regularization value to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=0.

UB=*number*
specifies the maximum L2 regularization value to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=10.

VALUES=*value-list*
specifies a list of L2 regularization values to consider during tuning, where *value-list* is a space-separated list of numbers greater than or equal to 0. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial L2 regularization value for the tuner to use.

By default, INIT=0.

EXCLUDE
excludes L2 regularization from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

SAMPLINGRATE (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**
specifies information about the portion of the training data for each boosted tree to use for tuning the gradient boosting model. For more information, see the SAMPLINGRATE= option in the PROC GRADBOOST statement.

You can specify the following additional suboptions:

LB=*number*
specifies the minimum sampling rate to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=0.1.

UB=*number*
specifies the maximum sampling rate to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=1.

VALUES=*value-list*
specifies a list of sampling rates to consider during tuning, where *value-list* is a space-separated list of numbers greater than 0 and less than or equal to 1. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial sampling rate for the tuner to use.

By default, INIT=0.5.

EXCLUDE
excludes the sampling rate from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

VARS_TO_TRY (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**
specifies information about the number of variables to consider at each split during tree growth. For more information, see the VARS_TO_TRY= option in the PROC GRADBOOST statement.

You can specify the following additional suboptions:

LB=*number*
specifies the minimum number of variables to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=1.

UB=*number*
specifies the maximum number of variables to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

The default is the number of input variables.

VALUES=*value-list*
specifies a list of numbers of variables to consider during tuning, where *value-list* is a space-separated list of positive integers. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial number of variables for the tuner to use.

The default is the total number of input variables.

EXCLUDE
excludes the number of variables from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

CODE Statement

CODE < *options* > **;**

The CODE statement writes SAS DATA step code for computing predicted values of the fitted model to a file. To score new data, you can then include the file in a DATA step.

Table 16.4 summarizes the *options* available in the CODE statement.

Table 16.4 CODE Statement Options

Option	Description
COMMENT	Adds comments to the generated code
FILE=	Names the file in which to save the generated code
FORMATWIDTH=	Specifies the numeric format width for the regression coefficients
INDENTSIZE=	Specifies the number of spaces to indent the generated code
LABELID=	Specifies a number used to construct names and labels
LINESIZE=	Specifies the line size for the generated code
NOTRIM	Compares formatted values, including blank padding

For more information about the syntax of the CODE statement, see the section "CODE Statement" on page 20 in Chapter 2, "Shared Concepts."

CROSSVALIDATION Statement

CROSSVALIDATION < *options* > **;**

The CROSSVALIDATION statement performs *k*-fold cross validation to find the average estimated validation error. You cannot specify this statement if you specify the PARTITION statement.

You can specify the following *options*:

KFOLD=*number*
specifies the number of partition folds in the cross validation process, where *number* is between 2 and 20, inclusive. By default, KFOLD=5.

NOPARALLEL
specifies that *k*-fold cross validation not be run in parallel. By default, the process runs in parallel.

FREQ Statement

FREQ *variable* **;**

The *variable* in the FREQ statement identifies a numeric variable in the input data table that contains the frequency of occurrence of each observation. PROC GRADBOOST treats each observation as if it appears f times, where f is the value of the FREQ *variable* for the observation. If f is not an integer, it is truncated to an integer. If f is less than 1 or missing, the observation is not used in the analysis. When you omit the FREQ statement, each observation is assigned a frequency of 1. The frequency variable does not affect the partitions that are created by the PARTITION and CROSSVALIDATION statements.

ID Statement

ID *variables* **;**

The ID statement lists one or more *variables* that are to be copied from the input data table to the output data tables that are specified in the OUT= option in the OUTPUT statement and the RSTORE= option in the SAVESTATE statement.

INPUT Statement

INPUT *variables* < / *options* > **;**

The INPUT statement names input *variables* that share common *options*. You can specify the INPUT statement multiple times.

You can specify the following option after a slash:

LEVEL=INTERVAL | NOMINAL
specifies the level of measurement of the *variables*. You can specify the following values:

INTERVAL specifies that the level of measurement of the *variables* is interval.

NOMINAL specifies that the level of measurement of the *variables* is nominal.

By default, LEVEL=INTERVAL for numeric variables and LEVEL=NOMINAL for categorical variables.

MONOTONIC=DECREASING | INCREASING
requests that the gradient boosting model be built by requiring monotonic constraints on the model with respect to the variables. You can specify the following values:

DECREASING requests that decreasing monotonic constraints be placed on the model with respect to the variables.

INCREASING requests that increasing monotonic constraints be placed on the model with respect to the variables.

For more information, see the section "Imposing Monotonicity Constraints" on page 423.

By default, no monotonic constraints are placed on model with respect to the variables.

MITIGATEBIAS Statement

MITIGATEBIAS SENSITIVEVAR=*variable* **PREDICTEDVARS=(***variable-names***) PREDICTEDEVENTS=***"event-list"* < *options* > **;**

The MITIGATEBIAS statement enables PROC GRADBOOST to iteratively train a model while minimizing the bias metric that you specify. To use this statement, you must also specify the SAVESTATE statement and a categorical response (target) variable. You cannot also specify the AUTOTUNE statement or the CROSSVALIDATION statement.

You must specify the following options:

PREDICTEDEVENTS="*event-list***"**

PEVENTS="*event-list***"**

specifies the events that correspond to each variable in the PREDICTEDVARS= option. You must specify the PREDICTEDEVENTS= and PREDICTEDVARS= options with the same number of items in the same order. Items in the *event-list* must be separated by the delimiter that you specify in the DELIMETER= option.

PREDICTEDVARS=(*variable-names***)**

PVARS=(*variable-names***)**

specifies the names of the variables that contain the posterior probability for each level in model prediction that corresponds to the response (target) variable. You must specify the PREDICTEDVARS= and PREDICTEDEVENTS= options with the same number of items in the same order. Duplicate variables are not allowed.

SENSITIVEVAR=*variable*

specifies the sensitive variable to use in order to reduce the value of the bias measurement that you specify in the BIASMETRIC= option.

You can also specify the following *options*:

BIASMETRIC=*bias-metric*

specifies the type of bias measurement. You can specify the following *bias-metrics*:

DEMOGRAPHICPARITY specifies demographic parity as the bias measurement.

EQUALIZEDODDS specifies equalized odds as the bias measurement.

EQUALOPPORTUNITY specifies equal opportunity as the bias measurement.

PREDICTIVEPARITY specifies predictive parity as the bias measurement.

By default, BIASMETRIC=DEMOGRAPHICPARITY.

BOUND=*number*

specifies the bound value for the exponentiated gradient reduction algorithm.

By default, BOUND=100.

DELIMITER="*character***"**

DLM="*character***"**

specifies the delimiter to be used to separate events that you specify in the PREDICTEDEVENTS= option. A delimiter is used when event names contain embedded spaces (such as "Fuel Oil") or special characters (such as ";" or "*"). You must use quotation marks around *character*. You can specify only one delimiter; combinations of delimiters are not supported. The valid values of the DELIMITER= option are " " (space), ";" (semicolon), "*" (asterisk), "." (period), and "," (comma).

By default, DELIMITER=" " (space).

If you specify a delimiter other than " " (space), PROC GRADBOOST treats any spaces (including empty spaces) as valid characters. You are responsible for mapping events correctly in order to obtain correct results.

LEARNINGRATE=*number*

specifies the step size to use in updating the exponentiated gradient reduction algorithm.

By default, LEARNINGRATE=0.01.

LOGLEVEL=0 | 1 | 2

specifies the level of log information to print.

0 prints only warnings and errors.

1 prints some notes.

2 prints notes for each iteration.

By default, LOGLEVEL=1.

MAXITER=*number*

specifies the maximum number of iterations to run the exponentiated gradient reduction algorithm.

By default, MAXITER=10.

SEED=*number*

specifies the seed for the pseudorandom number generator. By default, the random number stream is based on the time of day from the computer's clock. A 0 or negative seed value also results in random number streams that are based on the time of day from the computer's clock. If you want a reproducible random number sequence between runs, specify a value greater than 0.

By default, SEED=0.

TARGETEVENT="*event***"**

specifies the formatted value of the response (target) variable that represents the event of interest.

By default, *event* is the first level that you specify in the PREDICTEDEVENTS= option.

TOLERANCE=*number*
specifies the parity constraint violation tolerance. If you set *number* to 0, PROC GRADBOOST trains for the maximum number of iterations, which you specify in the MAXITER= option.

By default, TOLERANCE=0.005.

TUNEBOUND
specifies that the bound value must be tuned.

OUTPUT Statement

OUTPUT OUT=*libref.data-table* < *option* > **;**

The OUTPUT statement creates an output data table that contains the results of running PROC GRADBOOST.

You must specify the following option:

OUT=*libref.data-table*
names the output data table for PROC GRADBOOST to use. You must specify this option before any other options. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to where the data table is to be stored. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 393.

data-table specifies the name of the output data table.

You can also specify the following *option*:

COPYVAR=*variable*
COPYVARS=(*variables***)**
lists one or more *variables* from the input data table to be transferred to the output data table.

ROLE< =*name* >
generates a numeric variable that indicates the role played by each observation in fitting the model. By default, the variable is named _ROLE_. You can add an optional *=name* to change the name of this generated variable. For each observation, the interpretation of this variable is shown in Table 16.5.

Table 16.5 Role Interpretation

Value	Observation Role
0	Not used
1	Training
2	Validation
3	Testing

If you do not partition the input data by using a PARTITION statement, then the value of the role

variable is 1 for all observations.

PARTITION Statement

PARTITION *partition-option* **;**

The PARTITION statement specifies how observations in the input data set are logically partitioned into disjoint subsets for model training, validation, and testing. For more information, see the section "Using Validation and Test Data" on page 23 in Chapter 2, "Shared Concepts." Either you can designate a variable in the input data table and a set of formatted values of that variable to determine the role of each observation, or you can specify proportions to use for randomly assigning observations to each role.

You must specify exactly one of the following *partition-options*:

FRACTION(< TEST=*fraction* **> < VALIDATE=***fraction* **> < SEED=***number* **>)**
randomly assigns specified proportions of the observations in the input data table to the roles. You specify the proportions for testing and validation by using the TEST= and VALIDATE= suboptions. If you specify both the TEST= and VALIDATE= suboptions, then the sum of the specified fractions must be less than 1 and the remaining fraction is the proportion of the observations assigned to the training role. The SEED= option specifies an integer that is used to start the pseudorandom number generator for random partitioning of data for training, testing, and validation. If you do not specify SEED=*number* or if *number* is less than or equal to 0, the seed is generated by reading the time of day from the computer's clock.

ROLE=*variable* **(< TEST='***value***' > < TRAIN='***value***' > < VALIDATE='***value***' >)**
ROLEVAR=*variable* **(< TEST='***value***' > < TRAIN='***value***' > < VALIDATE='***value***' >)**
names the *variable* in the input data table whose values are used to assign roles to each observation. This *variable* cannot also appear as an analysis variable in other statements or options. The TEST=, TRAIN=, and VALIDATE= suboptions specify the formatted values of this variable that are used to assign observation roles. If you do not specify the TRAIN= suboption, then all observations whose role is not determined by the TEST= or VALIDATE= suboption are assigned to the training role.

SAVESTATE Statement

SAVESTATE RSTORE=*libref.data-table* **;**

The SAVESTATE statement creates an analytic store for the model and saves it as a binary object in a data table. You can use the analytic store in the ASTORE procedure to score new data. For more information, see Chapter 4, "ASTORE Procedure."

You must specify the following option:

RSTORE=*libref.data-table*
specifies a data table in which to save the analytic store for the model. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 393.

TARGET Statement

TARGET *variable* < / **LEVEL=NOMINAL | INTERVAL** > **;**

The TARGET statement names the *variable* whose values PROC GRADBOOST predicts.

You can specify the following option:

LEVEL=NOMINAL | INTERVAL
specifies the level of measurement. You can specify the following values:

NOMINAL specifies that the level of measurement of the variables is nominal.

INTERVAL specifies that the level of measurement of the variables is interval.

By default, LEVEL=INTERVAL for numeric variables and LEVEL=NOMINAL for categorical variables.

TRANSFERLEARN Statement

TRANSFERLEARN *variable* < / *options* > **;**

The TRANSFERLEARN statement enables you to train the gradient boosting model by using auxiliary data that are added to your training data. For more information, see the section "Transfer Learning" on page 429.

The TRANSFERLEARN statement names the *variable* whose value indicates whether the observation belongs to the training data or auxiliary data. A value between 0 and 1 indicates training data, and a value greater than or equal to 1 indicates auxiliary data. A value of 0 is given zero weight in the analysis. Observations that have missing values or negative values are ignored.

You can specify the following *options*:

BURN=*number*
specifies the *number* of trees to create before down-weighting any observation in the auxiliary data.

By default, BURN=0.

SHRINKAGE=*number*
specifies the *number* to apply as the weighting factor for down-weighting auxiliary data, where *number* must be between 0 and 1, exclusive.

By default, SHRINKAGE=0.9.

TRIMMING=*number*
specifies the *number* to use as a fraction of the distribution of gradients on the training data beyond which auxiliary observations are down-weighted, where *number* must be greater than 0 and less than or equal to 1/2.

By default, TRIMMING=0.01.

VIICODE Statement

VIICODE < *options* > **;**

The VIICODE statement writes SAS DATA step code to a file or to a catalog entry. The SAS DATA step code creates new variables on the basis of the detected variable interactions.

You can specify the following *options* in addition to all options in the CODE statement. For more information about the syntax of the CODE statement, see the section "CODE Statement" on page 20 in Chapter 2, "Shared Concepts."

ADD
requests that the newly created variables be of the form $V + W$.

LIMIT=*number*
specifies the maximum *number* of new variables to create. By default, LIMIT=200.

MISS
requests that the generated code handle missing values.

MULTIPLY
requests that the newly created variables be of the form $V \times W$.

SUBTRACT
requests that the newly created variables be of the form $V - W$.

THRESHOLD=*number*
requests that interactions with an importance less than *number* times the maximum interaction importance be ignored, where *number* must be between 0 and 1. By default, THRESHOLD=0.0001.

If the VII= option is not specified in the PROC GRADBOOST statement, then the VIICODE statement is ignored. For more information about variable interaction importance, see the section "Interaction Importance" on page 428.

WEIGHT Statement

WEIGHT *variable* **;**

The *variable* in the WEIGHT statement is used as a weight to perform a weighted analysis of the data. Observations that have nonpositive or missing weights are not included in the analysis. The values of the WEIGHT variable can be nonintegral and are not truncated. When you omit the WEIGHT statement, all observations that are used in the analysis are assigned a weight of 1.

Details: GRADBOOST Procedure

Subsampling the Data

A decision tree in a gradient boosting model trains on new training data that are derived from the original training data presented to the GRADBOOST procedure. Using different data to train different trees during the boosting process reduces the correlation of the predictions of the trees, which in turn should improve the predictions of the boosting model.

The GRADBOOST procedure samples the original data without replacement to create the training data for an individual tree. The GRADBOOST procedure performs the action of sampling multiple times throughout a run, and each set of training data created is referred to as a subsample. The subsampling process takes into account the frequency variable that you specify in the FREQ statement.

The SAMPLINGRATE= option in the PROC GRADBOOST statement specifies the fraction of observations to sample without replacement.

Training a Decision Tree

The GRADBOOST procedure trains a decision tree by splitting the subsampled data, then splitting each resulting segment, and so on recursively until some constraint is met.

Splitting involves the performing following tasks in order:

1. Select candidate inputs.
2. Compute the association of each input with the target.
3. Search for the best split that uses the most highly associated inputs.

PROC GRADBOOST randomly selects VARS_TO_TRY=*m* candidate input variables independently in every node. A split search is performed on all *m* variables, and the best rule is kept to split the node.

The split search seeks to maximize the reduction in the gain for a nominal target and the reduction in variance of an interval target.

Splitting Nominal Values

Methods

PROC GRADBOOST evaluates a nominal variable in a node in one of three ways:

- enumerate: evaluate all possible splits
- sort: evaluate splits that preserve a sort order of the values
- reject: reject the variable without considering any splits

The number of categories K that are present in a node of an input variable X determines the choice. A different choice might be used for a different variable in the same node, or for the same variable in a different node because K might be different in different nodes. Let s denote the value of the SORT= suboption of the NOMSEARCH option, and let c denote the value of the MAXCATEGORIES= suboption. The following rules determine which method PROC GRADBOOST uses:

- Enumerate when $K \leq \min(15, s, c)$.
- Sort when $\min(s, 16) \leq K < c$.
- Reject the variable when $c \leq K$.

The enumerate choice examines all possible splits. The number of splits grows exponentially with the number of categories K.

The sort choice first sorts the categories and then finds the best split that preserves the sort. The sort is on z_k, which is defined as

$$z_k = \frac{\Sigma_{i \text{ in } k}\, g_i + r\bar{g}}{n_k + r}$$

where

- k is a category
- Σ sums over observations in category k
- g_i is the gradient of observation i
- r is the NOMSEARCH(SHRINKAGE=) suboption value
- $\bar{g}$ is the average gradient across all categories
- n_k is the number of observations in category k

Risks of Many Categories

Algorithms that search for a splitting rule preferentially select nominal inputs that contain lots of categories because of the large number of ways the categories can be combined to match noise in the training data. Irrelevant nominal inputs commonly supplant predictive inputs, creating spurious splits that can degrade accuracy on new data.

Memory depletion is another risk, especially in the output model table. The branch assignment of every category is stored. The OUTMODEL= table contains a row for every node in the model and a column for every category assigned to that node. The table contains enough columns to accommodate the node with the most category assignments, even if only one node requires it. A single nominal input that contains lots of categories can create a model so large as to exhaust memory. The MAXCATEGORIES=*number* suboption prevents splitting a node with a nominal input that has more than *number* categories in the node. An input with more than *number* categories in one node and fewer than *number* categories in a second node is eligible for splitting the second node but not the first.

Scoring can take noticeably longer when a splitting rules include many categories.

Imposing Monotonicity Constraints

Sometimes it makes sense to impose a monotonic relationship between an input and the target. For example, someone might believe that the more stuff a person buys, the happier that person is. Assuming this, it would make sense to constrain splits on the amount of stuff a person buys to splits that increase a measure of happiness with increasing amounts of stuff bought. Although this approach does not guarantee that the relationship is exhibited everywhere in the final model, it typically is.

The MONOTONIC= option in the INPUT statement specifies a monotonicity constraint for interval inputs. The constraint is ignored for nominal inputs.

Monotonicity constraints can miss productive splits and sometimes produce no splits at all, thereby degrading accuracy. Increasing the value of the NUMBIN= option can mitigate the effect.

Boosting

A description of gradient boosting for decision trees can be found in Hastie, Tibshirani, and Friedman (2001) and Friedman (2001).

The GRADBOOST procedure creates a series of decision trees that together form a single predictive model. The trees are built sequentially. Each tree uses a subsample of the data and is built as described in the section "Training a Decision Tree" on page 421. The sequence of trees and how each tree affects a subsequent tree are discussed in Hastie, Tibshirani, and Friedman (2001), Friedman (2001), and Chen and Guestrin (2016).

A tree ensemble that seeks to predict an observed target, y_i, by using the input, x_i, can be written as

$$\hat{y}_i = F(x_i) = \Sigma_{k=1}^{K} f_k(x_i)$$

where f_k is the functional representation of the kth tree.

To learn the set of functions (the trees) that are used in the gradient boosting model, PROC GRADBOOST seeks to minimize the regularized objective function

$$L(F) = \Sigma_i l(\hat{y}_i, y_i) + \Sigma_k \Omega(f_k)$$

where $l(\hat{y}_i, y_i)$ is the loss function that measures the difference between the observed target, y_i, and the predicted value of the target, $\hat{y}_i$. The GRADBOOST procedure uses one of three loss functions as follows:

- For continuous targets, the loss function is the L2 loss.
- For binary nominal targets, the loss function is the logistic loss.
- For nonbinary nominal targets, the loss function is the multiclass logistic loss.

The term, $\Omega(f_k)$, in the objective function is the regularization term, which acts as a penalty on the model complexity. You can specify the regularization in PROC GRADBOOST in the RIDGE= and LASSO= options.

Early Stopping

In some cases, gradient boosting models can be overtrained and thus perform poorly on validation or test data. One method to combat overtraining in gradient boosting is early stopping. An additional advantage to early stopping is reduced training time in cases where the stopping criterion is met well before the specified maximum number of iterations occurs.

Early stopping takes advantage of the fact that boosting is an iterative process. This means that the prediction error can be measured on a validation data set at each iteration of the process. When the prediction error on the validation data meets specified criterion, the gradient boosting process stops, yielding a model that is less overtrained than if the boosting process were allowed to continue until completion.

For an interval target, the prediction error is average square error. For a nominal target, the METRIC= suboption of the EARLYSTOP option in the PROC GRADBOOST statement specifies the prediction error as misclassification rate or log loss. The error is measured on validation data that are specified in the PARTITION statement. Without validation data, early stopping is usually ineffective.

The GRADBOOST procedure uses two approaches to early stopping:

- stagnation: stops when consecutive decreases of the error satisfy a criterion
- threshold: stops when the error exceeds a threshold

The stagnation approach is used when the value of the STAGNATION=*number* suboption is greater than 0. If MINIMUM=NO and the value of the TOLERANCE=τ suboption is 0, then training stops after *number* iterations of nondecreasing error. If MINIMUM=YES, then training stops when the smallest error is not improved upon in *number* subsequent iterations. When $\tau > 0$, then if the relative change in error from iteration *i–1* to iteration *i* is less than τ,

$$\left|\frac{\mathrm{Err}_{i-1} - \mathrm{Err}_i}{\mathrm{Err}_{i-1}}\right| < \tau$$

for *number* iterations, the boosting process stops and returns the model from the last iteration in which the relative error difference was greater than or equal to τ.

The threshold approach is used when the value of the THRESHOLD=θ suboption is greater than 0 and the value of the THRESHOLDITER=T suboption is greater than 0. Training proceeds for T iterations and then stops when the error is greater than θ. The model that has the smallest error after at least T iterations is chosen.

Measuring Prediction Error

The GRADBOOST procedure computes the average square error measure of prediction error. For a nominal target, the procedure also computes the misclassification rate and the log loss.

The average square error for an interval target, the average square error for a nominal target, the misclassification rate, and the log loss are defined, respectively, as

$$\mathrm{ASE}_{\mathrm{int}} = \sum_{i=1}^{N} \frac{(y_i - \hat{y}_i)^2}{N}$$

$$\mathrm{ASE}_{\mathrm{cat}} = \sum_{i=1}^{N} \sum_{j=1}^{J} \frac{(\delta_{ij} - \hat{p}_{ij})^2}{JN}$$

$$\mathrm{MISC} = \sum_{i=1}^{N} \frac{1(y_i \neq \hat{y}_i)}{N}$$

$$\mathrm{LogLoss} = -\sum_{i=1}^{N} \sum_{j=1}^{J} \frac{\delta_{ij} \log(\ddot{p}_{ij})}{N}$$

where $\hat{y}_i$ is the target prediction of observation i, δ_{ij} equals 1 if the nominal target value j occurs in observation i or 0 if it does not, $\hat{p}_{ij}$ is the predicted probability of nominal target value j for observation i, N is the number of observations, J is the number of nominal target values (classes), and $\ddot{p}_{ij}$ is $\hat{p}_{ij}$ truncated away from 0 and 1,

$$\ddot{p}_{ij} = \max(\min(\hat{p}_{ij}, 1 - 10^{-10}), 10^{-10})$$

The definitions are valid whether $\hat{y}_i$ is the usual model prediction or the out-of-bag prediction. The $\mathrm{ASE}_{\mathrm{int}}$ that is based on the usual model predictions of the original training data is usually optimistic and smaller than what its value will be for future data.

Handling Missing Values

Strategies

Tree-based models use observations that have missing input values. The GRADBOOST procedure offers the following strategies for handling missing values:

- The simple strategy is to regard a missing value as a special nonmissing value. For a nominal input, a missing value simply constitutes a new categorical value. For an input whose values are ordered, each missing value constitutes a special value that is assigned a place in the ordering that yields the best split. That place is usually different in different nodes of the tree.

  This strategy is beneficial when missing values are predictive of certain target values. For example, people who have large incomes might be more reluctant to disclose their income than people who

have ordinary incomes. If income were predictive of a target, then a missing income value would be predictive of the target and the missing values would be regarded as a special large-income value. The strategy seems harmless when the distribution of missing values is uncorrelated with the target because no choice of branch for the missing values would help predict the target.

A linear regression could use the same strategy by adding binary indicator variables to designate whether a value is missing. Alternatively, and much more commonly, a linear regression could simply remove observations in which any input is missing. Let p denote the probability that a variable value is missing, and let v denote the number of input variables. The probability that an observation has one or more missing values is $1 - (1 - p)^v$ (assuming missingness is independent and identically distributed among the inputs). If $p = 0.1$ and $v = 10$, then 65% of the observations would have missing values and would be removed from linear regression.

- The alternative strategy for decision trees is to exclude from the search algorithm any observations that have a missing value in the single input variable that defines the splitting rule. If $p = 0.1$ and $v = 10$, then only 10% instead of 65% of the observations are excluded. Although this compares favorably with common linear regression, using observations that have missing values might still be better.

Specifics

If the value of a target variable is missing, the observation is excluded from training and from evaluating the model. If the value of an input variable is missing, PROC GRADBOOST uses the missing value as a legitimate value either by default or when ASSIGNMISSING=USEINSEARCH and the number of observations in which the splitting variable has missing values is at least as large as the value of the MINUSEINSEARCH= option. When ASSIGNMISSING=USEINSEARCH and the number of observations in which the splitting value has missing values is less than the value of the MINUSEINSEARCH= option, the splitting rule assigns observations that have missing values to the largest branch.

Measuring Variable Importance

The importance of a variable is the contribution it makes to the success of the model. For a predictive model, success means good prediction. Often the prediction relies mainly on a few variables. A good measure of importance reveals those variables. The better the prediction, the more closely the model represents reality, and the more plausible it is that the important variables represent the true cause of prediction. Some people prefer a simple model so that they can understand it. However, a simple model usually relinquishes details of reality. Sometimes it is better to first find a good model and then ask which variables are important than to first ask which model is good for variable importance and then train that model.

Van der Laan (2006) asks whether a predictive model is appropriate at all. He believes that if variable importance is your goal, then you should predict importance directly instead of fitting a model. If your goal is to select suspicious genes for further study in a laboratory or to find variables in an industrial process that might influence the quality of the product, then his argument is persuasive. However, the purpose of many predictive models is to make predictions. In these cases, gaining insight into causes can be useful.

Variable importance is also useful for selecting variables for a subsequent model. The comparative importance of the selected variables does not matter. Researchers often seek speed and simplicity from the first model and seek accuracy from the subsequent model.

The GRADBOOST procedure implements two methods for computing variable importance, which are described in the following subsections. By default, the variable importance is calculated by using the change in the residual sum of square errors. You can request that the GRADBOOST procedure also calculate the variable importance by random branch assignment (RBA) by specifying the RBAIMP option.

Residual Sum of Squares Importance Method

The residual sum of squares (RSS) for regression trees is defined as

$$\mathrm{RSS} = \sum_{\lambda} \sum_{i \in \lambda} \left(y_i - \hat{y}_{\lambda}^{T} \right)^2$$

where

- i is an observation on leaf λ
- y_i is the predicted value of the response variable of observation i
- $\hat{y}_{\lambda}^{T}$ is the actual value of the response variable on leaf λ

The residual sum of squares (RSS) for classification trees is defined as

$$\mathrm{RSS} = \sum_{\lambda} \sum_{\Phi} N_{\Phi}^{\lambda} \left[\sum_{\tau \neq \Phi} \left(P_{\tau}^{\lambda} \right)^2 + \left(1 - P_{\Phi}^{\lambda} \right)^2 \right]$$

where

- Φ is the actual response level
- N_{Φ}^{λ} is the number of observations on leaf λ that have response level Φ
- P_{τ}^{λ} is the posterior probability for the response level τ on leaf λ
- P_{Φ}^{λ} is the posterior probability for the actual response level Φ on leaf λ

For a single tree in the boosting model, the RSS-based metric measures variable importance based on the change in RSS when a split is found at a node. The change for variable v is

$$\Delta_d = \mathrm{RSS}_d - \sum_{i} \mathrm{RSS}_i^d$$

where

- d denotes the node
- i denotes the index of a child that this node includes
- RSS_d is the RSS if the node is treated as a leaf
- RSS_i^d is the RSS of the node after it has been split

If the change in RSS is negative (which is possible when you use the validation set), then the change is set to 0.

The RSS-based importance for a single variable, v, in a single tree is then defined as

$$\sum_{d=1}^{D} \Delta_d^v$$

where D is the total number of nodes in which v was used as the splitting variable.

The RSS variable importance for the boosting model is the average of the RSS variable importance across all trees in the boosting model.

Interaction Importance

In some cases, interactions of variables are of more interest than a single variable. When you specify the VII= option in the PROC GRADBOOST statement, the procedure computes variable interaction importance as follows.

The two-way interaction importance for variables v and w is

$$\sum_{d,e} \frac{\Delta_d^v + \Delta_e^w}{\Delta_d^v}$$

where

- d and e denote nodes, where node d is the parent of node e
- v and w are variables, where v is the splitting variable for node d and w is the splitting variable for node e
- This summation is across all parent-child node combinations with respect to the variables v and w.

The three-way interaction importance for variables v, w, and x is

$$\sum_{d,e,f} \frac{\Delta_d^v + \Delta_e^w + \Delta_f^x}{\Delta_d^v + \Delta_e^w}$$

where

- d, e, and f denote nodes, where node d is the parent of node e, which is the parent of node f
- v, w, and x are variables, where v is the splitting variable for node d, w is the splitting variable for node e, and x is the splitting variable for node f
- This summation is across all parent-child-grandchild node combinations with respect to the variables v, w, and x.

When comparing variable importance values, you should compare a two-way interaction only with another two-way interaction, or a three-way interaction only with another three-way interaction. Comparing a two-way interaction to a three-way interaction is not meaningful, because the values for the importance are not on the same scale.

In addition to calculating the variable interaction importance, you can also generate SAS DATA step code to create interaction variables and save the code to a file by using the VIICODE statement. When you run the generated SAS DATA step code on the original data, or new data, interaction variables are created.

Created interaction variables are of the form $W + V$, $W - V$, or $W \times V$, where the values used in the calculations come from the variables whose interactions are determined to be important. The values used are as follows:

- For continuous variables, the raw value of the variable is used.
- For categorical variables, the proportion of observations in the training set that is assigned to a specific branch is used.

Random Branch Assignment Importance Method

The random branch assignment (RBA) method computes the importance of an input variable v by comparing how well the data fit the predictions before and after they are modified. To modify the predictions, the GRADBOOST procedure replaces all splitting rules that use variable v by a rule that randomly assigns an observation to a branch. The probability of assigning an observation to a branch is proportional to the number of observations that are assigned to the branch in the current data. The current data are the training data when RBA is computed during training. Otherwise, the current data are those being scored on an existing model.

The RBA importance can be expressed mathematically as

$$I_{\mathrm{RBA}}(v) \propto \sum_{i=1}^{n} \mathrm{Loss}(y_i, \ddot{y}_i) - \sum_{i=1}^{n} \mathrm{Loss}(y_i, \hat{y}_i)$$

where $\ddot{y}_i$ is the modified prediction for observation i and $\hat{y}_i$ is the standard prediction.

For an interval target, PROC GRADBOOST computes the RBA importance of squared error loss. For a nominal target, PROC GRADBOOST uses the misclassification rate as the loss function.

Transfer Learning

Some applications of statistical and machine learning have only a scarce amount of training data available. One method of building models in situations that have small training sets is to supplement the scarce training data with more abundant auxiliary data. The auxiliary data should have the same target and input variables, but can come from a different context than the training data. For more discussion, see Dai et al. (2007).

In the GRADBOOST procedure, you can use the auxiliary data to increase the number of observations for training the model. In order to prevent the model from being overly biased toward the auxiliary data, the GRADBOOST procedure down-weights auxiliary observations that are dissimilar from the training observations. The factor that is applied to down-weight observations is controlled by the SHRINKAGE= option in the TRANSFERLEARN statement. To determine whether an observation is dissimilar, the GRADBOOST procedure computes a gradient for each observation. An observation in the auxiliary data that is dissimilar to the training data will have a gradient that is in either a low quantile or a high quantile of the gradients of the training data. The TRIMMING= option in the TRANSFERLEARN statement controls the fraction of training gradients beyond which auxiliary observations are down-weighted.

The GRADBOOST procedure begins down-weighting auxiliary observations only after a number of trees has been grown, as specified in the BURN= option in the TRANSFERLEARN statement.

To use transfer learning in the GRADBOOST procedure, you must concatenate your training data and your auxiliary data, and specify this combined data table in the DATA= option in the PROC GRADBOOST statement. To distinguish the two types of data, you must create an indicator variable as follows:

- When the value of the indicator variable is greater than 0 and less than 1, PROC GRADBOOST considers the observation to belong to the training data.
- When the value of the indicator variable is greater than or equal to 1, PROC GRADBOOST considers the observation to belong to the auxiliary data.
- When the value of the indicator variable is missing or is less than or equal to 0, PROC GRADBOOST gives the observation zero weight during model training.

Hyperparameter Tuning

The options that control the training algorithm are often called hyperparameters to distinguish them from the parameters of the trained model. The main parameters in tree-based models are the splitting rules. Optimal hyperparameter settings depend on the data and are never known for certain. Nevertheless, here are some recommendations:

BINMETHOD=QUANTILE
Using quantile bins instead of bins of equal width often results in a better fit, especially when some inputs have outliers.

NUMBIN=*number*
Using more bins rarely degrades and sometimes improves accuracy. However, training takes longer with more bins, often much longer with many more bins.

MAXDEPTH=*number*
A modeling algorithm that fits the training data well generalizes well if most of the data are signal, but it generalizes poorly if most of the data are noise. In the latter case for boosting, it is better to make small trees, make each tree contribute less, and make more trees. Accordingly, reduce the values of the MAXDEPTH= and LEARNINGRATE= options and increase the value of the NTREES= option when the data have a weak signal.

RIDGE=1
This option helps prevent overfitting splits.

For more information about hyperparameter tuning, see the section "Hyperparameter Tuning" on page 24 in Chapter 2, "Shared Concepts."

You can use the AUTOTUNE statement to tune the following options in the PROC GRADBOOST statement:

- LASSO= option for the L1 regularization parameter
- RIDGE= option for the L2 regularization parameter

- LEARNINGRATE= option for the learning rate parameter
- MAXDEPTH= option for the maximum depth of trees
- NTREES= option for the number of trees
- NUMBIN= option for the number of bins
- SAMPLINGRATE= option for the proportion of the training data to sample
- VARS_TO_TRY= option for the number of variables to randomly select at each node split for each tree

k-fold Cross Validation

The CROSSVALIDATION statement performs *k*-fold cross validation to assess the accuracy of a model. During cross validation, all data are divided into *k* subsets (folds), where *k* is the value of the KFOLD= option. For each fold, a new model is trained on the (*k*–1) folds and then validated using the selected (holdout) fold. The assessment metrics are then averaged across all the holdout folds. The CROSSVALIDATION statement creates a table that has *k*+1 rows. The first *k* rows contain the assessment metrics for each holdout fold, and the last row contains the average across all the holdout folds. Which metrics the table includes depends on whether the target is nominal or interval. For nominal targets, the table includes the values of square error, mean consequential error, and multiclass log loss. For interval targets, the table includes the values of square error, absolute error, and square logarithmic error.

TreeSHAP Values

TreeSHAP values are used to explain the contribution of each variable to the prediction of a single observation. The TreeSHAP algorithm is used by the Shapley value approximation method for tree-based models that was introduced by Lundberg et al. (2020). Compared to the algorithms that other model-agnostic Shapley value approximation methods use, the TreeSHAP algorithm is faster because it uses the specific structures of tree-based models.

When you use PROC GRADBOOST to train a model, you can save the trained model in an analytic store by using the SAVESTATE statement. You can then use the ASTORE procedure to calculate the TreeSHAP values when you score new data by using this saved analytic store. To enable the calculation of TreeSHAP values in PROC ASTORE, you must set the TREESHAP option value to 1 in the SETOPTION statement. See Chapter 4, "ASTORE Procedure," for more information about the SETOPTION statement.

NOTE: Currently, TreeSHAP values are available only for models that have a maximum of two splits per node. Additionally, the TreeSHAP algorithm is currently not available for models if you set the DISTRIBUTION= option to POISSON or TWEEDIE.

If the target variable is nominal, then the TreeSHAP algorithm calculates the Shapley values for the (scaled) log odds of the predicted probability for each target level. The (scaled) log odds are also saved in the output data set if you set the TREESHAP option value to 1 in the SETOPTION statement. You can use the DESCRIBE statement to see the variable and target that each generated TreeSHAP variable explains.

Bias Mitigation

With the widespread use of AI and machine learning models in decision-making processes, there is concern about the social impacts of these models. Bias can be embedded in both the training data and the trained model, either of which can lead to unfair or biased predictions for certain groups or populations. To reduce bias in the data and models, researchers have developed three main categories of bias mitigation methods:

- Preprocessing methods, which transform the data prior to model training. These methods try to eliminate the correlations between the input features and the sensitive feature while retaining as much information from the input data as possible.
- In-process methods, which consider fairness constraints during the model training process. These methods actively adjust the model parameters during training to create a model that produces fair predictions and classifications.
- Postprocessing methods, which do not directly alter the input data or the model training process. These methods adjust the predictive outputs from the learned models to compensate for the bias in the model.

You can reduce bias in the trained model by specifying the MITIGATEBIAS statement. This statement uses the exponentiated gradient reduction (EGR) algorithm, which is an in-process method of mitigating bias.

Exponentiated Gradient Reduction (EGR) Algorithm

The exponentiated gradient reduction (EGR) algorithm was introduced at ICML 2018 (Agarwal et al. 2018). As an in-process algorithm, it considers fairness constraints during the model training process. The EGR algorithm has three main advantages. First, it is model-agnostic, which means that you can apply it to a wide variety of machine learning models. The EGR algorithm works like a wrapper that is built around existing machine learning models. The wrapper iteratively reweights and possibly relabels the data, and then it trains a new classifier model on the basis of reweighted data in each iteration. By doing so, it reduces the complex optimization problem of training a machine learning model with fairness constraints to a series of standard model training problems without the fairness constraints. Second, the EGR algorithm supports various fairness measurements, such as demographic parity, equal opportunity, and equalized odds. Finally, the algorithm has strong theoretical support in terms of its optimization trade-off between fairness and accuracy. In theory, the EGR algorithm returns solutions that satisfy all the fairness constraints and also achieve the optimal error rate of the best fair solution.

Training a fair machine learning model is in general a nonlinear programming problem. In nonlinear optimization tasks, it is not uncommon to get suboptimal solutions because the constraints are infeasible or because the algorithm is stuck in a local minimum. As with many other machine learning algorithms, the effectiveness of the EGR algorithm depends on many factors, such as the quality of the input data, the selection of fairness constraints, the quality of the machine learning model that it wraps around, and the values of the hyperparameters.

PROC GRADBOOST supports bias mitigation for multilevel nominal target variables and multilevel nominal sensitive variables. Note that when you include a large number of levels in the sensitive variable, the EGR algorithm searches more complex optimization scenarios; this can result in suboptimal solutions. Therefore, the solutions from this algorithm might not always be desirable when the number of levels in the sensitive variable is high. You should review and validate your sensitive variable, training models, and fairness constraints before applying the EGR algorithm.

Fairness Constraints

PROC GRADBOOST supports the following fairness measurements: demographic parity, predictive parity, equal opportunity, and equalized odds. The goal of the EGR algorithm is to bring the fairness metrics of groups as close together as possible, as well as to convert fair machine learning to a sequence of standard cost-sensitive machine learning problems whose solutions yield a randomized classifier with the lowest error rate, subject to the desired constraints. As a result, the EGR algorithm iteratively trains the machine learning models by using different weights to reduce the fairness disparities without significantly increasing the loss of the final machine learning model. For example, when there are two protected groups in the sensitive variable, the fairness constraints that the EGR algorithm requires for demographic parity are as follows:

$$\mu_1^t - \mu_0^t \leq c$$
$$\mu_1^t - \mu_0^t \geq -c$$
$$\mu_2^t - \mu_0^t \leq c$$
$$\mu_2^t - \mu_0^t \geq -c$$

where μ_1^t is the demographic parity for the first protected group during iteration t; μ_2^t is the demographic parity for the second protected group during iteration t; μ_0^t is the demographic parity for the entire population during iteration t; and c is the predefined violation tolerance threshold.

Exponentiated Gradient Reduction Algorithm Workflow

Two key hyperparameters in the EGR algorithm are specified by the BOUND= option (denoted by B) and the LEARNINGRATE= option (denoted by lr); both are available in the MITIGATEBIAS statement. To derive the fair machine learning algorithm, the fairness constraints (see the section "Fairness Constraints" on page 433) are added to the loss function of the cost-sensitive machine learning model by introducing the Lagrange multipliers $\lambda_{j,t}$, where j is the index in the constraint set. The multipliers $\lambda_{j,t}$ are updated on the basis of the degree of constraint violations $\gamma_{j,t}$, B, and lr. Then $\lambda_{j,t}$ is used to derive the new weights for each observation in the input data set for the next round of training. Note that the values of B and lr play important roles in the performance of the EGR algorithm, based on experiments. The next section discusses how to come up with appropriate values for B and lr.

Bound Value Recommendation

Because the EGR algorithm is designed to strike a balance between model accuracy and fairness constraint violations, it does not perform well unless you specify appropriate values for the BOUND= and LEARNINGRATE= options in the MITIGATEBIAS statement. The BOUND= option is important for the sample reweighting and relabeling process. It controls how heavily the EGR algorithm penalizes fairness constraint violations, and therefore it determines the trade-off between the fairness constraints and the model accuracy. When the bound is too small, there is not enough momentum to drive the model close enough to the saddle points in the solution sets. In this case, the bias metric values of the different subgroups do not approach each other.

The LEARNINGRATE= option controls how quickly the algorithm reduces fairness constraint violations. If the learning rate is too small, the algorithm converges very slowly, resulting in a long training process; if the learning rate is too large, the algorithm becomes unstable and might not converge. In general, the learning rate and the bound value are coupled, and changing one of these values changes the optimal value of the other.

The MITIGATEBIAS statement offers a built-in bound value recommendation function to automatically search for a good candidate bound value when you request it. To eliminate the coupling effect of the bound

and the learning rate, PROC GRADBOOST fixes the initial learning rate at 0.01 during the tuning process. The procedure runs the bound value recommendation function if you specify the TUNEBOUND option. This function finds the best bound value through multiple iterations by analyzing the convergence patterns of the fairness metrics. PROC GRADBOOST first runs the EGR algorithm with some initial bound value for a few iterations, while monitoring the convergence pattern for signs of stagnation or overshooting to determine whether the bound value is too small or too large. It then adjusts the bound value until it finds an appropriate value or reaches the maximum number of iterations that are allowed. After the recommendation process, PROC GRADBOOST applies the recommended bound value to the EGR algorithm to train a final model. Note that because of the nonlinear nature of the optimization process, the bound value recommendation is also subject to the problem of local minima. This recommendation should be used as a general guideline for setting the bound value; you might find a better solution by further fine-tuning.

Displayed Output

The GRADBOOST procedure displays the parameters that are used to train the model, the fit statistics of the trained model, and other information. The output is organized into various tables, which are discussed here in order of their appearance.

Model Information

The “Model Information” table contains the settings of the training parameters. This table also contains some basic information about the trees in the resulting boosting model. This table is produced by default.

Number of Observations

The “Number of Observations” table contains the number of observations that are read from the input data table and the number of observations that are used in the analysis. When you specify the PARTITION statement, the table also indicates the number of observations that are used in each partition. This table is produced by default.

Variable Importance

The “Variable Importance” table displays variable importance based on residual sum of square errors, which is explained in the section “Measuring Variable Importance” on page 426. This table is produced by default.

RBA Variable Importance

The “RBA Variable Importance” table displays variable importance based on the random branch assignment (RBA) method, which is explained in the section “Random Branch Assignment Importance Method” on page 429. This table is produced by the RBAIMP option in the PROC GRADBOOST statement.

Fit Statistics

The "Fit Statistics" table contains statistics that measure the model's goodness of fit. The fit of the model to the data improves as the number of trees in the boosting model increases. Successive rows in the table contain fit statistics for a boosting model that has more trees. Fit statistics are described in the section "Measuring Prediction Error" on page 425. This table is produced by default.

Tuner Information

The "Tuner Information" table displays the setup values that the tuner uses. This table is produced by the AUTOTUNE statement.

Tuner Summary

The "Tuner Summary" table displays statistics about the tuning process. This table is produced by the AUTOTUNE statement.

Tuner Timing

The "Tuner Timing" table displays the total time spent on different tasks during tuning. This table is produced by the AUTOTUNE statement.

Best Configuration

The "Best Configuration" table displays the hyperparameters and objective function values for the best configuration. This table is produced by the AUTOTUNE statement.

Tuner Results

The "Tuner Results" table displays the values of the hyperparameters, the objective function for the default configuration (Iteration 0), and up to 10 of the best configurations that are found. This table is produced by the AUTOTUNE statement.

Evaluation History

The "Evaluation History" tables displays the values of the hyperparameters and the objective function for all configurations. This table is produced by the AUTOTUNE statement, either by default or when EVALHISTORY=ALL.

Cross-Validation Fit Statistics

The "Cross-Validation Fit Statistics" table contains the per-fold and average assessment metrics of k-fold cross validation.

OutputCasTables Table

The OutputCasTables table is a special table that has information about each table that is created. The information for each table consists of the table name, the library in which the table resides, and the number of columns and rows in the table.

Predicted Probability Names

The "Predicted Probability Names" table indicates the names of the created variables in either the score code or the data table that is specified in the OUT= option in the OUTPUT statement. This table is produced when you specify the PRINTTARGET option in the PROC GRADBOOST statement.

Constraint History

The "Constraint History" table lists the values of the various constraints at each iteration of the EGR algorithm. This table is produced by the MITIGATEBIAS statement.

Group Metrics History

The "Group Metrics History" table displays the metrics for accuracy and fairness for each subpopulation over each iteration of the EGR algorithm. This table is produced by the MITIGATEBIAS statement.

Iteration History

The "Iteration History" table displays the values of accuracy and bias metric at each iteration of the EGR algorithm. This table is produced by the MITIGATEBIAS statement.

Mitigation Information

The "Mitigation Information" table displays information about the options that you use for bias mitigation. This table is produced by the MITIGATEBIAS statement.

Timing

The "Timing" table displays the breakdown of how long bias mitigation spent on the various tasks of training models. This table is produced by the MITIGATEBIAS statement.

Weight History

The "Weight History" table displays the weights that are assigned by the EGR algorithm for each level of the response (target) variable and for each subpopulation. This table is produced by the MITIGATEBIAS statement.

ODS Table Names

Each table that the GRADBOOST procedure creates has a name associated with it. You must use this name to refer to the table when you use ODS statements. The name of each table and a short description of the contents are listed in Table 16.6.

Table 16.6 ODS Tables Produced by PROC GRADBOOST

Table Name	Description	Statement	Option
BestConfiguration	Hyperparameters and objective function values for the best configuration	AUTOTUNE	Default
ConstraintHistory	Values of the various constraints at each iteration of the EGR algorithm	MITIGATEBIAS	Default
CrossValidateMLFitStat	Per-fold and average assessment metrics of *k*-fold cross validation	CROSSVALIDATION	Default
EvaluationHistory	Values of the hyperparameters and the objective function for all configurations	AUTOTUNE	Default / EVALHISTORY=ALL
FitStatistics	Fit statistics from the model	PROC GRADBOOST	Default
GroupMetricsHistory	Metrics for accuracy and fairness for each subpopulation over each iteration of the EGR algorithm	MITIGATEBIAS	Default
IterationHistory	Values of the model accuracy and bias metric over each iteration of the EGR algorithm	MITIGATEBIAS	Default
MitigationInformation	Information about the options that are used for bias mitigation	MITIGATEBIAS	Default
ModelInfo	Model Information	PROC GRADBOOST	Default
Nobs	Number of Observations	PROC GRADBOOST	Default
OutputCASTables	See the section "OutputCasTables Table" on page 436	PROC GRADBOOST / OUTPUT	OUTMODEL=
PredName	Predicted name information for interval targets	PROC GRADBOOST	PRINTTARGET
PredProbName	Predicted probability name information for nominal targets	PROC GRADBOOST	PRINTTARGET

Table 16.6 *continued*

Table Name	Description	Statement	Option
PredIntoName	Predicted name information for nominal targets	PROC GRADBOOST	PRINTTARGET
RBAImportance	Random branch assignment variable importance	PROC GRADBOOST	RBAIMP
Timing	Breakdown of how long bias mitigation spent on the various tasks of training models	MITIGATEBIAS	Default
TunerInfo	Setup values used by the tuner	AUTOTUNE	Default
TunerResults	Values of the hyperparameters, the objective function for the default configuration (Iteration 0), and up to 10 best configurations found	AUTOTUNE	Default
TunerSummary	Statistics about the tuning process	AUTOTUNE	Default
TunerTiming	Total time spent on different tasks during tuning	AUTOTUNE	Default
VariableImportance	Residual sum of squares variable importance	PROC GRADBOOST	Default
WeightHistory	Weights that are assigned by the EGR algorithm for each level of the response (target) variable and for each subpopulation	MITIGATEBIAS	Default

ODS Graphics

You can refer to every graph that is produced through ODS Graphics by name. The name of the graph that PROC GRADBOOST generates is listed in Table 16.7.

Table 16.7 Graph Produced by PROC GRADBOOST

ODS Graph Name	Plot Description	Statement	Option
IterationHistoryPlot	Iteration history of the model accuracy and bias metric	MITIGATEBIAS	Default

Examples: GRADBOOST Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

Example 16.1: Scoring New Data by Using a Previous Boosting Model

This example illustrates how you can use the OUTMODEL= option to save a model table, and later use the model table to score a data table. It uses the JunkMail data set in the Sashelp library.

The JunkMail data set comes from a study that classifies whether an email is junk email (coded as 1) or not (coded as 0). The data set contains 4,601 observations with 59 variables. The response variable is a binary indicator of whether an email is considered spam or not. There are 57 predictor variables that record the frequencies of some common words and characters and the lengths of uninterrupted sequences of capital letters in emails.

You can load the Sashelp.JunkMail data set into your SAS library in the following DATA step:

```
data mylib.junkmail;
   set sashelp.junkmail;
run;
```

These statements assume that your SAS libref is named mylib, but you can substitute any appropriately defined SAS libref.

The following statements train a boosting model and score the training data table. The OUTPUT statement scores the training data and saves the results to a new table named *fit_at_runtime*.

```
proc gradboost data=mylib.junkmail outmodel=mylib.gradboost_model seed=12345;
   input Address Addresses All Bracket Business CS CapAvg CapLong
         CapTotal Conference Credit Data Direct Dollar Edu Email
         Exclamation Font Free George HP HPL Internet Lab Labs
         Mail Make Meeting Money Order Original Our Over PM Paren
         Parts People Pound Project RE Receive Remove Semicolon
         Table Technology Telnet Will You Your _000 _85 _415 _650
         _857 _1999 _3D / level = interval;
   target class /level=nominal;
   output out=mylib.score_at_runtime;
   ods output FitStatistics=fit_at_runtime;
run;
```

The preceding statements produce the table shown in Output 16.1.1. The table shows the training statistics.

Output 16.1.1 Fit Statistics, Fit at Run Time

Fit Statistics			
Number of Trees	Training Average Square Error	Training Misclassification Rate	Training Log Loss
1	0.2087	0.3940	0.6079
2	0.1842	0.2334	0.5569
3	0.1645	0.1098	0.5154
4	0.1479	0.0926	0.4793
5	0.1330	0.0880	0.4460
6	0.1216	0.0830	0.4197
7	0.1113	0.0815	0.3950
8	0.1024	0.0767	0.3728
9	0.0952	0.0769	0.3539
10	0.0888	0.0739	0.3368
.	.	.	.
.	.	.	.
.	.	.	.
91	0.0276	0.0367	0.1049
92	0.0273	0.0356	0.1040
93	0.0271	0.0361	0.1033
94	0.0270	0.0350	0.1030
95	0.0268	0.0343	0.1021
96	0.0267	0.0341	0.1017
97	0.0266	0.0346	0.1012
98	0.0264	0.0343	0.1006
99	0.0261	0.0322	0.0995
100	0.0259	0.0319	0.0989

The following statements use a previously saved model to score new data:

```
proc gradboost data=mylib.junkmail inmodel=mylib.gradboost_model;
   output out=mylib.score_later;
   ods output FitStatistics=fit_later;
run;
```

When you specify the INMODEL= option to use a previously created boosting model, you see the statistics for the scored data if the target exists in the newly scored data table. In this example, the scored data are the same as the training data, so you can see that the statistics in Output 16.1.2 match those previously seen in Output 16.1.1.

Output 16.1.2 Fit Statistics, Fit Later

Fit Statistics			
Number of Trees	Average Square Error	Misclassification Rate	Log Loss
1	0.2087	0.3940	0.6079
2	0.1842	0.2334	0.5569
3	0.1645	0.1098	0.5154
4	0.1479	0.0926	0.4793
5	0.1330	0.0880	0.4460
6	0.1216	0.0830	0.4197
7	0.1113	0.0815	0.3950
8	0.1024	0.0767	0.3728
9	0.0952	0.0769	0.3539
10	0.0888	0.0739	0.3368
.	.	.	.
.	.	.	.
.	.	.	.
91	0.0276	0.0367	0.1049
92	0.0273	0.0356	0.1040
93	0.0271	0.0361	0.1033
94	0.0270	0.0350	0.1030
95	0.0268	0.0343	0.1021
96	0.0267	0.0341	0.1017
97	0.0266	0.0346	0.1012
98	0.0264	0.0343	0.1006
99	0.0261	0.0322	0.0995
100	0.0259	0.0319	0.0989

This example demonstrates that the GRADBOOST procedure can score an input data table by using a previously saved boosting model, which was saved using the OUTMODEL= option in a previous procedure run. If you want to properly score a new data table, you must not modify the table mylib.gradboost_model, because doing so could invalidate the constructed boosting model. As with any scoring of new data, the variables that are used in the model creation must be present in order for you to score a new table.

Example 16.2: Transfer Learning

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

This example illustrates transfer learning. Transfer learning augments the training data with auxiliary data and attempts to down-weight the influence of observations that are not representative of the original training data. The original training data typically come from a *target* population from which data are hard to get. For simplicity, this example refers to auxiliary observations that are similar to the target population as *friends*, and the other auxiliary observations as*aliens*. PROC GRADBOOST tries to identify and down-weight the aliens. This example runs PROC GRADBOOST twice without using transfer learning: once with all the data, and once without the aliens. All models are evaluated with data from the target population that are not part of

the training data. The model incorporating transfer learning should provide a better fit than one without it, though not as good as the one in which the alien population is removed.

The following DATA step generates three data sets: one for training that includes the friends and the aliens, one without the aliens, and a third data set with test observations held out from training.

```
data mylib.train mylib.noAlien mylib.test;
     array x(2)  x1 - x2;
     keep datarole y x: constantZero;

     constantZero = 0;
     call streaminit(3331333);
     do datarole = -1 to 2;
        select(datarole);
           when(-1) do;  * test data;
                      y =1;
                      n = 1000;
                    end;
           when(0)  do;  * target population;
                      y =1;
                      n = 1000;
                    end;
           when(1)  do;  * friend population;
                      y  =1;
                      n = 2000;
                    end;
           when(2)  do;  * alien population;
                      y= -1;
                      n = 1000;
                    end;
        end;

        nhalf = n/2;
        mu    = 0.5;

        do i = 1 to n;
           do j = 1 to 2;
              x(j)  = rand('normal', mu);
           end;
           select(datarole);
               when(-1)  output mylib.test;
               when(0,1) output mylib.train mylib.noAlien;
               when(2)   output mylib.train;
           end;
           if i = nhalf then do;
              mu = -mu;
              y  = -y;
           end;
        end;
     end;
run;
```

The following macro invokes PROC GRADBOOST twice: first to train the model, and then again to apply the model to the test data and output fits statistics. The TRANSFERLEARN statement specifies the variable

that identifies the auxiliary observations. When that variable is zero for all observations, transfer learning is not done. This happens when constantZero is passed to the macro.

The BURN=10 option in the TRANSFERLEARN statement delays down-weighting until tree 11.

```
%macro runProc(data=, auxVar=, outfit=);
  proc gradboost data=&data. outmodel=mylib.model seed=3331333;
     input x: /level=interval;
     target y /level=nominal;
     transferLearn &auxVar. / burn=10;
  run;
  proc gradboost data=mylib.test inmodel=mylib.model;
     output out=mylib.score;
     ods output FitStatistics=&outfit.;
  run;
%mend;

%runProc(data=mylib.train,   auxVar=datarole, outfit=fit_train);
%runProc(data=mylib.train,   auxVar=constantZero, outfit=fit_noTransfer);
%runProc(data=mylib.noAlien, auxVar=constantZero, outfit=fit_noAlien);
```

The following code combines the average square error from the three models into a single table.

```
%macro extractFit(data=, infit=);
   data &data.;
     set &infit.;
     keep trees ase;
     rename ase = &data._ase;
   run;
%mend;

%extractFit(data=train, infit=fit_train);
%extractFit(data=noTransfer, infit=fit_noTransfer);
%extractFit(data=noAlien, infit=fit_noAlien);

data result;
  merge train noTransfer noAlien;
  by trees;
run;
```

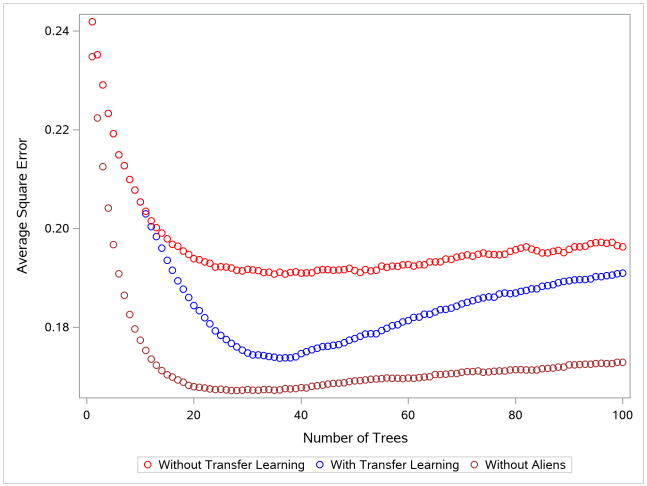
The following code plots the average square error for each model by the number of trees in the model:

```
proc template;
     define statgraph transferLearning;
         begingraph;
             layout overlay;
             scatterplot y=train_ase
                         x=trees / markerattrs=(color=blue)
                                   name='with'
                                   legendlabel="With Transfer Learning";
             scatterplot y=noTransfer_ase
                         x=trees / markerattrs=(color=red)
                                   name='without'
                                   legendlabel="Without Transfer Learning";
             scatterplot y=noAlien_ase
```

```
                        x=trees / markerattrs=(color=brown)
                                  name='noAliens'
                                  legendlabel="Without Aliens";

            discretelegend 'without' 'with' 'noAliens';
            endlayout;
        endgraph;
    end;
run;

proc sgrender data=result template=transferLearning;
run;
```

Output 16.2.1 shows that the fit with transfer learning is better than without it, though not as good as when all the alien observations are removed from the data. The fit with transfer learning is identical to the fit without it for the first 10 trees because down-weighting does not begin until tree 11 in this example.

Output 16.2.1 ASE versus Number of Trees for Three Models



NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

Example 16.3: Tuning a Boosting Model

This example illustrates how you can use the AUTOTUNE statement in the GRADBOOST procedure to tune a boosting model. It uses the JunkMail data set in the Sashelp library.

The JunkMail data set comes from a study that classifies whether an email is junk email (coded as 1) or not (coded as 0). The data set contains 4,601 observations with 59 variables. The response variable is a binary indicator of whether an email is considered spam or not. There are 57 predictor variables that record the frequencies of some common words and characters and the lengths of uninterrupted sequences of capital letters in emails.

You can load the Sashelp.JunkMail data set into your SAS library by naming your SAS libref in the first statement of the following DATA step:

```
data mylib.junkmail;
   set sashelp.junkmail;
run;
```

These statements assume that your SAS libref is named mylib, but you can substitute any appropriately defined libref.

The following statements use the AUTOTUNE statement in PROC GRADBOOST to run a tuning process on a boosting model:

```
ods graphics on;

proc gradboost data=mylib.junkmail outmodel=mylib.gradboost_model seed=12345;
   input Address Addresses All Bracket Business CS CapAvg CapLong
         CapTotal Conference Credit Data Direct Dollar Edu Email
         Exclamation Font Free George HP HPL Internet Lab Labs
         Mail Make Meeting Money Order Original Our Over PM Paren
         Parts People Pound Project RE Receive Remove Semicolon
         Table Technology Telnet Will You Your _000 _85 _415 _650
         _857 _1999 _3D / level = interval;
   target class /level=nominal;
   autotune
      /* Tuning Parameters
      You do not need to specify any tuning parameters for the default
      tuning process. If you want to make adjustments to the default
      tuning process, uncomment the following block of code and change
      any of the tuning parameters' attributes.

      tuningParameters=(
         BINMETHOD     (exclude)
         LASSO         (lb=0.0  ub=10.0 init=0.0 )
         LEARNINGRATE  (lb=0.01 ub=1.0  init=0.1 )
         MAXDEPTH      (lb=2    ub=7    init=5   )
         MINLEAFSIZE   (values=1 5 10 20 40 80 160 320 init=5 exclude)
         NTREES        (lb=20   ub=150  init=100 )
         NUMBIN        (lb=20   ub=100  init=50  )
         RIDGE         (lb=0.0  ub=10.0 init=1.0 )
         SAMPLINGRATE  (lb=0.1  ub=1.0  init=0.5 )
         VARS_TO_TRY   (lb=1    ub=56   init=7   )
      )
      */
   ;
   /* Remove this line to see all results */
   ods select BestConfiguration EvaluationHistoryPlot IterationHistoryPlot;
run;
```

The preceding statements produce the table and the plots shown in Output 16.3.1 through Output 16.3.3. The table in Output 16.3.1 displays the evaluation number, the values of the tuning parameters, and the error metric value for the best decision tree model that the tuner found. Note that the ODS SELECT statement limits the displayed results to a single table and two plots. You can remove this statement to display all tables and plots. For the full list of ODS tables that PROC GRADBOOST produces, see Table 16.6.

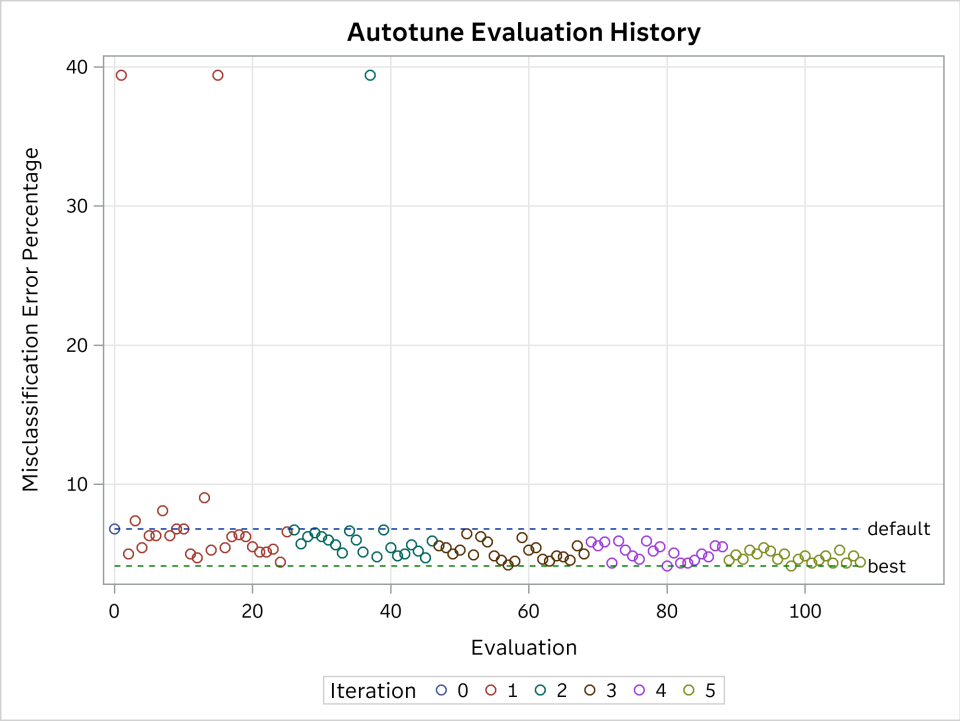
Output 16.3.1 Best Configuration Table

The GRADBOOST Procedure

Best Configuration	
Evaluation	80
Number of Variables to Try	43
Learning Rate	0.505
Sampling Rate	0.55
Lasso	2.5
Ridge	5
Number of Bins	60
Maximum Tree Levels	7
Misclassification Error Percentage	4.13

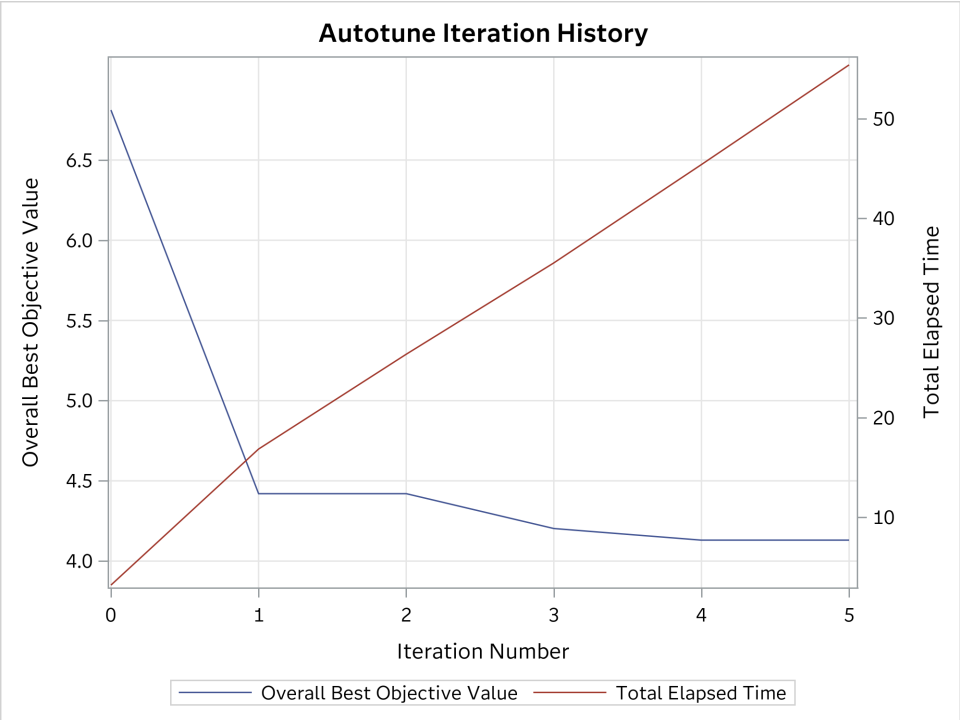
Output 16.3.2 displays a scatter plot of all configurations that the tuner tried. The objective values are shown on the Y axis, and the evaluation numbers are shown on the X axis.

Output 16.3.2 Evaluation History Plot



The plot in Output 16.3.3 displays how the best found objective value and the elapsed time changed with each iteration of the tuner.

Output 16.3.3 Iteration History Plot



NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

Example 16.4: Mitigating Bias of a Boosting Tree Model

This example uses the GRADBOOST procedure to create a gradient boosting tree model while minimizing the demographic parity for the Job variable. The response variable for the gradient boosting tree model is Bad, a classification variable that has two values (0 for payment of loan and 1 for default). The other variables are predictor variables for the model.

The following statements load the data table mylib.hmeq into your SAS library. For this example, the statements assume that your SAS library is named mylib, but you can substitute any appropriately defined SAS library.

```
data mylib.hmeq;
   set sampsio.hmeq;
run;
```

The following statements use PROC GRADBOOST to create a gradient boosting tree model without minimizing the demographic parity:

```
proc gradboost data=mylib.hmeq nTrees=20 seed=12345;
   input CLAge CLNo DebtInc Derog
         Loan MortDue Value / level=interval;
   input Reason / level=nominal;
```

```
    target Bad / level=nominal;
    output out=mylib.scored copyvars=(_ALL_);
    saveState rstore=mylib.model;
run;
```

The NTREES= option specifies that there are 20 trees to grow in the boosting model. The SEED= option specifies the random number seed for model building to be 12345. The two INPUT statements specify CLAge, CLNo, DebtInc, Derog, Loan, MortDue, Value, and Reason as the input variables. The TARGET statement specifies Bad as the target variable. The OUTPUT statement generates the data table mylib.scored, which contains the predicted mortgage default from the model for each observation and contains all variables from the input data table as copy variables. The SAVESTATE statement creates an analytic store for the model and saves it as a binary object in the data table mylib.model.

Output 16.4.1 displays the fit statistics for the boosting model.

Output 16.4.1 Fit Statistics Table

The GRADBOOST Procedure

Fit Statistics			
Number of Trees	Training Average Square Error	Training Misclassification Rate	Training Log Loss
1	0.1464	0.199	0.461
2	0.1359	0.199	0.432
3	0.1272	0.199	0.410
4	0.1202	0.199	0.392
5	0.1143	0.193	0.377
6	0.1095	0.175	0.364
7	0.1054	0.162	0.353
8	0.1021	0.156	0.344
9	0.0992	0.137	0.336
10	0.0966	0.128	0.329
11	0.0943	0.123	0.322
12	0.0925	0.116	0.317
13	0.0908	0.114	0.312
14	0.0896	0.113	0.308
15	0.0885	0.114	0.304
16	0.0872	0.113	0.300
17	0.0864	0.113	0.297
18	0.0857	0.113	0.294
19	0.0849	0.113	0.291
20	0.0842	0.112	0.289

After you train a gradient boosting tree model, you might want to view the demographic parity of the trained model. You can do this by running the ASSESSBIAS procedure. For more information about PROC ASSESSBIAS, see Chapter 3, "ASSESSBIAS Procedure."

The following statements use PROC ASSESSBIAS to assess the bias measurements of the model:

```
proc assessbias data=mylib.scored;
   target bad / event="1" level=nominal;
   var P_Bad1;
   fitstat pvar=P_BAD0 / pevent="0";
   sensitiveVar Job;
run;
```

Output 16.4.2 displays the bias metric values of the model. PROC ASSESSBIAS also produces other output not shown here.

Output 16.4.2 Bias Metrics Table

The ASSESSBIAS Procedure

Bias Metrics					
Bias Statistic	**Bias Statistic Label**	**Bias Statistic Value**	**Base Level**	**Compare Level**	**Note**
DemographicParity	Demographic Parity (Statistical Parity)	0.1170	Self	Office	
PredictiveParity	Predictive Parity	0.0836	Sales	Office	
EqualAccuracy	Equal Accuracy	0.1090	Office	Sales	
EqualizedOdds	Equalized Odds	0.1806	Self	Sales	The maximum TPR difference is greater than the maximum FPR difference.
EqualOpportunity	Equal Opportunity	0.1806	Self	Sales	

In order to use the MITIGATEBIAS statement to reduce the demographic parity, you must specify the levels of the response (target) variable and the corresponding predicted variables. You can use the DESCRIBE statement in the ASTORE procedure to obtain this information. The following statements use PROC ASTORE to describe the model:

```
proc astore;
   describe rstore=mylib.model;
run;
```

Output 16.4.3 lists the output variables created by the model. The predicted variables are P_BAD1 and P_BAD0. Their corresponding response (target) levels are 1 and 0, respectively.

Output 16.4.3 Output Variables Table

The ASTORE Procedure

Output Variables			
Name	**Length**	**Type**	**Label**
P_BAD1	8	Num	Predicted: BAD=1
P_BAD0	8	Num	Predicted: BAD=0
I_BAD	12	Character	Into: BAD
WARN	4	Character	Warnings

The following statements use PROC GRADBOOST to create a gradient boosting tree model, while minimizing the demographic parity for Job by using the MITIGATEBIAS statement:

```
ods graphics on;

proc gradboost data=mylib.hmeq nTrees=20 seed=12345;;
   input CLAge CLNo DebtInc Derog
         Loan MortDue Value / level=interval;
   input Reason / level=nominal;
   target Bad / level=nominal;
   output out=mylib.scored;
   saveState rstore=mylib.model;
   mitigateBias pevents="1 0"
                pvars=(P_BAD1 P_BAD0)
                seed=12345
                sensitivevar=Job
                targetEvent="1"
                maxiter=11;
run;
```

The PEVENTS= option lists the levels of the response (target) variable Bad. The predicted variables correspond to the levels that are listed in the PVARS= option. The PVARS= option lists the predicted variables that contain the posterior probability for each level in model prediction that corresponds to the response (target) variable Bad. The SEED= option specifies the seed for the pseudorandom number generator to be 12345. The SENSITIVEVAR= option names the variable Job as the sensitive variable for the purpose of reducing the value of the bias measurement. The TARGETEVENT= option specifies the value 1 as the event of interest. The MAXITER= option specifies 11 as the maximum number of iterations for the EGR algorithm.

Output 16.4.4 displays the information about the options that are used for bias mitigation. Output 16.4.5 displays the model accuracy and bias metric values over each iteration of the EGR algorithm. Output 16.4.6 shows the iteration history, which is a graphical representation of the iteration history table. The graph displays the bias metric and the misclassification rate for each iteration. PROC GRADBOOST selects the iteration that has the lowest bias metric. In this case, the 10th iteration is selected; the graph indicates this selection by using the legend and a vertical line. The MITIGATEBIAS statement also creates other tables, which are not shown here but are listed in the section "ODS Table Names" on page 437.

Output 16.4.4 Mitigation Information Table

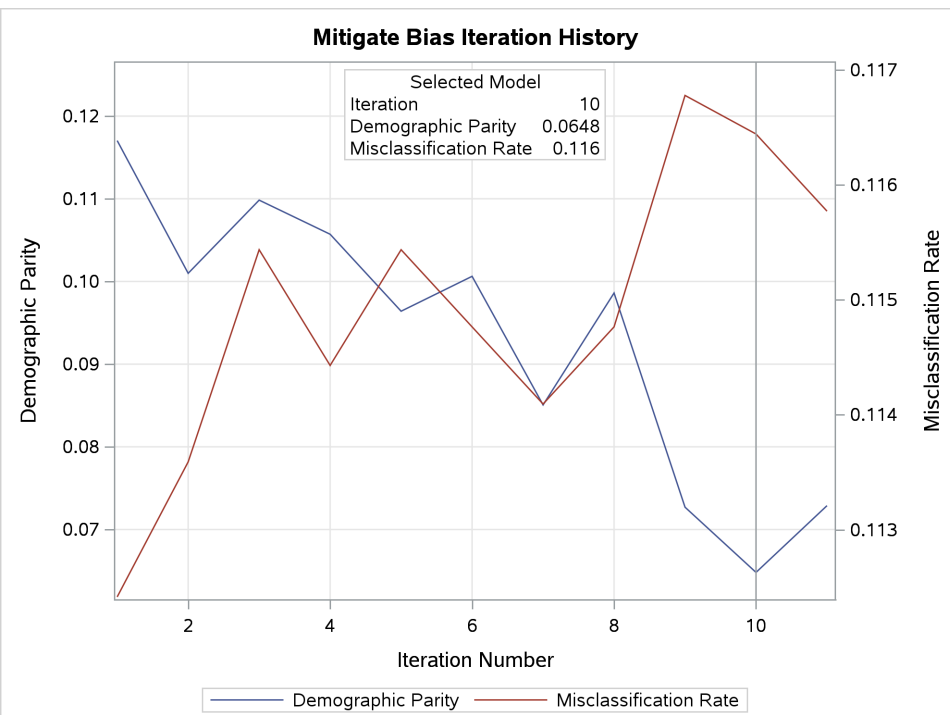
The GRADBOOST Procedure

Mitigation Information		
Description	**Character Value**	**Value**
Bias Metric	DEMOGRAPHICPARITY	.
Sensitive Variable	JOB	.
Response	BAD	.
Bound Value	100	100
Learning Rate	0.01	0.01
Maximum Iteration	11	11
Tolerance	0.005	0.005
Event Level	1	.

Output 16.4.5 Iteration History Table

EGR Iteration History			
Iteration	Demographic Parity (Statistical Parity)	Misclassification Rate	Multi-Class Log Loss
1	0.117034	0.1124	0.2887
2	0.100985	0.1136	0.2896
3	0.109836	0.1154	0.2901
4	0.105709	0.1144	0.2895
5	0.096401	0.1154	0.2902
6	0.100621	0.1148	0.2906
7	0.085066	0.1141	0.2917
8	0.098604	0.1148	0.2927
9	0.072697	0.1168	0.2933
10	0.064812	0.1164	0.2940
11	0.072883	0.1158	0.2937

Output 16.4.6 Iteration History Graph



References

Agarwal, A., Beygelzimer, A., Dudik, M., Langford, J., and Wallach, H. (2018). "A Reductions Approach to Fair Classification." In *Proceedings of the 35th International Conference on Machine Learning*, 60–69. PMLR.

Chen, T., and Guestrin, C. (2016). "XGBoost: A Scalable Tree Boosting System." In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 785–794. New York: ACM.

Dai, W., Yang, Q., Xue, G.-R., and Yu, Y. (2007). "Boosting for Transfer Learning." In *Proceedings of the 24th International Conference on Machine Learning*. Corvallis, OR: ACM.

Friedman, J. H. (2001). "Greedy Function Approximation: A Gradient Boosting Machine." *Annals of Statistics* 29:1189–1232.

Hastie, T. J., Tibshirani, R. J., and Friedman, J. H. (2001). *The Elements of Statistical Learning: Data Mining, Inference, and Prediction*. New York: Springer-Verlag.

Lundberg, S. M., Erion, G., Chen, H., DeGrave, A., Prutkin, J. M., Nair, B., Katz, R., Himmelfarb, J., et al. (2020). "From Local Explanations to Global Understanding with Explainable AI for Trees." *Nature Machine Intelligence* 2:56–67.

Van der Laan, M. J. (2006). "Statistical Inference for Variable Importance." *International Journal of Biostatistics* 2:1–31. Article 2.

Chapter 17
GVARCLUS Procedure

Contents

Overview: GVARCLUS Procedure **454**
- PROC GVARCLUS Features 454
- PROC GVARCLUS Compared to PROC VARCLUS 455
- Using SAS Viya Workbench 455

Getting Started: GVARCLUS Procedure **456**

Syntax: GVARCLUS Procedure **458**
- PROC GVARCLUS Statement 459
- DISPLAY Statement 462
- DISPLAYOUT Statement 463
- FREQ Statement 463
- INPUT Statement 464

Details: GVARCLUS Procedure **464**
- Missing Values 464
- Graphical Lasso 465
- Graphical Representation of Precision Matrix 466
- Preprocessing 466
- Selection 466
- Displayed Output 467
  - Number of Observations 467
  - Cluster Summary 467
- ODS Table Names 467

Examples: GVARCLUS Procedure **468**
- Example 17.1: Analyzing United States Senate Voting Data 468
- Example 17.2: Output a Covariance Matrix to a SAS Data File 470
- Example 17.3: Output the Covariance Matrix in LIL Format 471

References **473**

Overview: GVARCLUS Procedure

The GVARCLUS procedure performs variable clustering and graphical modeling in SAS Viya Workbench. The procedure divides a set of variables into disjoint clusters and creates tables that contain the edge and vertex information for defining an undirected graph. Variables in different clusters are conditionally independent given their own clusters. The procedure also provides the edge and vertex information for an undirected graphical model; this information expresses the relationships among all the variables. A regularization parameter is used to control the sparsity of connections among variables. Tuning the regularization parameter from low to high increases the number of disconnected components and splits larger clusters into smaller ones. Those divided clusters naturally form a hierarchical structure during this process.

The GVARCLUS procedure helps mine the relationships among variables and remove redundant variables. Removing redundant variables is especially important for high-dimensional data, which are increasingly common in the modern data mining world.

PROC GVARCLUS Features

The GVARCLUS procedure does the following:

- supports multiple INPUT statement for both continuous and categorical inputs
- provides a FREQ statement for grouped analysis
- provides the OUTTREE option to output the hierarchical clustering results in a tree structure
- provides the OUTEDGE and OUTVERT options to output the edges and vertices, respectively, for defining an undirected graph
- provides the OUTCP option to output a covariance matrix
- performs graphical lasso based on Friedman, Hastie, and Tibshirani (2008), which estimates the inverse covariance (IC) matrix at a specified regularization parameter. The inverse covariance matrix interprets the partial correlation between variables given other variables
- performs sequential steps of estimating the inverse covariance matrix by using regularization parameters from large to small, which impose different sparsity constraints on the estimation. The sequential steps eventually produce a set of nested clusters that are organized as a hierarchical tree
- performs covariance thresholding based on Friedman, Hastie, and Tibshirani (2008) as preprocessing, which thresholds the entries of the sample covariance matrix at the regularization parameter and decomposes the matrix into connected components

PROC GVARCLUS Compared to PROC VARCLUS

This section compares the GVARCLUS procedure with the VARCLUS procedure in SAS/STAT software.

PROC GVARCLUS interprets the conditional dependency among variables by estimating the inverse covariance matrix. The off-diagonal elements of an inverse covariance matrix correspond to partial correlations, so the zero elements imply conditional independence between the pair of variables. The conditional independence provides a better model for understanding the direct link between variables than does simple correlation analysis, which models each pair of variables without considering other variables.

PROC VARCLUS performs variable clustering based on latent components. It assigns a variable to its most correlated latent component, so eventually the variables within a cluster have high correlation.

The primary difference between the two procedures is that PROC GVARCLUS considers the association between variables after the other effects have been factored out.

Using SAS Viya Workbench

SAS Viya Workbench has greatly simplified the SAS programming experience by offering a syntax style nearly identical to that of SAS 9 and earlier releases. However, there are minor exceptions. In the past, SAS Viya procedures required the use of a CAS engine libref. This engine served multiple purposes, but it was primarily used to upload data to the CAS server and manage the data. Its additional purposes included support of new data types and improved performance for large data sets. No CAS server is needed for SAS Viya Workbench, because all computations are done in place for you in a single location.

In the following program, the LIBNAME statement creates a mylib libref that the examples in this chapter use to refer to your current or desired work directory:

```
libname mylib ".";
```

For more information, see *SAS Viya Workbench: Getting Started*.

Getting Started: GVARCLUS Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

This example demonstrates how you can use the GVARCLUS procedure to cluster variables into homogeneous groups and construct a hierarchy.

The following DATA step generates a data table. The table consists of 100 observations and 20 continuous variables (x1–x20). There are two groups of five variables, and 10 variables that are randomly generated from an uniform distribution. The initial variables for each group, x1 and x6, are independently generated from a normal distribution. Two of the variables from each group are linearly correlated to their initial variables, and the other two variables have a nonlinear relationship.

```
data mylib.getStarted;
    array x{20}  x1-x20;
    call streaminit(12345);
    do i = 1 to 100;
        x1=rand('NORMAL'); x2=0.9*x1+0.2; x3=0.7*x1+0.6; x4=x3*x2*x1; x5=x1*x1*x1;
        x6=rand('NORMAL'); x7=0.9*x6+0.2; x8=0.7*x6+0.6; x9=x8*x6; x10=x6*x6*x6;
        do j= 11 to 20;
            x{j} = rand('UNIFORM');
        end;
        output;
    end;
run;
```

The following statements execute the GVARCLUS algorithm on the mylib.getStarted data table:

```
proc gvarclus data=mylib.getStarted mincluster=2 maxsteps=8
              rho=0.8 outtree=mylib.dendrogram;
    input x1-x20 / level=interval;
run;
```

The INPUT statement defines the input variables x1 to x20 as interval variables. The MINCLUSTER= option in the PROC GVARCLUS statement specifies the minimum number of clusters as 2, so the clustering process stops when the number of clusters is less than or equal to 2. The MAXSTEPS= option specifies the maximum number of steps in the clustering process as 8. The RHO= option specifies a value of 0.8, which determines the sequence of regularization parameters $[0.8^1, 0.8^2, 0.8^3, \ldots]$ that are used in each step sequentially. The OUTTREE= option creates the mylib.dendrogram table to contain information about the tree structure of hierarchical clustering.

The output from this analysis is presented in Figure 17.1 and Figure 17.2.

Figure 17.1 displays the "Number of Observations" tables. This table shows that all 100 observations in the data table are used in the analysis.

Figure 17.1 Number of Observations

The GVARCLUS Procedure

Number of Observations Read	100
Number of Observations Used	100

Figure 17.2 shows the "Cluster Summary" table, which shows the cluster structure in each step. The variable (or level for nominal variables) is assigned to a unique cluster in each step, and the clusters from sequential steps are nested to form a hierarchical structure.

Figure 17.2 Cluster Summary

Cluster Summary								
Parameter	**Step 1**	**Step 2**	**Step 3**	**Step 4**	**Step 5**	**Step 6**	**Step 7**	**Step 8**
x1	1	19	35	51	66	81	96	109
x2	2	19	35	51	66	81	96	109
x3	3	19	35	51	66	81	96	109
x4	4	20	36	52	67	82	96	109
x5	5	21	37	53	68	83	97	109
x6	6	22	38	54	69	84	98	110
x7	6	22	38	54	69	84	98	110
x8	6	22	38	54	69	84	98	110
x9	7	23	39	55	70	85	98	110
x10	8	24	40	54	69	84	98	110
x11	9	25	41	56	71	86	99	111
x12	10	26	42	57	72	87	100	112
x13	11	27	43	58	73	88	101	113
x14	12	28	44	59	74	89	102	114
x15	13	29	45	60	75	90	103	115
x16	14	30	46	61	76	91	104	116
x17	15	31	47	62	77	92	105	117
x18	16	32	48	63	78	93	106	118
x19	17	33	49	64	79	94	107	119
x20	18	34	50	65	80	95	108	120

You can define a dendrogram in the Graph Template Language (GTL) and display the hierarchical clustering results from the GVARCLUS algorithm.

The following code copies mylib.dendrogram into the Work library:

```
data outtree;
    set mylib.dendrogram;
run;
```

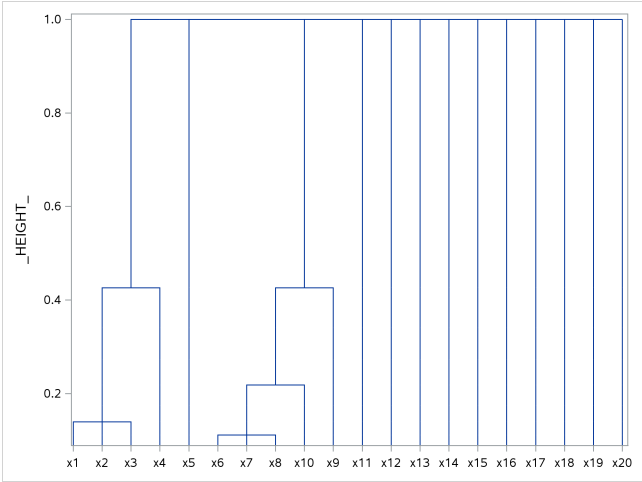
The following code defines the parent node, child node and the height in the dendrogram through the DENDROGRAM statement in GTL. The dendrogram is then plotted using the SGRENDER procedure.

```
proc template;
    define statgraph dendrogram;
    begingraph;
    layout overlay;
```

```
    dendrogram nodeID=_CHILD_ parentID=_PARENT_ clusterheight=_HEIGHT_;
    endlayout;
    endgraph;
end;

proc sgrender data=outtree template=dendrogram;
run;
```

Output 17.3 shows the dendrogram.

Figure 17.3 Hierarchical Clustering



Syntax: GVARCLUS Procedure

The following statements are available in the GVARCLUS procedure:

PROC GVARCLUS < *options* > **;**
  DISPLAY < *table-list* > < */options* > **;**
  DISPLAYOUT *table-spec-list* < */options* > **;**
  INPUT *variables* < **LEVEL= INTERVAL** | **NOMINAL** > **;**
  FREQ *variable* **;**

The PROC GVARCLUS statement and at least one INPUT statement are required.

The following sections describe the PROC GVARCLUS statement and then describe the other statements in alphabetical order.

PROC GVARCLUS Statement

PROC GVARCLUS < *options* > **;**

The PROC GVARCLUS statement invokes the procedure. Table 17.1 summarizes the important options in the PROC GVARCLUS statement by function. The options are then described fully in alphabetical order.

Table 17.1 PROC GVARCLUS Statement Options

Option	Description
Basic Options	
DATA=	Specifies the input data table
Options Related to Variable Clustering	
EXACT	Performs graphical variable clustering without preprocessing
MAXITER=	Specifies the maximum number of iterations for each estimation step
MAXMEMBER=	Specifies the maximum number of members within each final cluster
MAXSTEP=	Specifies the maximum number of clustering steps
MINCLUSTER=	Specifies the minimum number of clusters
RHO=	Specifies the initial value of the regularization parameter
SELECT=	Selects the optimal clustering on the basis of the specified criterion
STOP=	Requests that the clustering process stop if the clustering structure does not change in the previous number of specified steps
TOL=	Specifies the threshold of convergence
Options for Output Tables	
OUTCP=	Outputs the covariance matrix from the last step
OUTEDGE=	Outputs the edges for defining an undirected network
OUTTREE=	Outputs the hierarchical clustering results in a tree structure
OUTVERT=	Outputs the vertices and their size for defining an undirected network

You can specify the following *options*:

DATA=*libref.data-table*

names the input data table for PROC GVARCLUS to use. The default is the most recently created data table. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to the data. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 455.

data-table specifies the name of the input data table.

EXACT

performs graphical variable clustering without preprocessing by thresholding the sample covariance into connected components. By default, the preprocessing step is performed.

MAXITER=*n*

MAXITERS=*n*

specifies the maximum number of iterations for estimating the sparse precision matrix by using coordinate descent. By default, MAXITER=50.

MAXMEMBER=*n*

MAXMEMBERS=*n*

stops PROC GVARCLUS when the number of members within any cluster is greater than or equal to *n*.

MAXSTEP=*n*

MAXSTEPS=*n*

stops PROC GVARCLUS after it runs *n* steps. By default, MAXSTEP=3.

MINCLUSTER=*n*

MINCLUSTERS=*n*

stops PROC GVARCLUS when the number of clusters is less than or equal to *n*. By default, MINCLUSTER=3.

OUTCP=*libref.data-table* **< / LIST < EPS=***number* **> >**

creates a data table that contains a covariance matrix that depicts the relationships among variables and also creates a set of statistics about the input data table and variables. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 455.

When you specify the LIST option, the symmetric matrix is output in the list-of-list (LIL) format. In this format, the matrix is represented as a set of tuples (i, j, x), where x is an entry in the matrix and i and j denote its row and column indices, respectively. LIL format can be used when the output contains too many columns to fit in a data table. For example, in most database systems, the maximum number of columns in a table is usually limited to several thousand. If an output matrix contains more columns than the limit, you must use the LIST option in order to avoid errors that would arise from writing too many columns to the table. When LIL format is used, all entries in the matrix that have a value of 0 are ignored in the output.

When you specify EPS=*number* in the LIST suboption, matrix entries whose absolute value is smaller than *number* are ignored in the output. This feature helps omit unreliable estimations and generate a compact representation of the matrix. When you do not specify the EPS= option, only the 0 entries in the matrix are ignored in the output.

OUTEDGE=*libref.data-table*

creates a data table that contains the information that defines the edges in the network, _FROM_, _TO_, and _WEIGHT_. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 455.

OUTTREE=*libref.data-table*

creates a data table that depicts a tree diagram to display the hierarchical clustering results. The tree diagram can be plotted using the DENDROGRAM statement in the Graph Template Language. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 455.

OUTVERT=*libref.data-table*

creates a data table that contains the vertices in the network and their size. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 455.

RHO=*fraction*

specifies the value of ρ that determines the sequence of regularization parameters, $\{\rho^1, \rho^2, \rho^3, \ldots\}$, that are used on sequential clustering steps. The higher the ρ value, the more regularization and the sparser the inverse covariance. By default, RHO=0.8.

SELECT=NONE | PENALIZED

selects the optimal clustering on the basis of the specified criterion. You can specify the following values:

NONE performs no selection, and the output tables of the edges and vertices of the network reflect the clustering from the last step.

PENALIZED performs selection through minimizing the penalized log likelihood, as shown in the section "Selection" on page 466.

By default, SELECT=NONE.

STOP=*n*

requests that the procedure stop if the clustering results do not change in the previous *n* consecutive steps. By default, STOP=3.

TOLERANCE=*fraction*

TOL=*fraction*

specifies the minimal absolute tolerance at which an iteration stops. The tolerance number must be equal to or greater than 1.0E–12. By default, TOLERANCE=1.0E–3.

DISPLAY Statement

DISPLAY < *table-list* > < / *options* > **;**

The DISPLAY statement enables you to specify a list of display tables to display or exclude. This statement is similar to the ODS SELECT, ODS EXCLUDE, and ODS TRACE statements. However, the DISPLAY statement can improve performance when a large number of tables could be generated (such as in BY-group processing). The DISPLAY statement produces only the requested tables, but ODS statements generate all the tables and then keep the requested tables.

If you use both DISPLAY and ODS statements together, the DISPLAY statement takes precedence over the ODS statements. Note that the ODS EXCLUDE statement processes tables after they have been filtered by the DISPLAY statement. In some cases, it might appear that the ODS EXCLUDE statement is taking precedence because it can further filter the tables. For more information about ODS, see *SAS Output Delivery System: Procedures Guide*.

You can specify the *table-list* as a list of table names, paths, partial pathnames, and regular expressions.

The table names that you can specify are listed in the section "ODS Table Names" on page 467. A path is a table name that is prefixed with dot-separated grouping information. For example, a SelectionSummary table that a procedure produces during a selection routine might have the path *Bygroup1.Summary.SelectionSummary*. A partial pathname does not include all groups; for example, *SelectionSummary* and *Summary.SelectionSummary* are partial pathnames for *Bygroup1.Summary.SelectionSummary*.

When you specify a table name or partial pathname, all display tables whose paths end in the specified name are selected for display or exclusion. For example, both *SelectionSummary* and *Summary.SelectionSummary* select *Bygroup1.Summary.SelectionSummary*.

A regular expression is enclosed in forward slashes (/). For example, specifying "/tions/" selects all pathnames that contain the substring "tions"; in particular, the *Bygroup1.Summary.SelectionSummary* table is selected. Specifying "!/tions/" selects all pathnames that do not contain the substring "tions"; in particular, the *Bygroup1.Summary.SelectionSummary* table is not selected.

You can specify the following *options* after a slash (/):

CASESENSITIVE
performs a case-sensitive comparison of table names in the *table-list* to display table names when tables are subsetted for display. To preserve case, you must enclose table names in the *table-list* in quotation marks.

EXCLUDE
displays all display tables except those that you specify in the *table-list*.

EXCLUDEALL
suppresses display of all tables. This option takes precedence over the other options.

TRACE
displays the display table names, labels, and paths.

DISPLAYOUT Statement

DISPLAYOUT *table-spec-list* < / *options* > **;**

The DISPLAYOUT statement enables you to create output tables from your displayed output. This statement is similar to the ODS OUTPUT statement. For more information about ODS, see *SAS Output Delivery System: Procedures Guide*.

The *table-spec-list* specifies a list of output tables to create. Each entry in the list has either a *key*=*value* format or a *key* format:

key=*value* specifies *key* as the ODS table name, path, or partial pathname, and specifies *value* as the output table name.

key specifies *key* as the ODS table name and also as the output table name.

The ODS table names that you can specify are listed in the section "ODS Table Names" on page 467. You cannot specify the ODS table named OutputCasTables in the *table-spec-list*.

Table names and partial pathnames are discussed under the DISPLAY statement. The DISPLAYOUT statement does not support regular expressions.

You can specify the following *options* after a slash (/):

INCLUDEALL
creates output tables for all display tables. The name of the created output table is the same as the name of the corresponding display table. If you specify this option, the *table-spec-list* specification is ignored.

NOREPLACE
does not replace any existing output table of the same name.

FREQ Statement

FREQ *variable* **;**

The *variable* in the FREQ statement identifies a numeric variable in the input data table that contains the frequency of occurrence of each observation. PROC GVARCLUS treats each observation as if it appears f times, where f is the value of the FREQ *variable* for the observation. If f is not an integer, it is truncated to an integer. If f is less than 1 or missing, the observation is not used in the analysis. When you omit the FREQ statement, each observation is assigned a frequency of 1.

INPUT Statement

INPUT *variables* / < **LEVEL=INTERVAL | NOMINAL** > **;**

The INPUT statement specifies the names of *variables* to be used in training. Only interval, binary, and nominal variables are accepted. If you want to use different options for different variables, you can specify multiple INPUT statements.

You can specify the following *option* after a slash (/):

LEVEL=INTERVAL | NOMINAL

specifies whether the specified input *variables* are continuous or categorical. You can specify the following values:

INTERVAL specifies that the input *variables* are continuous.

NOMINAL specifies that the input *variables* are categorical.

By default, LEVEL=INTERVAL for numeric variables and LEVEL=NOMINAL for categorical variables. Binary variables are considered to be categorical variables.

Details: GVARCLUS Procedure

The GVARCLUS procedure analyzes the relationships among variables in a multivariate analysis. It estimates the inverse covariance matrix which encodes partial correlations between pairs of variables given the other variables. At each step, a regularization parameter is used to control the sparsity of the estimated inverse covariance matrix. Zero elements imply conditional independence between variables and the variables are divided into disjoint clusters. Tuning the regularization parameter from low to high increases the number of disconnected components and splits larger clusters into smaller one. The divided clusters from sequential steps naturally form a hierarchical structure during the process.

Missing Values

Any observation that has missing values for the frequency or effects is excluded from the analysis. Observations that have a frequency less than 1 are also excluded.

Graphical Lasso

PROC GVARCLUS uses the graphical lasso (Friedman, Hastie, and Tibshirani 2008) to estimate the inverse covariance matrix. Assume that the observations have a multivariate Gaussian distribution with mean μ and covariance Σ. Zeros in the inverse covariance matrix, also called the precision matrix, correspond to conditional independencies among the variables, given other variables. The precision matrix is estimated by solving a maximum likelihood problem with an added l_1-norm penalty to increase the sparsity of the precision matrix.

Suppose you have n multivariate normal observations of dimension p, $\mathbf{X} \sim \mathbf{N}(\mu, \Sigma)$. Let $\Theta = \Sigma^{-1}$, and let S be the empirical covariance matrix. The problem is to maximize the following penalized log likelihood over nonnegative definite matrices Θ:

$$\log\det\Theta - \mathrm{Trace}(S\Theta) - \lambda||\Theta||_1$$

Here det denotes the determinant and $||\Theta||_1$ is the L_1 norm, the sum of the absolute values of the elements of Σ^{-1}. The scalar parameter λ controls the size of the penalty. The penalty term is a proxy for the number of nonzero elements in Θ.

Let W be the estimate of Σ. Banerjee, El Ghaoui, and d'Aspremont (2008) show that the preceding problem is convex and can be solved by optimizing over each row and the corresponding column of W in a block coordinate descending fashion. They further show that the problem is equivalent to the following dual problem:

$$\min\{\frac{1}{2}||W_{11}^{1/2}\beta - b||^2 + \lambda||\beta||_1\}$$

where $b = W_{11}^{-1/2}$ and W is partitioned as follows:

$$W = \begin{bmatrix} W_{11} & w_{12} \\ w_{12}^T & w_{22} \end{bmatrix}$$

Permute the rows and columns so that the one to be updated is always the last. Solve the dual problem, and then update the estimate of W after each stage.

The graphical lasso algorithm solves the dual problem as follows:

1. Start with $W = S + \lambda I$, where S is partitioned as follows:

   $$S = \begin{bmatrix} S_{11} & s_{12} \\ s_{12}^T & s_{22} \end{bmatrix}$$

   The diagonal of W remains unchanged in the following steps.

2. For each $j = 1, 2, \ldots, p, \ldots$, solve the lasso problem by coordinate descent (Friedman, Hastie, and Tibshirani 2008), which takes as input the inner products W_{11} and s_{12}. This produces a $p - 1$ vector solution $\hat{\beta}$. Fill in the corresponding row and column of W using $w_{12} = W_{11}\hat{\beta}$.

3. Continue until convergence.

Graphical Representation of Precision Matrix

Elements that have the value 0 in a precision matrix imply conditional independence between variables given the other variables in the network. So the estimate of a precision matrix can recover the conditional independence structure, which can be displayed in an undirected graph. If node X_i is connected to node X_j by an arc, then X_i is called a neighbor of X_j. If X_i is connected to X_k though some chain of arcs, then X_i is called a connected component of X_k.

The regularization parameter in the penalized log likelihood can be seen as a quasi-measure for the strength of connection between X_i and X_j, because as λ increases from small to large, some connections drop out, and the connections that drop out should be weaker than those that stay in.

Preprocessing

Prior to estimating the precision matrix by using the graphical lasso at each step, PROC GVARCLUS creates a threshold for the entries of the sample covariance matrix at a regularization parameter λ and decomposes the covariance matrix into connected components. In other words, if an entry of sample covariance is less than λ in its absolute value, there is no graph edge between the variables. This graph is called the thresholded sample covariance graph. The thresholded covariance matrix can be reordered and decomposed into connected components.

If you already have the solution to maximize the penalized log likelihood, you can also present the edge set via a symmetric 0-1 matrix. Let $\hat{\Theta}^{\lambda}$ denote the solution at λ. The element (i, j) in the matrix is 1 when $\hat{\Theta}^{\lambda}_{ij} \neq 0$ and $i \neq j$ and is 0 otherwise. This graph is named the estimated concentration graph.

Mazumder and Hastie (2012) prove that the vertex partition of connected components that is obtained through thresholding the covariance matrix is exactly equal to the one that is obtained from the estimated concentration graph.

Selection

To select the optimal λ in the penalized log likelihood of the graphical lasso, PROC GVARCLUS minimizes the following fit statistics over all the steps,

$$-\log\det\hat{\Theta} + \mathrm{Trace}(S\hat{\Theta}) + \sum_{1\le i<j\le p} I(\hat{\theta}_{ij} \neq 0)$$

where $\hat{\Theta} = (\hat{\theta}_{ij})_{1\le i,j\le p}$ denotes the estimates that are obtained at each step.

Displayed Output

The following sections describe the output that PROC GVARCLUS produces by default. The output is organized into various tables, which are discussed in the order of their appearance.

Number of Observations

The "Number of Observations" table displays the number of observations read from the input data table and the number of observations used in the analysis.

Cluster Summary

The "Cluster Summary" table displays the cluster structure for each step.

ODS Table Names

Each table that the GVARCLUS procedure creates has a name associated with it. You must use this name to refer to the table when you use ODS statements. The name of each table and a short description of its contents are listed in Table 17.2.

Table 17.2 ODS Tables Produced by PROC GVARCLUS

Table Name	Description	Statement	Option
NObs	Number of observations read and used; number of events and trials, if applicable	PROC GVARCLUS	Default
ClusterSummary	Summary of clustering results	PROC GVARCLUS	Default
SelectionSummary	Selection summary	PROC GVARCLUS	Default
ClassInfo	Level information for classification variables	PROC GVARCLUS	Default

Examples: GVARCLUS Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

Example 17.1: Analyzing United States Senate Voting Data

This example analyzes United States Senate voting records data from the 106th Congress 1999–2000. The data set contains 100 variables, which correspond to 100 senators. Each of the 542 samples is a bill that was put to a vote. The votes are recorded as 0 for a No vote and 1 for a Yes vote.

Download the Senate data set to the Work library from the Github repository: `https://github.com/sassoftware/sas-viya-machine-learning/tree/master/data/senate`. You can then copy the Work.Senate data set into the mylib library by using the following DATA step:

```
data mylib.senate;
    set work.senate;
run;
```

The following statements invoke the GVARCLUS procedure:

```
proc gvarclus data=mylib.senate rho=0.8 maxstep=5 outtree=mylib.tree
              outedge=mylib.edge outvert=mylib.vert;
    input _ALL_/LEVEL=NOMINAL;
run;
```

By specifying the MAXSTEP= option to be more than 1, you can let PROC GVARCLUS perform variable clustering at multiple ρ values, which decrease at each step. RHO=0.8 specifies the initial ρ value for step 1, and then ρ is 0.8^2 for step 2, 0.8^3 for step 3, and so on. The regularization parameter ρ controls the sparsity of the estimation of the inverse covariance matrix at each step, and a sequence of steps yields a hierarchical structure of clustering. The hierarchies provided by PROC GVARCLUS are interpretable with tree output. The OUTTREE= option saves the tree to a table, and you can plot the dendrogram using the SAS Graph Template language (GTL). PROC GVARCLUS also outputs the edge and vertex information for representing the partial correlations in an undirected graph. The OUTEDGE= option saves the edges in the network to a table named mylib.edge, and the OUTVERT= option saves the vertices and their sizes in the network to a table named mylib.vert.

The following statements move the tree table to the Work library:

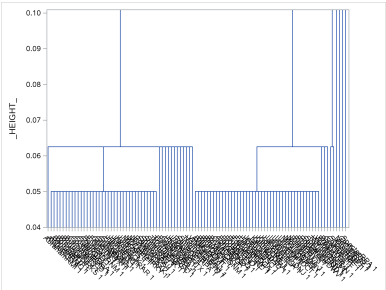
```
data work.tree (drop = last);
    set mylib.tree;
    last = scan(_CHILD_, -1);
    if last=0   then delete;
run;
```

The following code uses the DENDROGRAM statement in GTL to define the parent node, child node, and the height in the dendrogram. The dendrogram is then plotted using the SGRENDER procedure.

```
proc template;
    define statgraph dendrogram;
    begingraph;
        layout overlay/ yaxisopts=(linearopts=(viewmax=0.1));
            dendrogram nodeID=_CHILD_ parentID=_PARENT_ clusterheight=_HEIGHT_;
        endlayout;
    endgraph;
end;run;

ods graphics /width =2300px;
proc sgrender data=work.tree template=dendrogram;
run;
```

Output 17.1.1 shows that there are two large clusters in the dendrogram. All the Democratic Senate votes are united and in one cluster. Most of the Republican Senate votes are in the other cluster, but there are some small cliques.

Output 17.1.1 Senate Voting Records from the 106th Congress



Example 17.2: Output a Covariance Matrix to a SAS Data File

This example shows how to output a covariance matrix to a SAS data file. The OUTCP= option creates an output data table named mylib.cov.

The following DATA step generates a data table that has 2,000 observations and contains both interval variables (x1–x2) and nominal variables (a, c1, and c2):

```
data mylib.data1;
    array x{2};
    array c{2};
    do i=1 to 2000;
        a=int(rand('UNIFORM')*2);
        do j=1 to 2;
            x{j}=rand('UNIFORM');
            c{j}=int(rand('UNIFORM')*2);
       end;
       output;
    end;
run;
```

The following statements invoke the GVARCLUS procedure:

```
title  "Output of Covariance Matrix";

proc gvarclus data=mylib.data1 maxiter=4 outcp=mylib.cov;
    input c1 c2 /level=nominal;
    input x1 x2 /level=interval;
run;

proc print data=mylib.cov;
run;
```

Output 17.2.1 shows the content of the data file that PROC GVARCLUS generates.

Output 17.2.1 Output of Covariance Matrix

Output of Covariance Matrix

Obs	_ID_	_TYPE_	_VAR_	_vID_	v1	v2	v3	v4	v5	v6
1	1	MEAN/FREQ			0.49	0.50	1018.00	982.00	993.00	1007.00
2	2	N			2000.00	2000.00	2000.00	2000.00	2000.00	2000.00
3	3	COV	x1	v1	163.51	2.82	2.02	-2.02	-10.35	10.35
4	4	COV	x2	v2	2.82	163.96	3.15	-3.15	-1.05	1.05
5	5	COV	c1 0	v3	2.02	3.15	499.84	-499.84	-1.44	1.44
6	6	COV	c1 1	v4	-2.02	-3.15	-499.84	499.84	1.44	-1.44
7	7	COV	c2 0	v5	-10.35	-1.05	-1.44	1.44	499.98	-499.98
8	8	COV	c2 1	v6	10.35	1.05	1.44	-1.44	-499.98	499.98

The _VAR_ column displays the names of all variables and the levels of the nominal variables. Assuming that you have *n* effects (the total number of interval variables and the levels of categorical variables), the _vID_ column contains *n* markers, v1 to v*n*, where v*i* denotes the *i*th effect. The column _TYPE_ defines

the role of each row. When the _TYPE_ column displays MEAN/FREQ, the corresponding row contains either the mean for an interval variable or the frequency for a level of a nominal variable. When the _TYPE_ column displays N, the corresponding row contains the number of samples. And when the _TYPE_ column displays COV, the corresponding row contains a row of the covariance matrix. In this example, the covariance matrix is 4×4, and it resides in the table in rows 3–6 and columns 7–10.

Example 17.3: Output the Covariance Matrix in LIL Format

This example shows how to output a covariance matrix in list-of-list (LIL) format.

The following DATA step generates a data table that has 2,000 observations and contains both interval variables (x1–x2) and a categorical variable (a):

```
data mylib.data2;
    array x{2};
    do i=1 to 2000;
        a=int(rand('UNIFORM')*2);
        do j=1 to 2;
            x{j}=rand('UNIFORM');
        end;
        output;
    end;
run;
```

The following statements invoke the GVARCLUS procedure:

```
title  "Output of Covariance Matrix in LIL Format";

proc gvarclus data=mylib.data2 maxiter=4 outcp=mylib.cov_lil/list eps=0.01;
    input a /level=nominal;
    input x1-x2 /level=interval;
run;

proc print data=mylib.cov_lil;
run;
```

The OUTCP= option creates an output data table named mylib.cov_lil, which contains the covariance matrix. Output 17.3.1 shows the correlation matrix in LIL format.

Output 17.3.1 Output of Correlation Matrix in LIL Format

Output of Covariance Matrix in LIL Format

Obs	_TYPE_	_ID_	_NAME1_	_NAME2_	_VAL_
1	S	1	samples		2000.00
2	S	2	nVar		3.00
3	S	3	nEff		4.00
4	M	1	x1		0.49
5	M	2	x2		0.49
6	F	3	a 0		1022.00
7	F	4	a 1		978.00
8	V	1	1	1	162.69
9	V	2	2	1	7.60
10	V	3	2	2	162.63
11	V	4	3	1	-12.08
12	V	5	3	2	3.23
13	V	6	3	3	499.76
14	V	7	4	1	12.08
15	V	8	4	2	-3.23
16	V	9	4	3	-499.76
17	V	10	4	4	499.76

The column _TYPE_ defines the type of each row:

- When the _TYPE_ column displays S, the corresponding row contains the statistics of the data table. More specifically, when the _TYPE_ column displays S and the _NAME1_ column displays samples, the _VAL_ column in the corresponding row contains the number of samples in the data table. Similarly, when the _TYPE_ column displays S and the _NAME1_ column displays nVar, the _VAL_ column contains the number of variables. And when the _TYPE_ column displays S and the _NAME1_ column displays nEff, the _VAL_ column in the corresponding row contains the number of effects.
- When the _TYPE_ column displays F, the row contains the frequency of a level of a nominal variable. In this case, the _NAME1_ column contains the name and level of the nominal variable.
- When the _TYPE_ column displays M, the row contains the mean of an interval variable. In this case, the _NAME1_ column contains the name of the variable and the _NAME2_ column is empty.
- When the _TYPE_ column displays R, the row contains an entry in the correlation matrix. In this case, the _NAME1_ column contains the row ID, the _NAME2_ column contains the column ID, and the _VAL_ column contains the value.
- When the _TYPE_ column displays V or P, the corresponding row contains an entry of a COV matrix or an SSCP matrix, respectively.

Only entries in the lower triangle of the correlation matrix are written to the file, because the correlation matrix is symmetric. Also because EPS=0.01 is specified, any entry of the matrix whose value is smaller than 0.01 is ignored in the output; this saves storage space.

References

Banerjee, O., El Ghaoui, L., and d'Aspremont, A. (2008). "Model Selection through Sparse Maximum Likelihood Estimation for Multivariate Gaussian or Binary Data." *Journal of Machine Learning Research* 9:485–516.

Friedman, J. H., Hastie, T., and Tibshirani, R. (2008). "Sparse Inverse Covariance Estimation with the Graphical Lasso." *Biostatistics* 9:432–441.

Mazumder, R., and Hastie, T. (2012). "Exact Covariance Thresholding into Connected Components for Large-Scale Graphical Lasso." *Journal of Machine Learning Research* 13:781–794.

Chapter 18
KPCA Procedure

Contents

Overview: KPCA Procedure . . . **476**
- PROC KPCA Features . . . 476
- Using SAS Viya Workbench . . . 477

Getting Started: KPCA Procedure . . . **477**

Syntax: KPCA Procedure . . . **481**
- PROC KPCA Statement . . . 482
- DISPLAY Statement . . . 483
- DISPLAYOUT Statement . . . 484
- ID Statement . . . 485
- INPUT Statement . . . 485
- KERNEL Statement . . . 485
- LRAPPROXIMATION Statement . . . 486
- OUTPUT Statement . . . 487
- PREIMAGE Statement . . . 488
- SAVESTATE Statement . . . 489

Details: KPCA Procedure . . . **490**
- Details: The Exact Method . . . 490
- Low-Rank Approximation of KPCA Training Using the Nyström Method . . . 491
- Automatic Gaussian Kernel Bandwidth Selection of KPCA . . . 492
- Fast Scoring of KPCA . . . 492
- KPCA Iterative Pre-image Method . . . 493
- KPCA Mapping Pre-image Method . . . 494
- Displayed Output . . . 495
- ODS Table Names . . . 495
- Output Data Tables . . . 496

Example: KPCA Procedure . . . **497**
- Example 18.1: Letter Recognition . . . 497
- Example 18.2: Denoising Concentric Circles . . . 500
- Example 18.3: Denoising USPS Digits . . . 501

References . . . **508**

Overview: KPCA Procedure

Kernel principal component analysis (kernel PCA) is a nonlinear form of principal component analysis (Schölkopf, Smola, and Müller 1998). It uses the same basic idea as PCA; that is, it seeks to project the set of data onto a low-dimensional subspace that captures the highest possible amount of variance in the data. Whereas PCA performs a linear projection of the data onto a subset of the original space, kernel PCA uses a mapping function to embed the data in a high-dimensional reproducing kernel Hilbert space (RKHS) called $\mathcal{F}$ by a linear dimensionality reduction through the "kernel trick" in that space. Different kernels correspond to different mapping functions. This way, a nonlinear manifold (with respect to the original input space) can be found that contains the data. The applications of kernel PCA include nonlinear dimensionality reduction, nonlinear data classification, kernel principal component regression, image denoising, novelty detection, and so on.

The KPCA procedure performs kernel principal component analysis and stores the results in output tables that are produced by the EIGENVAL=, EIGENVEC=, and CENTROIDS= options. Also, the projection of the training data into the kernel principal components is stored by using the OUT= option in the OUTPUT statement.

PROC KPCA Features

The KPCA procedure has the following features:

- is multithreaded during all phases of analytic execution
- supports large-scale training data
- enables you to choose among linear, polynomial, and radial basis function (Gaussian) kernels
- enables you to perform fast training and fast scoring of KPCA models
- enables you to perform the analysis on selected columns of the data
- enables you to project new data onto principal components that have captured the nonlinear relationship in the data
- enables you to compute the pre-image of the points in the kernel principal component space back into the original input space

Using SAS Viya Workbench

SAS Viya Workbench has greatly simplified the SAS programming experience by offering a syntax style nearly identical to that of SAS 9 and earlier releases. However, there are minor exceptions. In the past, SAS Viya procedures required the use of a CAS engine libref. This engine served multiple purposes, but it was primarily used to upload data to the CAS server and manage the data. Its additional purposes included support of new data types and improved performance for large data sets. No CAS server is needed for SAS Viya Workbench, because all computations are done in place for you in a single location.

In the following program, the LIBNAME statement creates a mylib libref that the examples in this chapter use to refer to your current or desired work directory:

```
libname mylib ".";
```

For more information, see *SAS Viya Workbench: Getting Started*.

Getting Started: KPCA Procedure

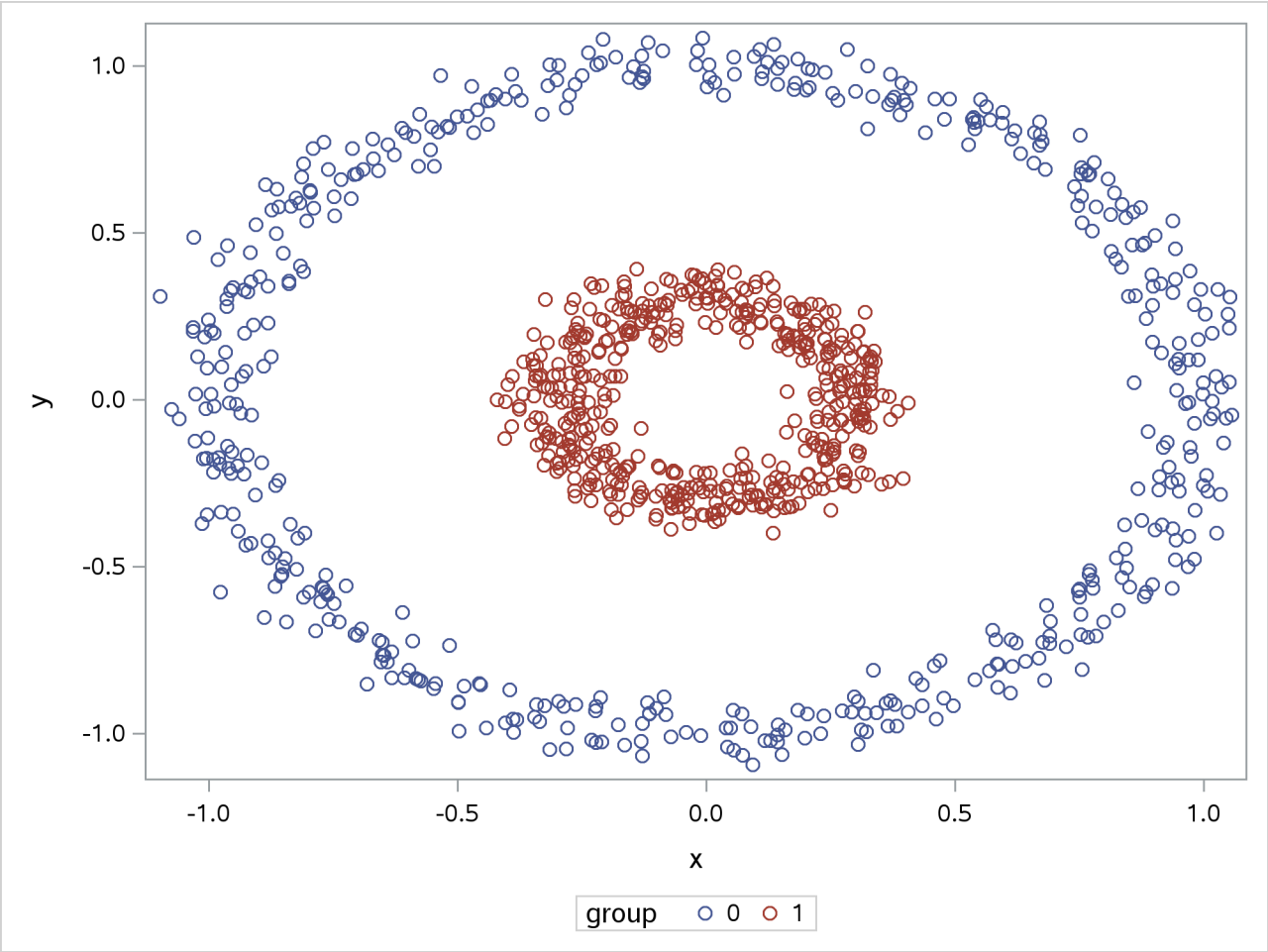
NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

This example shows how to use the KPCA procedure to project two concentric data sets onto the first two kernel principal components. The following code shows how to generate these data sets by using the IML procedure and plot it by using the SGPLOT procedure. The output is shown in Figure 18.1.

```
proc iml;
start Linspace(a, b, n);
   if n<2 then return( b );
   incr = (b-a) / (n-1);
   return( do(a, b, incr) );
finish;

start makecircles(X, y, n_samples, noise, random_state, factor);

   pi = constant("pi");

   lins=Linspace(0, 2*pi, floor(n_samples/2)+1);
   n_2=floor(n_samples/2);
   lins=remove(lins,n_2+1);
   outer_circ_x=cos(lins);
   outer_circ_y=sin(lins);
   inner_circ_x = outer_circ_x#factor;
   inner_circ_y = outer_circ_y#factor;

   X=(outer_circ_x||inner_circ_x)`||(outer_circ_y||inner_circ_y)`;
   y=(j(1,n_2,0)||j(1,n_2,1))`;
   Xy=X||y;
```

```
    /* shuffle observations */
    call randseed(random_state);
    /* sample size=5, rows chosen from 1:NumRows */
    obsIdx = sample(1:nrow(Xy), nrow(Xy), "NoReplace");
        X=Xy[obsIdx,1:ncol(Xy)-1];
        y=Xy[obsIdx,ncol(Xy)];
        Xn=j(nrow(X),2);
        if noise^=. then
        call randgen(Xn, "Normal", 0, noise);
        X=X+Xn;

finish;


 n_samples=1000; /*sample size*/
 noise=0.05;
 random_state=0;
 factor=0.3;  /*radius ratio of inner circle to the outer circle*/

 run makecircles(X, y, n_samples, noise, random_state, factor);

 circles=X||y;

 create  circles from circles[colname={"x" "y" "group"}];
 append from circles;
 close circles;

quit;

 proc sgplot data=circles;
   styleattrs datasymbols=(Circle X);
   scatter x=x y=y / group=group markerattrs=(size=7px);
 run;
```

Figure 18.1 Plot of the Two Concentric Circles



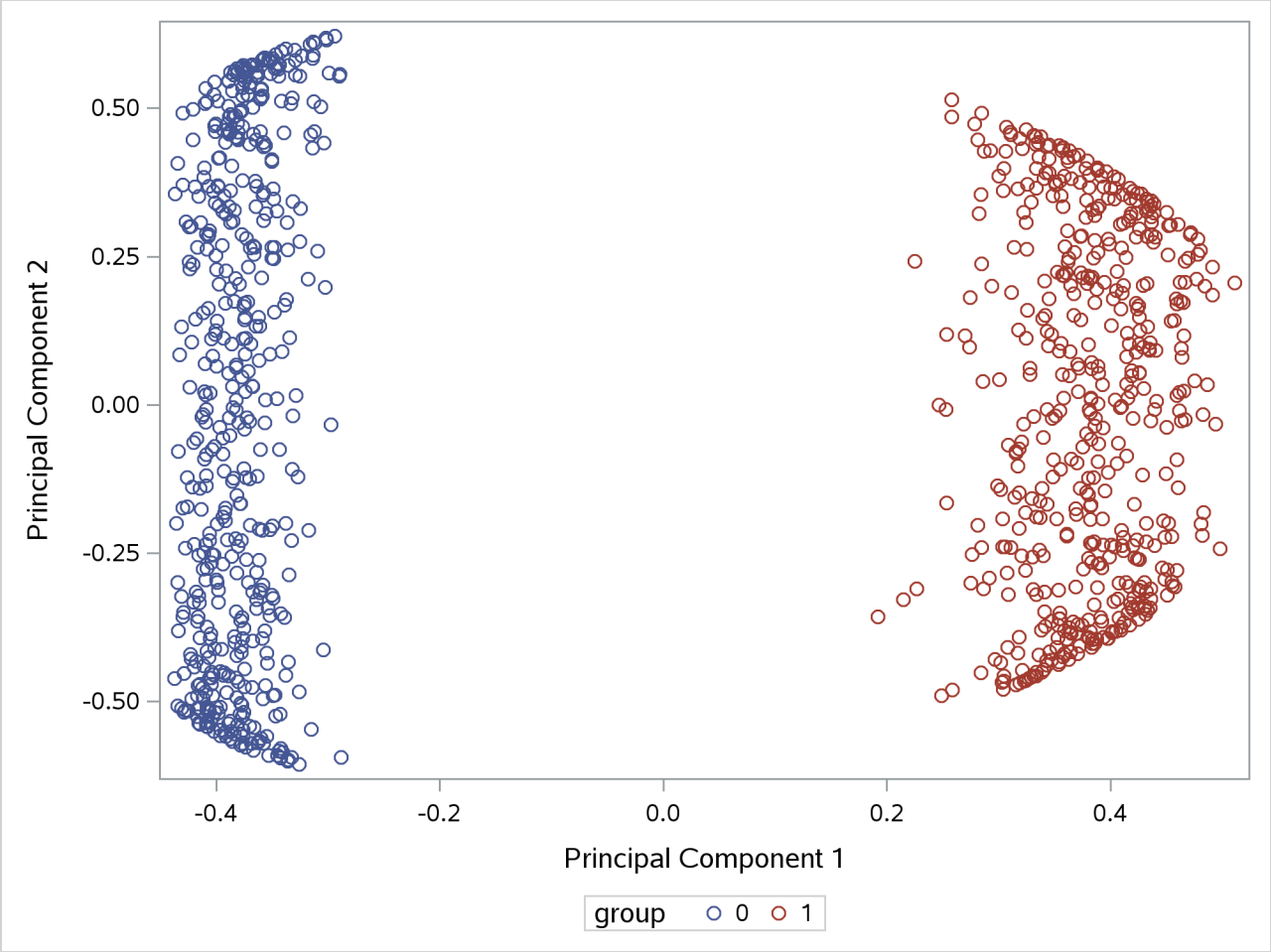
You can load the Circles data table into your SAS library and by using your libref in the first statement of the following DATA step:

```
data mylib.circles;
set circles;
run;
```

The following code uses the KPCA procedure to project the data in the Circles data table onto the first two kernel principal components and then use PROC SGPLOT to plot these data in the reduced dimensions. The output is shown in Figure 18.2.

```
proc KPCA data=mylib.circles method=exact;
   input x y;
   output out=mylib.scored copyvar=group npc=2;
run;

proc sgplot data=mylib.scored;
   styleattrs datasymbols=(Circle X);
   scatter x=_PCS_1 y=_PCS_2 / group=group markerattrs=(size=7px);
run;
```

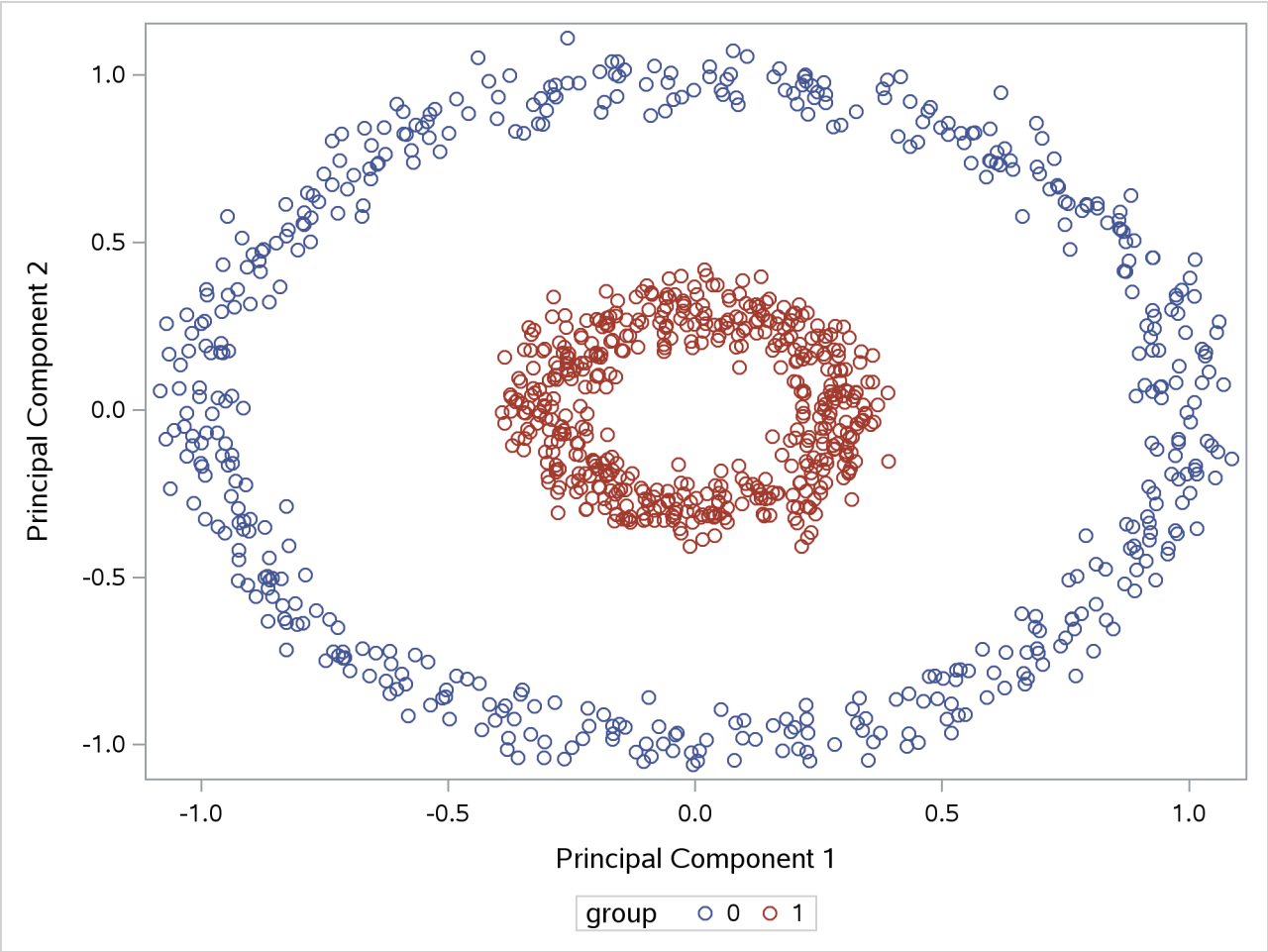
Figure 18.2 Projection of the Two Concentric Circles onto the First Two Kernel Principal Components



In Figure 18.2, you can see that after you project the data onto the first two kernel principal components, the two circles are linearly separable. However, as you can see in Figure 18.3, the projection of the data onto the first two regular principal components is not linearly separable. (Note that when you use the linear kernel type, a regular PCA is performed.) The following code runs the regular PCA on the Circles data table:

```
proc KPCA data=mylib.circles method=exact;
   input x y;
   kernel linear;
   output out=mylib.scoredpca copyvar=group npc=2;
run;

proc sgplot data=mylib.scoredpca;
   styleattrs datasymbols=(Circle X);
   scatter x=_PCS_1 y=_PCS_2 / group=group markerattrs=(size=7px);
run;
```

Figure 18.3 Projection of the Two Concentric Circles onto the First Two Regular Principal Components



Syntax: KPCA Procedure

The following statements are available in the KPCA procedure:

PROC KPCA < *options* > ;
  DISPLAY < *table-list* > < /*options* > ;
  DISPLAYOUT *table-spec-list* < /*options* > ;
  ID *variables* ;
  INPUT *variables* ;
  KERNEL *kernel-type* / *kernel-parameters* ;
  LRAPPROXIMATION < *options* > ;
  OUTPUT OUT=*libref.data-table* < *option* > ;
  PREIMAGE < *options* > ;
  SAVESTATE RSTORE=*libref.data-table* < *option* > ;

The INPUT statement is required. The following sections describe the PROC KPCA statement and then describe the other statements in alphabetical order.

PROC KPCA Statement

PROC KPCA < *options* > **;**

The PROC KPCA statement invokes the procedure. Table 18.1 summarizes the *options* available in the PROC KPCA statement.

Table 18.1 PROC KPCA Statement Options

Option	Description
Input Data Table Option	
DATA=	Specifies the input data table
KPCA Options	
CENTER	Centers the input data
METHOD=	Specifies the method of performing KPCA
NTHREADS=	Specifies the number of threads to use on each computation node
RANKTHRESHOLD=	Specifies the epsilon value to determine the rank of the kernel matrix
SCALE	Scales the input data
Output Table Options	
CENTROIDS=	Specifies the name of the centroids output table
EIGENVAL=	Specifies the name of the eigenvalue output table
EIGENVEC=	Specifies the name of the eigenvector output table

You can specify the following *options*:

CENTER
centers the observations by the mean of each column.

CENTROIDS=*libref.data-table*
specifies the name of the output table for the centroids that are generated by clustering. This table is generated if METHOD=APPROXIMATE. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 477.

DATA=*libref.data-table*
names the input data table for PROC KPCA to use. The default is the most recently created data table. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to the data. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 477.

data-table specifies the name of the input data table.

EIGENVAL=*libref.data-table*
specifies the name of the output table that contains the eigenvalues. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 477.

EIGENVEC=*libref.data-table*
specifies the name of the output table that contains the eigenvectors. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 477.

METHOD=APPROXIMATE | EXACT
specifies the method to use in performing KPCA. You can specify the following values:

APPROXIMATE uses the low-rank approximation method.

EXACT uses the exact method.

By default, METHOD=APPROXIMATE.

NTHREADS=*number-of-threads*
specifies the number of threads per computation node. The default value is the lesser of 16 and the number of threads available per computation node.

RANKTHRESHOLD=*number*
specifies the value of epsilon for determining the rank of the kernel matrix. The default *number* is 10^{-8}.

SCALE
scales the observations by the standard deviation of each column. If a constant variable exists (where all observations have the same value), the observations of this column are not scaled.

DISPLAY Statement

DISPLAY < *table-list* > < / *options* > **;**

The DISPLAY statement enables you to specify a list of display tables to display or exclude. This statement is similar to the ODS SELECT, ODS EXCLUDE, and ODS TRACE statements. However, the DISPLAY statement can improve performance when a large number of tables could be generated (such as in BY-group processing). The DISPLAY statement produces only the requested tables, but ODS statements generate all the tables and then keep the requested tables.

If you use both DISPLAY and ODS statements together, the DISPLAY statement takes precedence over the ODS statements. Note that the ODS EXCLUDE statement processes tables after they have been filtered by the DISPLAY statement. In some cases, it might appear that the ODS EXCLUDE statement is taking precedence because it can further filter the tables. For more information about ODS, see *SAS Output Delivery System: Procedures Guide*.

You can specify the *table-list* as a list of table names, paths, partial pathnames, and regular expressions.

The table names that you can specify are listed in the section "ODS Table Names" on page 495. A path is a table name that is prefixed with dot-separated grouping information. For example, a SelectionSummary table that a procedure produces during a selection routine might have the path *Bygroup1.Summary.SelectionSummary*. A partial pathname does not include all groups; for example, *SelectionSummary* and *Summary.SelectionSummary* are partial pathnames for *Bygroup1.Summary.SelectionSummary*.

When you specify a table name or partial pathname, all display tables whose paths end in the specified name are selected for display or exclusion. For example, both *SelectionSummary* and *Summary.SelectionSummary* select *Bygroup1.Summary.SelectionSummary*.

A regular expression is enclosed in forward slashes (/). For example, specifying "/tions/" selects all pathnames that contain the substring "tions"; in particular, the *Bygroup1.Summary.SelectionSummary* table is selected. Specifying "!/tions/" selects all pathnames that do not contain the substring "tions"; in particular, the *Bygroup1.Summary.SelectionSummary* table is not selected.

You can specify the following *options* after a slash (/):

CASESENSITIVE
performs a case-sensitive comparison of table names in the *table-list* to display table names when tables are subsetted for display. To preserve case, you must enclose table names in the *table-list* in quotation marks.

EXCLUDE
displays all display tables except those that you specify in the *table-list*.

EXCLUDEALL
suppresses display of all tables. This option takes precedence over the other options.

TRACE
displays the display table names, labels, and paths.

DISPLAYOUT Statement

DISPLAYOUT *table-spec-list* < / *options* > **;**

The DISPLAYOUT statement enables you to create output tables from your displayed output. This statement is similar to the ODS OUTPUT statement. For more information about ODS, see *SAS Output Delivery System: Procedures Guide*.

The *table-spec-list* specifies a list of output tables to create. Each entry in the list has either a *key*=*value* format or a *key* format:

key=*value* specifies *key* as the ODS table name, path, or partial pathname, and specifies *value* as the output table name.

key specifies *key* as the ODS table name and also as the output table name.

The ODS table names that you can specify are listed in the section "ODS Table Names" on page 495. You cannot specify the ODS table named OutputCasTables in the *table-spec-list*.

Table names and partial pathnames are discussed under the DISPLAY statement. The DISPLAYOUT statement does not support regular expressions.

You can specify the following *options* after a slash (/):

INCLUDEALL
creates output tables for all display tables. The name of the created output table is the same as the name of the corresponding display table. If you specify this option, the *table-spec-list* specification is ignored.

NOREPLACE
does not replace any existing output table of the same name.

ID Statement

ID *variables* **;**

The ID statement lists one or more variables to be copied from the input data table to the output data tables that are specified in the OUTPUT statement and the EIGENVEC= option in the PROC KPCA statement, and also to the file that is specified in the RSTORE= option in the SAVESTATE statement.

INPUT Statement

INPUT *variables* **;**

The INPUT statement specifies the names of *variables* to be considered in the KPCA procedure. The procedure accepts numeric variables as input. Nonnumeric variables are ignored. This statement is required.

KERNEL Statement

KERNEL *kernel-type / kernel-parameter* **;**

The KERNEL statement specifies the type of kernel and any associated parameters to use during training. You can specify one of the following *kernel-types* and its associated *kernel-parameters*:

LINEAR
specifies that the kernel type is linear. When the kernel type is linear, regular PCA is performed.

POLYNOMIAL / **DEGREE=***number* **INTERCEPT=***number*
specifies that the kernel type is in the polynomial form. The degree of the polynomial is a positive integer value that is specified in the DEGREE= option. By default, DEGREE=3. The intercept is a numeric value that is specified in the INTERCEPT= option. By default, INTERCEPT=1.

RBF / **BW=***s*

RBF / **BW=RANDOMCMSE(SEED=***number* **NSAMPLE=***number* **NPASS=***number***)**

RBF / **BW=SCMSE(SEED=***number* **NCLUS=***number* **CENTROIDS=***libref.data-table***)**
uses a radial basis function (also known as the Gaussian kernel function) during training. The kernel is defined as

$$K(x, y) = \exp \frac{-\|x - y\|^2}{2s^2}$$

where x and y are vectors and s is the bandwidth parameter. You can specify the bandwidth parameter in the form BW=*s*, where *s* is a positive nonzero real number.

The processing time usually increases as the value of *s* decreases. At very small values of *s*, the processing time can be very long.

To use random CMSE, specify the bandwidth parameter in the form

BW=RANDOMCMSE(SEED=*number* NSAMPLE=*number* NPASS=*number*)

where SEED=*number* specifies the value of seed, NSAMPLE=*number* specifies number of random samples to use in the random CMSE method, and NPASS=*number* specifies the number of times to repeat the process to tune the bandwidth. The values of the SEED=, NSAMPLE=, and NPASS= suboptions are integers. The values of the SEED= and NPASS= suboptions are nonnegative. The value of the NSAMPLE= suboption is greater than 1.

To use scalable CMSE, specify the bandwidth parameter in the form

BW=SCMSE(SEED=*number* NCLUS=*number* CENTROIDS=*libref.data-table*)

where SEED=*number* specifies the value of seed that is used in choosing initial centroids for k-means clustering, NCLUS=*number* specifies the number of clusters used in k-means clustering, and CENTROIDS=*libref.data-table* specifies the name of the output table for the centroids that are generated by clustering. For more information about this method, see the section "Automatic Gaussian Kernel Bandwidth Selection of KPCA" on page 492.

By default, the value of BW = $\sqrt{\frac{0.25}{n(n-1)}\sum_{i,j=1}^{n}\left\|x_i - x_j\right\|^2}$, where n is the number of rows in the training data. This choice of default value comes from an idea in Kwok and Tsang (2004). The default parameter is not guaranteed to produce good results. It is recommended that you tune this parameter by using some systematic approach, such as cross validation.

LRAPPROXIMATION Statement

LRAPPROXIMATION < *options* > **;**

The LRAPPROXIMATION statement specifies the parameters of the low-rank approximation method.

You can specify the following *options*:

CCRITERION=*number*
specifies a double value for the convergence criterion in k-means clustering. By default, CCRITERION=0.02.

CLUSMETHOD=FC | KMPP | RANDOM
specifies the method of seed initialization in k-means clustering to generate the centroids. You can specify one of the following methods:

FC
stands for fast clustering, which spreads out the initial seeds in a heuristic way.

KMPP / **<RANDOMSEED=***number***>**
spreads out the initial seeds by introducing a weighted probability distribution in seed selection. For more information, see Arthur and Vassilvitskii (2007).

RANDOM / **<RANDOMSEED=***number* **NRESTART=***number***>**
selects initial seeds by pure randomization.

Note that if CLUSMETHOD=KMPP or RANDOM, you can specify an integer value for the RANDOMSEED= option. If you do not specify a value for the RANDOMSEED= option, the seed is generated by reading the time of day from the computer's clock. If you want reproducible results, specify an integer for *number*. Also, if CLUSMETHOD=RANDOM, you can specify the number of restarts by using the NRESTART= option, whose value should be an integer. By default, NRESTART=5. Also, by default, CLUSMETHOD=KMPP.

MAXCLUS=*number*
specifies the maximum number of clusters to use in *k*-means clustering. By default, MAXCLUS=100.

MAXITER=*number*
specifies the maximum number of iterations to use in *k*-means clustering. By default, MAXITER=50.

OUTPUT Statement

OUTPUT OUT=*libref.data-table* < *option* > **;**

The OUTPUT statement creates an output data table to contain the results of the procedure run.

You must specify the following option:

OUT=*libref.data-table*
names the output data table for PROC KPCA to use. You must specify this option before any other options. *libref.data-table* is a two-level name, where

libref	refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to where the data table is to be stored. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 477.
data-table	specifies the name of the output data table.

You can also specify the following *option*:

NPC=*number*
specifies the number of principal components to display.

PREIMAGE Statement

PREIMAGE < *options* > **;**

The PREIMAGE statement specifies the pre-image method and any associated parameters to use during training. For more information about pre-image methods, see the sections "KPCA Iterative Pre-image Method" on page 493 and "KPCA Mapping Pre-image Method" on page 494. You can specify the following *options*:

METHOD=MAP(< *options* >**) | ITERATIVE**
specifies the method to use for pre-image training. You can specify one of the following values:

ITERATIVE
uses the iterative method to find the pre-image.

MAP(< *options* >**)**
uses the mapping method to find the pre-image. You can specify the following *options*:

CENTROIDSPC=*libref.data-table*
specifies the name of the output table to contain the centroids of the points in the principal component space that are generated by k-means clustering. This table is generated when you specify METHOD=APPROXIMATE in the PROC KPCA statement. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 477.

KERNEL=LINEAR | POLYNOMIAL(< *options* >**) | RBF(**< *option* >**)**
specifies the kernel type for the kernel ridge regression to be used in the mapping method. You can specify one of the following *options*:

LINEAR
uses a linear kernel for the kernel ridge regression in the mapping method.

POLYNOMIAL(DEGREE=*number* **INTERCEPT=***number***)**
specifies that the kernel type in the kernel ridge regression of the mapping method is in the polynomial form. The degree of the polynomial is a positive integer value that is specified in the DEGREE= option. By default, DEGREE=3. The intercept is a numeric value that is specified in the INTERCEPT= option. By default, INTERCEPT=1.

RBF(BW=*number***)**
uses a radial basis function (also known as the Gaussian kernel function) in the kernel ridge regression of the mapping method. The kernel is defined as

$$K(x, y) = \exp \frac{-\|x - y\|^2}{2s^2}$$

where x and y are vectors and s is the bandwidth parameter. You can specify the bandwidth parameter in the form BW=*s*, where *s* is a positive nonzero real number. The processing time usually increases as the value of *s* decreases. At very small values of *s*, the processing time can be very long.

By default, the bandwidth parameter value is $\sqrt{\frac{0.25}{n(n-1)}\sum_{i,j=1}^{n}\left\|x_i - x_j\right\|^2}$, where n is the number of rows in the training data set. The default value of the BW= option is calculated in the same way as the value described in the KERNEL statement. Note that the default parameter is not guaranteed to produce good results. It is recommended that you tune this parameter by using some systematic approach, such as cross validation.

L2=*number*
specifies a numeric value for the L2 regularization parameter of the kernel ridge regression to be used in the mapping method. By default, L2=1.

MAPCOEFFS=*libref.data-table*
specifies the name of the output table to contain the coefficients of the pre-image mapping method. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 477.

By default, METHOD=ITERATIVE.

NPC=*number*
specifies a positive integer value as the number of principal components to use in pre-image training. For the mapping method, it is also the number of principal components to use in pre-image scoring.

SAVESTATE Statement

SAVESTATE RSTORE=*libref.data-table* < *option* > **;**

The SAVESTATE statement creates an analytic store for the model and saves it as a binary object in a data table. You can use the analytic store in the ASTORE procedure to score new data. For more information, see Chapter 4, "ASTORE Procedure."

You must specify the following option:

RSTORE=*libref.data-table*
specifies the name of the table in which to save the analytic store. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 477.

You can also specify the following *option*:

EXACTSCORE
specifies the exact method of scoring. If this option is not specified and METHOD=APPROXIMATE in the PROC KPCA statement, then fast (approximate) scoring is used. However, if this option is not specified and METHOD=EXACT in the PROC KPCA statement, then exact scoring is used. For more information about fast scoring, see the section "Fast Scoring of KPCA" on page 492.

You can use the ASTORE procedure to specify the number of principal components to project the scoring observations. To do this, specify the option KPCA_NPC in the SETOPTION statement in PROC ASTORE. If you specify the PREIMAGE statement, you can use PROC ASTORE to obtain the pre-image of the data in

the original input space by specifying the value of SCORING_MODE as 1 in the SETOPTION statement in PROC ASTORE. You can also specify other pre-image scoring options, including ITER_NPC, ITER_NUM, and ITER_THRESH, in the SETOPTION statement in PROC ASTORE. For more information, see the sections "KPCA Iterative Pre-image Method" on page 493 and "KPCA Mapping Pre-image Method" on page 494.

Details: KPCA Procedure

Kernel principal component analysis (KPCA) is a nonlinear form of principal component analysis. In this section, the methods that PROC KPCA implements are described.

Details: The Exact Method

This section describes the exact KPCA method. For more information, see Schölkopf, Smola, and Müller (1998).

Suppose that the mean of the data in a reproducing kernel Hilbert space (RKHS) is

$$\mu = \frac{1}{n}\sum_{i=1}^{n}\phi(x_i) = 0$$

where x_i is a $q \times 1$ data point in the original space and $\phi(\cdot)$ is the mapping function that maps the original data to some high-dimensional RKHS. Then, assuming that the data vector x_i is already centered, the covariance matrix is

$$\mathrm{C} = \frac{1}{n}\sum_{i=1}^{n}\phi(x_i)\phi(x_i)^T$$

The eigensystem of C is

$$\mathrm{C}v = \lambda v$$

Schölkopf, Smola, and Müller (1998) show that the eigenvectors can be expressed as a linear combination of features in RKHS:

$$v = \sum_{i=1}^{n}\alpha_i\phi(x_i)$$

Therefore, finding the eigenvectors is equivalent to finding the coefficients α_i.

By substituting the preceding equation into the equation immediately before it, for the jth eigenvector, you get

$$\frac{1}{n}\sum_{i=1}^{n}\phi(x_i)\phi(x_i)^T\left(\sum_{l=1}^{n}\alpha_{jl}\phi(x_l)\right) = \lambda_j\sum_{l=1}^{n}\alpha_{jl}\phi(x_l)$$

Then, plugging in the kernel and rearranging the equation, you get

$$K^2\boldsymbol{\alpha}_j = n\lambda_j K\boldsymbol{\alpha}_j$$

where the $n \times n$ matrix K is called the kernel matrix, whose klth element is the value of kernel function $k(x_k, x_l) = \phi(x_k)^T\phi(x_l)$; and $\boldsymbol{\alpha}_j$ is an $n \times 1$ vector, whose lth element is the coefficient α_{jl}.

Next, factor K from both sides is removed, and finally the eigensystem can be written in terms of kernel matrix K as

$$K\boldsymbol{\alpha}_j = n\lambda_j\boldsymbol{\alpha}_j$$

Then, a normalization condition is applied such that the eigenvector v_j in the RKHS satisfies

$$v_j^T v_j = 1 \Longrightarrow \sum_{k=1}^{n}\sum_{l=1}^{n}\alpha_{jl}\alpha_{jk}\phi(x_l)^T\phi(x_k) = 1 \Longrightarrow \boldsymbol{\alpha}_j^T K\boldsymbol{\alpha}_j = 1$$

By left-multiplying $K\boldsymbol{\alpha}_j = n\lambda_j\boldsymbol{\alpha}_j$ by $\boldsymbol{\alpha}_j^T$ and using the normalization condition, you get

$$n\lambda_j\boldsymbol{\alpha}_j^T\boldsymbol{\alpha}_j = 1 \qquad \forall j$$

The projection of a new or old data point x onto the jth principal component is

$$\phi(x)^T v_j = \sum_{i=1}^{n}\alpha_{ji}\phi(x)^T\phi(x_i) = \sum_{i=1}^{n}\alpha_{ji}K(x, x_i)$$

Low-Rank Approximation of KPCA Training Using the Nyström Method

To train an exact KPCA model on the input matrix M of size $n \times m$, the full kernel matrix $K \in \mathbb{R}^{n\times n}$ needs to be constructed, and the computationally expensive eigendecomposition operation, with a computational complexity of $\mathcal{O}(n^3)$, must be applied to K. Thus, for large values of n, the exact KPCA method can be very slow. To address this issue, an approximation method called the Nyström method was proposed by Baker (1977). This method is based on sampling from the rows of the input matrix. Instead of applying the eigendecomposition method to the full kernel matrix K, you apply it to a smaller matrix of size $c \times c$, where c is the sampling size.

The sampling scheme is important in the performance of the Nyström method. In Kwok and Tsang (2004), an error analysis is conducted, and the resulting error bound suggests using k-means clustering as the sampling scheme. The greater the number of clusters that are chosen in k-means clustering, the more accurate the reconstruction of K will be. If the number of clusters equals the number of observations, the reconstruction will be exact. In k-means clustering, the algorithm for choosing the initial values (or seeds) has significant impact on the centroids that are obtained. PROC KPCA includes several algorithms for clustering initialization, among them k-means++ (Arthur and Vassilvitskii 2007). Low-rank approximation reduces the memory complexity of KPCA training from $\mathcal{O}(n^2)$ to $\mathcal{O}(nc)$ and reduces the computational complexity of kernel PCA from $\mathcal{O}(n^3)$ to $\mathcal{O}(nc^2)$, where n is the data size and c is the number of clusters.

Automatic Gaussian Kernel Bandwidth Selection of KPCA

For KPCA with Gaussian kernel, the choice of the kernel bandwidth is extremely important. Bandwidth values that are either too large or too small fail to serve the purpose of data dimension reduction. The bandwidth selection method that the KPCA procedure implements is called the criterion of maximum sum of eigenvalues (CMSE). It is known that KPCA expands the features by nonlinear transformations and then applies PCA in the RKHS. It is also commonly acknowledged that when PCA is applied to a covariance matrix, the first k eigenvalues (k should be large enough to identify a major variance pattern) explain the majority of variance underlying the data. For KPCA, each kernel bandwidth corresponds to a unique RKHS, and the first k eigenvalues explain a certain amount of variance of features in a certain RKHS. The key idea of the CMSE method is thus that for a given value of k, the bandwidth that has the maximum sum of the first k largest eigenvalues is the ideal bandwidth, because it tends to explain the largest amount of variance in the data set.

Although the CMSE method provides a systematic way to perform Gaussian kernel bandwidth selection for KPCA, the time that it takes to apply CMSE to an even moderately large data table is still too long, given that the time complexity of eigendecomposition is $\mathcal{O}(n^3)$, where n is the size of the data table. To make the CMSE method useful in practical applications that involve moderate to large data tables, the KPCA procedure provides two scalable and fast implementations of the CMSE method:

- **Random CMSE** works by randomly sampling the data and running the CMSE method on the random sample instead of the full input data table. Because this approach is faster than the CMSE method, it can be repeated through multiple passes in the hope of achieving better results—that is, a bandwidth value that produces a higher sum of eigenvalues.
- **Scalable CMSE** borrows the idea of low-rank approximation from KPCA training. It works by using k-means clustering to select the centroids of the data set and then find the low-rank approximation of the original kernel matrix. Then, an eigendecomposition can be calculated on a much smaller kernel matrix of size $c \times c$, where c is the sampling size (that is, the number of centroids).

Fast Scoring of KPCA

In addition to enabling fast and memory-efficient KPCA training, the low-rank approximation method also provides a fast and memory-efficient way to do scoring—namely, calculating KPCA score values for test data. In exact KPCA, to score test data the whole training data set needs to be retained and the kernel matrix between test data and training data needs to be calculated. However, in the low-rank approximation method, only the centroids that result from the k-means clustering of the training data need to be retained, and only the kernel matrix between the test data and these centroids needs to be calculated. This reduces both the amount of memory required and the computational complexity of KPCA scoring from $\mathcal{O}(n)$ to $\mathcal{O}(c)$, where n is the data set size and c is the number of clusters.

KPCA Iterative Pre-image Method

Although the mapping $\phi(\cdot)$ from input space to feature space is of primary importance in KPCA, the reverse mapping from feature space back to input space, known as the "pre-image" technique, is useful as well. For example, the pre-image technique proves to be useful in pattern denoising. The rationale behind this is that the directions of eigenvectors that correspond to small eigenvalues are assumed to contain only information about noise, because small eigenvalues suggest small variances. In contrast, leading eigenvectors that correspond to large eigenvalues are assumed to contain relevant pattern information. The pre-image of linear PCA can also be applied to the denoising problem; however, the linear limitation makes it impossible to identify noise in a nonlinear pattern. The way that kernel PCA denoises the data involves first applying linear PCA in the feature space and then implementing denoising by projecting the data onto the subspace spanned by the leading eigenvectors. However, these projections still lie in the feature space, and you have to map them back to the input space to recover the denoised patterns. Note that the eigenvectors for KPCA, and hence the projections in the feature space, are expressed in terms of mapped input data. The mapping $\phi(\cdot)$ is usually nonlinear and therefore not necessarily invertible. Hence, it cannot generally be guaranteed that the exact pre-image exists, and in most cases you must settle for only an approximate solution (Mika et al. 1999).

There are several existing methods of pre-image calculation. The most commonly used methods are the iterative method (Mika et al. 1999) and the mapping method (Bakir, Weston, and Schölkopf 2003). The iterative method directly finds the pre-image by minimizing the Euclidean distance between $\phi(\boldsymbol{z})$ and the projection $P_J\phi(\boldsymbol{x})$ in the feature space, where P_J denotes the projection operator in the feature space with J principal components. In mathematical form, the iterative method aims to find the pre-image $\boldsymbol{z}^*$ such that

$$\boldsymbol{z}^* = \underset{\boldsymbol{z}}{\operatorname{argmin}} \|\phi(\boldsymbol{z}) - P_J\phi(\boldsymbol{x})\|^2$$

Exact Iterative KPCA Pre-image Method

For the Gaussian kernel, after some rearrangements, the preceding minimization problem reduces to a fixed-point iteration problem. Currently only the Gaussian kernel is supported by the iterative pre-image method, because for other types of kernels the nonconvergence issue is frequently encountered.

Approximate Iterative KPCA Pre-image Method

In the exact iterative pre-image algorithm, the whole training data set and the eigenvectors of its kernel matrix need to be saved for pre-image scoring. This can impose a memory burden on the machine when the training data set is large. To alleviate this problem, this method borrows the idea from KPCA fast scoring and devises a way to achieve approximate but fast iterative pre-image scoring. To implement this method, the low-rank approximation method must be used in KPCA training. With this approximate iterative pre-image scoring, only the k-means clustering centroids and the eigenvectors of a matrix of dimension $c \times c$ are required, where c is the number of centroids in k-means clustering.

KPCA Mapping Pre-image Method

The pre-image mapping method was first proposed by Bakir, Weston, and Schölkopf (2003). The objective is to learn the pre-image map of the form

$$\mathbf{\Gamma} = [\mathbf{\Gamma}_1(P_k\phi(x)), \ldots, \mathbf{\Gamma}_m(P_k\phi(x))]^T$$

where P_k is the KPCA projection operator with kernel function k, m is the number of features of the training data set, $\mathbf{\Gamma}_j(P_k\phi(x)) = \sum_{i'}^{n} \beta_{i'}^{j} k'(P_k\phi(x), P_k\phi(x_{i'}))$, n is the training data set size, and k' is a new kernel function different from k. Unlike the iterative pre-image method, the mapping method has no nonconvergence issues because it does not involve iterations. The mapping pre-image method is also faster than the iterative method, because once the mapping function is learned, you can score a batch of data all at the same time, instead of one observation at a time as in the iterative pre-image method.

Exact KPCA Mapping Pre-image Method

From the form of the mapping function $\mathbf{\Gamma}(\cdot)$, you see that the exact KPCA mapping pre-image method depends solely on the coefficients $\beta_{i'}^{j}$ and the training data set. Define the coefficient matrix as $\mathbf{B} = [\boldsymbol{\beta}^1, \ldots, \boldsymbol{\beta}^m]$, where $\boldsymbol{\beta}^j = (\beta_1^j, \ldots, \beta_n^j)^T$. To find the coefficients, a kernel ridge regression is then suggested:

$$\mathbf{\Gamma} = \underset{\mathbf{\Gamma}}{\operatorname{argmin}} \sum_{i=1}^{n} \|x_i - \mathbf{\Gamma}(P_k\phi(x))\|^2 + \lambda \sum_{j=1}^{m} \|\boldsymbol{\beta}^j\|^2$$

Thus the solution of $\mathbf{B}$ is formulated as

$$\mathbf{B} = (\boldsymbol{K}_{n\times n}^{'T} \boldsymbol{K}_{n\times n}^{'} + \lambda \boldsymbol{I}_n)^{-1} \boldsymbol{K}^{'T} \boldsymbol{X}$$

where $\boldsymbol{K}_{n\times n}' = (k'(P_k\phi(x_i), P_k\phi(x_j)))$ is the new kernel matrix for kernel ridge regression, $\boldsymbol{I}_n$ is an $n \times n$ identity matrix, and $\boldsymbol{X}$ is the training data matrix. For a scoring data point x, define $k_x' = [k'(P_k\phi(x), P_k\phi(x_1)), \ldots, k'(P_k\phi(x), P_k\phi(x_N))]$. Then the pre-image mapping method learns the pre-image $\hat{x}$ of $P_k\phi(x)$ by

$$\hat{x} = \mathbf{\Gamma}(P_k\phi(x)) = (k_x'\mathbf{B})^T$$

Approximate KPCA Mapping Pre-image Method

The mapping method requires solving a linear system of size $n \times n$, where n is the training data size. When n is very large, the $\boldsymbol{K}^{'T}\boldsymbol{K}^{'}$ matrix can occupy too much memory or even be too large to fit in memory at all. To handle this case, the approximate mapping pre-image method based on Nyström kernel ridge regression (Chen et al. 2016) can be implemented. It reduces the linear system size from $n \times n$ to $c \times c$, where c is the subsample size. If you use a low-rank approximation for KPCA, then the specified mapping pre-image method is also implemented in the aforementioned approximate way.

Displayed Output

The KPCA procedure displays various tables that are related to input and results. The following sections describe the output tables in the order of their appearance.

Model Information

The "Model Information" table displays basic information about the parameters that are used in the procedure. This information includes the number of interval variables, the kernel type and kernel parameters, the threshold for nonzero eigenvalues, and the number of principal components that are generated.

k-means Clustering Information

The "*k*-means Clustering Information" table displays the parameters that are used in the *k*-means clustering method. This table is displayed only if METHOD=APPROXIMATE in the PROC KPCA statement. These parameters include the number of clusters, random seed, seed initialization method for *k*-means clustering, maximum number of iterations, and convergence criterion.

Number of Observations

The "Number of Observations" table displays the number of observations that are read and used in the model. Observations that have missing values for at least one of the input variables are ignored.

Descriptive Statistics

The "Descriptive Statistics" table displays the mean and standard deviation of the input variables.

Task Timing

The "Task Timing" table displays the time in seconds and the percentage of time consumed by the various steps of the KPCA method. These steps include kernel matrix construction, *k*-means clustering, eigendecomposition, and scoring the training data.

ODS Table Names

Each table that the KPCA procedure creates has a name associated with it. You must use this name to refer to the table when you use ODS statements. The names of each table and a short description of the contents are listed in Table 18.2.

Table 18.2 ODS Tables Produced by PROC KPCA

Table Name	Description	Statement	Option
ClusteringInfo	*k*-means parameters	PROC KPCA	Default
DescStatsInt	Input variable information	PROC KPCA	Default
ModelInfo	Model information	PROC KPCA	Default
NObs	Number of observations	PROC KPCA	Default
TaskTiming	Breakdown of the time spent in each step	PROC KPCA	Default

Output Data Tables

The KPCA procedure creates output tables for eigenvalues (if you specify the EIGENVAL= option in the PROC KPCA statement), eigenvectors (if you specify the EIGENVEC= option in the same statement), or centroids (if you specify the CENTROIDS= option in the same statement). Also, you can score the training data and store the results in the table that you specify in the OUT= option in the OUTPUT statement. Table 18.3 lists details about these data tables.

Table 18.3 Output Data Tables Produced by PROC KPCA

Data Table	Content	Dimensions
Eigenval	Eigenvalues of the kernel matrix	Number of eigenvalues × 1
Eigenvec	Eigenvectors of the kernel matrix	Number of observations × number of eigenvalues
Centroids	Centroids obtained from clustering	Number of centroids × number of variables
Out	Projection of training observations onto the kernel principal components	Number of observations × number of eigenvalues

Example: KPCA Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

Example 18.1: Letter Recognition

In this example, the KPCA procedure is used as a preprocessor to the linear discriminant model for letter recognition.

The letter recognition data set that this example uses is from the UCI Machine Learning Repository (Dua and Graff 2019). The data set includes a large number of black-and-white pixel images of rectangular shape, each of them corresponding to one of the 26 capital letters in the English alphabet. These letter pictures have 20 different fonts, and each picture was randomly altered to generate a total of 20,000 unique instances. Each instance was then transformed into 16 statistical attributes (edge counts and statistical moments). All these attributes were further scaled to a range of integer values from 0 to 15 (Frey and Slate 1991). The objective is to classify each pixel image as one of the 26 capital letters. In this example, a training set of size 16,000 is randomly selected, and the remaining 4,000 instances are used as a test set. Because this is a big data set, fast KPCA is a more appropriate method to use for training while still achieving performance comparable to that of the exact method. To apply fast KPCA, 200 centroids are selected. After the model is trained, the projection values of training and scoring data onto the kernel principal components undergo a multilabel linear discriminant analysis (LDA) for classification.

In the following code, the PROC KPCA statement applies exact and fast KPCA to the data table mylib.letter_train. This DATA step assumes that your SAS libref is named mylib, but you can substitute any appropriately defined SAS libref. Next, the PROC DISCRIM statement applies linear discriminant analysis to the projections onto the principal component directions. Here letter_train and letter_test are the names of the training data set and test data set, respectively.

```
   data letter;
infile datalines delimiter=',';
input capital $ var1-var16;
datalines;
T,2,8,3,5,1,8,13,0,6,6,10,8,0,8,0,8

   ... more lines ...



data letter;
  set letter;
  obsid=_n_;
run;

/* Split data set into train (16000) and test(4000) */
proc surveyselect data=letter
  method=srs n=16000
  seed=100 out=letter_train;
```

```
run;

proc sql;
create table letter_test as
select * from letter
where obsid not in (select obsid from letter_train);
quit;

data mylib.letter_train; set letter_train; run;
data mylib.letter_test; set letter_test; run;


/* Apply exact KPCA to training data, extract 200 principal components*/
proc kpca data=mylib.letter_train  method=exact;
  input var1-var16;
  kernel  RBF/bw=7.071;
  output out=mylib.score copyvars=(capital obsid) NPC=200;
  savestate rstore=mylib.state;
run;

/* Score test data */
proc astore;
  setoption kpca_npc 200;
  score data=mylib.letter_test rstore=mylib.state
  out=mylib.outscore copyVars=(capital obsid);
quit;

/* Apply linear discriminant analysis and get multilabel classification error */
proc discrim data=mylib.score method=normal pool=yes short
  testdata=mylib.outscore;
  class capital;
  testclass capital;
run;


/* Apply fast KPCA to training data, extract 200 principal components*/
proc kpca data=mylib.letter_train method=approximate;
  input var1-var16;
  lrapproximation  clusmethod=KMPP  maxclus= 200;
  kernel  RBF/bw=7.071;
  output out=mylib.score_fast copyvars=(capital obsid) NPC=200;
  savestate rstore=mylib.state_fast;
run;

/* Fast score test data */
proc astore;
  setoption kpca_npc 200;
  score data=mylib.letter_test rstore=mylib.state_fast
  out=mylib.outscore_fast copyVars=(capital obsid);
quit;

/* Apply linear discriminant analysis and get multilabel classification error */
proc discrim data=mylib.score_fast method=normal pool=yes short
  testdata=mylib.outscore_fast;
```

```
  class capital;
  testclass capital;
run;
```

In this example, the multilabel classification errors that are obtained by fast KPCA and exact KPCA are close: 0.1662 for fast KPCA and 0.1593 for exact KPCA. However, the training time of fast KPCA is only 10.57 seconds, compared to 272.78 seconds for exact KPCA. The example demonstrates the efficiency of fast KPCA in significantly reducing the run time while not compromising much on the quality of the principal components that it generates. Moreover, in exact KPCA, all the training data along with the eigenvector matrix in the state file must be stored for scoring the test data, whereas in fast KPCA only the k-means centroids and eigenvector matrix must be stored in the state file. This greatly reduces the space requirement to score new data.

You can use the following code to also apply PCA (equivalent to KPCA with linear kernel) to the training data and extract all 16 components to use in linear discriminant analysis:

```
/* Apply PCA to training data, extract 16 principal components*/
proc kpca data=mylib.letter_train  method=exact;
   input var1-var16;
   kernel  linear;
   output out=mylib.score_pca copyvars=(capital obsid) NPC=16;
   savestate rstore=mylib.state_pca;
run;

/* Score test data */
proc astore;
   setoption kpca_npc 16;
   score data=mylib.letter_test rstore=mylib.state_pca
   out=mylib.outscore_pca copyVars=(capital obsid);
quit;

/* Apply linear discriminant analysis and get multilabel classification error */
proc discrim data=mylib.score_pca method=normal pool=yes short
   testdata=mylib.outscore_pca;
   class capital;
   testclass capital;
run;
```

Using the preceding code, you can see that even when all components are used, PCA achieves a multilabel misclassification rate of 0.3011, which is much higher than that of KPCA.

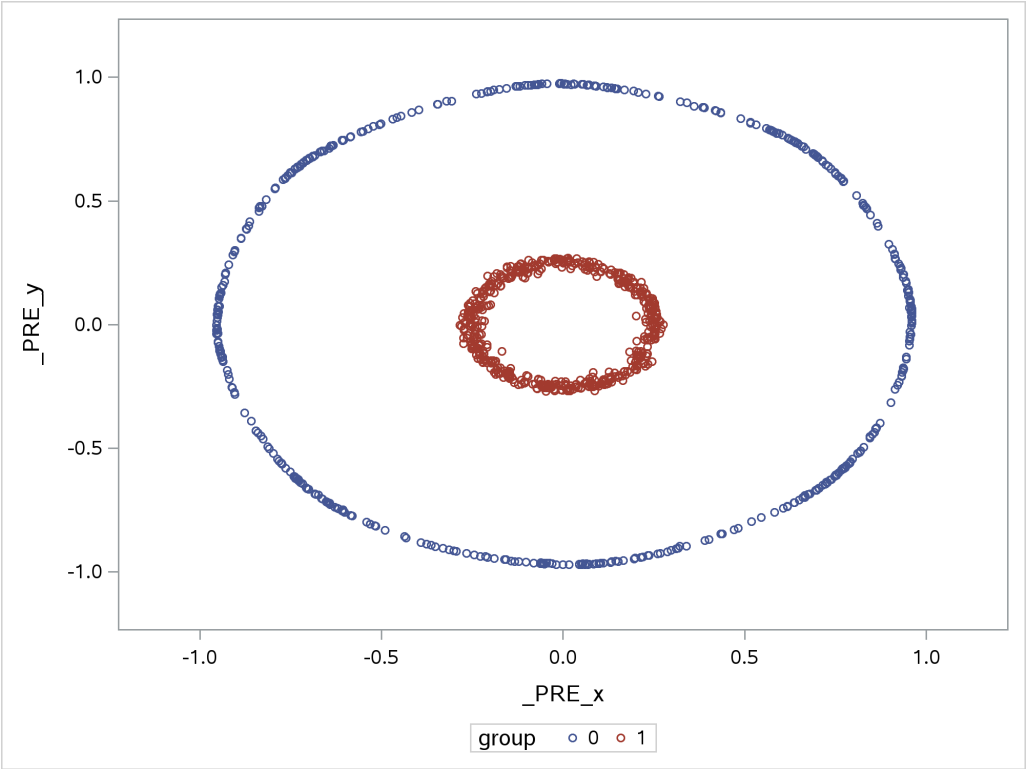
Example 18.2: Denoising Concentric Circles

This example shows how to use the KPCA procedure to obtain the pre-image of the KPCA projection of the noisy concentric pattern. Because the KPCA projection along the first 10 eigendirections (PREIMAGE NPC=10) should contain the major part of the variance of the pattern, the pre-image of this projection in input space is expected to be much less noisy than the original pattern. The noisy pattern is generated using the same code as in Figure 18, and the noisy concentric plot is shown in Output 18.1.

In the example in the section "Getting Started: KPCA Procedure" on page 477, the Circles data are already loaded into the mylib library, so here you just call the KPCA procedure with the iterative pre-image method specified and then call the ASTORE procedure to find the pre-image of the noisy concentric pattern. The pre-image output is shown in Output 18.2.1.

```
/* First call PROC KPCA */
proc KPCA data=mylib.circles method=exact;
   input x y;
   kernel RBF/bw=0.3;
   /* Use iterative pre-image method with 10 principal components */
   preimage npc=10 method=iterative;
   savestate rstore=mylib.state;
run;


/* Then use PROC ASTORE to calculate pre-image score,
   here test and train are the same data sets */
proc astore;
   /* scoring_mode=1 for pre-image scoring, default value is 0 for
      projection on the principal components */
   setoption SCORING_MODE 1;
   setoption ITER_NPC 10;
   setoption ITER_THRESH 1e-5;
   setoption ITER_NUM 1000;
   score data=mylib.circles rstore=mylib.state out=mylib.preimage_results
         copyvars=(x y group);
run;
quit;

/* Plot the pre-image of the noisy concentric circles */
proc sgplot data=mylib.preimage_results;
   styleattrs datasymbols=(Circle X);
   scatter x=_pre_x y=_pre_y / group=group markerattrs=(size=5px);
   yaxis min=-1.2 max=1.2;
   xaxis min=-1.2 max=1.2;
run;
```

Output 18.2.1 Denoised Concentric Pattern



In Output 18.2.1, you can see that pre-image plot of the concentric pattern is indeed smoother (less noisy) than the original noisy concentric pattern.

Example 18.3: Denoising USPS Digits

This example shows how to use the KPCA procedure to get the pre-image of the noisy USPS digit data. It contains 16 × 16-pixel labeled images of handwritten digits scanned by the US Postal Service. There are a total of 9,298 digits, and each digit is represented by a 256-dimensional real value vector and the corresponding integer label. For each of the 10 digits, 300 examples were randomly selected for training and 50 examples for testing. The test data were rendered noisy by adding Gaussian noise, which is characterized by a zero mean and a standard deviation of 0.5.

The following code first uses the IML procedure to randomly select the training and test data sets and then plots the second appearance of each digit in the test data set, with and without Gaussian noise, respectively:

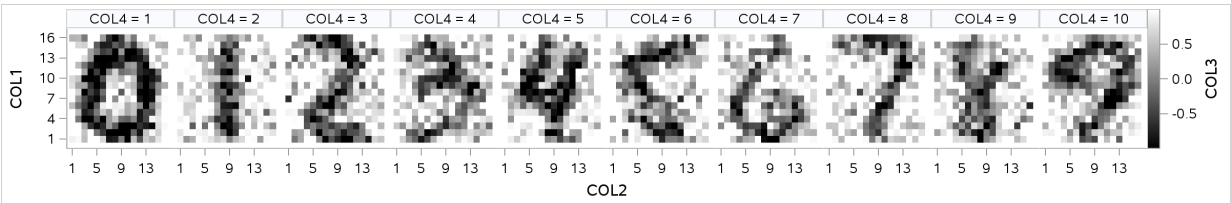
```
proc iml;
option source2;
ods graphics on/ reset=all;
title;
/* Read variables from a clean USPS digit data set into a matrix */
use usps_clean;
read all var _all_ into digitClean;

/* Flip pixel value so that the digit is black and background is white */
```

```
digitClean[,2:257]=-digitClean[,2:257];

/* Read variables from a noisy USPS digit data set into a matrix */
use usps_noise;
read all var _all_ into digitNoise;

/* Flip pixel values so that the digit is black and background is white */
digitNoise[,2:257]=-digitNoise[,2:257];

/* Define a utility function that plots the digit data*/
start implot_digit(image_m, data_m, coln);
   n=nrow(data_m);
   do i=1 to n;
     row=shape(data_m[i,1:coln-1],16,16);
      m_sub=j(16##2,4);
       do j=1 to 16;
         do k=1 to 16;
           m_sub[k+16*(j-1),1]=(17-j);
           m_sub[k+16*(j-1),2]=k;
           m_sub[k+16*(j-1),3]=row[j,k];
           m_sub[k+16*(j-1),4]=i;
         end;
       end;
     image_m=image_m//m_sub;
  end;
finish;



/* Get the features for clean digits (first column is observation ID) */
digitClean=digitClean[,2:ncol(digitClean)];

/* Get the features for noisy digits (first column is observation ID) */
digitNoise=digitNoise[,2:ncol(digitNoise)];

/* row and column number of the digit data
   (noise and clean digit data sets have same dimension) */
coln=ncol(digitClean);
rown=nrow(digitClean);

/* Allocate space for noisy test data */
digitNoiseTest={};

/* Allocate space for clean test data*/
digitCleanTest={};

/* Allocate space for clean training data*/
digitTrain={};


/* random seed for generating the training and the test data set */
call randseed(54323,1);

/* For each digit, choose 300 samples for the training data and 50 samples
```

```
   for the test data */
do i=0 to 9;
   s = sample(loc(digitClean[,coln]=i), 350);
   /* last column is digit label */
   digitCleanTest=digitCleanTest//digitClean[s[301:350],1:coln-1];
   digitNoiseTest=digitNoiseTest//digitNoise[s[301:350],1:coln-1];
   digitTrain=digitTrain//digitClean[s[1:300],];
end;


/* digit labels */
label={0, 1, 2, 3, 4, 5, 6, 7, 8, 9};

/* second appearance of noisy test digits */
digitNoiseTest_one=digitNoiseTest[{2,52,102,152,202,252,302,352,402,452},];
digitNoiseTest_one=digitNoiseTest_one||label;

/* second appearance of clean test digits */
digitCleanTest_one=digitCleanTest[{2,52,102,152,202,252,302,352,402,452},];
digitCleanTest_one=digitCleanTest_one||label;


/* Convert the digit pixel matrix to SAS data table of the form that
   HEATMAP statement can accept */
run implot_digit(digitNoiseTest_trans, digitNoiseTest_one, coln);
create digitNoiseTest_trans from digitNoiseTest_trans;
append from digitNoiseTest_trans;
close digitNoiseTest_trans;


run implot_digit(digitCleanTest_trans, digitCleanTest_one, coln);
create digitCleanTest_trans from digitCleanTest_trans;
append from digitCleanTest_trans;
close digitCleanTest_trans;

/* Save digitTrain to SAS data table */
create digitTrain from digitTrain;
append from digitTrain;
close digitTrain;

/* Save digitNoiseTest to SAS data table */
create digitNoiseTest from digitNoiseTest;
append from digitNoiseTest;
close digitNoiseTest;

/* Save digitCleanTest to SAS data table */
create digitCleanTest from digitCleanTest;
append from digitCleanTest;
close digitCleanTest;

/* Plot test digits with Gaussian noise */

ods graphics /width=8in height=1.3in;
proc sgpanel data=digitNoiseTest_trans;
```

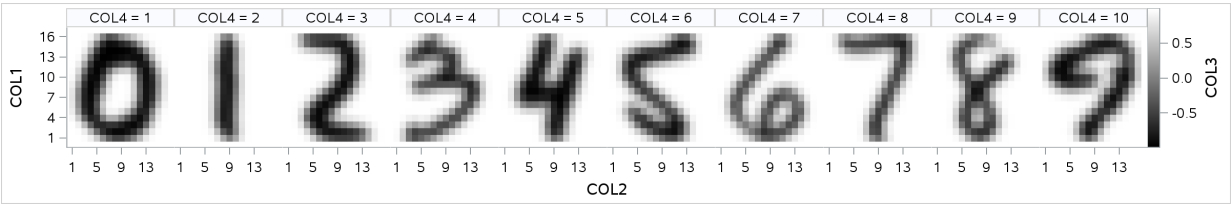
```
   panelby COL4/noborder columns=10 proportional ;
   heatmap x=COL2 y=COL1 /colorresponse=COL3 colormodel=(black white)
           showybins showxbins xbinstart=0 ybinstart=0;
run;
```

Output 18.3.1 Plot of Noisy USPS Digit Data




```
/* Plot clean test digits without Gaussian noise */
ods graphics /width=8in height=1.3in;
proc sgpanel data=digitCleanTest_trans;
   panelby COL4/noborder columns=10 proportional ;
   heatmap x=COL2 y=COL1 /colorresponse=COL3 colormodel=(black white)
           showybins showxbins xbinstart=0 ybinstart=0;
run;
```

Output 18.3.2 Plot of Clean USPS Digit Data



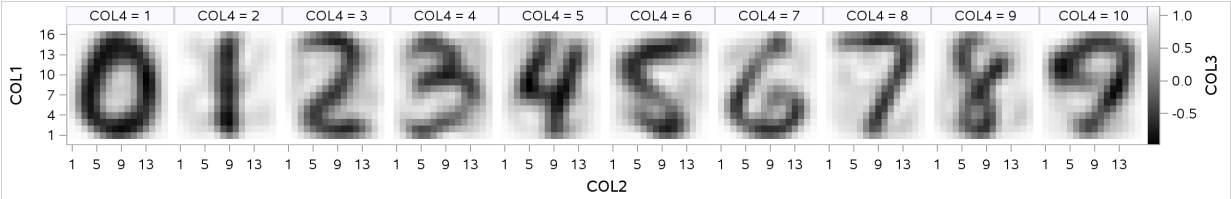
In Output 18.3.1 and Output 18.3.2, noisy digits are blurry and clean digits are sharp and clear.

Next, run the KPCA procedure with the mapping pre-image method specified. In the following code, the RBF kernel is used in both KPCA (BW=8) and kernel ridge regression (BW=1), and 80 principal components are used for KPCA projection. When the mapping pre-image training is completed, the ASTORE procedure is called to compute the pre-image score of the noisy test digits. To get a sense of how well the mapping pre-image method performs, you can plot the pre-image of the second appearance of all 10 digits.

```
data mylib.digitTrain;
  set digitTrain;
run;

data mylib.digitNoiseTest;
  set digitNoiseTest;
run;

proc KPCA data=mylib.digitTrain method=exact;
  input COL1-COL256;
  kernel RBF/bw=8;
  preimage npc=80 method=map(L2=1 kernel=RBF(bw=1));
  savestate rstore=mylib.state;
```

```
  displayout modelinfo="TaskTiming";
run;


data mylib.digitNoiseTest;
   set digitNoiseTest;
   id =_N_;
run;

proc ASTORE;
    setoption scoring_mode 1;
    score data=mylib.digitNoiseTest rstore=mylib.state out=mylib.preimage_results
               copyvars=id;
run;
quit;

data preimage_results;
    set mylib.preimage_results;
run;

proc sort data=preimage_results;
    by id;
run;

proc iml;

/* Read pre-image results into an iml matrix */
use preimage_results;
read all var _all_ into preimage_results;

/* second appearance of the pre-image of noisy test digits */
preimage_results=preimage_results[{2,52,102,152,202,252,302,352,402,452},];
preimage_results=preimage_results||label;

coln=ncol(preimage_results);
run implot_digit(preimage_results_trans, preimage_results, coln);
create preimage_results_trans from preimage_results_trans;
append from preimage_results_trans;
close preimage_results_trans;


/* Plot pre-image of noisy digits (denoising) */

ods graphics /width=8in height=1.3in;
proc sgpanel data=preimage_results_trans;
   panelby COL4/noborder columns=10 proportional ;
   heatmap x=COL2 y=COL1 /colorresponse=COL3 colormodel=(black white)
           showybins showxbins xbinstart=0 ybinstart=0;
run;
```

Output 18.3.3 Plot of Denoised USPS Digits



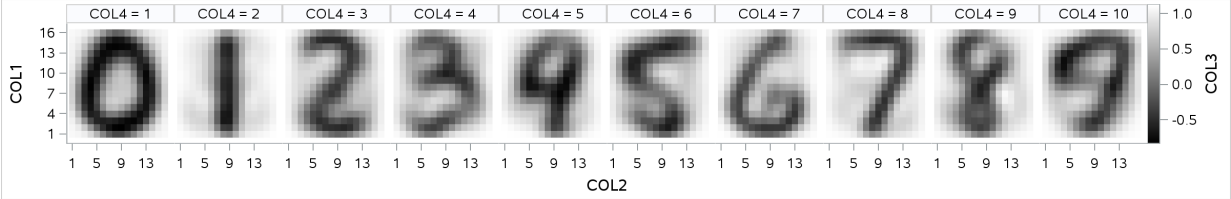
In Output 18.3.3, you can see that the denoised digits are much clearer and easier to identify than the original noisy digits shown in Output 18.3.1.

The following code implements the approximate mapping pre-image method by specifying METHOD=APPROXIMATE. Here the *k*-means clustering uses KMPP (*k*-means plus plus) as the clustering centroids initialization method and uses the default number of centroids (100).

```
/* Run mapping pre-image method using Nystrom approximation with 100 centroids */

proc KPCA data=mylib.digitTrain method=approximate;
  input COL1-COL256;
  kernel RBF/bw=8;
  lrapproximation clusmethod = KMPP;
  preimage npc=80 method=map(L2=1 kernel=RBF(bw=1));
  savestate rstore=mylib.state;
  displayout modelinfo="TaskTiming";
run;


data mylib.digitNoiseTest;
   set digitNoiseTest;
   id =_N_;
run;

proc ASTORE;
    setoption scoring_mode 1;
    score data=mylib.digitNoiseTest rstore=mylib.state
          out=mylib.preimage_results_approx copyvars=id;
run;
quit;


data preimage_results_approx;
    set mylib.preimage_results_approx;
run;

proc sort data=preimage_results_approx;
    by id;
run;

/* Read approximate pre-image results data into iml matrix */
proc iml;

use preimage_results_approx;
```

```
read all var _all_ into preimage_results_approx;

/* second appearance of the pre-image of noisy test digits */
preimage_results_approx=
       preimage_results_approx[{2,52,102,152,202,252,302,352,402,452},];
preimage_results_approx=preimage_results_approx||label;


coln=ncol(preimage_results_approx);
run implot_digit(preimage_results_approx_trans, preimage_results_approx, coln);
create preimage_results_approx_trans from preimage_results_approx_trans;
append from preimage_results_approx_trans;
close preimage_results_approx_trans;


/* Plot the approximated pre-image of noisy digits (denoising) */


ods graphics /width=8in height=1.3in;
proc sgpanel data=preimage_results_approx_trans;
  panelby COL4/noborder columns=10 proportional ;
  heatmap x=COL2 y=COL1 /colorresponse=COL3 colormodel=(black white)
          showybins showxbins xbinstart=0 ybinstart=0;
run;


quit;
```

Output 18.3.4 Plot of Denoised USPS Digits with Approximate Mapping Method



Output 18.3.4 shows that the denoised digits that are obtained by the approximate mapping pre-image method are similar to those obtained by the exact method and shown in Output 18.3.3.

In terms of the amount of time the two methods take, the training time for the approximate mapping method is much less than the training time for the exact mapping method, as shown in the "Task Timing" tables in Output 18.3.5 and Output 18.3.6, respectively. This demonstrates the efficiency of the approximate mapping method in reducing the training time while not compromising much in denoising performance. In addition to that improvement, in the exact mapping pre-image method all the training data must be stored in the state file to get the pre-image scores of the test data, whereas in the approximate mapping pre-image method only the k-means centroids of the KPCA projection of the training data must be stored in the state file. This significantly reduces the time required for creating the scoring state from the state file.

Output 18.3.5 Timing Table—Approximate Mapping Pre-image Method

Task Timing		
Task	**Seconds**	**Percent**
Kernel Matrix Construction	0.01	0.33%
k-means Clustering	2.20	70.60%
Eigendecomposition	0.00	0.11%
k-means Clustering for Principal Components	0.66	21.06%
Pre-image Mapping Method Training	0.02	0.69%
Other	0.22	7.21%
Total	3.12	100.00%

Output 18.3.6 Timing Table—Exact Mapping Pre-image Method

Task Timing		
Task	**Seconds**	**Percent**
Kernel Matrix Construction	0.34	6.65%
Eigendecomposition	2.34	46.47%
Pre-image Mapping Method Training	2.13	42.34%
Other	0.23	4.54%
Total	5.04	100.00%

References

Arthur, D., and Vassilvitskii, S. (2007). "k-Means++: The Advantages of Careful Seeding." In *Proceedings of the Eighteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, 1027–1035. Philadelphia: SIAM.

Baker, C. T. H. (1977). *The Numerical Treatment of Integral Equations*. Oxford: Clarendon Press.

Bakir, G., Weston, J., and Schölkopf, B. (2003). "Learning to Find Pre-images." In *NIPS'03: Proceedings of the 16th International Conference on Neural Information Processing Systems*, 449–456.

Chen, H., Xia, H., Huang, H., and Cai, W. (2016). "Error Analysis of Generalized Nyström Kernel Regression." In *Advances in Neural Information Processing Systems 29 (NIPS 2016)*, 2541–2549. Cambridge, MA: MIT Press.

Dua, D., and Graff, C. (2019). "UCI Machine Learning Repository." School of Information and Computer Sciences, University of California, Irvine. `http://archive.ics.uci.edu/ml`.

Frey, P. W., and Slate, D. J. (1991). "Letter Recognition Using Holland-Style Adaptive Classifiers." *Machine Learning* 6:161–182.

Kwok, J. T., and Tsang, I. W. (2004). "The Pre-image Problem in Kernel Methods." *IEEE Transactions on Neural Networks* 15:1517–1525.

Mika, S., Schölkopf, B., Smola, A. J., Müller, K.-R., Scholz, M., and Rätsch, G. (1999). "Kernel PCA and De-noising in Feature Spaces." In *Advances in Neural Information Processing Systems 11*, edited by M. J. Kearns, S. A. Solla, and D. A. Cohn, 536–542. Cambridge, MA: MIT Press.

Schölkopf, B., Smola, A., and Müller, K.-R. (1998). "Nonlinear Component Analysis as a Kernel Eigenvalue Problem." *Neural Computation* 10:1299–1319.

Chapter 19
LIGHTGRADBOOST Procedure

Contents

Overview: LIGHTGRADBOOST Procedure **511**
PROC LIGHTGRADBOOST Features 512
Using SAS Viya Workbench 512
Getting Started: LIGHTGRADBOOST Procedure **513**
Syntax: LIGHTGRADBOOST Procedure **517**
PROC LIGHTGRADBOOST Statement 517
AUTOTUNE Statement 521
INPUT Statement 528
OUTPUT Statement 529
SAVESTATE Statement 529
TARGET Statement 530
Details: LIGHTGRADBOOST Procedure **530**
Training a LightGBM Model 530
ODS Table Names 530
Examples: LIGHTGRADBOOST Procedure **531**
Example 19.1: Train a LightGBM Model 531
Example 19.2: Tuning the Hyperparameters of a LightGBM Model 532
References **535**

Overview: LIGHTGRADBOOST Procedure

The LIGHTGRADBOOST procedure creates a predictive model called a light gradient boosting machine (LightGBM) model in SAS Viya Workbench. This new model, which was proposed by Ke et al. (2017), consists of multiple decision trees. A predictive model defines a relationship between input variables and a target variable. Its purpose is to predict a target value from inputs. The LIGHTGRADBOOST procedure creates the model by using training data in which the target values are known. You can then apply the model to observations in which the target is unknown. If the predictions fit the new data well, the model is said to generalize well. Good generalization is the primary goal of predictive tasks. A predictive model might fit the training data well but generalize poorly.

LightGBM is a high-performance gradient boosting framework based on decision tree algorithms that is used for regression, classification, and many other machine learning tasks. It is designed to be efficient, and it offers the following advantages:

- higher training speed and greater efficiency

- lower memory usage
- better accuracy
- ability to handle large-scale data

The LightGBM model introduces two novel techniques: gradient-based one-side sampling (GOSS) and exclusive feature bundling (EFB).

The GOSS technique excludes a significant proportion of data instances that have small gradients, and it uses the rest of the instances only to estimate the information gain. This is a novel sampling method that down-samples the instances on the basis of gradients. A naive approach to down-sampling is to discard instances with small gradients by focusing only on instances with large gradients, but this would alter the data distribution. In a nutshell, GOSS retains data instances with large gradients while performing random sampling on instances with small gradients.

The EFB technique bundles mutually exclusive features in order to reduce the number of features. In real applications, although there are a large number of features, the feature space is usually quite sparse, which offers a possibility of designing a nearly lossless approach to reduce the number of effective features. Specifically, in a sparse feature space, many features are (almost) exclusive; that is, they rarely take nonzero values simultaneously.

The LightGBM algorithm splits the tree leafwise, as opposed to other boosting algorithms that grow trees levelwise.

PROC LIGHTGRADBOOST Features

The LIGHTGRADBOOST procedure has the following features:

- processes data in a single thread as well as in multiple threads
- supports the SAVESTATE statement for scoring by the ASTORE procedure

Using SAS Viya Workbench

SAS Viya Workbench has greatly simplified the SAS programming experience by offering a syntax style nearly identical to that of SAS 9 and earlier releases. However, there are minor exceptions. In the past, SAS Viya procedures required the use of a CAS engine libref. This engine served multiple purposes, but it was primarily used to upload data to the CAS server and manage the data. Its additional purposes included support of new data types and improved performance for large data sets. No CAS server is needed for SAS Viya Workbench, because all computations are done in place for you in a single location.

In the following program, the LIBNAME statement creates a mylib libref that the examples in this chapter use to refer to your current or desired work directory:

```
libname mylib ".";
```

For more information, see *SAS Viya Workbench: Getting Started*.

Getting Started: LIGHTGRADBOOST Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

A common use of light gradient boosting machine (LightGBM) models is to predict whether a mortgage applicant will default on a loan. The home equity data table Hmeq, which is in the Sampsio library that SAS provides, contains observations for 5,960 mortgage applicants. A variable named Bad indicates whether the applicant, after being approved for a loan, paid off or defaulted on the loan.

This example uses the Hmeq data table to build a LightGBM model that scores the data and can be used to score data about new loan applicants. Table 19.1 describes the variables in Hmeq.

Table 19.1 Variables in the Home Equity (Hmeq) Data Table

Variable	Role	Level	Description
Bad	Response	Binary	1 = applicant defaulted on the loan or is seriously delinquent 0 = applicant paid off the loan
CLAge	Predictor	Interval	Age of oldest credit line in months
CLNo	Predictor	Interval	Number of credit lines
DebtInc	Predictor	Interval	Debt-to-income ratio
Delinq	Predictor	Interval	Number of delinquent credit lines
Derog	Predictor	Interval	Number of major derogatory reports
Job	Predictor	Nominal	Occupational category
Loan	Predictor	Interval	Requested loan amount
MortDue	Predictor	Interval	Amount due on mortgage
nInq	Predictor	Interval	Number of recent credit inquiries
Reason	Predictor	Binary	`'DebtCon'` = debt consolidation `'HomeImp'` = home improvement
Value	Predictor	Interval	Value of property
YoJ	Predictor	Interval	Years at present job

For this example, the statements assume that your SAS libref is named mylib, but you can substitute any appropriately defined SAS libref. You can load the hmeq data set into your SAS library in the following DATA step:

```
data mylib.hmeq;
   length Bad Loan MortDue Value 8 Reason Job $7
              YoJ Derog Delinq CLAge nInq CLNo DebtInc 8;
   set sampsio.hmeq;
run;
```

```
proc print data=mylib.hmeq(obs=10); run;
```

Output 19.1 shows the first 10 observations of mylib.hmeq.

Figure 19.1 Partial Listing of the mylib.hmeq Data

Obs	Bad	Loan	MortDue	Value	Reason	Job	YoJ	Derog	Delinq	CLAge	nInq	CLNo	DebtInc
1	1	1100	25860	39025	HomeImp	Other	10.5	0	0	94.367	1	9	.
2	1	1300	70053	68400	HomeImp	Other	7.0	0	2	121.833	0	14	.
3	1	1500	13500	16700	HomeImp	Other	4.0	0	0	149.467	1	10	.
4	1	1500	.	.			.	.	.	.	.	.	.
5	0	1700	97800	112000	HomeImp	Office	3.0	0	0	93.333	0	14	.
6	1	1700	30548	40320	HomeImp	Other	9.0	0	0	101.466	1	8	37.1136
7	1	1800	48649	57037	HomeImp	Other	5.0	3	2	77.100	1	17	.
8	1	1800	28502	43034	HomeImp	Other	11.0	0	0	88.766	0	8	36.8849
9	1	2000	32700	46740	HomeImp	Other	3.0	0	2	216.933	1	12	.
10	1	2000	.	62250	HomeImp	Sales	16.0	0	0	115.800	0	13	.

PROC LIGHTGRADBOOST treats numeric variables as interval inputs unless you specify otherwise. Character variables are always treated as nominal inputs. The following statements run PROC LIGHTGRADBOOST and save the model in a table named mylib.lgbmStore:

```
proc lightgradboost data=mylib.hmeq validdata=mylib.hmeq
   boosting=DART objective=binary deterministic;
   input Delinq Derog Job nInq Reason / level = nominal;
   input CLAge CLNo DebtInc Loan Mortdue Value YoJ / level = interval;
   target Bad / level = nominal;
   SAVESTATE RSTORE=mylib.lgbmStore ;
   output out=mylib.procout copyvars=(bad loan reason);
   ods output IterHistory=IterHistory;
run;
```

No parameters are specified in the PROC LIGHTGRADBOOST statement; therefore, the procedure uses all default values. For example, the number of trees in the boosting model is 100, and the number of bins for interval input variables is 255.

The INPUT and TARGET statements are required in order to run PROC LIGHTGRADBOOST. The INPUT statement indicates which variables to use to build the model, and the TARGET statement indicates which variable the procedure predicts.

Figure 19.2 displays the “Model Information” table. This table shows the values of the training parameters in the first six rows, in addition to some basic information about the trees in the model.

Figure 19.2 Model Information

The LIGHTGRADBOOST Procedure

Model Information	
	cValue
Boosting Method	dart
Objective Type	binary
Number of Bins	255
Minimum Leaf Size	20
Lasso (L1) Penalty	0
Ridge (L2) Penalty	0
Maximum Categories	32

Figure 19.3 displays the "Number of Observations" table, which shows how many observations were read and used.

Figure 19.3 Number of Observations

Number of Observations	
Number of Training Observations Read	5960
Number of Training Observations Used	5960
Number of Validation Observations Read	5960
Number of Validation Observations Used	5960

Figure 19.4 displays the "Iteration History" table, which shows evaluation for training and validation data.

Figure 19.4 Iteration History

Iteration History		
Number of Trees	Training Accuracy Metric	Validation Accuracy Metric
1	0.455459	0.455459
2	0.423271	0.423271
3	0.396965	0.396965
4	0.374807	0.374807
5	0.357085	0.357085
6	0.359752	0.359752
7	0.344340	0.344340
8	0.330293	0.330293
9	0.317816	0.317816
10	0.305986	0.305986
.	.	.
.	.	.
.	.	.
91	0.190224	0.190224
92	0.191454	0.191454
93	0.188470	0.188470
94	0.188844	0.188844
95	0.185942	0.185942
96	0.186890	0.186890
97	0.183837	0.183837
98	0.185237	0.185237
99	0.187095	0.187095
100	0.188650	0.188650

Figure 19.5 displays the "Encoded Name Information" table, which shows the names of predicted probabilities.

Figure 19.5 Encoded Name Information

Encoded Name Information		
Level Name	Level ID	Variable Name
0	**0**	P_Bad0
1	**1**	P_Bad1

Syntax: LIGHTGRADBOOST Procedure

The following statements are available in the LIGHTGRADBOOST procedure:

PROC LIGHTGRADBOOST < *options* > **;**
  AUTOTUNE < *options* > **;**
  INPUT *variables* < / *options* > **;**
  OUTPUT OUT=*libref.data-table* < *option* > **;**
  SAVESTATE RSTORE=*libref.data-table* **;**
  TARGET *variable* < / **LEVEL=NOMINAL** | **INTERVAL** > **;**

The PROC LIGHTGRADBOOST, INPUT, and TARGET statements are required. The INPUT statement can appear multiple times.

The rest of this section provides detailed syntax information about each of the available statements, beginning with the PROC LIGHTGRADBOOST statement. The remaining statements are described in alphabetical order.

PROC LIGHTGRADBOOST Statement

PROC LIGHTGRADBOOST < *options* > **;**

The PROC LIGHTGRADBOOST statement invokes the procedure. Table 19.2 summarizes the options in this statement.

Table 19.2 PROC LIGHTGRADBOOST Statement Options

Option	Description
Basic Options	
BAGGINGFRACTION=	Randomly selects a portion of the observations without resampling
BAGGINGFREQUENCY=	Specifies the frequency of bagging
BOOSTING=	Specifies the type of boosting
DATA=	Specifies the input data table
DETERMINISTIC	Ensures stable results when the same data and same parameters are used
INPUTFRACTION=	Randomly selects a subset of features on each iteration or tree. If you set it to 0.8, the LightGBM algorithm selects 80% of features before training each tree.
LASSO=	Specifies the L1-norm regularization parameter
LEAFSIZE=	Specifies the minimum number of observations per leaf
LEARNINGRATE=	Specifies the learning rate for each tree
MAXCATEGORIES=	Limits the number of split points considered for categorical features
MAXDEPTH=	Limits the maximum depth of the tree model. A value less than or equal to 0 means no limit.
MAXITERS=	Specifies the number of iterations for the boosting
MONOTONELIST=	Specifies constraints of monotone features

Table 19.2 *continued*

Option	Description
NTHREADS=	Specifies the number of threads to use in the computation
NUMBIN=	Specifies the number of bins to use for numeric variables in calculating the decision tree
OBJECTIVE=	Specifies the type of objective
RIDGE=	Specifies the L2-norm regularization parameter
SEED=	Specifies the random number seed to use for model building
USEMISSING	Specifies the use of special handling of missing values
VALIDATIONDATA=	Specifies the validation data table

You can specify the following *options*:

BAGGINGFRACTION=*number*

randomly selects a portion of the observations without resampling. If *number* is nonzero, the LightGBM algorithm enables bagging.

By default, BAGGINGFRACTION=1.0. You can tune this value by using the AUTOTUNE statement.

BAGGINGFREQUENCY=*number*

specifies the frequency of bagging. If *number* is greater than 0, the LightGBM algorithm performs bagging. If *number* is n, the LightGBM algorithm randomly selects a value (*number* specified in the BAGGINGFRACTION= option $\times$ 100% of the data) to use for the next n iterations.

By default, BAGGINGFREQUENCY=0. You can tune this value by using the AUTOTUNE statement.

BOOSTING=DART | GBDT | GOSS | RF

specifies the type of boosting to use. You can specify the following values:

DART uses dropouts meet multiple additive regression trees.

GBDT uses the traditional gradient boosting decision tree.

GOSS uses gradient-based one-side sampling.

RF uses the random forest. You must specify the BAGGINGFRACTION= and BAGGINGFREQUENCY= options in order to use this boosting type.

By default, BOOSTING=GBDT.

DATA=*libref.data-table*

names the input data table for PROC LIGHTGRADBOOST to use. The default is the most recently created data table. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to the data. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 512.

data-table specifies the name of the input data table.

DETERMINISTIC
ensures stable results when you use the same data and the same parameters.

INPUTFRACTION=*number*
FEATUREFRACTION=*number*
randomly selects a subset of features on each iteration or tree. If *number* is 0.8, the LightGBM algorithm selects 80% of features before training each tree.

By default, INPUTFRACTION=1.0.

LEAFSIZE=*number*
MINLEAFSIZE=*number*
specifies the minimum number of observations that each child of a split must contain in the training data table in order for the split to be considered.

By default, LEAFSIZE=20. You can tune this value by using the AUTOTUNE statement.

LEARNINGRATE=*number*
specifies the learning rate for each tree, where *number* must be greater than 0.

By default, LEARNINGRATE=0.1. You can tune this value by using the AUTOTUNE statement.

MAXCATEGORIES=*number*
limits the number of split points to be considered for categorical features. By default, MAXCATEGORIES=32. This value must be greater than 0.

MAXDEPTH=*number*
limits the maximum depth of the tree model. A *number* less than or equal to 0 means no limit.

You can tune this value by using the AUTOTUNE statement.

MAXITERS=*number*
NUMITERS=*number*
specifies the maximum number of iterations for the boosting.

By default, MAXITERS=100. You can tune this value by using the AUTOTUNE statement.

MONOTONELIST=(–1 0 1)
MONOTONECONSTRAINTS=(–1 0 1)
specifies series of constraints of monotone features, where 1 represents increasing constraints, –1 represents decreasing constraints, and 0 represents no constraints.

NTHREADS=*number*
specifies the number of threads to use in the computation. The default number of threads is calculated for best performance according to the data.

NUMBIN=*number*
NBINS=*number*
INTERVALBINS=*number*
specifies the number of bins to use for binning the interval input variables. PROC LIGHTGRADBOOST bins continuous predictors to a fixed bin size. This option controls the number of bins and thus also the size of the bins.

By default, NUMBIN=255. You can tune this value by using the AUTOTUNE statement.

OBJECTIVE=*function*

specifies the objective function to use. You can specify the following values:

BINARY	uses binary classification.
CROSS_ENTROPY	uses cross-entropy.
CROSS_ENTROPY_LAMBDA	uses an alternative parameterization of cross-entropy.
FAIR	uses fair loss.
GAMMA	uses gamma regression.
HUBER	uses Huber regression.
MAPE	uses mean absolute percentage error regression.
MULTICLASS	uses multiclass classification.
MULTICLASSOVA	uses one-versus-all multiclass classification.
POISSON	uses Poisson regression.
QUANTILE	uses quantile regression.
REGRESSION	uses regression.
REGRESSION_L1	uses L1 regression.
TWEEDIE	uses Tweedie regression.

By default, OBJECTIVE=REGRESSION.

REGL1=*number*

L1=*number*

LASSO=*number*

specifies the L1-norm regularization parameter, where *number* must be nonnegative.

By default, REGL1=0. You can tune this value by using the AUTOTUNE statement.

REGL2=*number*

L2=*number*

RIDGE=*number*

specifies the L2-norm regularization parameter on prediction. The value of *number* must be nonnegative.

By default, REGL2=1. You can tune this value by using the AUTOTUNE statement.

SEED=*number*

specifies the initial seed for random number generation for model building.

USEMISSING

specifies the use of special handling of missing values. You can disable this option by omitting it. The LightGBM model uses not a number (NaN) values to represent missing values by default.

VALIDATIONDATA=*data*

VALIDDATA=*data*

specifies that validation data be used. You can use this option to avoid overfitting the training data, combined with parameters such as early stopping rounds.

AUTOTUNE Statement

AUTOTUNE < *options* > **;**

The AUTOTUNE statement searches for the best combination of values of the BAGGINGFRACTION=, BAGGINGFREQUENCY=, INPUTFRACTION=, LASSO=, LEAFSIZE=, LEARNINGRATE=, MAXDEPTH=, MAXITERS=, NUMBIN=, and RIDGE= options in the PROC LIGHTGRADBOOST statement. These options control the training algorithm and are often called hyperparameters to distinguish them from the parameters of the trained model. Optimal hyperparameter settings depend on the data and are never known for certain.

Table 19.3 summarizes the *options* that you can specify in the AUTOTUNE statement. For more information about all options except the TUNINGPARAMETERS= option, see the option's description in the section "AUTOTUNE Statement" on page 6 in Chapter 2, "Shared Concepts." The TUNINGPARAMETERS= option is described in Table 19.3.

Table 19.3 AUTOTUNE Statement Options

Option	Description
APPENDLOOKUP	Specifies that the table specified in the HISTORYTABLE= option contain the rows from the table specified in the LOOKUPTABLE= option
CONSTRAINTS=	Specifies the constraints on the output metrics
EVALHISTORY=	Specifies how to report the evaluation history of the tuner
FOLDCOLUMN=	Specifies the column in the data table in which the cross validation fold value is indicated
FRACTION=	Specifies the fraction of observations to use for validation
HISTORYTABLE=	Specifies the table that contains the evaluation history
KFOLD=	Specifies the number of folds for *k*-fold cross validation
LIVEUPDATE	Specifies that the table specified in the HISTORYTABLE= option be updated at every evaluation
LOOKUPTABLE=	Specifies the table to use for evaluation lookup
MAXBAYES=	Specifies the maximum number of points in the kriging model
MAXEVALS=	Specifies the maximum number of evaluations
MAXITER=	Specifies the maximum number of iterations when SEARCHMETHOD=GA or SEARCHMETHOD=BAYESIAN
MAXTIME=	Specifies the maximum time for all iterations
MAXTRAINTIME=	Specifies the maximum time for a model training
NCONVITER=	Specifies the number of convergence iterations
NOGRIDSHUFFLE	Requests that the grid points not be shuffled
NOLOCALSEARCH	Disables local search optimization
NPARALLEL=	Specifies the number of parallel sessions

Table 19.3 *continued*

Option	Description
NSUBSESSIONWORKERS=	Specifies the number of workers in parallel sessions
OBJECTIVE=	Specifies the objective function
POPSIZE=	Specifies the population size when SEARCHMETHOD=GA or SEARCHMETHOD=BAYESIAN
SAMPLESIZE=	Specifies the sample size when SEARCHMETHOD=LHS or SEARCHMETHOD=RANDOM
SEARCHMETHOD=	Specifies the search method that the optimizer uses
SECONDOBJECTIVE=	Specifies the second objective to use for tuning
SELECTINITPOINT	Specifies that the tuner select the best evaluation from the lookup table
TARGETEVENT=	Specifies the target event for ROC-based calculations
TRAINFRACTION=	Specifies the fraction of observations to use for training
TUNINGPARAMETERS=	Specifies the custom tuning parameters
USEPARAMETERS=	Specifies how to handle the TUNINGPARAMETERS= option

TUNINGPARAMETERS=(*suboption* | . . . | < *suboption* >**)**

TUNEPARMS=(*suboption* | . . . | < *suboption* >**)**

specifies which parameters to tune and which ranges to tune over. If you specify USEPARAMETERS=STANDARD, this option is ignored.

You can specify one or more of the following *suboptions*:

BAGGINGFRACTION (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**

specifies information about the BAGGINGFRACTION= option values to use in tuning the light gradient boosting model. For more information, see the BAGGINGFRACTION= option in the PROC LIGHTGRADBOOST statement.

You can specify the following additional suboptions:

LB=*number*

specifies the minimum BAGGINGFRACTION= option value to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=0.2

UB=*number*

specifies the maximum BAGGINGFRACTION= option value to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=0.95

VALUES=*value-list*

specifies a list of BAGGINGFRACTION= option values to consider during tuning, where *value-list* is a space-separated list of numbers greater than or equal to 0 and less than or equal to 1. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial BAGGINGFRACTION= option value for the tuner to use.

By default, INIT=0.95

EXCLUDE
excludes the BAGGINGFRACTION= option from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

BAGGINGFREQUENCY (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**
specifies information about the BAGGINGFREQUENCY= option values to use in tuning the light gradient boosting model. For more information, see the BAGGINGFREQUENCY= option in the PROC LIGHTGRADBOOST statement.

You can specify the following additional suboptions:

LB=*number*
specifies the minimum BAGGINGFREQUENCY= option value to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=0

UB=*number*
specifies the maximum BAGGINGFREQUENCY= option value to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=7

VALUES=*value-list*
specifies a list of BAGGINGFREQUENCY= option values to consider during tuning, where *value-list* is a space-separated list of numbers greater than or equal to 0. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial BAGGINGFREQUENCY= option value for the tuner to use.

By default, INIT=0

EXCLUDE
excludes the BAGGINGFREQUENCY= option from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

INPUTFRACTION (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**
specifies information about the INPUTFRACTION= option values to use in tuning the light gradient boosting model. For more information, see the INPUTFRACTION= option in the PROC LIGHTGRADBOOST statement.

You can specify the following additional suboptions:

LB=*number*
specifies the minimum INPUTFRACTION= option value to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=0.05

UB=*number*
specifies the maximum INPUTFRACTION= option value to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=1.0

VALUES=*value-list*
specifies a list of INPUTFRACTION= option values to consider during tuning, where *value-list* is a space-separated list of numbers greater than 0 and less than or equal to 1. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial INPUTFRACTION= option value for the tuner to use.

By default, INIT=1.0

EXCLUDE
excludes the INPUTFRACTION= option from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

LASSO (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**
specifies information about the LASSO= option values to use in tuning the light gradient boosting model. For more information, see the LASSO= option in the PROC LIGHTGRADBOOST statement.

You can specify the following additional suboptions:

LB=*number*
specifies the minimum LASSO= option value to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=0.

UB=*number*
specifies the maximum LASSO= option value to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=10.

VALUES=*value-list*
specifies a list of LASSO= option values to consider during tuning, where *value-list* is a space-separated list of numbers greater than or equal to 0. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial LASSO= option value for the tuner to use.

By default, INIT=0.

EXCLUDE
excludes the LASSO= option from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

LEAFSIZE (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**
specifies information about the LEAFSIZE= option values to use in tuning the light gradient boosting model. This option is not tuned by default. If it is specified without any suboptions, then the tuner uses the following values: 1, 5, 10, 20, 40, 80, 160, and 320. For more information, see the LEAFSIZE= option in the PROC LIGHTGRADBOOST statement.

You can specify the following additional suboptions:

LB=*number*
specifies the minimum LEAFSIZE= option value to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=1.

UB=*number*
specifies the maximum LEAFSIZE= option value to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=320.

VALUES=*value-list*
specifies a list of LEAFSIZE= option values to consider during tuning, where *value-list* is a space-separated list of integers greater than or equal to 0. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial LEAFSIZE= option for the tuner to use.

By default, INIT=20.

EXCLUDE
excludes the LEAFSIZE= option from the tuning process.

LEARNINGRATE (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**
specifies information about the LEARNINGRATE= option values to use in tuning the light gradient boosting model. For more information, see the LEARNINGRATE= option in the PROC LIGHTGRADBOOST statement.

You can specify the following additional suboptions:

LB=*number*
specifies the minimum LEARNINGRATE= option value to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=0.01.

UB=*number*
specifies the maximum LEARNINGRATE= option value to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=1.

VALUES=*value-list*
specifies a list of LEARNINGRATE= option values to consider during tuning, where *value-list* is a space-separated list of numbers greater than 0 and less than or equal to 1. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial LEARNINGRATE= option value for the tuner to use.

By default, INIT=0.1.

EXCLUDE
excludes the LEARNINGRATE= option from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

MAXDEPTH (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**
specifies information about the MAXDEPTH= option value to use in tuning the light gradient boosting model. If it is specified without any suboptions, then the tuner uses the following values: –1, 3, 4, 5, 6, and 7. For more information, see the MAXDEPTH= option in the PROC LIGHTGRADBOOST statement.

You can specify the following additional suboptions:

LB=*number*
specifies the minimum MAXDEPTH= option value to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=–1.

UB=*number*
specifies the maximum MAXDEPTH= option value to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=7.

VALUES=*value-list*
specifies a list of MAXDEPTH= option values to consider during tuning, where *value-list* is a space-separated list of integers greater than or equal to 1. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial MAXDEPTH= option value for the tuner to use.

By default, INIT=–1.

EXCLUDE
excludes the MAXDEPTH= option from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

NUMBIN (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**
specifies information about the NUMBIN= option values to use in tuning the light gradient boosting model. For more information, see the NUMBIN= option in the PROC LIGHTGRADBOOST statement.

You can specify the following additional suboptions:

LB=*number*
specifies the minimum NUMBIN= option value to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=2.

UB=*number*
specifies the maximum NUMBIN= option value to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=255.

VALUES=*value-list*
specifies a list of the NUMBIN= option values to consider during tuning, where *value-list* is a space-separated list of positive integers. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial NUMBIN= option value for the tuner to use.

By default, INIT=255.

EXCLUDE
excludes the NUMBIN= option from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

RIDGE (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**
specifies information about the RIDGE= option values to use in tuning the light gradient boosting model. For more information, see the RIDGE= option in the PROC LIGHTGRADBOOST statement.

You can specify the following additional suboptions:

LB=*number*
specifies the minimum RIDGE= option value to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=0.

UB=*number*
specifies the maximum RIDGE= option value to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=10.

VALUES=*value-list*
specifies a list of RIDGE= option values to consider during tuning, where *value-list* is a space-separated list of numbers greater than or equal to 0. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial RIDGE= option value for the tuner to use.

By default, INIT=0.

EXCLUDE
excludes the RIDGE= option from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

INPUT Statement

INPUT *variables* < / *options* > **;**

The INPUT statement names input *variables* that share common *options*. You can specify the INPUT statement multiple times.

You can specify the following *options*:

LEVEL=INTERVAL | NOMINAL
specifies the level of measurement of the variables. You can specify the following values:

INTERVAL specifies that the level of measurement of the variables is interval.

NOMINAL specifies that the level of measurement of the variables is nominal.

By default, LEVEL=INTERVAL for numeric variables and LEVEL=NOMINAL for categorical variables.

OUTPUT Statement

OUTPUT OUT=*libref.data-table* < *option* > **;**

The OUTPUT statement creates an output data table that contains the results of running PROC LIGHTGRADBOOST.

You must specify the following option:

OUT=*libref.data-table*
names the output data table for PROC LIGHTGRADBOOST to use. You must specify this option before any other options. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to where the data table is to be stored. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 512.

data-table specifies the name of the output data table.

You can also specify the following *option*:

COPYVAR=*variable*
COPYVARS=(*variables***)**
lists one or more *variables* from the input data table to transfer to the output data table.

SAVESTATE Statement

SAVESTATE RSTORE=*libref.data-table* **;**

The SAVESTATE statement creates an analytic store for the model and saves it as a binary object in a data table. You can use the analytic store in the ASTORE procedure to score new data. For more information, see Chapter 4, "ASTORE Procedure."

You must specify the following option:

RSTORE=*libref.data-table*
specifies a data table in which to save the analytic store for the model. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 512.

TARGET Statement

TARGET *variable* < / **LEVEL=NOMINAL | INTERVAL** > **;**

The TARGET statement names the variable whose values PROC LIGHTGRADBOOST predicts.

You can specify the following option:

LEVEL=NOMINAL | INTERVAL
specifies the level of measurement. You can specify the following values:

NOMINAL specifies that the level of measurement of the variables is nominal.

INTERVAL specifies that the level of measurement of the variables is interval.

By default, LEVEL=INTERVAL for numeric variables and LEVEL=NOMINAL for categorical variables.

Details: LIGHTGRADBOOST Procedure

Training a LightGBM Model

The LIGHTGRADBOOST procedure uses histogram-based algorithms, which put continuous feature values into discrete bins. This speeds up training and reduces memory usage. These histogram-based algorithms find the best split points on the basis of the feature histograms. There are costs for building histograms (number of rows × number of features) and for finding split points (number of bins × number of features). Because the number of bins is usually much smaller than the number of rows, histogram building is more computationally complex. If you can reduce the number of rows or number of features, you can substantially speed up the training.

LightGBM models support many applications, such as regression, binary classification, multiclassification, and cross-entropy.

ODS Table Names

Each table that the LIGHTGRADBOOST procedure creates has a name associated with it. You must use this name to refer to the table when you use ODS statements. The name of each table and a short description of the contents are listed in Table 19.4.

Table 19.4 ODS Tables Produced by PROC LIGHTGRADBOOST

Table Name	Description	Statement
BestConfiguration	Hyperparameters and objective function values for the best configuration	AUTOTUNE

Table 19.4 *continued*

Table Name	Description	Statement
EvaluationHistory	Values of the hyperparameters and the objective function for all configurations	AUTOTUNE
IterHistory	Accuracy metric values of the training and validation data	PROC LIGHTGRADBOOST
ModelInfo	Model information	PROC LIGHTGRADBOOST
Nobs	Number of observations	PROC LIGHTGRADBOOST
PredName	Predicted name information for interval or nominal targets	PROC LIGHTGRADBOOST
TunerInfo	Setup values that are used by the tuner	AUTOTUNE
TunerResults	Values of the hyperparameters, the objective function for the default configuration (iteration 0), and up to 10 best configurations found	AUTOTUNE
TunerSummary	Statistics about the tuning process	AUTOTUNE
TunerTiming	Total time spent on different tasks during tuning	AUTOTUNE

Examples: LIGHTGRADBOOST Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

Example 19.1: Train a LightGBM Model

This example illustrates how you can use the LIGHTGRADBOOST procedure to train a light gradient boosting machine (LightGBM) model. It uses the home equity data set Hmeq, which is in the Sampsio library.

The Hmeq data set contains 5,960 observations with 13 variables. The response variable is a binary indicator to predict which clients will default on their loan.

You can load the sampsio.hmeq data set into your SAS library in the following DATA step:

```
data mylib.hmeq;
   set sampsio.hmeq;
run;
```

These statements assume that your SAS libref is named mylib, as in the section "Using SAS Viya Workbench" on page 512, but you can substitute any appropriately defined SAS libref.

The following statements train a LightGBM model and score the training data table. The SAVESTATE statement creates an analytic store for the model and saves it as a binary object in the mylib.lgbmStore table. You can use the analytic store later in the ASTORE procedure for scoring.

```
proc lightgradboost data=mylib.hmeq validdata=mylib.hmeq
   boosting=DART objective=binary deterministic;
   input Delinq Derog Job nInq Reason / level = nominal;
   input CLAge CLNo DebtInc Loan Mortdue Value YoJ / level = interval;
   target Bad / level = nominal;
   SAVESTATE RSTORE=mylib.lgbmStore ;
   output out=mylib.lgbmout copyvars=(bad loan reason);
   ods output ModelInfo=ModelInfo;
run;
```

The following statements use a previously saved model to score the new data table mylib.lgbmScoreout by using PROC ASTORE:

```
proc astore;
   score data=mylib.hmeq out=mylib.lgbmScoreout rstore=mylib.lgbmStore ;
run; quit;
```

You can see the scoring results shown in Output 19.1.1 by using the PRINT statement:

```
proc print data=mylib.lgbmScoreout(obs=3);
run;
```

Output 19.1.1 Scoring Results from the Score File

Obs	I_BAD	P_BAD0	P_BAD1
1	1	0.16002	0.83998
2	1	0.25107	0.74893
3	1	0.11124	0.88876

In Output 19.1.1, the generated columns P_BAD0 and P_BAD1 contain the predicted probabilities of the target variable Bad with respective labels, and the generated column I_BAD contains the predicted label.

Example 19.2: Tuning the Hyperparameters of a LightGBM Model

This example illustrates how you can use the AUTOTUNE statement in the LIGHTGRADBOOST procedure to tune hyperparameters of a light gradient boosting machine (LightGBM) model. It uses the home equity data set Hmeq, which is in the Sampsio library.

The Hmeq data set contains 5,960 observations with 13 variables. The response variable is a binary indicator to predict which clients will default on their loan.

You can load the sampsio.hmeq data set into your SAS library by specifying your SAS libref in the first statement in the following DATA step:

```
data mylib.hmeq;
   set sampsio.hmeq;
run;
```

These statements assume that your SAS libref is named mylib, as in the section "Using SAS Viya Workbench" on page 512, but you can substitute any appropriately defined SAS libref.

The following statements use PROC LIGHTGRADBOOST with the AUTOTUNE statement to automatically tune the hyperparameters of a LightGBM model:

```
proc lightgradboost data=mylib.hmeq validdata=mylib.hmeq
   boosting=DART objective=binary deterministic;
   input Delinq Derog Job nInq Reason / level = nominal;
   input CLAge CLNo DebtInc Loan Mortdue Value YoJ / level = interval;
   target Bad / level = nominal;
   autotune
      /* Tuning Parameters
      You do not need to specify any tuning parameters for the default
      tuning process. If you want to make adjustments to the default
      tuning process, uncomment the following block of code and change
      any of the tuning parameters' attributes.

      tuningParameters=(
         LEARNINGRATE     ( lb=0.01  ub=1.0   init=0.1  )
         LASSO            ( lb=0.0   ub=10.0  init=0.0  )
         RIDGE            ( lb=0.0   ub=10.0  init=0.0  )
         NBINS            ( lb=2     ub=255   init=255  )
         LEAFSIZE         ( lb=1     ub=320   init=20   )
         MAXDEPTHMAXDEPTH ( lb=-1    ub=7     init=-1   )
         FEATUREFRACTION  ( lb=0.05  ub=1.0   init=1.0  )
         BAGGINGFRACTION  ( lb=0.2   ub=0.95  init=0.95 )
         BAGGINGFREQUENCY ( lb=0     ub=7     init=0    )
         NUMITERS         ( lb=1     ub=100   init=100  )
      )
      */
   ;
   /* Remove this line to see all results */
   ods select BestConfiguration EvaluationHistoryPlot IterationHistoryPlot;
run;
```

The preceding statements produce the table and the plots shown in Output 19.2.1 through Output 19.2.3. The table in Output 19.2.1 displays the evaluation number, the values of the tuning parameters, and the error metric value for the best LightGBM model that the tuner found. Note that the ODS SELECT statement limits the displayed results to a single table and two plots. You can remove this statement to display all tables and plots. For the full list of ODS tables that PROC LIGHTGRADBOOST produces, see Table 19.4.

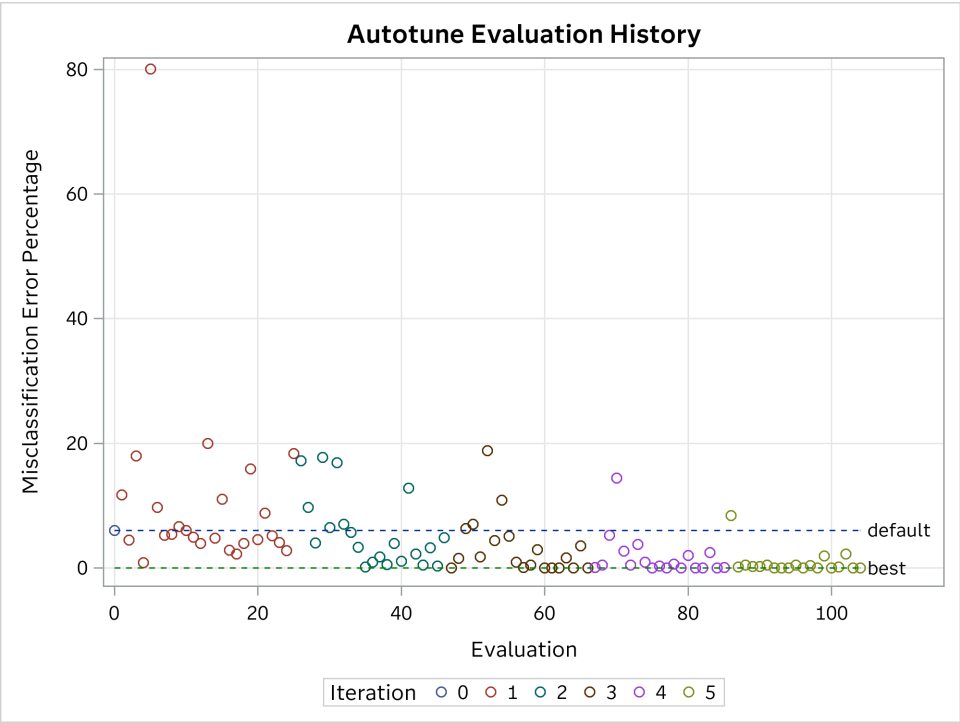
Output 19.2.1 Best Configuration Table

The LIGHTGRADBOOST Procedure

Best Configuration	
Evaluation	75
Learning Rate	0.80285636
Number of Bins	79
Maximum Tree Depth	7
Fraction of Features	0.53576104
Bagging Fraction	0.81942468
Frequency of Bagging	0
Maximum Number of Iterations	100
Misclassification Error Percentage	0.00

Output 19.2.2 displays a scatter plot of all configurations that the tuner tried. The objective values are shown on the Y axis, and the evaluation numbers are shown on the X axis.

Output 19.2.2 Evaluation History Plot



The plot in Output 19.2.3 displays how the best found objective value and the elapsed time changed with each iteration of the tuner.

Output 19.2.3 Iteration History Plot

References

Ke, G., Meng, Q., Finley, T., Wang, T., Chen, W., Ma, W., Ye, Q., and Liu, T. Y. (2017). "LightGBM: A Highly Efficient Gradient Boosting Decision Tree." In *Proceedings of the 31st International Conference on Neural Information Processing Systems*. Long Beach, CA: NIPS. https://papers.nips.cc/paper/2017/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf.

Chapter 20
LIME Procedure

Contents

Overview: LIME Procedure **538**
- PROC LIME Features 538
- Using SAS Viya Workbench 538

Getting Started: LIME Procedure **539**

Syntax: LIME Procedure **541**
- PROC LIME Statement 541
- ASTOREMODEL Statement 543
- CODE Statement 543
- DISTANCE Statement 544
- EXPLAINER Statement 544
- FREQ Statement 545
- INPUT Statement 545
- PREDICTEDTARGET Statement 545
- WEIGHT Statement 546

Details: LIME Procedure **546**
- Required Specifications 546
- LIME Value Calculation 547
- Handling Missing Data 548
- Frequency and Weight Variables 549
- Displayed Output 550
- ODS Table Names 550

Examples: LIME Procedure **551**
- Example 20.1: Modify DS2 Code That Has a DROP or KEEP Statement 551

References **553**

Overview: LIME Procedure

The LIME procedure implements the local interpretable model-agnostic explanations (LIME) method that was introduced by Ribeiro, Singh, and Guestrin (2016). The procedure explains how a machine learning model behaves near a specific query observation. To do so, PROC LIME randomly generates observations on the basis of query data and the distribution of the input data, scores the generated data by using the machine learning model, and fits a local regression model to the scored data. The local regression model uses a linear regression model with the LASSO penalty. Additionally, the observations are weighted according to their proximity to the query observation. This is in contrast to a global regression model, which does not use a query observation for a weighting scheme or data generation. The estimated coefficients from the local regression model explain how fast the model prediction changes near the query observation.

PROC LIME Features

PROC LIME has the following features:

- computes the local interpretable model-agnostic explanations (LIME) values
- computes an explainer fidelity value for model interpretation

Using SAS Viya Workbench

SAS Viya Workbench has greatly simplified the SAS programming experience by offering a syntax style nearly identical to that of SAS 9 and earlier releases. However, there are minor exceptions. In the past, SAS Viya procedures required the use of a CAS engine libref. This engine served multiple purposes, but it was primarily used to upload data to the CAS server and manage the data. Its additional purposes included support of new data types and improved performance for large data sets. No CAS server is needed for SAS Viya Workbench, because all computations are done in place for you in a single location.

In the following program, the LIBNAME statement creates a mylib libref that the examples in this chapter use to refer to your current or desired work directory:

```
libname mylib ".";
```

For more information, see *SAS Viya Workbench: Getting Started*.

Getting Started: LIME Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

This example uses the LIME procedure to explain a prediction that is made by a forest model.

The following DATA step creates the input data table mylib.dmagecr:

```
data mylib.dmagecr;
   set sampsio.dmagecr;
   id = _N_;
   keep age amount coapp duration foreign job good_bad id;
run;
```

The following DATA step creates the query data table mylib.query in your SAS library:

```
data mylib.query;
   set sampsio.dmagecr;
   keep age amount coapp duration foreign job good_bad;
   if _N_ = 20;
run;
```

The following statements use the FOREST procedure to build a forest model to predict whether the credit rating of each individual in the mylib.dmagecr data table is good or bad. The procedure also outputs an analytic store named forest_astore.

```
proc forest
   data=mylib.dmagecr
   seed=1234;
   input age amount duration / level=interval;
   input coapp foreign job /level=nominal;
   target good_bad;
   savestate rstore=mylib.forest_astore;
run;
```

The following statements run PROC LIME and produce ODS tables of the results:

```
proc lime
   data=mylib.query
   referenceData=mylib.dmagecr
   seed=12345;
   input age amount duration / level=interval;
   input coapp foreign job /level=nominal;
   predictedTarget P_good_badbad;
   astoreModel rstore=mylib.forest_astore;
run;
```

The DATA= option names the input query data table, which specifies the data to use for LIME value calculation. The REFERENCEDATA= option names the reference data table. The SEED= option specifies the seed to use for pseudorandom number generation. The two INPUT statements specify that age, amount, and duration are the interval variables and coapp, foreign, and job are the nominal variables. The

PREDICTEDTARGET statement specifies that P_good_badbad be used as the predicted target variable. The ASTOREMODEL statement specifies the analytic store that contains the model to explain.

Figure 20.1 displays the explainer information. Figure 20.2 displays the LIME values for the query observation, which are the estimated parameters of the local regression model. Figure 20.3 displays the explainer fidelity information.

Figure 20.1 Explainer Information

The LIME Procedure

Explainer Information	
Data Generation Method	Query Centered
Distance Method	Normalized Euclidean with Exponential Kernel
Explainer Type	Regression with LASSO
Binary Encoding	None
Standardize Parameter Estimates	None
Include Missing as a Level	No
Number of Samples	3000
Exponential Kernel Divisor	1.8371173071
Mixed Distance Weight	1
Seed	12345

Figure 20.2 Parameter Estimates

Parameter Estimates					
	Nominal Variables				
Variable	**coapp**	**foreign**	**job**	**Estimate**	**Query Level**
Intercept	.	.	.	0.3190324769	.
age	.	.	.	-0.005548298	31
amount	.	.	.	-5.169254E-6	3430
duration	.	.	.	0.0067004505	24
coapp	2	.	.	0.1128066607	0
coapp	3	.	.	-0.039284815	0
coapp	1	.	.	0	1
foreign	.	2	.	-0.155467523	0
foreign	.	1	.	0	1
job	.	.	1	0.0752579772	0
job	.	.	2	0.0391428793	0
job	.	.	3	0	1

Figure 20.3 Explainer Fidelity

Explainer Fidelity		
Model Prediction	**Explainer Prediction**	**Explainer RMSE**
0.3276871873	0.2901154994	0.0947577628

Syntax: LIME Procedure

The following statements are available in the LIME procedure:

PROC LIME DATA=*libref.data-table* **REFERENCEDATA=***libref.data-table* < *options* > **;**
ASTOREMODEL RSTORE=*libref.data-table* **;**
CODE FILE=*filename* | **TABLE=***libref.data-table* **;**
DISTANCE < *options* > **;**
EXPLAINER < *options* > **;**
FREQ *variable* **;**
INPUT *variables* < / **LEVEL=INTERVAL** | **NOMINAL** > **;**
PREDICTEDTARGET *variable* **;**
WEIGHT *variable* **;**

The PROC LIME, INPUT, and PREDICTEDTARGET statements are required. The ASTOREMODEL and INPUT statements can appear multiple times. The rest of this section provides detailed syntax information about each statement, beginning with the PROC LIME statement. The remaining statements are described in alphabetical order.

PROC LIME Statement

PROC LIME DATA=*libref.data-table* **REFERENCEDATA=***libref.data-table* < *options* > **;**

The PROC LIME statement invokes the procedure. Table 20.1 summarizes the options in this statement.

Table 20.1 PROC LIME Statement Options

Option	Description
DATA=	Specifies the query input data table, which contains the query observation
INCLUDEMISSING	Specifies that the missing values of nominal variables should be treated as a valid level
LOGLEVEL=	Specifies a value to control the quantity and type of notes to print to the client log
NTHREADS=	Specifies the number of threads to use in the computation
REFERENCEDATA=	Specifies the reference data table
SAMPLESIZE=	Specifies the number of observations to generate
SEED=	Specifies the seed for generating the pseudorandom numbers

You must specify the following options:

DATA=*libref.data-table*
names the input data table for PROC LIME to use. The default is the most recently created data table. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to the data. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 538.

data-table specifies the name of the input data table.

REFERENCEDATA=*libref.data-table*
specifies the reference data table. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the input data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 538.

You can also specify the following *options*:

INCLUDEMISSING
treats missing values of nominal variables as a valid level.

LOGLEVEL=0 | 1 | 2
controls the quantity and type of notes to print to the client log.

0 prints only warnings and errors.

1 prints some notes.

2 prints many notes.

By default, LOGLEVEL=1.

NTHREADS=*number-of-threads*
specifies the number of threads to use in the computation. The default value is the number of CPUs available on the machine.

SAMPLESIZE=*number*
specifies the number of observations to generate. The default is 500 times the number of variables that are listed in the INPUT statements or 1,000,000, whichever value is smaller.

SEED=*number*
specifies the seed value for pseudorandom number generation. If you do not specify a seed, or if you specify a value less than or equal to 0, the seed is generated by reading the time of day from the computer's clock.

ASTOREMODEL Statement

ASTOREMODEL RSTORE=*libref.data-table* **;**

The ASTOREMODEL statement names an analytic store that contains the model to explain. If you specify multiple ASTOREMODEL statements, you must also specify the CODE statement to name the DS2 code that is used along with the analytic stores. For more information about DS2 code and using DS2 code with analytic stores, see *SAS DS2 Language Reference*, *SAS DS2 Programmer's Guide*, and Chapter 4, "ASTORE Procedure."

You must specify the following option:

RSTORE=*libref.data-table*
specifies the analytic store that contains the model to explain. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 538.

CODE Statement

CODE FILE=*filename* | **TABLE=***libref.data-table* **;**

The CODE statement names the DATA step code that describes the model or the DS2 code that is used along with an analytic store that you specify in the ASTOREMODEL statement. If you specify multiple ASTOREMODEL statements, you must also specify the CODE statement to name the DS2 code that is used along with the analytic stores. For more information about DS2 code and using DS2 code with analytic stores, see *SAS DS2 Language Reference*, *SAS DS2 Programmer's Guide*, and Chapter 4, "ASTORE Procedure."

You must specify one of the following options:

FILE=*filename*
names the external file that contains the DATA step code that describes the model or the DS2 code to be used along with an analytic store that you specify in the ASTOREMODEL statement. When you enclose the *filename* in a quoted string (for example, FILE="c:\mydir\scorecode.sas"), this option specifies the path and file name for reading the code from an external file. You can also specify an unquoted *filename* of no more than eight characters. If the *filename* is assigned as a fileref in a Base SAS FILENAME statement, the file that you specify in the FILENAME statement is used.

TABLE=*libref.data-table*
names the data table that contains the DATA step code that describes the model or the DS2 code to be used along with an analytic store that you specify in the ASTOREMODEL statement. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the input data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 538.

DISTANCE Statement

DISTANCE < *options* > **;**

The DISTANCE statement specifies the options that are related to distance calculation.

You can specify the following *options*:

EXPONENTIALKERNEL=*number*
specifies the denominator in the exponential kernel. The default value is based on the input data, as described in the section "Distance and Weight Calculation" on page 547.

IMPUTE=MEAN | NONE | WEIGHTEDMEAN
specifies the method to use for imputing the missing observations. You can specify the following methods:

MEAN	uses the mean to impute missing values of interval variables, and uses the mode to impute missing values of nominal variables.
NONE	does not impute missing values.
WEIGHTEDMEAN	uses the weighted mean to impute missing values of interval variables, and uses the mode to impute missing values of nominal variables.

By default, IMPUTE=NONE.

MIXEDDISTANCEWEIGHT=*number*
specifies the distance calculation weighting that balances interval and nominal inputs.

By default, MIXEDDISTANCEWEIGHT=1.

EXPLAINER Statement

EXPLAINER < *options* > **;**

The EXPLAINER statement specifies the options to use in the LASSO regression.

You can specify the following *options*:

MAXEFFECTS=*number*
specifies the maximum number of effects that the LASSO regression can select.

MINEFFECTS=*number*
specifies the minimum number of effects that the LASSO regression can select.

STANDARDIZEESTIMATES=INTERVALS | NONE
specifies the variables whose parameter estimates should be standardized. You can specify the following values:

INTERVALS	specifies that the parameter estimates of all interval variables should be standardized.
NONE	specifies that no parameter estimates of any variable should be standardized.

FREQ Statement

FREQ *variable* **;**

The *variable* in the FREQ statement identifies a numeric variable in the input data table that contains the frequency of occurrence of each observation. PROC LIME treats each observation as if it appears f times, where f is the value of the FREQ *variable* for the observation. If f is not an integer, it is truncated to an integer. If f is less than 1 or missing, the observation is not used in the analysis. When you omit the FREQ statement, each observation is assigned a frequency of 1.

INPUT Statement

INPUT *variables* < / **LEVEL=INTERVAL | NOMINAL** > **;**

The INPUT statement specifies one or more *variables* as input variables. You can specify this statement multiple times.

You can specify the following option after a slash:

LEVEL=INTERVAL | NOMINAL
specifies the type of all the *variables* in the statement. You can specify the following values:

INTERVAL treats all the *variables* in the statement as interval variables.

NOMINAL treats all the *variables* in the statement as nominal variables.

By default, LEVEL=INTERVAL for numeric variables and LEVEL=NOMINAL for character variables.

PREDICTEDTARGET Statement

PREDICTEDTARGET *variable* **;**

The PREDICTEDTARGET statement names the variable that contains the model's predictions. The specified *variable* typically has a P_ prefix.

WEIGHT Statement

WEIGHT *variable* **;**

The *variable* in the WEIGHT statement is used as a weight to perform a weighted analysis of the data. Observations that have nonpositive or missing weights are not included in the analysis. The values of the WEIGHT variable can be nonintegral and are not truncated. When you omit the WEIGHT statement, all observations that are used in the analysis are assigned a weight of 1.

Details: LIME Procedure

Required Specifications

The following specifications are required in order to run the LIME procedure:

- An input data table that has a single row, which is known as the query table. This row is the observation that acts as the focus of the model explanation.
- A reference data table. Typically, this is the data table that you use to train the model.
- A list of input variables. These are the model inputs that are used to train the model. Input variables can be interval or nominal, but you must identify any numeric inputs that you want to treat as nominal by using the LEVEL=NOMINAL option in the INPUT statement. String variables are always treated as nominal.
- A scoring function. You must provide either an analytic store, several analytic stores and DS2 code, or DATA step score code that is created by the modeling algorithm.
- A predicted target variable. This is the output variable that contains the model's predictions when the scoring function is applied to the input data table. The values of the variable can represent numeric predictions or predicted probabilities for an interval or nominal target, respectively. In many SAS Viya machine learning models, the name of the variable is constructed as follows: for an interval target, "P_" is prefixed to the name of the target variable (for example, "P_MSRP"); for a nominal target, "P_" is prefixed to the name of the target variable, and the value that corresponds to the target event is appended (for example, "P_Bad1").

LIME Value Calculation

PROC LIME fits a local surrogate regression model to data that are generated around the query observation and the corresponding output predicted values from the machine learning model. The specification of the local regression model follows Ribeiro, Singh, and Guestrin (2016) and computes the parameter estimate values that are used in the LIME method. PROC LIME calculates the LIME values by using the following steps:

1. Data generation: Randomly generate observations near the query data, based on the distribution of the input data.
2. Distance and weight calculation: Calculate the distance between each generated observation and the query observation, and calculate the weight for each generated observation according to the distance.
3. Data scoring: Score the generated data by using the machine learning model.
4. Surrogate model fitting: Fit a weighted local regression model with the LASSO penalty to the scored data. The regression model uses the predicted target variable as the target variable.

Data Generation

For interval variables, PROC LIME generates data by independently sampling from a normal distribution that is centered at the value of the query observation. The variance of the normal distribution is equal to that of the reference data that you specify in the REFERENCEDATA= option. For nominal variables, PROC LIME calculates the value for each variable in a simulated observation by independently sampling from the empirical distribution of that variable as determined from the reference data that you specify in the REFERENCEDATA= option.

You can specify the number of generated observations by using the SAMPLESIZE= option. For information about how the procedure handles missing data during data generation, see the section "Handling Missing Data" on page 548.

Distance and Weight Calculation

PROC LIME uses the normalized Euclidean distance in the exponential kernel to calculate the weight of an observation. The normalized Euclidean distance, with nominal variable support, is given by

$$\mathrm{Dist}(x, q) = \|(\hat{x}_{\mathrm{interval}} - \hat{q}_{\mathrm{interval}})\|_2 + \gamma * \mathrm{Hamming}(x_{\mathrm{nominal}}, q_{\mathrm{nominal}})$$

where

- $\|\hat{x}_{\mathrm{interval}} - \hat{q}_{\mathrm{interval}}\|_2$ is the normalized Euclidean distance between the interval variables of the observation x and the query q, where
  - $\hat{x}_{\mathrm{interval}}$ represents the interval variables of the observation x, normalized over the standard deviation of each corresponding variable in the reference data, where $\hat{x}_i = \frac{x_i}{\sigma_i}$
  - $\hat{q}_{\mathrm{interval}}$ represents the interval variables of the query observation q, normalized over the standard deviation of each corresponding variable in the reference data, where $\hat{q}_i = \frac{q_i}{\sigma_i}$

  - σ_i is the standard deviation of the ith interval variable in the reference data

- Hamming$(x_{\text{nominal}}, q_{\text{nominal}})$ is the Hamming distance between the nominal variables of the observation x and the query q

- γ is the mixed-distance weight

The weight is then calculated as

$$w = e^{-\frac{\text{Dist}_{(x,q)}}{\omega^2}}$$

where ω is the exponential kernel divisor, x is the observation for which the weight is being calculated, and q is the query observation as specified in the DATA= option.

You can specify the exponential kernel divisor and mixed-distance weight by using the EXPONENTIALKERNEL= and MIXEDDISTANCEWEIGHT= options in the DISTANCE statement, respectively. By default, the exponential kernel divisor and mixed-distance weight are calculated as follows:

- By default, the value of the exponential kernel divisor is $0.75 * \sqrt{(p_1 + p_2)}$, where p_1 is the number of interval input variables that have nonzero variance and p_2 is the number of nominal input variables.

- By default, the value of the mixed-distance weight is 1.

For information about how PROC LIME handles missing data for distance and weight calculation, see the section "Handling Missing Data" on page 548.

Handling Missing Data

You can control how the procedure handles missing data by specifying the INCLUDEMISSING option in the PROC LIME statement and the IMPUTE= option in the DISTANCE statement. By default, the IMPUTE= option is set to NONE.

The INCLUDEMISSING option specifies that the procedure should consider a missing value of a nominal variable to be a valid level. This option does not affect the handling of missing data in interval variables.

- When you omit this option, a missing value of a nominal variable is not considered to be a valid level. PROC LIME ignores missing values when calculating the distances. It does not generate new data that have missing values of the interval or nominal variables.

- When you specify this option, a missing value of a nominal variable is considered to be a valid level. If the query data have a missing nominal variable, this variable is considered in the distance calculation, where the Hamming distance is 0 only if the reference data also have a missing value of the corresponding variable. PROC LIME can also generate new data that have missing values of nominal variables, if the input reference data table has missing values of the corresponding nominal variables. The proportion of the missing values is determined by the reference data table. Interval variables are still generated without missing values.

The IMPUTE= option specifies how PROC LIME imputes a missing value. When you omit the INCLUDEMISSING option, the IMPUTE= option controls the imputation behavior in both the interval and nominal variables. When you specify the INCLUDEMISSING option, the IMPUTE= option controls the imputation behavior in only the interval variables. The IMPUTE= option takes the value MEAN, NONE, or WEIGHTEDMEAN. The imputation behavior for each value is as follows:

- MEAN indicates that PROC LIME imputes missing values in the reference data and the query data by using the variable mean or the variable mode of the input reference data when applicable. If you omit the INCLUDEMISSING option, the procedure replaces missing values of an interval variable with the mean value of the corresponding variable, and it replaces missing values of a nominal variable with the most frequent level of the corresponding variable. If you specify the INCLUDEMISSING option, the procedure replaces only missing values of an interval variable with the mean value of the corresponding variable. Note that although PROC LIME imputes the missing values in the query data before sending the data to fit the local regression model, the calculation of the distance between the imputed input data and the original query data is done without imputation. This means that if the query data have a missing value of a variable and the missing values of this variable are not considered to be a valid level, PROC LIME ignores the variable when calculating the distance.
- NONE indicates that the procedure does not impute missing values in the input data or the query data.
- WEIGHTEDMEAN indicates that the procedure imputes missing values in the reference data and the query data by using the weighted variable mean or the variable mode of the input reference data when applicable. PROC LIME use the same imputation process as for the MEAN value, except that it uses the weighted variable mean for interval variables.

Frequency and Weight Variables

The FREQ statement is an optional statement that names a numeric variable that provides the frequency of each observation in the reference data table. If you do not specify a frequency variable, then the procedure assumes that the frequency of all observations is 1. PROC LIME expects nonnegative integer frequency values. If the frequency of an observation is not an integer, the procedure truncates the frequency value to the largest integer that is smaller than or equal to the value. It treats negative and missing frequencies as 0.

The WEIGHT statement is an optional statement that names a numeric variable that provides the weight of each observation in the reference data table. If you do not specify a weight variable, then PROC LIME assumes that the weight of all observations is 1. The procedure expects nonnegative weight values. It treats negative and missing weights as 0.

PROC LIME uses both the frequency and weight variables to calculate the variable mean and variance for interval variables. For nominal variables, only the frequency variable is used in the data generation and weight calculation process. PROC LIME does not use the frequency or weight variable from the query data even if the variables exist. The query data are always treated as if they have a frequency and weight of 1.

Displayed Output

The following sections describe the output that PROC LIME produces.

Explainer Fidelity Table

The "Explainer Fidelity" table contains information about how accurate the regression model is in comparison to the machine learning model. The table displays the machine learning model and regression model predictions on the query observation, as well as the overall root mean square error for the regression model.

Explainer Information Table

The "Explainer Information" table contains information about the settings that were used in the procedure. This table helps you verify that the procedure used your intended settings.

Parameter Estimates Table

The "Parameter Estimates" table contains the parameter estimates from the local regression model, along with information about the values of the input variables for the query. These estimates are the LIME values.

ODS Table Names

Each table that PROC LIME creates has a name that is associated with it. You must use this name to refer to the table when you use ODS statements. The name of each table and a short description of the contents are listed in Table 20.2.

Table 20.2 ODS Tables Produced by PROC LIME

Table Name	Description	Statement	Option
ExplainerFidelity	Indicator of the accuracy of the weighted linear regression model	PROC LIME	Default
ExplainerInfo	Information about what settings were used	PROC LIME	Default
ParameterEstimates	Parameter estimates for the weighted linear regression. These are the LIME values	PROC LIME	Default

Examples: LIME Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

Example 20.1: Modify DS2 Code That Has a DROP or KEEP Statement

This example shows how to use the LIME procedure to calculate LIME values by using an analytic store and DS2 code. It shows how to include DS2 code alongside an analytic store for model specification, as well as how to avoid a possible error.

The following DATA step creates the input data table mylib.dmagecr:

```
data mylib.dmagecr;
   set sampsio.dmagecr;
   id = _N_;
   keep age amount coapp duration foreign job good_bad id;
run;
```

The following DATA step creates the query data table mylib.query in your SAS library:

```
data mylib.query;
   set sampsio.dmagecr;
   keep age amount coapp duration foreign job good_bad;
   if _N_ = 20;
run;
```

The following statements use the FOREST procedure to build a forest model to predict whether the credit rating of each individual in the mylib.dmagecr data table is good or bad. The procedure also outputs an analytic store named forest_astore.

```
proc forest
   data=mylib.dmagecr
   maxDepth=4
   seed=1234;
   input age amount duration / level=interval;
   input coapp foreign job /level=nominal;
   target good_bad;
   savestate rstore=mylib.forest_astore;
run;
```

The following statements run the ASTORE procedure to generate DS2 code for the analytic store and save the DS2 code to a file named DS2Code.sas:

```
proc astore;
   describe rstore=mylib.forest_astore epcode="DS2Code.sas";
run;
```

The DS2 code that PROC ASTORE creates contains a KEEP statement that does not keep all the model inputs. Because of this KEEP statement, if you use the DS2 code with the analytic store in PROC LIME,

the analysis in the procedure will fail. When you use DS2 code in PROC LIME and the DS2 code includes a KEEP statement, the KEEP statement must include all input variables, and likewise a DROP statement cannot include any input variables. The full set of input variables is needed in the scored table as part of the analysis that PROC LIME performs. If you have a KEEP or DROP statement in your DS2 code, the procedure issues a warning. An error can occur if the required variables do not exist when PROC LIME uses the model for scoring.

There are several ways to ensure that your DS2 code is compatible with PROC LIME. You can manually save the DS2 code and edit any KEEP or DROP statements that it includes. Alternatively, you can use regular expressions to remove the KEEP or DROP statements.

The following DATA step modifies the DS2 code to remove the KEEP statement, and it outputs the modified DS2 code to a file named DS2code_modified.sas. The regular expression that is used here is an example; it might not always work for the DS2 code that you have.

```
data _null_;
   length allcode $30000.;
   retain allcode;
   file "DS2code_modified.sas";
   infile "DS2code.sas" end=eof;
   input;
   allcode = cats(allcode, _infile_, '0A'x);
   if eof then do;
      allcode = prxchange('s/keep[^;]*;//',1,allcode);
      put allcode;
   end;
run;
```

The following statements run PROC LIME and produce ODS tables of the results:

```
proc lime
   data=mylib.query
   referenceData=mylib.dmagecr
   seed=12345;
   input age amount duration / level=interval;
   input coapp foreign job /level=nominal;
   predictedTarget P_good_badbad;
   astoreModel rstore=mylib.forest_astore;
   code file="DS2Code_modified.sas";
run;
```

The DATA= option names the input query data table, which specifies the data to use for calculating LIME values. The REFERENCEDATA= option names the reference data table. The SEED= option specifies the seed to use for pseudorandom number generation. The two INPUT statements specify that age, amount, and duration are the interval variables and coapp, foreign, and job are the nominal variables. The PREDICTEDTARGET statement specifies that P_good_badbad be used as the predicted target variable. The ASTOREMODEL and CODE statements specify the analytic store and the DS2 code that contain the model to explain.

Output 20.1.1 displays the explainer information. Output 20.1.2 displays the LIME values for the query observation, which are the estimated parameters of the local regression model. Output 20.1.3 displays the explainer fidelity information.

Output 20.1.1 Explainer Information

The LIME Procedure

Explainer Information	
Data Generation Method	Query Centered
Distance Method	Normalized Euclidean with Exponential Kernel
Explainer Type	Regression with LASSO
Binary Encoding	None
Standardize Parameter Estimates	None
Include Missing as a Level	No
Number of Samples	3000
Exponential Kernel Divisor	1.8371173071
Mixed Distance Weight	1
Seed	12345

Output 20.1.2 Parameter Estimates

Parameter Estimates

Nominal Variables

Variable	coapp	foreign	job	Estimate	Query Level
Intercept	.	.	.	0.2931126743	.
age	.	.	.	-0.002966058	31
amount	.	.	.	7.6573524E-6	3430
duration	.	.	.	0.0028286862	24
coapp	2	.	.	0.044016531	0
coapp	3	.	.	-0.015015446	0
coapp	1	.	.	0	1
foreign	.	2	.	-0.108113816	0
foreign	.	1	.	0	1
job	.	.	1	0.0239071866	0
job	.	.	3	0	1

Output 20.1.3 Explainer Fidelity

Explainer Fidelity

Model Prediction	Explainer Prediction	Explainer RMSE
0.2824168939	0.2953180482	0.0421468777

References

Ribeiro, M. T., Singh, S., and Guestrin, C. (2016). "'Why Should I Trust You?' Explaining the Predictions of Any Classifier." In *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 1135–1144. New York: ACM.

Chapter 21
MBANALYSIS Procedure

Contents

Overview: MBANALYSIS Procedure **556**
- PROC MBANALYSIS Features 556
- Using SAS Viya Workbench 556

Getting Started: MBANALYSIS Procedure **557**

Syntax: MBANALYSIS Procedure **558**
- PROC MBANALYSIS Statement 558
- CUSTOMER Statement 560
- HIERARCHY Statement 560
- OUTPUT Statement 560
- SAVESTATE Statement 561
- TARGET Statement 561

Details: MBANALYSIS Procedure **562**
- FP-Growth Algorithm 563
- Scoring with PROC ASTORE 564
- Displayed Output 564
- Results 564
- Limitations 565

Examples: MBANALYSIS Procedure **565**
- Example 21.1: Training a Model 565

References **567**

Overview: MBANALYSIS Procedure

The MBANALYSIS (market basket analysis) procedure performs association rule mining on a transaction database. Association rules are in the form of *if* A *then* B, where A and B are items in the transaction database. Association rules helps determine association or correlation between various items in a database. These rules are used for finding the latest trends and patterns in data and also help in business decision-making processes. The MBANALYSIS procedure uses the frequent-pattern growth (FP-growth) algorithm of Han, Pei, and Yin (2000) for finding frequent item sets and then generates rules that are based on these frequent item sets. For more information about the FP-growth algorithm, see the section "FP-Growth Algorithm" on page 563.

PROC MBANALYSIS Features

The MBANALYSIS procedure has the following features:

- processes data in a single thread as well as in multiple threads
- supports hierarchical transactional data as input
- uses the FP-growth algorithm for frequent item set mining
- supports various measures of interestingness to find useful rules
- supports the SAVESTATE statement for scoring by the ASTORE procedure

Using SAS Viya Workbench

SAS Viya Workbench has greatly simplified the SAS programming experience by offering a syntax style nearly identical to that of SAS 9 and earlier releases. However, there are minor exceptions. In the past, SAS Viya procedures required the use of a CAS engine libref. This engine served multiple purposes, but it was primarily used to upload data to the CAS server and manage the data. Its additional purposes included support of new data types and improved performance for large data sets. No CAS server is needed for SAS Viya Workbench, because all computations are done in place for you in a single location.

In the following program, the LIBNAME statement creates a mylib libref that the examples in this chapter use to refer to your current or desired work directory:

```
libname mylib ".";
```

For more information, see *SAS Viya Workbench: Getting Started*.

Getting Started: MBANALYSIS Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

This example finds frequent item sets in the sampsio.assocs data set and generates rules for finding association between those frequent item sets. The sampsio.assocs data set contains 1,000 observations, each of which contains products purchased by a customer.

You can load the sampsio.assocs data set into your SAS library in the following DATA step:

```
data mylib.assocs;
   set sampsio.assocs;
run;
```

These statements assume that your SAS libref is named mylib, as in the section "Using SAS Viya Workbench" on page 556, but you can substitute any appropriately defined SAS libref.

The following statements execute the MBANALYSIS procedure on the mylib.assocs data table and generate output in the mylib.out data table:

```
proc mbanalysis data=mylib.assocs items=3 support=100;
 output out=mylib.out outfreq=mylib.outfreq outrule=mylib.outrule;
 customer Customer;
 target product;
 run;
```

The PROC MBANALYSIS statement specifies mylib.assocs as the input data table and specifies the rule generation criteria: the number of items and level of support. The OUTPUT statement specifies output tables for storing the results. The CUSTOMER statement identifies Customer as identification variable for the transactions. The TARGET statement defines Product as the target variable.

The procedure stores the list of frequent items in the mylib.outfreq table, the frequent item sets in the mylib.out table, and the generated rules in the mylib.outrule table. The mylib.outrule table contains the rules that are generated for the frequent item sets after rule generation criteria are applied.

Syntax: MBANALYSIS Procedure

The following statements are available in the MBANALYSIS procedure:

PROC MBANALYSIS *options* **;**
  CUSTOMER *variable* **;**
  HIERARCHY DATA=*libref.data-table* < *libref.data-table. . .* > **;**
  OUTPUT *options* **;**
  SAVESTATE RSTORE=*libref.data-table* **;**
  TARGET *variable* **;**

The PROC MBANALYSIS statement, the CUSTOMER statement, the TARGET statement, and the OUTPUT statement are required.

The following sections describe the PROC MBANALYSIS statement and then describe the other statements in alphabetical order.

PROC MBANALYSIS Statement

PROC MBANALYSIS *options* **;**

The PROC MBANALYSIS statement invokes the procedure.

You must specify the following *option*:

DATA=*libref.data-table*
names the input data table for PROC MBANALYSIS to use. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to the data. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 556.

data-table specifies the name of the input data table.

You must specify either of the following *options*:

SUPPORT=*number*
specifies the minimum level of support (minimum frequency of an item) for a rule, where *number* must be an integer greater than or equal to 1. This option overrides the specification of the PCTSUP= option.

PCTSUP=*number*
SUPPCT=*number*
SUP_PCT=*number*
PCTSUPPORT=*number*
specifies the minimum level of support for a rule as a percentage of the number of baskets in the input data table, where *number* must be a real number between 0 and 100, inclusive. This option is ignored if the SUPPORT= option is specified.

You can also specify the following *options*:

ANTECEDENTLIST=(*string* < *string. . .* >**)**
specifies the regular expression strings to match in the antecedent (left-hand side) of a rule.

CONF=*number*
specifies the minimum confidence for the rules, where *number* must be a real number between 0 and 100.

By default, CONF=50.

CONSEQUENTLIST=(*string* < *string. . .* >**)**
specifies the regular expression strings to match in the consequent (right-hand side) of a rule.

ITEMS=*number*
specifies the number of items in a rule, where *number* must be an integer between 1 and 1,000. By default, ITEMS=2 when you specify either an OUT= or OUTRULE= option in the OUTPUT statement; otherwise, ITEMS=1 by default.

LIFT=*number*
specifies the minimum lift value necessary to generate a rule, where *number* must be a positive, real number between 0 and 100, inclusive.

By default, LIFT=1.

MAXBSKTSZ=*number*
specifies the maximum basket size, where *number* must be a positive integer between 0 and 1,000, inclusive. Baskets whose size is larger than *number* are rejected.

By default, MAXBSKTSZ=1000.

MINBSKTSZ=*number*
specifies the minimum basket size, where *number* must be a positive integer between 0 and 20, inclusive. Baskets whose size is smaller than *number* are rejected.

By default, MINBSKTSZ=1.

NLHS_RANGE=(*number***,** *number***)**
specifies the range of number of items in the left-hand side (LHS) of a rule.

By default, NLHS_RANGE=(1,1000).

NORM
normalizes the values of the items in the output tables that are specified in the OUTPUT statement.

NRHS_RANGE=(*number***,** *number***)**
specifies the range of number of items in the right-hand side (RHS) of a rule.

By default, NRHS_RANGE=(1,1000).

SEPARATOR=*string*
specifies the separator character in the antecedent (left-hand side) or consequent (right-hand side) of a rule.

By default, SEPARATOR="&".

SUP_LIFT=*number*
specifies the minimum support lift necessary to generate a rule, where *number* must be a positive, real number.

By default, SUP_LIFT=0.

CUSTOMER Statement

CUSTOMER *variable* **;**

The CUSTOMER statement specifies the variable that is used to group the target *variable* into a basket. You can also specify this statement as ID *variable*.

HIERARCHY Statement

HIERARCHY DATA=*libref.data-table* < *libref.data-table. . .* > **;**

The HIERARCHY statement specifies one or more hierarchical input data tables. If this statement is not specified, then PROC MBANALYSIS performs simple association analysis without a hierarchy. A maximum of five levels of hierarchy is supported. Each level of hierarchy is specified in a separate data table.

You must specify the following option:

DATA=*libref.data-table* < *libref.data-table. . .* >
specifies one or more data tables that contain the hierarchy data. A maximum of five levels of hierarchy is supported. Each level of hierarchy is specified in a separate data table. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the input data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 556.

OUTPUT Statement

OUTPUT *options* **;**

The OUTPUT statement specifies output tables that contain result of the MBANALYSIS procedure.

You must specify at least one of the following options:

OUT=*libref.data-table*
creates a data table that contains information about frequent item sets with their transaction counts and support. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 556.

OUTFREQ=*libref.data-table*
creates a data table that contains information about the unique frequent items with their transaction counts and support. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 556.

OUTRULE=*libref.data-table*
creates a data table that contains information about the rules. This table includes variables that identify the left-hand side and right-hand sides of the rule, the support, and the lift. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 556.

SAVESTATE Statement

SAVESTATE RSTORE=*libref.data-table* **;**

The SAVESTATE statement creates an analytic store for the model and saves it as a binary object in a data table. You can use the analytic store in the ASTORE procedure to score new data. For more information, see Chapter 4, "ASTORE Procedure."

You must specify the following option:

RSTORE=*libref.data-table*
specifies a data table in which to save the analytic store for the model. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 556.

TARGET Statement

TARGET *variable* **;**

The TARGET statement specifies a single nominal *variable* to use as the target variable. If this *variable* is not present in the input data table, then PROC MBANALYSIS exits with an error.

Details: MBANALYSIS Procedure

The MBANALYSIS procedure performs association rule mining on transaction data in conjunction with an item that is specified in one or more hierarchies (also called groups or categories). Association rule mining is popularly known as market basket analysis. Market basket analysis is useful in retail marketing scenarios that involve tens of thousands of distinct items that are grouped in a hierarchy. PROC MBANALYSIS uses the transaction and hierarchical data and generates rules at multiple levels. These rules are called generalized association rules; they solve some of the problems, which are described below, that arise in simple market basket analysis.

Simple market basket analysis that is performed on transaction or point-of-sale data might miss potentially useful and significant associations. For example, consider a supermarket that sells different types of breads and a wide selection of wines, and assume that a large number of customers buy some type of bread with some type of wine. Simple market basket analysis might fail to spot a rule that links bread and wine at the transaction level. It computes the support for specific types of breads to specific types of wines, but it is possible that none of these support levels is large enough to generate a rule. To overcome this problem, the MBANALYSIS procedure combines hierarchies with the transaction data and computes the support for any type of bread to any type of wine.

In addition, simple association mining tends to create a large number of obvious and uninteresting rules along with the useful rules. A solution to the preceding problem that reduces the support would only exacerbate that problem. In practice, this tendency is one of the major drawbacks of association rule mining. If the support is set high, fewer rules are generated but more of them are obvious and thus useless. If the support is set low, too many rules are generated and the domain experts must evaluate the rules to determine which rules are useful. One solution to this drawback is generalized association rule mining.

The MBANALYSIS procedure enables you to generate an interestingness measure, which is called support lift. This measure is based on the deviation of a rule's support from its estimated support, which is computed based on the support of the ancestors of the rule's items. Objectively, greater deviation is more likely to be interesting. If no hierarchy of ancestors is specified, then PROC MBANALYSIS generates simple association rules.

For example, the following rule for supermarket means that if customers buy butter and bread, they also buy milk:

$$\text{butter}, \text{bread} => \text{Milk}$$

Support (coverage) is an indication of how frequently the items appear in the database. Support of a rule for A and B is expressed as

$$A => B : P(A, B) = X/Y$$

where X is the number of transactions that contain both A and B, and Y is the total number of transactions.

Confidence (accuracy) indicates the number of times the conditional statement of rule have been found to be true. Confidence of a rule for A and B is expressed as

$$A => B : P(B|A) = X/Z$$

where X is the number of transactions that contain both A and B, and Z is the number of transactions that contain A.

FP-Growth Algorithm

Association rule mining (ARM) is an important data mining task that tries to find interesting rules from a transactional data set. Association rule mining is formally described in Agrawal, Imieliński, and Swami (1993).

Many association rule mining algorithms are based on a "support-confidence" framework, which has two major steps: First, minimum support is applied to find all frequent item sets in a data set. Second, these frequent item sets and the minimum confidence constraint are used to generate rules. Discovering all frequent item sets in a data set is difficult because the task is combinatorial.

The Apriori algorithm in Agrawal and Srikant (1994) uses a breadth-first search (BFS) strategy to count the support of item sets and uses a candidate generation function that exploits the downward closure property of support in Agrawal, Imieliński, and Swami (1993). The basic idea of the downward-closure property is that the support of an item set is less than a particular threshold if the support of any subset of this item set is less than that threshold. In this way, many small item sets are excluded during the generation-and-test process. The Apriori algorithms can reduce the search space by doing a restricted generation-and-test process. The Apriori algorithm is the one of the most efficient association rule mining algorithms.

The Apriori algorithm of Agrawal and Srikant (1994) and many similar algorithms need to scan the data set n–2 times when generating an item set that contains n items. The frequent pattern growth (FP-growth) algorithm builds a frequent-pattern tree (FP-tree) by scanning the data set twice (Han, Pei, and Yin 2000; Han et al. 2004). Then it discovers the frequent item sets by traversing the FP-tree in a depth-first search (DFS). DFS can find all the frequent item sets without enumerating all the candidates in 2^n search space. Therefore, FP-growth algorithms are faster than BFS-based algorithms. The nature of the FP-growth algorithm is restricted-test-only without generation of the n-item set.

The FP-tree construction algorithm first scans the data set and then collects the set of frequent items, F, and their supports. Next, it sorts F in support-descending order as L, the list of frequent items. Frequent items are sorted in frequency-descending order so that there is a better chance that more prefix strings can be shared.

An FP-tree has two components: an item prefix tree and a header table of frequent items. Each node in the item prefix tree consists of three fields: item-name, count, and node-link. Here item-name registers which item this node represents, count registers the number of transactions represented by the portion of the path that reaches this node, and node-link links to the next node in the FP-tree that carries the same item name, or null if there is none. Each entry in the frequent-item header table consists of two (or three) fields: item-name, count (frequency count of this one-item set), and head of node-link.

When items are sorted in support-descending order, more frequent items are arranged closer to the top of the FP-tree and thus are more likely to be shared. This arrangement suggests that an FP-tree is usually a highly compact projection of the entire data set. However, if the data set is large, the FP-tree might not fit in memory. There are two ways to distribute the data:

- Parallel projection is done in parallel: Each observation is projected to every node. The problem is that the total size of the projected data sets could be multiple times the size of the original data set.
- Partition projection is done in sequence: An observation T is projected to the X-projected data set only if X is a frequent item in T and there is no other item after X in the list of frequent items that appears in the observation. The advantage of partition projection is that the total size of the projected data sets at each level is smaller than the original data set.

The frequent item sets are mined by traversing a node's conditional pattern base (under the condition of the node's existence) and thus is called conditional FP-tree (CFPT). The mining function is defined as mine(CFPT|node), which is also a recursive function that uses a divide-and-conquer process.

Scoring with PROC ASTORE

The SAVESTATE statement stores the analytical results in a binary file, which can be used as an input file for scoring with the ASTORE procedure. The transactional data table is in long format and PROC ASTORE accepts input in wide format, so the input data format must be converted from long to wide before the data are provided as input to PROC ASTORE.

The scoring process is supported by three algorithms, which you can specify in the SETOPTION statement in the ASTORE procedure. Select the algorithm you want to use by specifying SETOPTION ARM_SCORING_ALGO *number*, where *number* is one of the following values:

1 uses a rule association algorithm. For a particular input item set, a rule is selected if its antecedent and consequent item sets are included in the input item set.

2 uses an exclusive recommendation algorithm. For a particular input item set, a rule is selected if its antecedent item set is a subset of the input item set and its consequent item set is not a subset of the input item set.

3 uses a recommendation algorithm. For a particular input item set, a rule is selected if its antecedent item set is a subset of the input item set.

In order to maintain consistency, the output of PROC ASTORE needs to be converted from wide to long format.

Displayed Output

The MBANALYSIS procedure does not produce any displayed output tables.

Results

The results that the MBANALYSIS procedure produces are stored in tables. You can access these tables by their names, which are shown in Table 21.1.

Table 21.1 Results Tables Produced by PROC MBANALYSIS

Table Name	Description	Procedure	Option
out	Contains general information about frequent item sets	PROC MBANALYSIS	OUT
outfreq	Contains information about the items' frequencies	PROC MBANALYSIS	OUTFREQ
outrule	Contains detailed information about rules	PROC MBANALYSIS	OUTRULE

Table 21.1 *continued*

Table Name	Description	Procedure	Option
savestate	Contains the output binary scoring file	PROC MBANALYSIS	SAVESTATE

Limitations

When the number of observations is greater than 10 million and the number of items in a rule (as specified in the ITEMS= option in the PROC MBANALYSIS statement) is considerably large (more than 10), the amount of memory required to process the analysis might be so great that the procedure stops responding.

When one item has multiple parents, only the last parent is considered for generating rules.

The maximum number of rules that are generated per thread is one million.

Examples: MBANALYSIS Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

Example 21.1: Training a Model

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

This example trains the model by using data that are available in the sampsio.Assocs data set. This data set contains 1,000 transactions of 1,000 customers; each transaction contains information about the customer, time, and item or product. The customer (ID) variable is named Customer. The target variable is named Product, and the time variable is ignored. To perform association discovery, the input data table must have a separate observation for each product that is purchased by each customer. You must assign the ID role to one variable and the target model role to another variable when you create the data source. Only one variable is supported as the customer variable and only one variable is supported as the target variable.

You can load the sampsio.assocs data set into your SAS library in the following DATA step. These statements assume that your SAS libref is named mylib, but you can substitute any appropriately defined SAS libref.

```
data mylib.assocs;
   set sampsio.assocs;
run;
```

The following DATA steps create two hierarchies, mylib.firstLevel and mylib.secondLevel:

```
data mylib.firstLevel;
  length Item $20 Category $20;
    Item='apples'; Category='Fruits'; output;
    Item='artichok'; Category='Vegetables'; output;
    Item='avocado'; Category='Fruits'; output;
    Item='baguette'; Category='Bread'; output;
    Item='bordeaux'; Category='Wine'; output;
    Item='bourbon'; Category='Whisky'; output;
    Item='chicken'; Category='Non-Veg'; output;
    Item='coke'; Category='Cold-Drink'; output;
    Item='corned_b'; Category='Non-Veg'; output;
    Item='cracker'; Category='Vegetables'; output;
    Item='ham'; Category='Non-Veg'; output;
    Item='heineken'; Category='Beer'; output;
    Item='hering'; Category='Non-Veg'; output;
    Item='ice_crea'; Category='Cold-Drink'; output;
    Item='olives'; Category='Vegetables'; output;
    Item='peppers'; Category='Fruits'; output;
    Item='sardines'; Category='Non-Veg'; output;
    Item='soda'; Category='Cold-Drink'; output;
    Item='steak'; Category='Non-Veg'; output;
    Item='turkey'; Category='Non-Veg'; output;
 run;
data mylib.secondLevel;
  length Category $20 Dept $20;
    Category='Fruits'; Dept='Breakfast'; output;
    Category='Vegetables'; Dept='Breakfast'; output;
    Category='Bread'; Dept='Breakfast'; output;
    Category='Wine'; Dept='Drinks'; output;
    Category='Whisky'; Dept='Drinks'; output;
    Category='Non-Veg'; Dept='Breakfast'; output;
    Category='Cold-Drink'; Dept='Drinks'; output;
    Category='Beer'; Dept='Drinks'; output;
 run;
```

The following statements run the association analysis on the mylib.assocs data table:

```
proc mbanalysis data=mylib.assocs items=10 support=1 separator="\*"
   nLHS_range=(2,3) nRHS_range=(4,5) antecedentList=("Fruits")
   consequentList=("Non-Veg");
   output out=mylib.out outfreq=mylib.outfreq outrule=mylib.outrule;
   customer Customer;
   target product;
   hierarchy data = mylib.firstLevel mylib.secondLevel;
   savestate rstore=mylib.myStateArm;
run;
```

The DATA= option names the input data table to be analyzed. The ITEMS= option specifies the number of items in a rule. The SUPPORT= option specifies the minimum level of support for a rule. The SEPARATOR= option specifies separator character in the antecedent (left-hand side) or consequent (right-hand side) of a rule. The NLHS_RANGE= option specifies range of number of items in the left-hand side (LHS) of a rule. The NRHS_RANGE= option specifies range of number of items in the right-hand side (RHS) of a rule. The

ANTECEDENTLIST= option specifies the regular expression strings to match in the antecedent of a rule. The CONSEQUENTLIST= option specifies the regular expression strings to match in the consequent of a rule. The OUTPUT statement specifies output tables to contain results of the MBANALYSIS procedure. The CUSTOMER statement requests that the Customer variable be used to group the target variable into baskets. The TARGET statement requests that the Product variable be used as the target variable. The HIERARCHY statement specifies the item's hierarchies. The RSTORE= option in the SAVESTATE statement specifies the binary scoring file to be used for scoring.

References

Agrawal, R., Imieliński, T., and Swami, A. (1993). "Mining Association Rules between Sets of Items in Large Databases." In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, 207–216. New York: ACM Press. https://doi.org/10.1145/170035.170072.

Agrawal, R., and Srikant, R. (1994). "Fast Algorithms for Mining Association Rules in Large Databases." In *Proceedings of the 20th International Conference on Very Large Data Bases*, 487–499. San Francisco: Morgan Kaufmann. http://www.vldb.org/conf/1994/P487.PDF.

Han, J., Pei, J., and Yin, Y. (2000). "Mining Frequent Patterns without Candidate Generation." In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*. New York: ACM Press. https://www.cs.sfu.ca/~jpei/publications/sigmod00.pdf.

Han, J., Pei, J., Yin, Y., and Mao, R. (2004). "Mining Frequent Patterns without Candidate Generation: A Frequent-Pattern Tree Approach." *Data Mining and Knowledge Discovery* 8:53–87.

Chapter 22
MTLEARN Procedure

Contents

Overview: MTLEARN Procedure **570**
PROC MTLEARN Features 570
Using SAS Viya Workbench 570
Getting Started: MTLEARN Procedure **571**
Syntax: MTLEARN Procedure **574**
PROC MTLEARN Statement 575
AUTOTUNE Statement 577
DISPLAY Statement 581
DISPLAYOUT Statement 582
INPUT Statement 583
OUTPUT Statement 583
SAVESTATE Statement 584
TARGET Statement 584
Details: MTLEARN Procedure **584**
Examples of Using the GRAPHTYPE= Option 585
Missing Values 590
Displayed Output 590
ODS Table Names 591
Output Data Tables 592
Examples: MTLEARN Procedure **592**
Example 22.1: Tuning the Hyperparameters of a Multitask Learning Model 592

Overview: MTLEARN Procedure

The MTLEARN procedure implements the multitask learning technique for least squares loss with ℓ_1 and graph structure penalizations. It solves multiple related sparse linear regression problems simultaneously. A graph structure encodes the relationships between the problems. PROC MTLEARN shares the data and model parameters among different regression problems and solves the problems simultaneously in order to produce a more robust and accurate predictive model. When used with independent graph tables, PROC MTLEARN also serves as an elastic net regularization solver, which is widely used in feature selection.

PROC MTLEARN Features

PROC MTLEARN offers the following features:

- returns multiple related linear regression models by using an implementation of the accelerated proximal gradient descent optimizing algorithm
- exploits all the available cores and concurrent threads

Using SAS Viya Workbench

SAS Viya Workbench has greatly simplified the SAS programming experience by offering a syntax style nearly identical to that of SAS 9 and earlier releases. However, there are minor exceptions. In the past, SAS Viya procedures required the use of a CAS engine libref. This engine served multiple purposes, but it was primarily used to upload data to the CAS server and manage the data. Its additional purposes included support of new data types and improved performance for large data sets. No CAS server is needed for SAS Viya Workbench, because all computations are done in place for you in a single location.

In the following program, the LIBNAME statement creates a mylib libref that the examples in this chapter use to refer to your current or desired work directory:

```
libname mylib ".";
```

For more information, see *SAS Viya Workbench: Getting Started*.

Getting Started: MTLEARN Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

This example shows how to use the MTLEARN procedure to obtain regression models from observations in a data table. This example assumes that the SAS 9 engine libref "mylib" and the SASVIYA engine libref "mylib" have already been created.

The following DATA step creates the input data table, toyData. This data table contains seven variables: id is the row ID variable; X1, X2, and X3 are the input variables; and Y1, Y2, and Y3 are the target variables.

```
data mylib.toydata;
   input id X1 X2 X3 Y1 Y2 Y3;
   cards;
   0  0.912621   1.288759   1.568040   2.813182   2.792080    2.792080
   1 -0.696839  -1.487805   1.383056   0.187085   0.559484    0.559484
   2  0.970062   2.405601  -1.638553  -1.903736  -2.301208   -2.301208
   3  0.291515  -0.176170  -0.175393   3.842113   3.799427    3.799427
   4  0.721477   0.472668  -0.522688   5.171737   4.906125    .
   5  1.563781  -1.04760   -0.298725  21.160617  20.762561    .
   6  0.426927   1.319816  -0.022467  -2.231491  -2.361400    .
   7 -1.046538  -0.190017  -1.75929   -9.438538  -9.388092    .
   8  0.155960   0.299715  -0.366969   0.109865  -0.00258535  .
   9  1.568768  -0.900524   0.334290  20.422254  19.799980   19.799980
   ;
run;
```

The following DATA step creates the input graph table, mylib.toyr. This data table encodes the relationships between the targets. In this example, the targets Y2 and Y3 are connected, meaning that the regression weights for Y2 and Y3 are expected to be similar. This data table contains four variables: id is the row ID variable, and Y1, Y2, and Y3 are the target variables.

```
data mylib.toyr;
   input id Y1 Y2 Y3;
   cards;
   0   0   1  -1
   ;
run;
```

The following statements run PROC MTLEARN and output the results to ODS tables:

```
proc mtlearn data = mylib.toydata
   graphType = CUSTOM
   regL1 = 0.5
   regL2 = 1.0
   maxIter = 10
   tolerance = 1e-2
   seed = 123
   graphTable = mylib.toyr
   modelOut = mylib.mtl_outW
   graphOut = mylib.mtl_outR
```

```
    ;
    input X1 X2 X3;
    target Y1 Y2 Y3;
    output out = mylib.mtl_out copyvars = (id X1 X2 X3);
    savestate rstore = mylib.mtl_ast;
    ods select ModelInfo NObs DescStats OptIterHistory;
run;
```

GRAPHTYPE=CUSTOM indicates the use of a custom graph table that is specified in the GRAPHTABLE= option; REGL1=0.5 specifies the value of the ℓ_1 penalization weight; REGL2=1.0 specifies the value of the ℓ_2 penalization weight; MAXITER=10 specifies the maximum number of iterations; TOLERANCE=1E–2 specifies the optimization tolerance as a stopping criterion; SEED=123 specifies the seed to use for pseudorandom number generation; GRAPHTABLE=MYLIB.TOYR specifies the user-defined graph table; MODELOUT=MYLIB.MTL_OUTW writes the estimated multitask regression weights to the data table mylib.mtl_outW; GRAPHOUT=MYLIB.MTL_OUTR writes the graph table to the data table mylib.mtl_outR. The INPUT statement specifies that the X1, X2, and X3 variables be used as inputs. The TARGET statement specifies that the Y1, Y2, and Y3 variables be used as targets. The OUTPUT statement writes the scored results to the data table mylib.mtl_out, and the COPYVARS= option copies the id, X1, X2, and X3 variables to the output. The SAVESTATE statement with the RSTORE= option stores the estimated model in the data table mylib.mtl_ast for future scoring.

Figure 22.1 shows the values of the parameters that are used in multitask learning.

Figure 22.1 Model Information

The MTLEARN Procedure

Model Information	
Seed	123
L1 Regularization	0.5
L2 Regularization	1
Tolerance	0.01
Maximum Iterations	10

Figure 22.2 shows the "Number of Observations" information. Note that missing values are not allowed in the input variables, and any row that has at least one nonmissing target is used in the optimization. Because all rows in the mylib.toyData data table have at least one target that is not missing, the number of observations read and the number of observations used are equal in this example.

Figure 22.2 Number of Observations

Number of Observations Read	10
Number of Observations Used	10

Figure 22.3 shows statistics for each interval variable in the INPUT and TARGET statements, including the mean and standard deviation.

Figure 22.3 Interval Variable Statistics

Interval Variables		
Variable	Mean	Std Dev
X1	0.486773	0.863056
X2	0.198444	1.215561
X3	-0.149870	1.083567
Y1	4.013309	9.723587
Y2	3.856637	9.545267
Y3	4.929953	8.636507

Figure 22.4 shows the iteration history of the objective function.

Figure 22.4 Iteration History

Iteration History	
Iteration	Objective Function
0	381.53329458
1	168.56137127
2	76.435700052
3	39.95948095
4	27.044926296
5	23.227784499
6	22.553144392
7	22.695112662
8	22.769024386
9	22.603979523

The following statements use the PRINT procedure to extract the first 10 observations from the output score table. The results are shown in Figure 22.5.

```
proc print noobs data=mylib.mtl_out(obs=10);
run;
```

Figure 22.5 Multitask Learning Score Table

id	X1	X2	X3	P_Y1	P_Y2	P_Y3
0	0.91262	1.28876	1.56804	2.8956	2.6659	2.6911
3	0.29152	-0.17617	-0.17539	3.8039	3.7416	3.7899
6	0.42693	1.31982	-0.02247	-2.1305	-2.2776	-2.3144
9	1.56877	-0.90052	0.33429	20.2176	19.9016	20.1579
1	-0.69684	-1.48781	1.38306	0.2040	0.4411	0.4553
4	0.72148	0.47267	-0.52269	4.9828	4.7928	4.8505
7	-1.04654	-0.19002	-1.75929	-9.6041	-9.3888	-9.5053
2	0.97006	2.40560	-1.63855	-1.9203	-2.2583	-2.3010
5	1.56378	-1.04760	-0.29873	20.8952	20.5755	20.8414
8	0.15596	0.29972	-0.36697	0.1176	0.0649	0.0640

The following PROC PRINT statements display the estimated regression weights table, which is shown in Figure 22.6:

```
proc print noobs data=mylib.mtl_outW;
run;
```

Figure 22.6 Output Regression Weights Table

Variable	_W_Y1_	_W_Y2_	_W_Y3_
X1	10.0906	9.86395	9.98828
X2	-4.8786	-4.91644	-4.98455
X3	-0.0165	0.00000	-0.00036

The following PROC PRINT statements display the graph table, which is shown in Figure 22.7.

```
proc print noobs data=mylib.mtl_outR;
run;
```

Figure 22.7 Output Graph Table

id	Y1	Y2	Y3
0	0	1	-1

Syntax: MTLEARN Procedure

The following statements are available in the MTLEARN procedure:

PROC MTLEARN < *options* > **;**
  AUTOTUNE < *options* > **;**
  DISPLAY < *table-list* > < / *options* > **;**
  DISPLAYOUT *table-spec-list* < / *options* > **;**
  INPUT *variables* < **LEVEL=INTERVAL** > **;**
  TARGET *variables* < **LEVEL=INTERVAL** > **;**
  OUTPUT OUT=*libref.data-table* < *options* > **;**
  SAVESTATE RSTORE=*libref.data-table* < *options* > **;**

The PROC MTLEARN statement, the INPUT statement, and the TARGET statement are required. You can specify multiple INPUT statements. You can specify one or multiple targets in the TARGET statement.

The following sections describe the PROC MTLEARN statement and then describe the other statements in alphabetical order.

PROC MTLEARN Statement

PROC MTLEARN < *options* > **;**

The PROC MTLEARN statement invokes the procedure. Table 22.1 summarizes the *options* available in the PROC MTLEARN statement.

Table 22.1 PROC MTLEARN Statement Options

Option	Description
Input Data Table Options	
DATA=	Specifies the input data table
GRAPHTABLE=	Specifies the user-defined graph table
Multitask Learning Options	
GRAPHTYPE=	Specifies the type of graph table
MAXITER=	Specifies the maximum number of iterations
NTHREADS=	Specifies the number of threads to use
REGL1=	Specifies the ℓ_1 (LASSO) penalization weight
REGL2=	Specifies the ℓ_2 graph penalization weight
SEED=	Specifies the seed to use for pseudorandom number generation
TOLERANCE=	Specifies the optimization tolerance (absolute ℓ_2 difference of solution) as a stopping criterion
Output Data Table Options	
GRAPHOUT=	Specifies the output data table in which to save the graph table
MODELOUT=	Specifies the output data table in which to save the estimated multitask regression weights

You can specify the following *options*:

DATA=*libref.data-table*

names the input data table for PROC MTLEARN to use. The default is the most recently created data table. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to the data. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 570.

data-table specifies the name of the input data table.

The data table must include one variable named id as the row ID variable.

GRAPHOUT=*libref.data-table*

specifies the output data table in which to save the graph table. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 570.

GRAPHTABLE=*libref.data-table*

specifies the user-defined graph table. The first column of the graph table must contain the row ID variable. The graph table must also include all target variables that you specify in the TARGET statement. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the input data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 570.

GRAPHTYPE=CLUSTER | CUSTOM | FUSE | INDEP

specifies the type of graph table.

You can specify one of the following values:

CLUSTER generates a cluster graph table, where all tasks are connected (to a virtual mean task).

CUSTOM uses the graph table that you specify.

FUSE generates a fuse graph table, where each task is connected to the next task in the target list.

INDEP generates an independent graph table, where all tasks are independent.

If you specify the GRAPHTABLE= option, the value of the GRAPHTYPE= option is automatically overwritten with CUSTOM.

By default, GRAPHTYPE=CLUSTER.

MAXITER=*number*

specifies the maximum number of iterations for the algorithm to perform, where *number* is an integer greater than or equal to 1.

By default, MAXITER=100. You can tune this value by using the AUTOTUNE statement.

MODELOUT=*libref.data-table*

specifies the output data table in which to save the estimated multitask regression weights. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 570.

NTHREADS=*number*

specifies the number of threads to use for the computation, where *number* is an integer greater than 0. The default value is the maximum number of available threads per computer.

REGL1=*number*

specifies the ℓ_1 (LASSO) penalization weight, where *number* is greater than or equal to 0. The default value is 0.01. You can tune this value by using the AUTOTUNE statement.

REGL2=*number*

specifies the ℓ_2 graph penalization weight, where *number* is greater than or equal to 0. The default value is 0.01. You can tune this value by using the AUTOTUNE statement.

SEED=*random-seed*

specifies an integer that is used to start the pseudorandom number generator. This option enables you to reproduce the same sample output, but only when NTHREADS=1. If you do not specify a seed, or if you specify a value less than or equal to 0, the seed is generated from reading the time of day from the computer's clock.

TOLERANCE=*number*

specifies the optimization tolerance (absolute ℓ_2 difference of solution) as a stopping criterion, where *number* is greater than 0. The default value is 1E–6. You can tune this value by using the AUTOTUNE statement.

AUTOTUNE Statement

AUTOTUNE < *options* > **;**

The AUTOTUNE statement searches for the best combination of values of the MAXITER=, REGL1=, REGL2=, and TOLERANCE= options in the PROC MTLEARN statement.

Table 22.2 summarizes the *options* that you can specify in the AUTOTUNE statement. For more information about all options except the TUNINGPARAMETERS= option, see the option's description in the section "AUTOTUNE Statement" on page 6 in Chapter 2, "Shared Concepts." The TUNINGPARAMETERS= option is described following Table 22.2.

Table 22.2 AUTOTUNE Statement Options

Option	Description
APPENDLOOKUP	Specifies that the table specified in the HISTORYTABLE= option contain the rows from the table specified in the LOOKUPTABLE= option
CONSTRAINTS=	Specifies the constraints on the output metrics
EVALHISTORY=	Specifies how to report the evaluation history of the tuner
FOLDCOLUMN=	Specifies the column in the data table in which the cross validation fold value is indicated
FRACTION=	Specifies the fraction of observations to use for validation
HISTORYTABLE=	Specifies the table that contains the evaluation history
KFOLD=	Specifies the number of folds for *k*-fold cross validation
LIVEUPDATE	Specifies that the table specified in the HISTORYTABLE= option be updated at every evaluation
LOOKUPTABLE=	Specifies the table to use for evaluation lookup
MAXBAYES=	Specifies the maximum number of points in the kriging model
MAXEVALS=	Specifies the maximum number of evaluations
MAXITER=	Specifies the maximum number of iterations when SEARCHMETHOD=GA or SEARCHMETHOD=BAYESIAN
MAXTIME=	Specifies the maximum time for all iterations
MAXTRAINTIME=	Specifies the maximum time for a model training
NCONVITER=	Specifies the number of convergence iterations

Table 22.2 *continued*

Option	Description
NOGRIDSHUFFLE	Requests that the grid points not be shuffled
NOLOCALSEARCH	Disables local search optimization
NPARALLEL=	Specifies the number of parallel sessions
NSUBSESSIONWORKERS=	Specifies the number of workers in parallel sessions
OBJECTIVE=	Specifies the objective function
POPSIZE=	Specifies the population size when SEARCHMETHOD=GA or SEARCHMETHOD=BAYESIAN
SAMPLESIZE=	Specifies the sample size when SEARCHMETHOD=LHS or SEARCHMETHOD=RANDOM
SEARCHMETHOD=	Specifies the search method that the optimizer uses
SECONDOBJECTIVE=	Specifies the second objective to use for tuning
SELECTINITPOINT	Specifies that the tuner select the best evaluation from the lookup table
TARGETEVENT=	Specifies the target event for ROC-based calculations
TRAINFRACTION=	Specifies the fraction of observations to use for training
TUNINGPARAMETERS=	Specifies the custom tuning parameters
USEPARAMETERS=	Specifies how to handle the TUNINGPARAMETERS= option

TUNINGPARAMETERS=(*suboption* | **...** | **<** *suboption* **>)**

TUNEPARMS=(*suboption* | **...** | **<** *suboption* **>)**

specifies which parameters to tune and which ranges to tune over. If USEPARAMETERS=STANDARD, this option is ignored.

You can specify one or more of the following *suboptions*:

MAXITER (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**

specifies information about the maximum number of iterations to use for tuning the multitask learning model. For more information, see the MAXITER= option in the PROC MTLEARN statement.

You can specify the following additional suboptions:

LB=*number*

specifies the minimum number of iterations to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=100.

UB=*number*

specifies the maximum number of iterations to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=5000.

VALUES=*value-list*
specifies a list of numbers of trees to consider during tuning, where *value-list* is a space-separated list of positive integers. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial number of iterations for the tuner to use.

By default, INIT=100.

EXCLUDE
excludes the number of iterations from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

REGL1 (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**
specifies information about the ℓ_1 penalization weight to use for tuning the multitask learning model. For more information, see the REGL1= option in the PROC MTLEARN statement.

You can specify the following additional suboptions:

LB=*number*
specifies the minimum ℓ_1 penalization weight to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=0.

UB=*number*
specifies the maximum ℓ_1 penalization weight to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=1000.

VALUES=*value-list*
specifies a list of values to consider for the ℓ_1 penalization weight during tuning, where *value-list* is a space-separated list of numbers greater than or equal to 0. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial ℓ_1 penalization weight for the tuner to use.

By default, INIT=0.01.

EXCLUDE
excludes the ℓ_1 penalization weight from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

REGL2 (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**
specifies information about the ℓ_2 penalization weight to use for tuning the multitask learning model. For more information, see the REGL2= option in the PROC MTLEARN statement.

You can specify the following additional suboptions:

LB=*number*
specifies the minimum ℓ_2 penalization weight to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=0.

UB=*number*
specifies the maximum ℓ_2 penalization weight to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=1000.

VALUES=*value-list*
specifies a list of values to consider for the ℓ_2 penalization weight during tuning, where *value-list* is a space-separated list of numbers greater than or equal to 0. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial ℓ_2 penalization weight for the tuner to use.

By default, INIT=0.01.

EXCLUDE
excludes the ℓ_2 penalization weight from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

TOLERANCE (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**
specifies information about the optimization tolerance to use for tuning the multitask learning model. For more information, see the TOLERANCE= option in the PROC MTLEARN statement.

You can specify the following additional suboptions:

LB=*number*
specifies the minimum optimization tolerance to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=1E–6.

UB=*number*
specifies the maximum optimization tolerance to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=1E–3.

VALUES=*value-list*
specifies a list of values to consider for the optimization tolerance during tuning, where *value-list* is a space-separated list of numbers greater than 0. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial optimization tolerance for the tuner to use.

By default, INIT=1E–6.

EXCLUDE
excludes the optimization tolerance from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

DISPLAY Statement

DISPLAY < *table-list* > < / *options* > **;**

The DISPLAY statement enables you to specify a list of display tables to display or exclude. This statement is similar to the ODS SELECT, ODS EXCLUDE, and ODS TRACE statements. However, the DISPLAY statement can improve performance when a large number of tables could be generated (such as in BY-group processing). The DISPLAY statement produces only the requested tables, but ODS statements generate all the tables and then keep the requested tables.

If you use both DISPLAY and ODS statements together, the DISPLAY statement takes precedence over the ODS statements. Note that the ODS EXCLUDE statement processes tables after they have been filtered by the DISPLAY statement. In some cases, it might appear that the ODS EXCLUDE statement is taking precedence because it can further filter the tables. For more information about ODS, see *SAS Output Delivery System: Procedures Guide*.

You can specify the *table-list* as a list of table names, paths, partial pathnames, and regular expressions.

The table names that you can specify are listed in the section "ODS Table Names" on page 591. A path is a table name that is prefixed with dot-separated grouping information. For example, a SelectionSummary table that a procedure produces during a selection routine might have the path *Bygroup1.Summary.SelectionSummary*. A partial pathname does not include all groups; for example, *SelectionSummary* and *Summary.SelectionSummary* are partial pathnames for *Bygroup1.Summary.SelectionSummary*.

When you specify a table name or partial pathname, all display tables whose paths end in the specified name are selected for display or exclusion. For example, both *SelectionSummary* and *Summary.SelectionSummary* select *Bygroup1.Summary.SelectionSummary*.

A regular expression is enclosed in forward slashes (/). For example, specifying "/tions/" selects all pathnames that contain the substring "tions"; in particular, the *Bygroup1.Summary.SelectionSummary* table is selected. Specifying "!/tions/" selects all pathnames that do not contain the substring "tions"; in particular, the *Bygroup1.Summary.SelectionSummary* table is not selected.

You can specify the following *options* after a slash (/):

CASESENSITIVE
performs a case-sensitive comparison of table names in the *table-list* to display table names when tables are subsetted for display. To preserve case, you must enclose table names in the *table-list* in quotation marks.

EXCLUDE
displays all display tables except those that you specify in the *table-list*.

EXCLUDEALL
suppresses display of all tables. This option takes precedence over the other options.

TRACE
displays the display table names, labels, and paths.

DISPLAYOUT Statement

DISPLAYOUT *table-spec-list* < / *options* > **;**

The DISPLAYOUT statement enables you to create output tables from your displayed output. This statement is similar to the ODS OUTPUT statement. For more information about ODS, see *SAS Output Delivery System: Procedures Guide*.

The *table-spec-list* specifies a list of output tables to create. Each entry in the list has either a *key*=*value* format or a *key* format:

key=*value* specifies *key* as the ODS table name, path, or partial pathname, and specifies *value* as the output table name.

key specifies *key* as the ODS table name and also as the output table name.

The ODS table names that you can specify are listed in the section "ODS Table Names" on page 591. You cannot specify the ODS table named OutputCasTables in the *table-spec-list*.

Table names and partial pathnames are discussed under the DISPLAY statement. The DISPLAYOUT statement does not support regular expressions.

You can specify the following *options* after a slash (/):

INCLUDEALL
creates output tables for all display tables. The name of the created output table is the same as the name of the corresponding display table. If you specify this option, the *table-spec-list* specification is ignored.

NOREPLACE
does not replace any existing output table of the same name.

INPUT Statement

INPUT *variables* < **LEVEL=INTERVAL** > **;**

The INPUT statement specifies the names of the *variables* to use in the multitask regression. It names one or more input variables that use common options. If you want to use different options for different variables, you can specify multiple INPUT statements.

You can include the following option in each INPUT statement:

LEVEL=INTERVAL
specifies the level of measurement of the *variables*. You can specify the following value:

INTERVAL specifies that the level of measurement of the *variables* is interval.

You must specify at least one interval input variable. PROC MTLEARN currently accepts only interval input variables.

OUTPUT Statement

OUTPUT OUT=*libref.data-table* < *options* > **;**

The OUTPUT statement creates an output data table to contain the results of the procedure run.

You must specify the following *option*:

OUT=*libref.data-table*
names the output data table for PROC MTLEARN to use. You must specify this option before any other options. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to where the data table is to be stored. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 570.

data-table specifies the name of the output data table.

You can also specify the following *option*:

COPYVAR=*variable*
COPYVARS=(*variables***)**
copies one or more *variables* from the input data table to the output data table.

SAVESTATE Statement

SAVESTATE RSTORE=*libref.data-table* < *options* > **;**

The SAVESTATE statement creates an analytic store for the model and saves it as a binary object in a data table. You can use the analytic store in the ASTORE procedure to score new data. For more information, see Chapter 4, "ASTORE Procedure."

You must specify the following option:

RSTORE=*libref.data-table*
specifies a data table in which to save the analytic store for the model. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 570.

TARGET Statement

TARGET *variables* < **LEVEL=INTERVAL** > **;**

The TARGET statement names the target variables. The targets must be of interval type and must be different from the variables in the INPUT statement. You must specify at least one target variable. Missing values in the target variables are allowed. You can specify the following option:

LEVEL=INTERVAL
specifies that the level of measurement of the *variables* is interval.

PROC MTLEARN currently accepts only interval target variables.

Details: MTLEARN Procedure

Consider a total number of K different tasks. For each task $i = 1, \ldots, K$, there are n_i observations in the observation matrix $X_i \in \mathbb{R}^{n_i \times p}$. The targets are denoted by $y_i \in \mathbb{R}^{n_i}$. Let $W = [W_1, \ldots, W_K] \in \mathbb{R}^{p \times K}$ be the model that you want to learn, such that

$$y_i = X_i W_i + \epsilon_i$$

where ϵ_i is random noise, and you want to learn W_i from the observations X_i and the targets y_i.

The model W is estimated by solving the following optimization problem,

$$\widehat{W} = \arg\min_{W} \frac{1}{2} \sum_{i=1}^{K} \|X_i W_i - y_i\|_2^2 + \frac{1}{2}\rho_{\ell_2} \|W R^{\top}\|_F^2 + \rho_{\ell_1} \|W\|_1$$

where R is the graph table (matrix) that represents the relationships between K tasks. Note that the specific form of R can be generated in various ways. One simple way is to model the relationships among tasks by

using an undirected graph, where each task is a node, and the two nodes are connected if the two tasks are related. To encode the connections, let

$$\mathcal{E} = \{e_k | k = 1, \ldots, n_e\}$$

be the set of edges in the graph, where $n_e = |\mathcal{E}|$ is the total number of edges in the graph. Define a matrix

$$R \in \mathbb{R}^{|\mathcal{E}| \times K}$$

where

$$R_{k,i} = \begin{cases} 1, \text{ if } e_k \text{ connects targets } i \text{ and } j, i < j \\ -1, \text{ if } e_k \text{ connects targets } i \text{ and } j, i > j \\ 0, \text{ otherwise} \end{cases}$$

Examples of Using the GRAPHTYPE= Option

The GRAPHTYPE= option values that you can specify are CLUSTER, INDEP, FUSE, and CUSTOM. When the GRAPHTABLE= option is not specified, the MTLEARN procedure can automatically generate a graph table if GRAPHTYPE=CLUSTER, INDEP, or FUSE. If GRAPHTYPE=CUSTOM, the MTLEARN procedure does not generate a graph table and instead expects a graph table to be specified in the GRAPHTABLE= option. If a graph table is specified in GRAPHTABLE= option, PROC MTLEARN uses the user-specified graph table, and the value of the GRAPHTYPE= option is overwritten with CUSTOM.

- When GRAPHTYPE=CLUSTER and no GRAPHTABLE= option is specified, PROC MTLEARN generates a cluster graph table, where all tasks are connected (to a virtual mean task). The graph table $R \in \mathbb{R}^{K \times K}$ is defined as

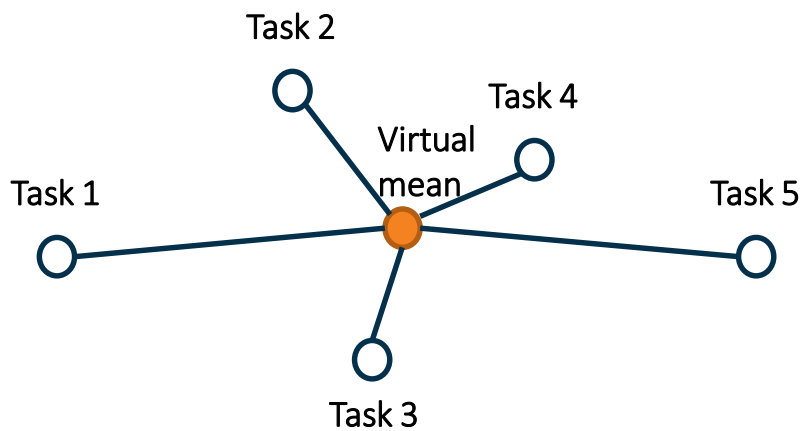
  $$R_{k,i} = \begin{cases} 1, \text{ if } k = i \\ -\frac{1}{K-1}, \text{ otherwise} \end{cases}$$

  When $K = 1$ (only one target variable), the value of the graph table is set to 1. Figure 22.8 shows an example of a cluster graph structure of five tasks, whose corresponding graph table matrix is

  $$R = \begin{bmatrix} 1 & -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} \\ -\frac{1}{4} & 1 & -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} \\ -\frac{1}{4} & -\frac{1}{4} & 1 & -\frac{1}{4} & -\frac{1}{4} \\ -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & 1 & -\frac{1}{4} \\ -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & -\frac{1}{4} & 1 \end{bmatrix}$$

  The cluster graph type is best used when all tasks are similarly related to one another.

Figure 22.8 Cluster Graph Structure with Five Targets (Tasks 1 to 5)



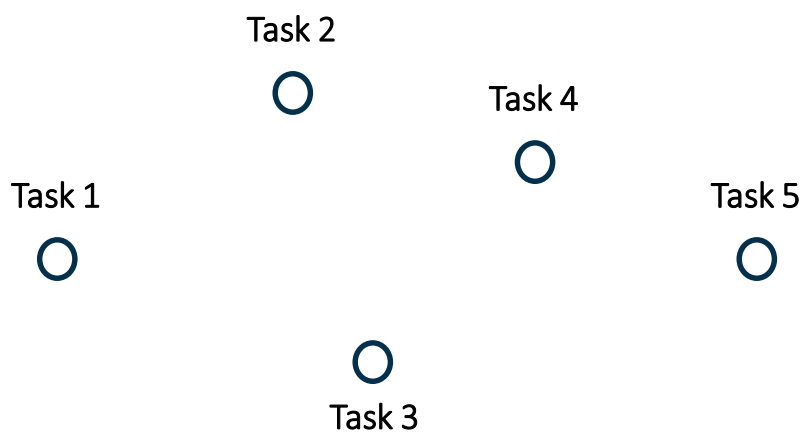
- When GRAPHTYPE=INDEP and no GRAPHTABLE= option is specified, PROC MTLEARN generates an independent graph table, where all tasks are independent. The graph table matrix $R \in \mathbb{R}^{K \times K}$ is an identity matrix,

$$R_{k,i} = \begin{cases} 1, & \text{if } k = i \\ 0, & \text{otherwise} \end{cases}$$

When $K = 1$ (only one target variable), the value of the graph table is set to 1. Figure 22.9 shows an example of an independent graph structure of five tasks, whose corresponding graph table matrix is

$$R = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{bmatrix}$$

Figure 22.9 Independent Graph Structure with Five Targets (Tasks 1 to 5)



The independent graph type is best used when all tasks are independent of one another. It can also be used for elastic net regression problems.

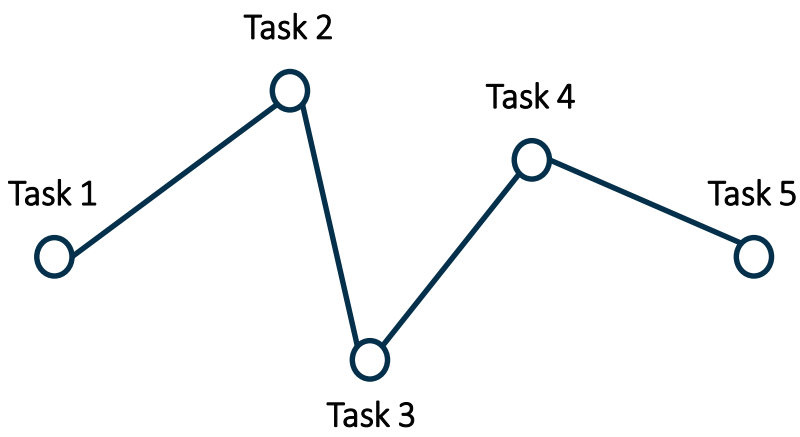
- When GRAPHTYPE=FUSE and no GRAPHTABLE= option is specified, PROC MTLEARN generates a fuse graph table, where each task is connected to the next task in the target list. The graph table $R \in \mathbb{R}^{(K-1)\times K}$ is defined as

$$R_{k,i} = \begin{cases} 1, \text{ if } k = i \\ -1, \text{ if } k = i+1 \\ 0, \text{ otherwise} \end{cases}$$

When $K = 1$ (only one target variable), the value of the graph table is set to 0. Figure 22.10 shows an example of a fuse graph structure of five tasks, whose corresponding graph table matrix is

$$R = \begin{bmatrix} 1 & -1 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 1 & -1 \end{bmatrix}$$

Figure 22.10 Fuse Graph Structure with Five Targets (Tasks 1 to 5)



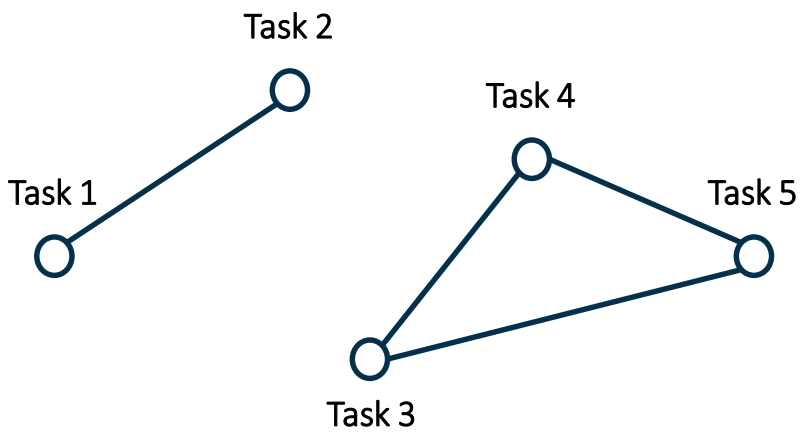
The fuse graph type is best used when tasks have known orders, such as temporal relationships.

- When GRAPHTYPE=CUSTOM, the MTLEARN procedure uses the graph table you specify in the GRAPHTABLE= option. The first column of the custom graph table must contain the row ID variable. The custom graph table must also include all target variables that you specify in the TARGET statement.

  Figure 22.11 shows an example of a custom graph. Using the definition of R near the beginning of the section "Details: MTLEARN Procedure" on page 584, the corresponding graph table matrix is

$$R = \begin{bmatrix} 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 1 & 0 & -1 \\ 0 & 0 & 0 & 1 & -1 \end{bmatrix}$$

Figure 22.11 Example of a Custom Graph Structure with Five Targets (Tasks 1 to 5)



The following DATA step creates the custom graph table shown in Figure 22.8. This data table contains six variables: id is the row ID variable, and Task1, Task2, Task3, Task4, and Task5 are the target variables.

```
data customR;
  input id Task1 Task2 Task3 Task4 Task5;
  cards;
  0   1  -1   0   0   0
  1   0   0   1  -1   0
  2   0   0   1   0  -1
  3   0   0   0   1  -1
  ;
run;
```

The following statements display the generated graph table, which is shown in Output 22.12.

```
proc print noobs data=customR;
run;
```

Figure 22.12 Example of a Custom Graph Table

id	Task1	Task2	Task3	Task4	Task5
0	1	-1	0	0	0
1	0	0	1	-1	0
2	0	0	1	0	-1
3	0	0	0	1	-1

Missing Values

In the MTLEARN procedure, missing values are not allowed in the input variables, but they are allowed in the target variables. For any observation in the input table x_k, let the corresponding target variables be $y_i, i = 1, \ldots, K$. Let $\mathcal{S} = \{1, \ldots, K\}$ be the set of all task indices, $\mathcal{S}_M$ be the set of indices whose target variables are missing, and $\mathcal{S}_N = \mathcal{S} \backslash \mathcal{S}_M$ be the set of indices whose target variables are present. PROC MTLEARN considers that the observation $x_k \in X_i$, where $i \in \mathcal{S}_N$, and $x_k \notin X_j$, where $j \in \mathcal{S}_M$.

If all target variables are missing for an observation, the observation is skipped and not used in the optimization. However, the observation is still scored in the output data table.

Displayed Output

The MTLEARN procedure displays various tables that are related to the multitask regression. The following sections describe the output tables in the order of their appearance when the related options are specified.

Model Information

The "Model Information" table displays basic information about the parameters that are used in the multitask regression analysis. This information includes the seed value, ℓ_1 penalization weight, ℓ_2 penalization weight, tolerance value, and maximum number of iterations.

Number of Observations

The "Number of Observations" table displays the number of observations that are read from the input data table and used.

Interval Variables

The "Interval Variables" table shows the mean and the standard deviation of the interval variables.

Iteration History

The "Iteration History" table displays the iteration history of the objective function.

OutputCasTables Table

The OutputCasTables table is a special table that has information about each table that is created. The information for each table consists of the table name, the library in which the table resides, and the number of columns and rows in the table. Because this table is not a typical ODS table that contains analytical results, you cannot include it in the *table-spec-list* in the DISPLAYOUT statement.

ODS Table Names

Each table that the MTLEARN procedure creates has a name associated with it, and you must use this name to refer to the table when you use ODS statements. The name of each table and a short description of its contents are listed in Table 22.3.

Table 22.3 ODS Tables Produced by PROC MTLEARN

Table Name	Description	Statements	Option
BestConfiguration	Hyperparameters and objective function values for the best configuration	AUTOTUNE	Default
DescStats	Descriptive statistics for interval variables	INPUT and TARGET	LEVEL=INTERVAL
EvaluationHistory	Values of the hyperparameters and the objective function for all configurations	AUTOTUNE	Default
HyperparameterImportance	Relative importance of hyperparameters	AUTOTUNE	Default
IterationHistory	Tuner iteration history	AUTOTUNE	Default
ModelInfo	Model information	PROC MTLEARN	Default
NObs	Number of observations	PROC MTLEARN	Default
OptIterHistory	Iteration history	PROC MTLEARN	Default
OutputCASTables	See the section "OutputCasTables Table" on page 590	PROC MTLEARN	Default
TunerCasOutputTables	Tables created during tuning	AUTOTUNE	Default
TunerInfo	Setup values that are used by the tuner	AUTOTUNE	Default
TunerResults	Values of the hyperparameters, the objective function for the default configuration (iteration 0), and up to 10 best configurations found	AUTOTUNE	Default
TunerSummary	Statistics about the tuning process	AUTOTUNE	Default
TunerTiming	Total time spent on different tasks during tuning	AUTOTUNE	Default

Output Data Tables

The MTLEARN procedure creates two data tables:

- The output graph table lists the value of the graph table. You specify the name of this data table in the GRAPHOUT= option in the PROC MTLEARN statement.
- The output model table lists the value of the estimated multitask regression weights. You specify the name of this data table in the MODELOUT= option in the PROC MTLEARN statement.

Examples: MTLEARN Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

Example 22.1: Tuning the Hyperparameters of a Multitask Learning Model

This example illustrates how you can use the AUTOTUNE statement in the MTLEARN procedure to tune hyperparameters of a multitask learning model.

This example assumes that the SAS 9 engine libref mylib and the SASVIYA engine libref mylib have already been created.

The following DATA step creates the input data table mylib.toyData. This data table contains seven variables: id is the row ID variable; X1, X2, and X3 are the input variables; and Y1, Y2, and Y3 are the target variables.

```
data mylib.toydata;
   input id X1 X2 X3 Y1 Y2 Y3;
   cards;
   0  0.912621   1.288759   1.568040   2.813182   2.792080    2.792080
   1 -0.696839  -1.487805   1.383056   0.187085   0.559484    0.559484
   2  0.970062   2.405601  -1.638553  -1.903736  -2.301208   -2.301208
   3  0.291515  -0.176170  -0.175393   3.842113   3.799427    3.799427
   4  0.721477   0.472668  -0.522688   5.171737   4.906125    4.923472
   5  1.563781  -1.04760   -0.298725  21.160617  20.762561   20.592648
   6  0.426927   1.319816  -0.022467  -2.231491  -2.361400   -2.472938
   7 -1.046538  -0.190017  -1.75929   -9.438538  -9.388092   -9.392649
   8  0.155960   0.299715  -0.366969   0.109865  -0.00258535 -0.0210573
   9  1.568768  -0.900524   0.334290  20.422254  19.799980   19.799980
   ;
run;
```

The following DATA step creates the input graph table mylib.toyr. This data table encodes the relationships between the targets. In this example, the targets Y2 and Y3 are connected, meaning that the regression weights for Y2 and Y3 are expected to be similar. This data table contains four variables: id is the row ID variable; and Y1, Y2, and Y3 are the target variables.

```
data mylib.toyr;
   input id Y1 Y2 Y3;
   cards;
   0   0   1  -1
   ;
run;
```

The following statements run PROC MTLEARN and display the results in ODS tables:

```
proc mtlearn data = mylib.toydata
   graphType  = CUSTOM
   regL1      = 20
   regL2      = 500
   maxIter    = 1000
   tolerance  = 5e-4
   seed       = 12345
   graphTable = mylib.toyr
   modelOut   = mylib.mtl_outW
   graphOut   = mylib.mtl_outR
   ;
   input X1 X2 X3;
   target Y1 Y2 Y3;
   autotune
      /* Tuning Parameters
      You do not need to specify any tuning parameters for the default
      tuning process. If you want to make adjustments to the default
      tuning process, uncomment the following block of code and change
      any of the tuning parameters' attributes.

      tuningParameters=(
         MAXITERS  ( lb=100  ub=5000   init=100  )
         TOLERANCE ( lb=1E-6 ub=1E-3   init=1E-6 )
         REGL1     ( lb=0.0  ub=1000.0 init=0.01 )
         REGL2     ( lb=0.0  ub=1000.0 init=0.01 )
      )
      */
   ;
   /* Remove this line to see all results */
   ods select BestConfiguration EvaluationHistoryPlot IterationHistoryPlot;
run;
```

The preceding statements produce the table and the plots shown in Output 22.1.1 through Output 22.1.3. The table in Output 22.1.1 displays the evaluation number, the values of the tuning parameters, and the error metric value for the best multitask learning model that the tuner found. Note that the ODS SELECT statement limits the displayed results to a single table and two plots. You can remove this statement to display all tables and plots. For the full list of ODS tables that PROC MTLEARN produces, see Table 22.3.

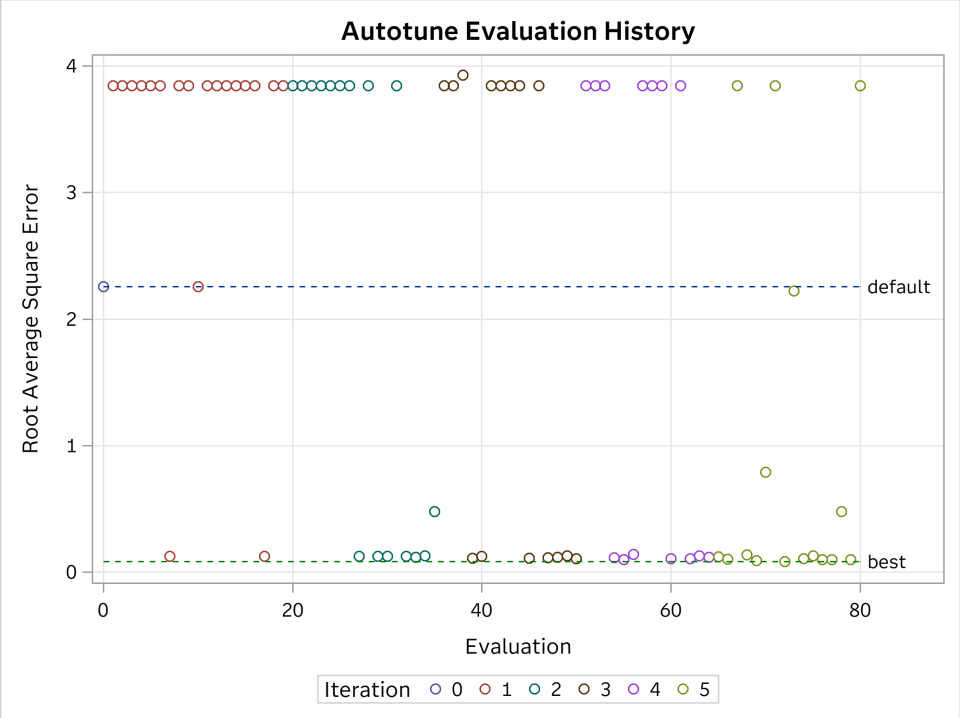
Output 22.1.1 Best Configuration Table

The MTLEARN Procedure

Best Configuration	
Evaluation	72
Maximum Number of Iterations	459
Tolerance	0.001
L1 (LASSO) Penalization Weight	0
Graph Penalization Weight	814.141471
Root Average Square Error	0.0825157251

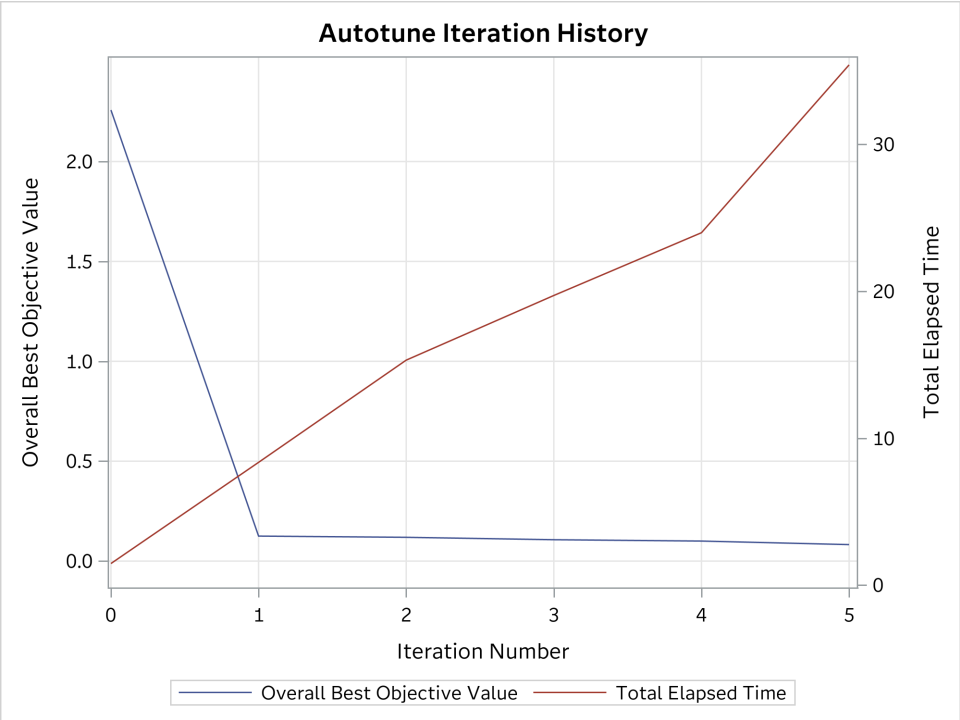
Output 22.1.2 displays a scatter plot of all configurations that the tuner tried. The objective values are shown on the Y axis, and the evaluation numbers are shown on the X axis.

Output 22.1.2 Evaluation History Plot



The plot in Output 22.1.3 displays how the best found objective value and the elapsed time changed with each iteration of the tuner.

Output 22.1.3 Iteration History Plot



Chapter 23
MWPCA Procedure

Contents

Overview: MWPCA Procedure **598**
- PROC MWPCA Features 598
- Using SAS Viya Workbench 598

Getting Started: MWPCA Procedure **599**

Syntax: MWPCA Procedure **602**
- PROC MWPCA Statement 603
- DISPLAY Statement 604
- DISPLAYOUT Statement 605
- ID Statement 606
- INPUT Statement 606
- OUTPUT Statement 606
- RPCA Statement 609
- SVD Statement 610

Details: MWPCA Procedure **610**
- Cumulative Eigenvalue Percentage Tolerance 611
- Incremental Singular Value Decomposition 611
- Displayed Output 611
- ODS Table Names 612

Examples: MWPCA Procedure **612**
- Example 23.1: Anomaly Detection of Parking Lot Floodlights 612
- Example 23.2: Comparing the Incremental SVD Solver with the Full SVD Solver . . 616

References **619**

Overview: MWPCA Procedure

The MWPCA procedure implements principal component analysis over sliding windows that consist of a specified number of observations. You can use this procedure to capture changes in principal components over time. Also, you can choose to perform robust principal component analysis (RPCA) on each window; in RPCA, the outliers and noise are excluded from each window before the analysis is performed.

One important application of MWPCA is for detecting relative changes in parts of a system compared to the overall system. By tracing the principal components over time, you can determine whether significant changes in the principal components can be a cause for concern.

PROC MWPCA stores the requested principal components for each window in an output table that is produced by the OUTPUT statement.

PROC MWPCA Features

The MWPCA procedure has the following features:

- is multithreaded during all phases of analytic execution
- supports large-scale data
- enables analysis to be done on selected columns of the data
- provides the option to perform robust principal component analysis (RPCA) in addition to the standard principal component analysis (PCA) on each window
- tracks changes of the first principal component over sliding windows

Using SAS Viya Workbench

SAS Viya Workbench has greatly simplified the SAS programming experience by offering a syntax style nearly identical to that of SAS 9 and earlier releases. However, there are minor exceptions. In the past, SAS Viya procedures required the use of a CAS engine libref. This engine served multiple purposes, but it was primarily used to upload data to the CAS server and manage the data. Its additional purposes included support of new data types and improved performance for large data sets. No CAS server is needed for SAS Viya Workbench, because all computations are done in place for you in a single location.

In the following program, the LIBNAME statement creates a mylib libref that the examples in this chapter use to refer to your current or desired work directory:

```
libname mylib ".";
```

For more information, see *SAS Viya Workbench: Getting Started*.

Getting Started: MWPCA Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

The following simple example shows how to use the MWPCA procedure to monitor the operation of four wind turbines. The data, which are simulated, are the hourly energy (in kilowatts) that each turbine produces.

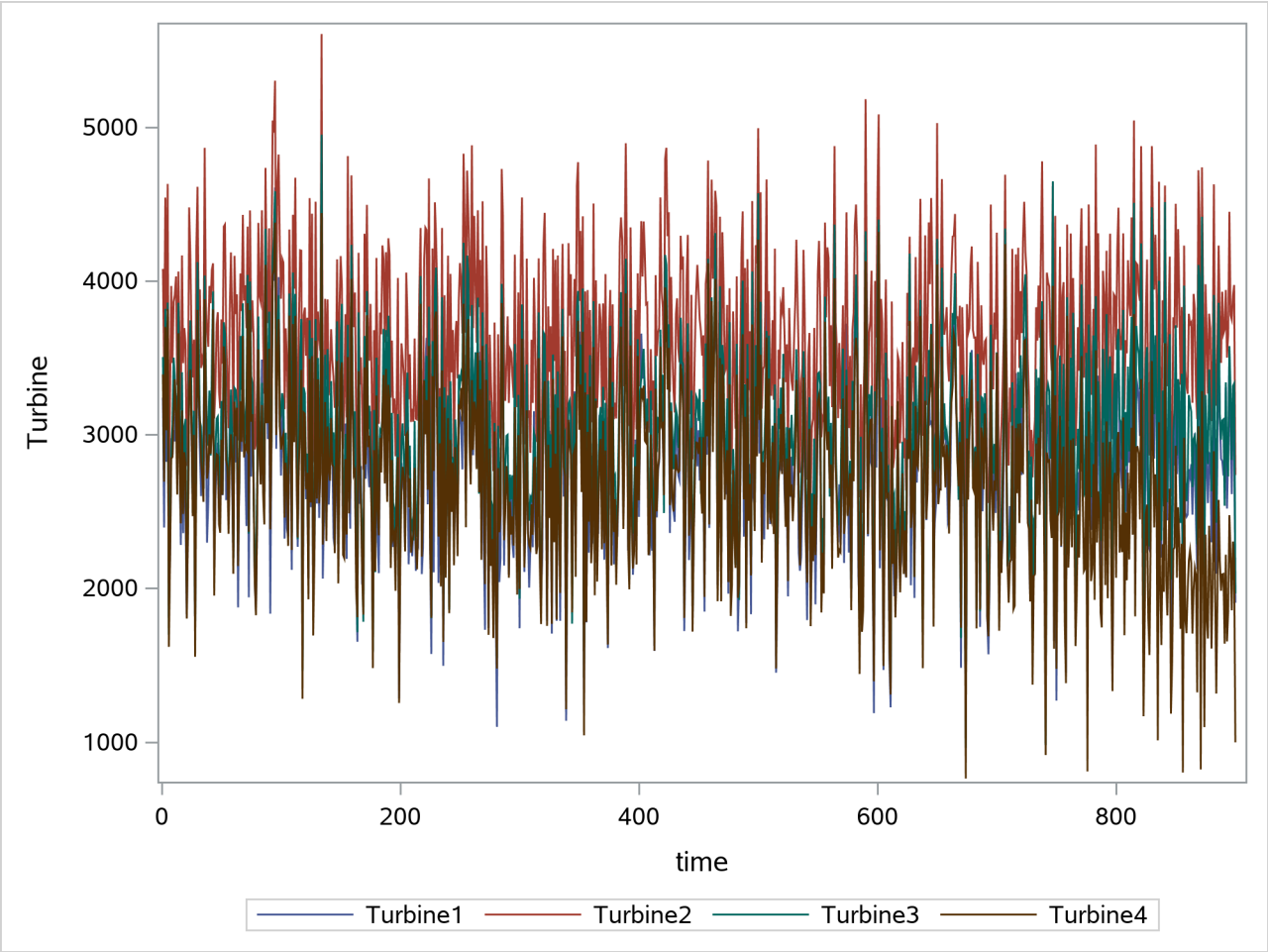
The following DATA step creates the Turbines data set:

```
data turbines;
input time Turbine1 Turbine2 Turbine3 Turbine4;
datalines;
1   3241.250445   4079.490559   3505.854227   3392.565663
2   2395.881828   3410.130913   2783.411447   2694.114311
3   3628.878452   4543.677282   3818.729737   3701.380754
4   2823.608204   3806.751908   3221.480613   3029.188791
5   3743.196614   4632.571843   3859.732421   3817.242149
6   1807.899427   2536.174173   1698.306145   1619.159414
7   2240.006055   3271.723656   2477.616826   2210.066151

... more lines ...
```

The following statements plot the amount of energy that each turbine produces over time. Output 23.1 shows the resulting plot.

```
proc sgplot data =turbines;
   series x= time y= Turbine1/legendlabel = "Turbine1" lineattrs = (thickness = 0.1);
   series x= time y= Turbine2/legendlabel = "Turbine2" lineattrs = (thickness = 0.1);
   series x= time y= Turbine3/legendlabel = "Turbine3" lineattrs = (thickness = 0.1);
   series x= time y= Turbine4/legendlabel = "Turbine4" lineattrs = (thickness = 0.1);
   yaxis label="Turbine";
run;
```

Figure 23.1 Energy Generated (Kilowatts)



You can load Turbines into your SAS library by using your libref in the first statement of the following DATA step:

```
data mylib.turbines;
set turbines;
run;
```

The following statements run PROC MWPCA:

```
proc mwpca data=mylib.turbines windowsize=200 stepsize=1;
id time;
output out=mylib.windowpcs standardpc pcangles;
run;
```

Note that the STANDARDPC option is used in the OUTPUT statement. Using this option is recommended.

Figure 23.2 displays the "Model Information," "Dimensions," and "Results Summary" ODS tables. In the "Model Information" table, you can see the values of the WINDOWSIZE= and STEPSIZE= options and some other options that are set to their default values. A window size of 200 and step size of 1 indicates that the procedure starts with the first 200 observations in the data table, calculates the principal components, moves

ahead one observation and uses observations 2–201 to calculate the principal components, and continues in this manner until the entire data table has been used.

In the "Dimensions" table, you can see the number of observations and variables in the input table. The MWPCA procedure ignores observations that have missing values; however, the ID column cannot be missing for any observation.

In the "Results Summary" table, you can see that the mylib.windowpcs data table contains 701 windows, each of which contains 200 observations. You can also see that the solution status is optimal; that is, the PCA algorithm ran successfully for all windows.

Figure 23.2 ODS Tables

Model Information	
Data Source	TURBINES
Window Size	200
Step Size	1
SVD Method	Eigenvalue Decomposition
Cumulative Eigenvalue Percent Tolerance	1

Number of Observations	900
Number of Variables	4

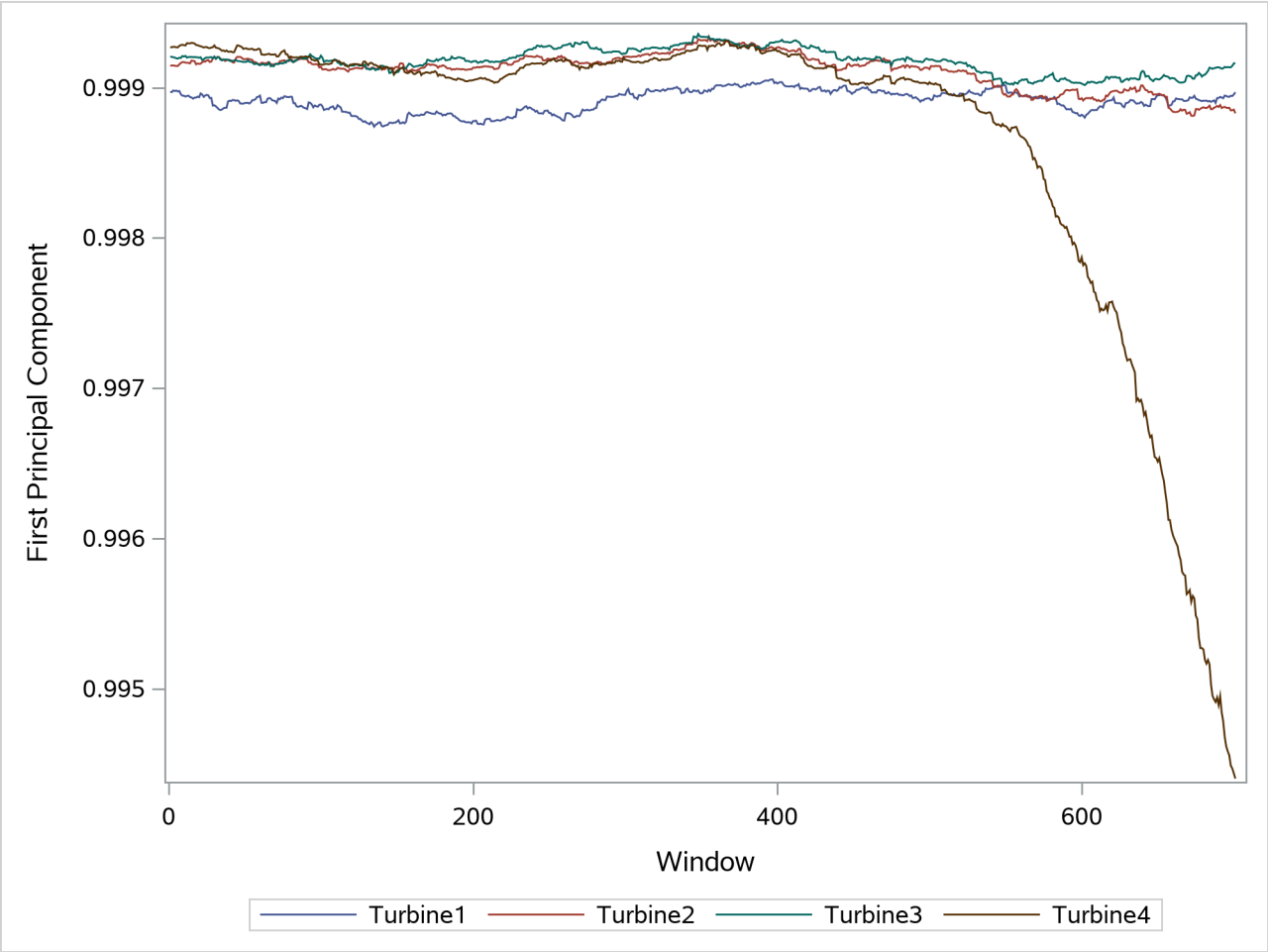
Results Summary	
Number of Windows	701
Solution Status	Optimal
Run Time (Seconds)	1.65

You can see the output table windowpcs in your mylib library. This table contains the elements of the first principal component for each turbine and information about how it changes over time. The following statements plot the value of the first principal component over time for each turbine. Figure 23.3 shows the resulting plot.

```
proc sgplot data=mylib.windowpcs;
series x=window_id y=Turbine1/legendlabel="Turbine1" lineattrs=(thickness=1);
series x=window_id y=Turbine2/legendlabel="Turbine2" lineattrs=(thickness=1);
series x=window_id y=Turbine3/legendlabel="Turbine3" lineattrs=(thickness=1);
series x=window_id y=Turbine4/legendlabel="Turbine4" lineattrs=(thickness=1);
yaxis label="First Principal Component";
xaxis label="Window";
run;
```

As you can see in Figure 23.3, the first principal component displays the same pattern of changes for the first three turbines. However, toward the end, the first principal component for the fourth turbine starts to behave differently from the other three turbines, indicating an abnormality in the operation of the fourth turbine.

Figure 23.3 First Principal Component



Syntax: MWPCA Procedure

The following statements are available in the MWPCA procedure:

```
PROC MWPCA STEPSIZE=number WINDOWSIZE=number < options > ;
    ID variable ;
    INPUT variables ;
    RPCA < options > ;
    SVD < options > ;
    OUTPUT OUT=libref.data-table < options > ;
    DISPLAY < table-list > < / options > ;
    DISPLAYOUT table-spec-list < / options > ;
```

The PROC MWPCA and the ID statements are required. The following sections describe the PROC MWPCA statement and then describe the other statements in alphabetical order.

PROC MWPCA Statement

PROC MWPCA STEPSIZE=*number* **WINDOWSIZE**=*number* < *options* > **;**

The PROC MWPCA statement invokes the procedure. Table 23.1 summarizes the *options* available in the PROC MWPCA statement.

Table 23.1 PROC MWPCA Statement Options

Option	Description
Input Data Table Option	
DATA=	Specifies the input data table
Other Options	
CENTER	Centers the input data
CUMEIGPCTTOL=	Specifies the significance level of eigenvalues that determine the rank of the low-rank matrix in each window
NTHREADS=	Specifies the maximum number of threads to use on each computation node
ROBUST	Uses robust principal component analysis on each window
STEPSIZE=	Specifies the step size
WINDOWSIZE=	Specifies the window size

You must specify the following options:

STEPSIZE=*number*
specifies the step size, which is the number of observations between the beginnings of two consecutive windows.

WINDOWSIZE=*number*
specifies the number of observations in each window.

You can also specify the following *options*:

CENTER
centers the observations by the mean of each column for each window.

CUMEIGPCTTOL=*number*
specifies the significance level of eigenvalues that determine the rank of the low-rank matrix in each window. This value must be between 0 and 1. By default, CUMEIGPCTTOL=1. For more information, see the section "Cumulative Eigenvalue Percentage Tolerance" on page 611.

DATA=*libref.data-table*
names the input data table for PROC MWPCA to use. The default is the most recently created data table. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to the data. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 598.

data-table specifies the name of the input data table.

NTHREADS=*number-of-threads*
specifies the number of threads to use in the computation. The default value is the lesser of the number of threads available per computation node and 16.

ROBUST
uses the robust principal component analysis (RPCA) method to obtain principal components for each window. You can specify the RPCA options in the RPCA statement. For more information about the RPCA method, see Chapter 30, "RPCA Procedure." If you do not specify this option, the regular PCA method is used on each window. It is recommended not to use this option for anomaly detection.

DISPLAY Statement

DISPLAY < *table-list* > < / *options* > **;**

The DISPLAY statement enables you to specify a list of display tables to display or exclude. This statement is similar to the ODS SELECT, ODS EXCLUDE, and ODS TRACE statements. However, the DISPLAY statement can improve performance when a large number of tables could be generated (such as in BY-group processing). The DISPLAY statement produces only the requested tables, but ODS statements generate all the tables and then keep the requested tables.

If you use both DISPLAY and ODS statements together, the DISPLAY statement takes precedence over the ODS statements. Note that the ODS EXCLUDE statement processes tables after they have been filtered by the DISPLAY statement. In some cases, it might appear that the ODS EXCLUDE statement is taking precedence because it can further filter the tables. For more information about ODS, see *SAS Output Delivery System: Procedures Guide*.

You can specify the *table-list* as a list of table names, paths, partial pathnames, and regular expressions.

The table names that you can specify are listed in the section "ODS Table Names" on page 612. A path is a table name that is prefixed with dot-separated grouping information. For example, a SelectionSummary table that a procedure produces during a selection routine might have the path *Bygroup1.Summary.SelectionSummary*. A partial pathname does not include all groups; for example, *SelectionSummary* and *Summary.SelectionSummary* are partial pathnames for *Bygroup1.Summary.SelectionSummary*.

When you specify a table name or partial pathname, all display tables whose paths end in the specified name are selected for display or exclusion. For example, both *SelectionSummary* and *Summary.SelectionSummary* select *Bygroup1.Summary.SelectionSummary*.

A regular expression is enclosed in forward slashes (/). For example, specifying "/tions/" selects all pathnames that contain the substring "tions"; in particular, the *Bygroup1.Summary.SelectionSummary* table is selected. Specifying "!/tions/" selects all pathnames that do not contain the substring "tions"; in particular, the *Bygroup1.Summary.SelectionSummary* table is not selected.

You can specify the following *options* after a slash (/):

CASESENSITIVE
performs a case-sensitive comparison of table names in the *table-list* to display table names when tables are subsetted for display. To preserve case, you must enclose table names in the *table-list* in quotation marks.

EXCLUDE
displays all display tables except those that you specify in the *table-list*.

EXCLUDEALL
suppresses display of all tables. This option takes precedence over the other options.

TRACE
displays the display table names, labels, and paths.

DISPLAYOUT Statement

DISPLAYOUT *table-spec-list* < / *options* > **;**

The DISPLAYOUT statement enables you to create output tables from your displayed output. This statement is similar to the ODS OUTPUT statement. For more information about ODS, see *SAS Output Delivery System: Procedures Guide*.

The *table-spec-list* specifies a list of output tables to create. Each entry in the list has either a *key*=*value* format or a *key* format:

key=*value* specifies *key* as the ODS table name, path, or partial pathname, and specifies *value* as the output table name.

key specifies *key* as the ODS table name and also as the output table name.

The ODS table names that you can specify are listed in the section "ODS Table Names" on page 612. You cannot specify the ODS table named OutputCasTables in the *table-spec-list*.

Table names and partial pathnames are discussed under the DISPLAY statement. The DISPLAYOUT statement does not support regular expressions.

You can specify the following *options* after a slash (/):

INCLUDEALL
creates output tables for all display tables. The name of the created output table is the same as the name of the corresponding display table. If you specify this option, the *table-spec-list* specification is ignored.

NOREPLACE
does not replace any existing output table of the same name.

ID Statement

ID *variable* **;**

The ID statement lists one *variable* from the data set that specifies the order of the observations in the input data set. This variable is not considered as part of the analysis to compute the principal components. Also, the values of *variable* should be unique with a common difference.

The following examples show values of the variables that you can use as ID variables:

- 1, 3, 5, 7, . . .
- 2.2, 2.7, 3.2, 3.7, 4.2, 4.7, . . .

The following examples show values of the variables that you cannot use as ID variables:

- 1, 2, 5, 6, 7, . . . (these values do not have a common difference)
- 1, 2, 3, 3, 4, 5, 6, 7, . . . (these values are not unique)

The ID statement is required. Without this statement, windows cannot be identified.

INPUT Statement

INPUT *variables* **;**

The INPUT statement specifies the names of *variables* to be considered in the MWPCA procedure. Only numeric *variables* are accepted. If you do not specify this statement, all numeric *variables* in the input data set are considered (except the one used in the ID statement).

OUTPUT Statement

OUTPUT OUT=*libref.data-table* < *options* > **;**

The OUTPUT statement creates an output data table to contain the results of the procedure run.

You must specify the following option:

OUT=*libref.data-table*
names the output data table for PROC MWPCA to use. You must specify this option before any other options. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to where the data table is to be stored. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 598.

data-table specifies the name of the output data table.

You can also specify the following *options*:

NPC=*number*

NPCDISPLAYED=*number*

specifies the number of principal components to display for each window in the output table.

NPCUSED=*number*

specifies the number of principal components to be used to compute angle changes for each window in the output table.

PCANGLES

displays the angle change of the principal components and the absolute angle of the first principal component in the output table. The angle change and the absolute angle track the change of the principal components over sliding windows. The range of both values is between 0 and 1, because both values are normalized by $\frac{2}{\pi}$, as shown by the following definitions:

- The *angle change* of the first principal component is calculated by

$$\theta = \arccos\left(\frac{P_1(k-1)\cdot P_1(k)}{|P_1(k-1)||P_1(k)|}\right)\left(\frac{2}{\pi}\right)$$

  where $P_1(k-1)$ and $P_1(k)$ are the first principal component vectors of two consecutive windows. If the value of the NPCUSED= option is $k >= 2$, you use the following logic to compute the angle change. Consider the first k principal components that are associated with each of the windows. Consider these sets of principal components as sets of vectors. Each set of vectors defines a vector subspace. The subspace that a set of vectors spans is the set of all linear combinations of those vectors.

  For example, the subspace that is associated with the vectors $(0,1,0)^T$ and $(1,0,0)^T$ in three-dimensional $(x,y,z)^T$ space is the x, y plane. Each principal component is an $n \times 1$ vector, and k principal components define a k-dimensional subspace of an n-dimensional space.

  After the subspaces $\mathcal{V}$ and $\mathcal{U}$ are defined, you can calculate the angles between them. The first principal angle, θ_1, is calculated as follows:

$$\cos(\theta_1) = \max_{a\in\mathcal{V}}\max_{b\in\mathcal{U}}\frac{a\cdot b}{||a||\cdot||b||}$$

  Let $\mathcal{V}_1$ and $\mathcal{U}_1$ be the subspaces of $\mathcal{V}$ and $\mathcal{U}$, respectively, which are orthogonal to the values of a and b that maximize the previous equation. Then the following equation holds:

$$\cos(\theta_2) = \max_{a\in\mathcal{V}_1}\max_{b\in\mathcal{U}_1}\frac{a\cdot b}{||a||\cdot||b||}$$

  This process continues, with each new pair of subspaces being defined as orthogonal to all previously selected a and b. What results are k different angles.

  You can rewrite the optimizations by using Q_U and Q_V, matrices whose columns are an orthonormal basis for $\mathcal{U}$ and $\mathcal{V}$, respectively. Then you can write $a_i = Q_U e_i$ and $b_i = Q_V f_i$, which finds the e_i, f_i that maximizes the following expression:

$$\frac{e_i Q_U^T Q_V f_i}{||e_i||\cdot||f_i||}$$

  subject to $e_i^T e_j = 0$ and $f_i^T f_j = 0$ for all $j < i$. You can perform the optimization by using the singular value decomposition of $Q_U^T Q_V$.

It is useful to convert the collection of angles to a single number for anomaly detection. Let θ be a random variable that represents the collection of angles. If μ_θ and $\sum_\theta$ are the mean and covariance of the distribution of θ, respectively, then $\tau = (\theta - \mu_\theta)^T \sum_\theta^{-1} (\theta - \mu_\theta)$ can be calculated as a single summary number. If θ is a multivariate normal distribution, then τ is a χ^2 distribution with degrees of freedom equal to the dimension of θ, which in this case is k. In practice, the mean or covariance of θ is unknown, so they are estimated using a sliding-window sampling of a warmup data set. If no warmup data are specified, then μ_θ and $\sum_\theta$ are set to the zero vector and identity matrix, respectively.

- The *absolute angle* is calculated by

$$\theta = \arccos\left(\frac{v_1 \cdot P_1(k)}{|v_1||P_1(k)|}\right)\left(\frac{2}{\pi}\right)$$

where v_1 is the diagonal vector [1, 1, 1, . . . , 1] and $P_1(k)$ is the first principal component vector of the new observation.

SEED=*number*

RANDSEED=*number*

specifies the seed of the pseudorandom number generator to use in warmup steps. If you do not specify a seed, or if you specify a value less than or equal to 0, the seed is generated by reading the time of day from the computer's clock.

STANDARDPC

standardizes the output principal components. If the window matrix is W and the ith principal component vector is P_i, then the jth element of the standard principal component P_i^S is

$$P_{ij}^S = \frac{P_{ij} \times \sigma_i}{\sqrt{(W^T W)_{jj}}}$$

where σ_i is the ith-largest eigenvalue of matrix W.

WARMUPSIZE=*number*

specifies the number of observations to be used in warmup steps.

WINSUMMARY=*libref.data-table*

specifies the name of the windows summary table that includes information about each window, such as rank of the low-rank matrix, number of iterations, number of observations that have missing values, and solution status. As a side note, observations that have missing values are omitted from the window.

libref.data-table is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 598.

RPCA Statement

RPCA < *options* > **;**

If you specify the ROBUST option in the PROC MWPCA statement, then you can use this statement to specify the following RPCA *options*:

FIXEDMU
uses a fixed value for μ in each iteration of RPCA when METHOD=APG. Otherwise, μ is updated dynamically in each iteration.

LAMBDA=*number*
specifies a value for λ, where *number* is a positive real number. The default value is computed by $\frac{1}{\sqrt{n}}$, where n is the greater of the number of observations and the number of variables in the input data set. This parameter affects the sparsity of the sparse matrix. For more information, see Candès et al. (2011).

LAMBDAWEIGHT=*number*
specifies the value of λ_{weight}. The final value of the λ that is used in the RPCA algorithm is calculated by multiplying λ_{weight} by λ. You can use this value to control the sparsity of the sparse matrix. For more information about the sparse matrix in RPCA, see Chapter 30, "RPCA Procedure."

By default, LAMBDAWEIGHT=1.

MAXITER=*number*
specifies the maximum number of iterations before the RPCA algorithm stops, where *number* is a positive integer. By default, MAXITER=1000.

METHOD=ALM | APG
specifies the method to perform RPCA. You can specify the following values:

ALM	specifies the augmented Lagrange multiplier method.
APG	specifies the accelerated proximal gradient method.

For more information about each method, see the section "Details: RPCA Procedure" on page 795.

By default, METHOD=ALM.

MU=*number*
specifies an initial value of μ when METHOD=APG. By default, MU=10^{-3}.

TOLERANCE=*number*
specifies the convergence criterion for the RPCA algorithm on each window. By default, TOLERANCE=10^{-7}.

SVD Statement

SVD < *options* > **;**

The SVD statement calls a singular value decomposition (SVD) solver to calculate the principal components for each window. You can specify the following *options*:

INCREMENTAL
uses the incremental SVD solver. For more information, see the section "Incremental Singular Value Decomposition" on page 611.

MAXRANK=*number*
specifies the maximum value of the rank that is considered by the SVD solver. By default, *number* is the number of input variables.

METHOD=EIGEN | ITERATIVE
specifies the method that the SVD solver uses. You can specify the following methods:

EIGEN uses the eigenvalue decomposition method.

ITERATIVE uses the iterative SVD method.

By default, METHOD=EIGEN.

Details: MWPCA Procedure

Principal component analysis (PCA) is a statistical approach that converts a set of correlated variables to a set of linearly uncorrelated variables known as principal components. Because the first few principal components (which are associated with the largest eigenvalues) usually capture most of the variability in the data, they can be tracked over time to assess whether any changes have taken place in the subspace that is spanned by the data. Moving windows PCA (MWPCA) is designed to track whether any such changes occur. MWPCA is useful in monitoring systems that contain many correlated measures for which the correlations hold over time in a properly functioning system. A significant change in the elements of the most important principal components indicates that there has been a change in the data structure. MWPCA is particularly useful in situations where similar machines or machine parts are operating in a similar environment and thus should be functioning similarly as the environment changes. Wind turbines on the same wind farm illustrate a useful application of MWPCA. Although each turbine might produce different output depending on its specific location in the farm, changes in wind conditions should result in highly correlated changes in output from the turbines. In this case, only the elements of the first principal component need to be tracked. If any one of the loading factors in the first principal component of one turbine starts to behave differently than the same loading factor of the other turbines, you should probably investigate whether the associated machine performance is deviating from the others and needs attention.

MWPCA performs better than the Hotelling T^2 approach in detecting degradations when it is applied to systems that have nonstationary measures and systems that have measures that are affected by seasonality. Also, MWPCA can give you a relatively clear indication of which part of the system is getting out of control.

The optimal choice for window size and step size varies by application and frequency of sampling.

Cumulative Eigenvalue Percentage Tolerance

Sometimes when the input has noise, the rank that is reported in the WINSUMMARY= table is higher than the true rank. This occurs because the existence of noise in the input can increase the number of eigenvalues. To address this issue, you can eliminate the insignificant eigenvalues by specifying a tolerance on the cumulative percentage of eigenvalues.

Suppose that the eigenvalues that are computed for the low-rank matrix of a window are 1,000, 100, and 1. If CUMEIGPCTTOL=1, then the rank of the low-rank matrix is reported as 3. However, the third eigenvalue is much smaller than the other two eigenvalues. If CUMEIGPCTTOL=0.99, then the first two eigenvalues cumulatively contribute to more than 99.90% of the sum of the eigenvalues. In this case, the smallest eigenvalue can be ignored, and the rank of the low-rank matrix is 2.

Incremental Singular Value Decomposition

The incremental singular value decomposition (SVD) solver that the MWPCA procedure provides uses an online method of capturing changes in principal components in real time. It combines Brand's method (Brand 2006) with Stor et al.'s rank-one modifications of SVD (Stor, Slapnicar, and Barlow 2015).

The incremental SVD solver updates an existing SVD model instead of running SVD for each window. The window size is maintained by including incoming observations and discarding aging observations. Whenever such operations are performed, the original SVD model is updated accordingly. By using incremental SVD, the MWPCA procedure runs a great deal faster without compromising much accuracy.

Displayed Output

The MWPCA procedure displays various tables that are related to input and results. The following sections describe the output tables in the order of their appearance.

Model Information

The "Model Information" table displays basic information about the options that the MWPCA procedure uses. This information includes the data source, window size, step size, RPCA method, SVD method used in RPCA, and values of λ and λ_{weight} used in RPCA.

Dimensions

The "Dimensions" table displays the number of observations and variables in the input data set.

Results Summary

The "Results Summary" table displays the summary of the PROC MWPCA results, including the number of windows, solution status, and run time.

The solution status is "optimal" only if optimization ends successfully for all windows. Otherwise, the solution status is "Optimization was unsuccessful in at least one window."

ODS Table Names

Each table that the MWPCA procedure creates has a name associated with it. You must use this name to refer to the table when you use ODS statements. The name of each table and a short description of the contents are listed in Table 23.2.

Table 23.2 ODS Tables Produced by PROC MWPCA

Table Name	Description	Statement	Option
Dimensions	Dimensions of the input table	PROC MWPCA	Default
ModelInfo	Model information	PROC MWPCA	Default
Summary	Summary of the results	PROC MWPCA	Default

Examples: MWPCA Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

Example 23.1: Anomaly Detection of Parking Lot Floodlights

This example applies the MWPCA procedure to a data set that consists of energy consumption every five minutes for six floodlight circuits in a parking lot at SAS headquarters in Cary, North Carolina, over a span of about three months. Each of these light circuits has a sensor that measures the energy consumption of the lights.

The following code includes the DATA step that generates the data table mylib.ParkingLotLights.

```
        data mylib.parkinglotlights;
input time e49548 e49549 e49551 e49552 e49553 e49554;
datalines;

   ... more lines ...
```

Figure 23.4 displays the energy consumption of each of these light circuits in the first 16 days. The energy consumption throughout the day is zero because the lights are off during the day. In this figure, you can see that the orange lights (e49549) have a dip in energy consumption for a couple of hours on the sixth night, indicating that one of the lights controlled by the circuit was not functioning properly. Also, you can see that during day 15, all lights have a spike in energy consumption during the day. A storm occurred on that day, creating enough darkness to trigger the lights to come on briefly. There was no malfunction in the lights.

Figure 23.4 Energy Consumption (Kilowatts) of Floodlights, First 16 Days

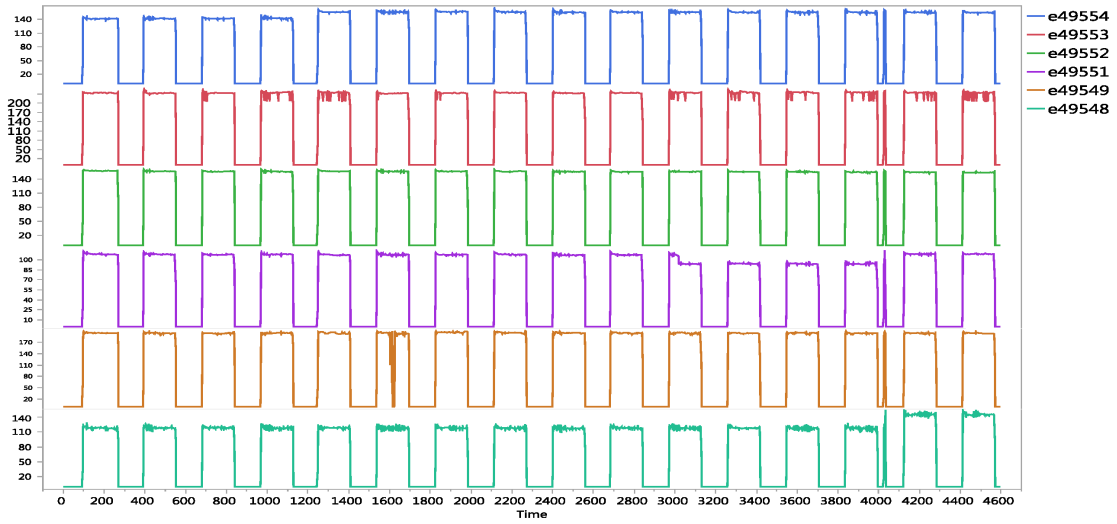
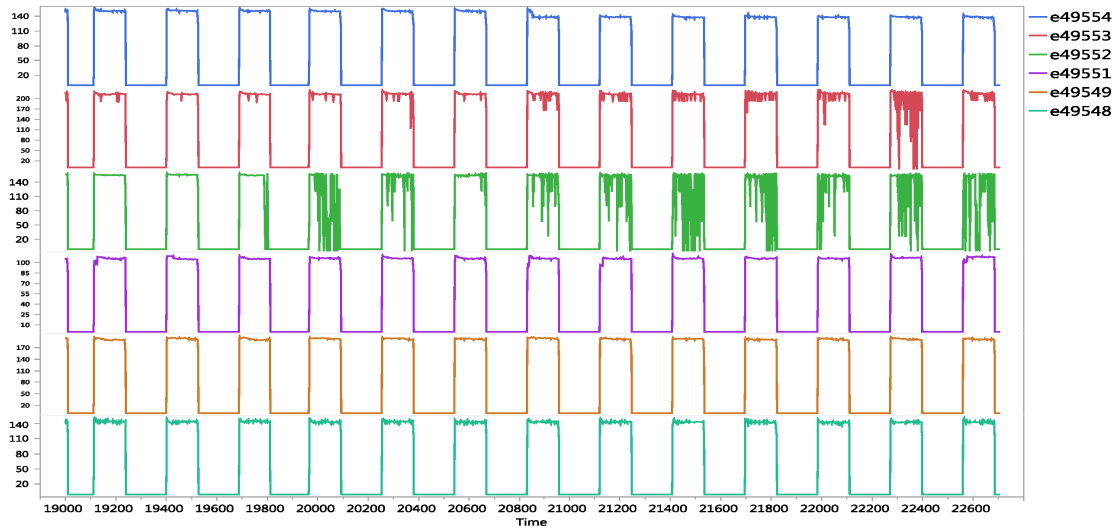


Figure 23.5 displays the energy consumption of each light circuit in the last 13 days. In this figure, you can see that the green lights (e49552) start to have many dips in energy consumption in the last 11 nights; several of the lights on this circuit are starting to fail. Also, toward the end, the red lights (e49553) have dips in energy consumption, especially on the second-last night, indicating that problems are occurring in this circuit as well.

Figure 23.5 Energy Consumption (Kilowatts) of Floodlights, Last 13 Days



The following code uses PROC MWPCA to illustrate how moving-windows principal component analysis

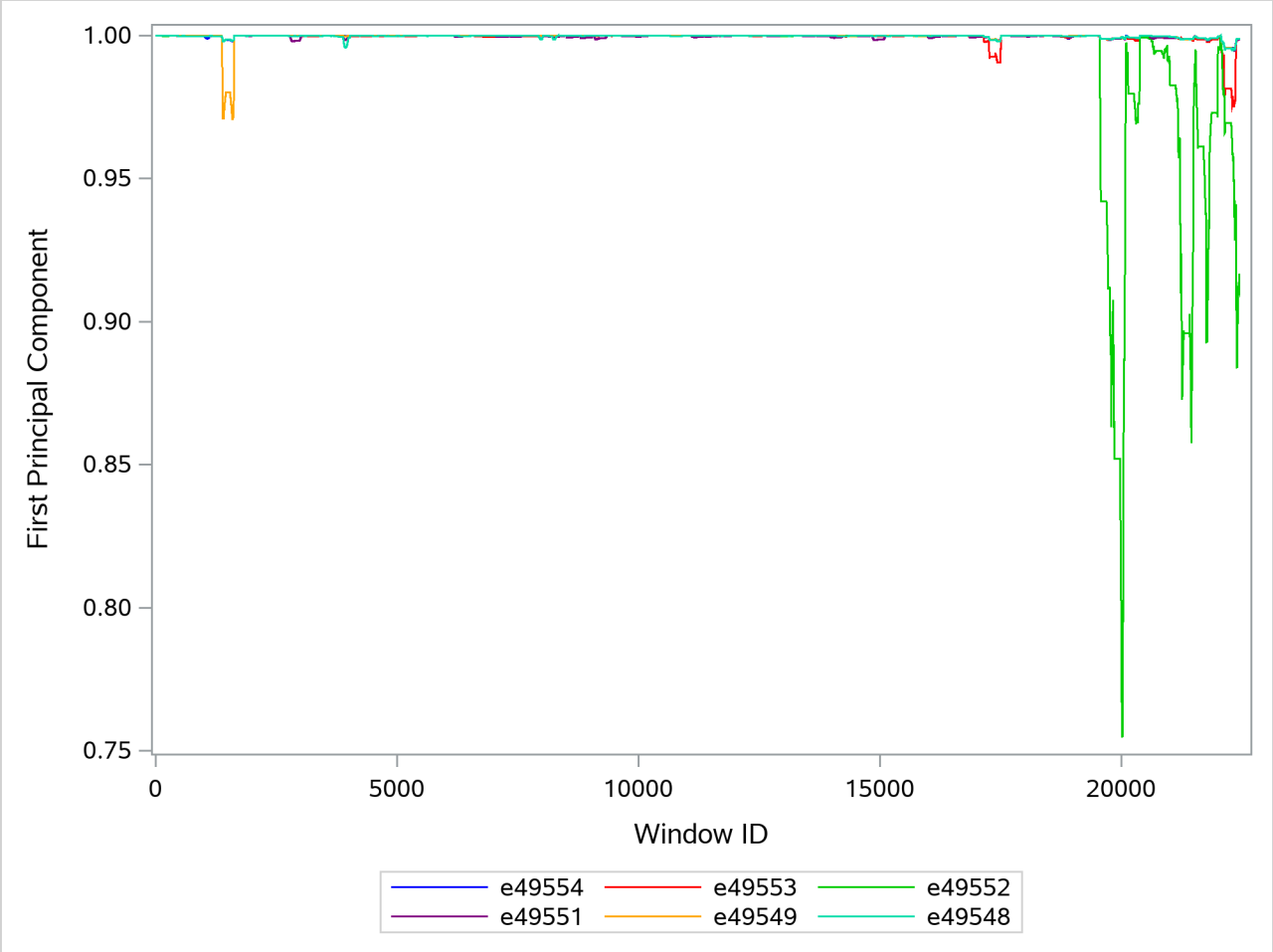
can help determine when the lights begin to fail:

```
proc MWPCA data=mylib.parkinglotlights windowsize = 240 stepsize = 1;
       id time;
       input e49548 e49549 e49551 e49552 e49553 e49554;
       output out=mylib.output standardpc pcangles;
run;
```

Output 23.1.1 displays the first principal component that is obtained from the output table that results from applying PROC MWPCA to the mylib.ParkingLotLights data table over the sliding windows of data. In this figure, you can observe that the first principal component that is associated with the orange lights starts deviating from the first principal components that are associated with the other lights around the same time that the dips were observed on the sixth night for these lights. Also, toward the end of the data, the first principal component that is associated with the green lights starts deviating from the first principal components that are associated with the other lights again around the same time that the dips for this light were observed toward the end. The same is true of the dips in the energy consumption of the red lights toward the end of the sample. Note that during the storm, when all lights came on unexpectedly (on day 15), the first principal components that are related to the lights do not deviate from each other, and thus there is no signal of failure in this instance. This result is correct: the unexpected behavior of the lights was not caused by a light malfunction. Output 23.1.1 shows how PROC MWPCA detects malfunctions in each light circuit by tracking the first principal components of their energy consumption.

```
proc sgplot data=mylib.output;
   series x=window_id y=e49554/legendlabel = "e49554"
    lineattrs=(color=blue thickness = 1) ;
   series x=window_id y=e49553/legendlabel = "e49553"
    lineattrs=(color=red thickness = 1) ;
   series x=window_id y=e49552/legendlabel = "e49552"
    lineattrs=(color=CX00CC00 thickness = 1) ;
   series x=window_id y=e49551/legendlabel = "e49551"
    lineattrs=(color=purple thickness = 1) ;
   series x=window_id y=e49549/legendlabel = "e49549"
    lineattrs=(color=orange thickness = 1) ;
   series x=window_id y=e49548/legendlabel = "e49548"
    lineattrs=(color=CX00DDAA thickness = 1) ;
   yaxis label="First Principal Component";
   xaxis label="Window ID";
run;
```

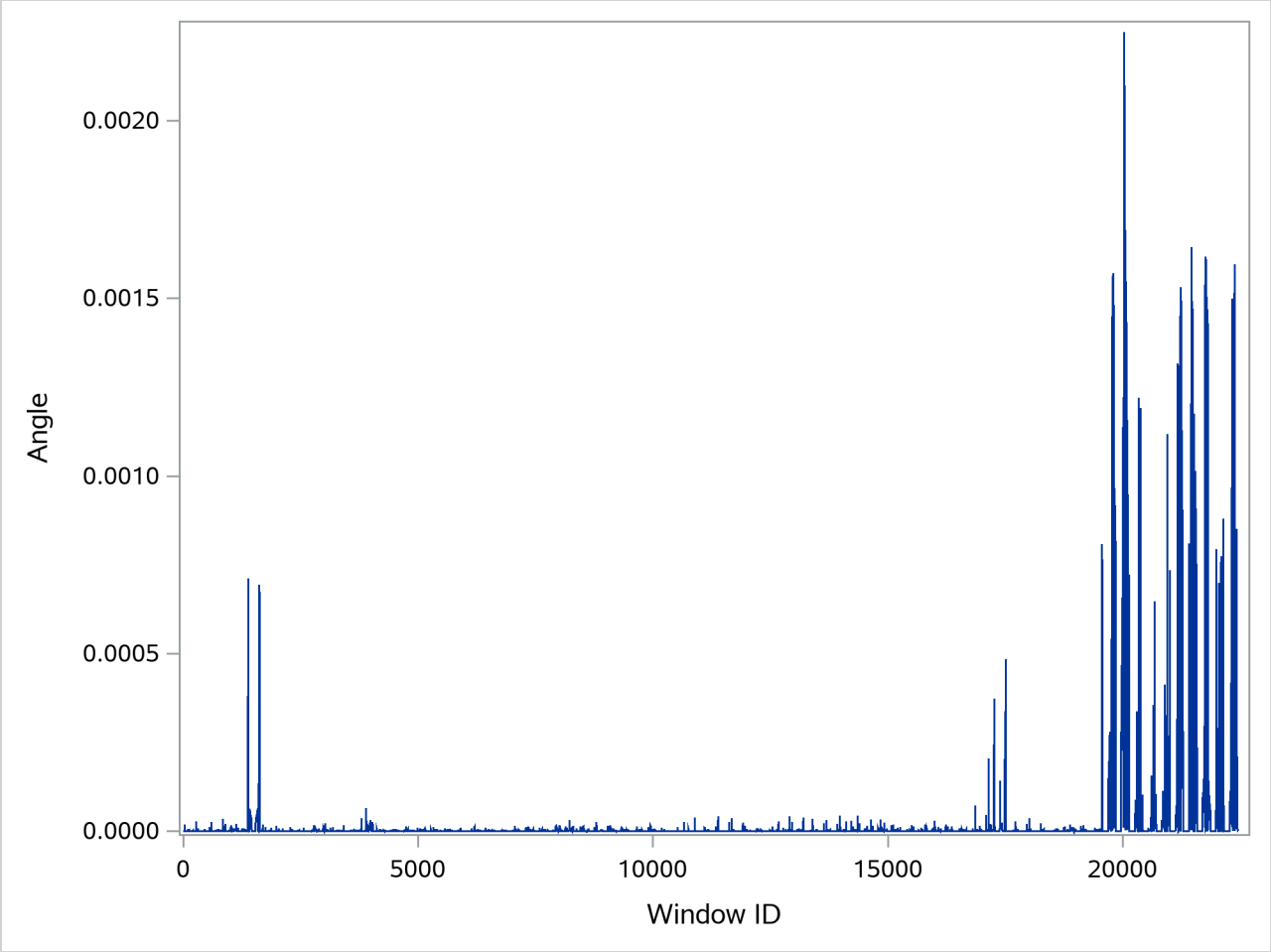
Output 23.1.1 First Principal Component over Sliding Windows



Output 23.1.2 plots the angle change of the first principal component between consecutive windows. When the observation that enters a window contains the outliers, its value of angle change is relatively higher than that of previous windows. By checking the value of the angle change in Output 23.1.2, you can easily detect when a light circuit is not functioning properly.

```
proc sgplot data =mylib.output;
   series x= window_id y= angle_chg/legendlabel = "angle change"
    lineattrs = (thickness = 1);
   yaxis label="Angle";
   xaxis label="Window ID";
run;
```

Output 23.1.2 Angle Change of the First Principal Component over Sliding Windows



Example 23.2: Comparing the Incremental SVD Solver with the Full SVD Solver

This example compares the speed and accuracy of the incremental SVD solver with that of the full SVD solver.

First, the following code is used to generate a simulation data set. The data set contains 3,000 observations and 10 variables.

```
proc iml;
   call randseed(2022);
   M = j(3000, 10);                /* allocate (3000 x 10) vector */
   call randgen(M, "Normal");      /* fill it */

   create M from M;
   append from M;
   close M;
quit;
```

Next, the simulated data set is loaded into the mylib library, and the MWPCA procedure is run using the incremental SVD solver on the simulated data set. The results are shown in Output 23.2.1.

```
data mylib.M;
   set M;
   id = _N_;
run;

proc mwpca data=mylib.M windowsize=100 stepsize=1;
   id id;
   svd incremental maxrank=3;
   output out=mylib.incremental_res nPC=1;
run;
```

Output 23.2.1 Incremental SVD Solver Results

Model Information	
Data Source	M
Window Size	100
Step Size	1
SVD Method	Eigenvalue Decomposition
Cumulative Eigenvalue Percent Tolerance	1

Number of Observations	3000
Number of Variables	10

Results Summary	
Number of Windows	2901
Solution Status	Optimal
Run Time (Seconds)	0.08

Then the MWPCA procedure is run using the full SVD solver on the simulated data set. The results are shown in Output 23.2.2.

```
proc mwpca data=mylib.M windowsize=100 stepsize=1;
   id id;
   svd MaxRank=3;
   output out=mylib.full_res nPC=1;
run;

proc mwpca data=mylib.M windowsize=100 stepsize=1;
   id id;
   svd MaxRank=3;
   output out=mylib.full_res nPC=1;
run;
```

Output 23.2.2 Full SVD Solver Results

Model Information	
Data Source	M
Window Size	100
Step Size	1
SVD Method	Eigenvalue Decomposition
Cumulative Eigenvalue Percent Tolerance	1

Number of Observations	3000
Number of Variables	10

Results Summary	
Number of Windows	2901
Solution Status	Optimal
Run Time (Seconds)	4.00

From Output 23.2.1 and Output 23.2.2, you can see that in this case, the incremental SVD solver is much faster than the full SVD solver.

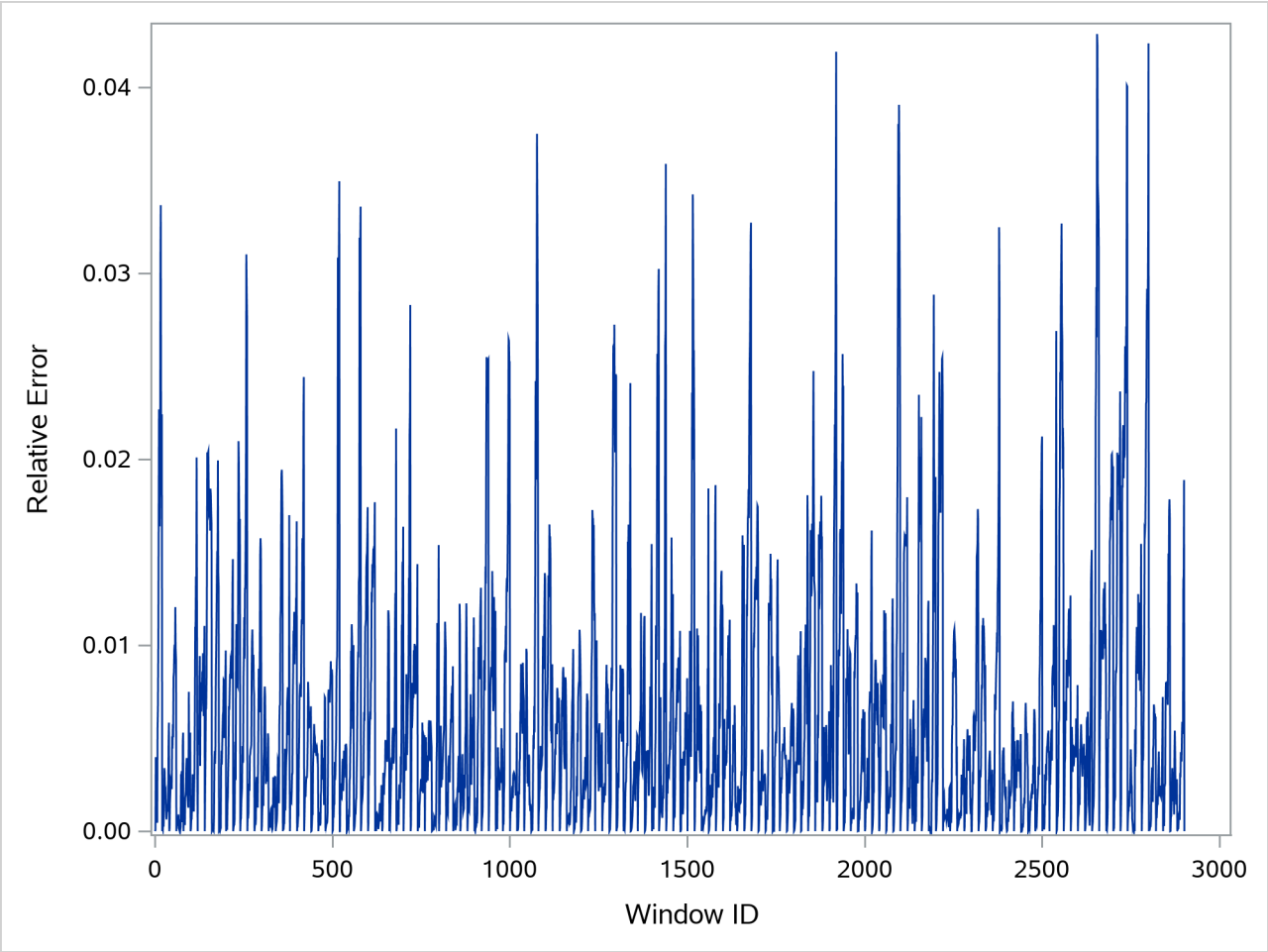
Finally, the accuracy of the incremental SVD solver is evaluated. The following code extracts the singular values from the incremental SVD and full SVD results, and then it computes the relative error of the singular values:

```
data eigvals;
   merge
      mylib.incremental_res(rename=(eigen_value=inc_eig))
      mylib.full_res(rename=(eigen_value=full_eig));
   keep window_id inc_eig full_eig;
run;

data eigvals;
   set eigvals;
   relative_err=abs(inc_eig-full_eig)/full_eig;
run;

proc sgplot data=eigvals;
   series x=window_id y=relative_err/lineattrs=(thickness=0.1);
   yaxis label='Relative Error';
run;

data eigvals;
   merge
      mylib.incremental_res(rename=(eigen_value=inc_eig))
      mylib.full_res(rename=(eigen_value=full_eig));
   keep window_id inc_eig full_eig;
run;

data eigvals;
   set eigvals;
   relative_err=abs(inc_eig-full_eig)/full_eig;
run;
```

```
proc sgplot data=eigvals;
   series x=window_id y=relative_err/lineattrs=(thickness=0.1);
   yaxis label='Relative Error';
run;
```

From Output 23.2.3, you can see that the relative error is always less than 5%. This means that by using the incremental SVD solver, the MWPCA procedure is much faster without losing much accuracy.

Output 23.2.3 Full SVD Solver Results



References

Brand, M. (2006). "Fast Low-Rank Modifications of the Thin Singular Value Decomposition." *Linear Algebra and Its Applications* 415:20–30.

Candès, E. J., Li, X., Ma, Y., and Wright, J. (2011). "Robust Principal Component Analysis?" *Journal of the Association for Computing Machinery* 58:1–37.

Stor, N. J., Slapnicar, I., and Barlow, J. L. (2015). "Accurate Eigenvalue Decomposition of Real Symmetric Arrowhead Matrices and Applications." *Linear Algebra and Its Applications* 464:62–89.

Chapter 24
NNET Procedure

Contents

Overview: NNET Procedure **622**
PROC NNET Features 622
Using SAS Viya Workbench 623
Getting Started: NNET Procedure **623**
Syntax: NNET Procedure **625**
PROC NNET Statement 625
ARCHITECTURE Statement 627
AUTOTUNE Statement 628
CODE Statement 633
CROSSVALIDATION Statement 634
FREQ Statement 634
HIDDEN Statement 634
INPUT Statement 635
MITIGATEBIAS Statement 636
OPTIMIZATION Statement 638
OUTPUT Statement 641
PARTITION Statement 642
SAVESTATE Statement 643
TARGET Statement 643
TRAIN Statement 644
WEIGHT Statement 646
Details: NNET Procedure **647**
Computational Method 647
Hyperparameter Tuning 648
k-fold Cross Validation 648
Autoencoder 649
Bias Mitigation 649
Exponentiated Gradient Reduction (EGR) Algorithm 649
Displayed Output 651
Optimization Iteration History 651
Convergence Status 651
Model Information 652
Score Information 652
Cross Validation Results 652
PredProbName 652
PredIntoName 652

PredName 652
Constraint History 652
Group Metrics History 652
Iteration History 652
Mitigation Information 653
Timing 653
Weight History 653
ODS Table Names 653
ODS Graphics 655
Examples: NNET Procedure **656**
Example 24.1: Binary Target Classification with Partition 656
Example 24.2: Finding the Best Neural Network Configuration 658
Example 24.3: Mitigating Bias of a Classification Neural Network 662
References **665**

Overview: NNET Procedure

The NNET procedure trains a multilayer perceptron neural network in SAS Viya Workbench. For more information about multilayer perceptron neural networks, see Bishop (1995). PROC NNET can also use a previously trained network to score a data table (referred to as stand-alone scoring), or it can generate SAS DATA step statements that can be used to score a data table.

Training a multilayer perceptron neural network requires the unconstrained minimization of a nonlinear objective function. Because there are currently no practical methods to guarantee finding a global minimum of that objective function, one way to be reasonably sure of finding a good solution is to train the network multiple times by using different sets of initial values for the weights.

PROC NNET Features

The NNET procedure was designed with two goals in mind: to perform efficient, high-speed training of neural networks, and to be as easy to use as possible while still creating models that fit the training data well and generalize well.

PROC NNET has the following basic features:

- multithreading during all phases of training and scoring
- intelligent defaults for most neural network parameters, such as activation and error functions
- either automatic or manual selection and use of a validation data partition to prevent overfitting during training
- automatic termination of training when the validation error stops improving
- ability to train the model while minimizing the bias metrics that you specify

Using SAS Viya Workbench

SAS Viya Workbench has greatly simplified the SAS programming experience by offering a syntax style nearly identical to that of SAS 9 and earlier releases. However, there are minor exceptions. In the past, SAS Viya procedures required the use of a CAS engine libref. This engine served multiple purposes, but it was primarily used to upload data to the CAS server and manage the data. Its additional purposes included support of new data types and improved performance for large data sets. No CAS server is needed for SAS Viya Workbench, because all computations are done in place for you in a single location.

In the following program, the LIBNAME statement creates a mylib libref that the examples in this chapter use to refer to your current or desired work directory:

```
libname mylib ".";
```

For more information, see *SAS Viya Workbench: Getting Started*.

Getting Started: NNET Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

This example shows how to use the NNET procedure to train a neural network to predict the type of iris plant. The iris data published by Fisher (1936) have been widely used for examples in discriminant and cluster analyses. The sepal length, sepal width, petal length, and petal width are measured in millimeters on 50 iris specimens from each of three species: *Iris setosa, I. versicolor, and I. virginica*. The data set is available in the Sashelp library.

You can load the sashelp.iris data into your SAS library by naming your SAS library in the first statement of the following DATA step:

```
data mylib.iris;
   set sashelp.iris;
run;
```

These statements assume that your SAS library is named mylib, but you can substitute any appropriately defined SAS library.

The following statements run PROC NNET and output the results to ODS tables:

```
proc nnet data=mylib.iris;
   input SepalLength SepalWidth PetalLength PetalWidth;
   target Species / level=nominal;
   hidden 2;
   train outmodel=mylib.nnetModel_gs seed=635117188;
   partition fraction(validate=0.3 seed=103873735);
run;
```

Figure 24.1 shows the model information for the neural network.

Figure 24.1 Model Information

The NNET Procedure

Model Information	
Model	Neural Net
Number of Observations Used	93
Number of Observations Read	93
Target/Response Variable	Species
Number of Nodes	9
Number of Input Nodes	4
Number of Output Nodes	3
Number of Hidden Nodes	2
Number of Hidden Layers	1
Number of Weight Parameters	14
Number of Bias Parameters	5
Architecture	MLP
Seed for Initial Weight	635117188
Optimization Technique	LBFGS
Number of Neural Nets	1
Objective Value	0.2194992373
Misclassification Rate for Validation	0

Figure 24.2 shows the misclassification rate for the training sample.

Figure 24.2 Score Information for Training Data

Score Information for Training	
Number of Observations Read	93
Number of Observations Used	93
Misclassification Rate	0.0215

Figure 24.3 shows the misclassification rate for the validation sample.

Figure 24.3 Score Information for Validation Data

Score Information for Validation	
Number of Observations Read	57
Number of Observations Used	57
Misclassification Rate	0

Syntax: NNET Procedure

The following statements are available in the NNET procedure:

PROC NNET < *options* > ;
- **ARCHITECTURE** *architecture-options* ;
- **AUTOTUNE** < *options* > ;
- **CODE** < *options* > ;
- **CROSSVALIDATION** < *options* > ;
- **FREQ** *variable* ;
- **HIDDEN** *number* < /*options* > ;
- **INPUT** *variables* < / **LEVEL=INTERVAL** | **NOMINAL** > ;
- **MITIGATEBIAS** *mitigate-options* ;
- **OPTIMIZATION** < *options* > ;
- **PARTITION** < *partition-options* > ;
- **SAVESTATE RSTORE=***libref.data-table* ;
- **OUTPUT OUT=***libref.data-table* < *option* > ;
- **TARGET** *variables* < /*options* > ;
- **TRAIN OUTMODEL=***libref.data-table* < *options* > ;
- **WEIGHT** *variable* ;

When you train a neural network, the PROC NNET, INPUT, TARGET, and TRAIN statements are required. At least one HIDDEN statement is required unless you use the GLIM architecture; in that case the HIDDEN statement is not allowed.

When you use a previously trained neural network to score a data table, only the PROC NNET, OUTPUT, and CODE statements are allowed.

PROC NNET Statement

PROC NNET < *options* > ;

The PROC NNET statement invokes the procedure. You can specify the following options in the PROC NNET statement:

You can specify the following *options*:

DATA=*libref.data-table*

names the input data table for PROC NNET to use. The default is the most recently created data table. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to the data. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 623.

data-table specifies the name of the input data table.

NOTE: The data set options WHERE, TEMPNAMES, and TEMPEXPRESS are not supported.

INMODEL=*libref.data-table*

specifies a model for stand-alone scoring or coding. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the input data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 623.

LISTNODE=INPUT | HIDDEN | OUTPUT

specifies the values of the layers to be saved in the scored output or code. Values are normalized values after standardization, if any. This option is valid only if you specify the OUTPUT or CODE statement.

You can specify the following values:

INPUT outputs the values of the input layer.

HIDDEN outputs the values of the hidden layer(s).

OUTPUT outputs the values of the output layer.

By default, LISTNODE=HIDDEN for autoencoder networks. For non-autoencoder networkers, no layer information is saved if the LISTNODE= option is not specified.

MAXNOMINALLEVELS=*number*

specifies the maximum number of levels for each nominal variable. PROC NNET generates an error if any nominal variable exceeds this limit. This option is ignored when no nominal variables are included in the training.

By default, as the limit, PROC NNET uses the largest integer that the computer supports.

MISSING=MAX | MEAN | MIN

specifies the statistic with which to impute missing values for interval input variables. If you specify this option, missing values for nominal input variables are treated as a valid level. PROC NNET excludes any observations that have missing values in the target variable.

You can specify the following values:

MAX substitutes missing values with the maximum value of the corresponding interval input variable.

MEAN substitutes missing values with the mean value of the corresponding interval input variable.

MIN substitutes missing values with the minimum value of the corresponding interval input variable.

By default, observations that have missing values are excluded from the analysis.

NTHREADS=*number-of-threads*

specifies the number of threads to use for the computation. The default value is the number of CPUs available. The value of *number-of-threads* can be from 1 to 64, inclusive.

PRINTTARGET

prints out the ODS tables that contain the mapping of the target variable or its events to variables PROC NNET predicts. The ODS names are PredProbName, PredIntoName, and PredName in Table 24.3.

STANDARDIZE=NONE | STD | MIDRANGE

specifies the method to use for standardizing the interval inputs.

You can specify the following values:

NONE	does not alter the variables.
STD	scales the variables such that their mean is 0 and the standard deviation is 1.
MIDRANGE	scales the variable such that their midrange is 0 and the half-range is 1. That is, the variables have a minimum of –1 and a maximum of 1. You can also use the keyword RANGE for this method.

By default, STANDARDIZE=MIDRANGE.

TARGETSTANDARDIZE=NONE | STD | MIDRANGE

TARGETSTD=NONE | STD | MIDRANGE

specifies the method to use for standardizing the interval target.

You can specify the following values:

NONE	does not alter the variables.
STD	scales the variables such that their mean is 0 and the standard deviation is 1.
MIDRANGE	scales the variable such that their midrange is 0 and the half-range is 1. That is, the variables have a minimum of –1 and a maximum of 1. You can also use the keyword RANGE for this method.

By default, TARGETSTANDARDIZE=MIDRANGE.

ARCHITECTURE Statement

ARCHITECTURE *architecture-option* **;**

The ARCHITECTURE statement specifies the architecture of the neural network to be trained.

You can specify one of the following *architecture-options* (GLIM and MLP DIRECT are not allowed when an autoencoder is used; for more information about the autoencoder, see the section "Autoencoder" on page 649):

GLIM

specifies a neural network architecture that has no hidden layers (this is equivalent to a generalized linear model). If you specify this *architecture-option*, the HIDDEN statement is not allowed.

MLP
specifies a multilayer perceptron architecture that has one or more hidden layers. This is the default architecture. This option is the default for autoencoding.

MLP DIRECT
specifies that direct connections between each input and each target neuron be included when the MLP architecture is used.

When you use PROC NNET to train a neural network, this statement is optional. This statement is not allowed when you use PROC NNET to perform stand-alone scoring.

NOTE: If you specify the AUTOTUNE statement, PROC NNET uses the architecture that is suggested by the tuning optimization. As a result, it might use a different architecture from the one that is specified in the ARCHITECTURE statement.

AUTOTUNE Statement

AUTOTUNE < *options* > **;**

The AUTOTUNE statement activates the tuning optimization algorithm, which searches for the best hidden layers and regularization parameters based on the problem and specified options. If ALGORITHM=SGD or ALGORITHM=ADAM, the algorithm also searches for the best values of the learning rate and annealing rate. When you specify the AUTOTUNE statement, PROC NNET might ignore any specified HIDDEN statement, depending on the values of the USEPARAMETERS= and TUNINGPARAMETERS= options. You cannot specify both the AUTOTUNE statement and the CROSSVALIDATION statement in the same procedure run.

Table 24.1 summarizes the *options* that you can specify in the AUTOTUNE statement. For more information about all options except the TUNINGPARAMETERS= option, see the section "AUTOTUNE Statement" on page 6 in Chapter 2, "Shared Concepts." The TUNINGPARAMETERS= option is described in detail following Table 24.1.

Table 24.1 AUTOTUNE Statement Options

Option	Description
APPENDLOOKUP	Specifies that the table specified in the HISTORYTABLE= option contain the rows from the table specified in the LOOKUPTABLE= option
CONSTRAINTS=	Specifies the constraints on the output metrics
EVALHISTORY=	Specifies how to report the evaluation history of the tuner
FOLDCOLUMN=	Specifies the column in the data table in which the cross validation fold value is indicated
FRACTION=	Specifies the fraction of observations to use for validation
HISTORYTABLE=	Specifies the table that contains the evaluation history
KFOLD=	Specifies the number of folds for *k*-fold cross validation
LIVEUPDATE	Specifies that the table specified in the HISTORYTABLE= option be updated at every evaluation
LOOKUPTABLE=	Specifies the table to use for evaluation lookup
MAXBAYES=	Specifies the maximum number of points in the kriging model

Table 24.1 *continued*

Option	Description
MAXEVALS=	Specifies the maximum number of evaluations
MAXITER=	Specifies the maximum number of iterations when SEARCHMETHOD=GA or SEARCHMETHOD=BAYESIAN
MAXTIME=	Specifies the maximum time for all iterations
MAXTRAINTIME=	Specifies the maximum time for a model training
NCONVITER=	Specifies the number of convergence iterations
NOGRIDSHUFFLE	Requests that the grid points not be shuffled
NOLOCALSEARCH	Disables local search optimization
NPARALLEL=	Specifies the number of parallel sessions
NSUBSESSIONWORKERS=	Specifies the number of workers in parallel sessions
OBJECTIVE=	Specifies the objective function
POPSIZE=	Specifies the population size when SEARCHMETHOD=GA or SEARCHMETHOD=BAYESIAN
SAMPLESIZE=	Specifies the sample size when SEARCHMETHOD=LHS or SEARCHMETHOD=RANDOM
SEARCHMETHOD=	Specifies the search method that the optimizer uses
SECONDOBJECTIVE=	Specifies the second objective to use for tuning
SELECTINITPOINT	Specifies that the tuner select the best evaluation from the lookup table
TARGETEVENT=	Specifies the target event for ROC-based calculations
TRAINFRACTION=	Specifies the fraction of observations to use for training
TUNINGPARAMETERS=	Specifies the custom tuning parameters
USEPARAMETERS=	Specifies how to handle the TUNINGPARAMETERS= option

TUNINGPARAMETERS=(*suboption* **| . . . | <** *suboption* **>)**

TUNEPARMS=(*suboption* **| . . . | <** *suboption* **>)**

specifies which parameters to tune and which ranges to tune over. If USEPARAMETERS=STANDARD, this option is ignored.

You can specify one or more of the following *suboptions*:

ANNEALINGRATE (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**

specifies the range of the annealing rates to use in the tuning process. This option is valid only when ALGORITHM=SGD or ADAM in the OPTIMIZATION statement.

You can specify the following additional *suboptions*:

LB=*number*

specifies the minimum annealing rate to use in the tuning process, where *number* is a nonnegative double. If you specify this suboption, you cannot also specify the VALUES= suboption.

By default, LB=1E–13.

UB=*number*
specifies the maximum annealing rate to use in the tuning process, where *number* is a nonnegative double. If you specify this suboption, you cannot also specify the VALUES= suboption.

By default, UB=1E–2.

VALUES=*value-list*
specifies a list of annealing rates to use in the tuning process, where *value-list* is a space-separated list of nonnegative doubles. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial annealing rate in the tuning process.

By default, INIT=1E–06.

EXCLUDE
excludes the annealing rate from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

LEARNINGRATE (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**
specifies the range of the learning rates to use in the tuning process. This option is valid only when ALGORITHM=SGD or ADAM in the OPTIMIZATION statement.

You can specify the following additional *suboptions*:

LB=*number*
specifies the minimum learning rate to use for tuning, where *number* is a nonnegative double. If you specify this suboption, you cannot also specify the VALUES= suboption.

By default, LB=1E–6.

UB=*number*
specifies the maximum learning rate to use for tuning, where *number* is a nonnegative double. If you specify this suboption, you cannot also specify the VALUES= suboption.

By default, UB=1E–1.

VALUES=*value-list*
specifies a list of learning rates to use in the tuning process, where *value-list* is a space-separated list of nonnegative doubles. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial learning rate to use for tuning.

By default, INIT=1E–03.

EXCLUDE
excludes the learning rate from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

NHIDDEN (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**
specifies the tuning range of the number of hidden layers in the network.

You can specify the following additional *suboptions*:

LB=*number*
specifies the minimum number of hidden layers, where *number* is an integer between 0 and 5. If you specify this suboption, you cannot also specify the VALUES= suboption.

By default, LB=0.

UB=*number*
specifies the maximum number of hidden layers, where *number* is an integer between 0 and 5. If you specify this suboption, you cannot also specify the VALUES= suboption.

By default, UB=2.

VALUES=*value-list*
specifies a list of numbers of hidden layers to be searched in the tuning process, where *value-list* is a space-separated list of nonnegative integers. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial number of hidden layers for the tuning process.

By default, INIT=0.

EXCLUDE
excludes the number of hidden layers from the tuning process. If you specify this suboption, any specified LB= and UB= suboptions are ignored and you cannot specify any NUNITS*i* suboption.

NUNITS*i* (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**
specifies tuning information for neurons in the *i*th hidden layer, where *i* is any integer 1–5, inclusive. By default, up to two hidden layers are tried during tuning. An NUNITS*i* suboption takes effect only when the value of the UB= suboption in the NHIDDEN suboptions is greater than or equal to *i*.

You can specify the following additional *suboptions*:

LB=*number*
specifies the minimum number of neurons in the *i*th hidden layer, where *number* must be a nonnegative integer. If you specify this suboption, you cannot also specify the VALUES= suboption.

By default, LB=1.

UB=*number*
specifies the maximum number neurons in the *i*th hidden layer, where *number* must be a nonnegative integer. If you specify this suboption, you cannot also specify the VALUES= suboption.

By default, UB=$\min(3n, 100)$, where n is the number of model inputs.

VALUES=*value-list*
specifies a list of candidate numbers of neurons in the *i*th hidden layer to be searched in the tuning process, where *value-list* is a space-separated list of nonnegative integers. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial number of neurons in the *i*th hidden layer.

By default, INIT=1.

EXCLUDE
excludes the number of neurons in the *i*th hidden layer from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored. If you specify this suboption, you cannot specify the NHIDDEN suboption or any other NUNITS*i* option.

REGL1 (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**
specifies the range of L1 regularization values in the tuning process.

You can specify the following additional *suboptions*:

LB=*number*
specifies the minimum L1 regularization value in the tuning process, where *number* is a nonnegative double. If you specify this suboption, you cannot also specify the VALUES= suboption.

By default, LB=0.

UB=*number*
specifies the maximum L1 regularization value in the tuning process, where *number* is a nonnegative double. If you specify this suboption, you cannot also specify the VALUES= suboption.

By default, UB=10.

VALUES=*value-list*
specifies a list of L1 regularization values to be searched in the tuning process, where *value-list* is a space-separated list of nonnegative doubles. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial L1 regularization value in the tuning process.

By default, INIT=0.

EXCLUDE
excludes L1 regularization from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

REGL2 (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**
specifies the range of L2 regularization values in the tuning process.

You can specify the following additional *suboptions*:

LB=*number*
specifies the minimum L2 regularization value in the tuning process, where *number* is a nonnegative double. If you specify this suboption, you cannot also specify the VALUES= suboption.

By default, LB=0.

UB=*number*
specifies the maximum L2 regularization value in the tuning process, where *number* is a nonnegative double. If you specify this suboption, you cannot also specify the VALUES= suboption.

By default, UB=10.

VALUES=*value-list*
specifies a list of L2 regularization values to be searched in the tuning process, where *value-list* is a space-separated list of nonnegative doubles. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial L2 regularization value in the tuning process.

By default, INIT=0.

EXCLUDE
excludes L2 regularization from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

CODE Statement

CODE < *options* > **;**

The CODE statement returns the SAS score code that can be used to score data similar to the input data.

You can specify the following *options*:

FILE=*filename*
specifies the name of the file where PROC NNET is to write the SAS score code.

NOCOMPPGM

omits the logic of the option FRACTION option in the PARTITION statement from the score code.

If you do not specify this option, the logic of the FRACTION option in the PARTITION statement is included in the score code.

CROSSVALIDATION Statement

CROSSVALIDATION < *options* > **;**

The CROSSVALIDATION statement performs *k*-fold cross validation to find the average estimated validation error. You cannot specify this statement if you specify the PARTITION statement.

You can specify the following *options*:

KFOLD=*number*

specifies the number of partition folds in the cross validation process, where *number* is between 2 and 20, inclusive. By default, KFOLD=5.

NOPARALLEL

specifies that *k*-fold cross validation not be run in parallel. By default, the process runs in parallel.

FREQ Statement

FREQ *variable* **;**

The *variable* in the FREQ statement identifies a numeric variable in the input data table that contains the frequency of occurrence of each observation. PROC NNET treats each observation as if it appears *f* times, where *f* is the value of the FREQ statement *variable* for the observation. If *f* is not an integer, it is truncated to an integer. If *f* is less than 1 or missing, the observation is not used in the analysis. When the FREQ statement is omitted, each observation is assigned a frequency of 1.

HIDDEN Statement

HIDDEN *number* < /*options* > **;**

The HIDDEN statement specifies the number of neurons or units in a hidden layer. You must specify a *number*, and it must be a positive integer. You can specify multiple HIDDEN statements; each HIDDEN statement represents a hidden layer. This statement is required if you specify the MITIGATEBIAS statement. It is ignored if you specify the AUTOTUNE statement.

You can also specify the following *options*:

ACT=EXP | IDENTITY | LOGISTIC | RECTIFIER | SIN | TANH
specifies the activation function for the hidden layer.

You can specify the following measures:

EXP	specifies the exponential function.
IDENTITY	specifies the identity function.
LOGISTIC	specifies the logistic function.
RECTIFIER	specifies the rectifier activation function.
SIN	specifies the sine function.
TANH	specifies the hyperbolic tangent function.

By default, ACT=TANH.

COMB=ADD | LINEAR
specifies the combination function for the hidden layer.

You can specify the following functions:

ADD	specifies the additive combination function.
LINEAR	specifies the linear combination function.

By default, COMB=LINEAR.

INPUT Statement

INPUT *variables* < / **LEVEL=INTERVAL** | **NOMINAL** > **;**

The INPUT statement identifies the *variables* in the input data table that are input to the neural network. You can specify multiple INPUT statements.

You can specify the following option:

LEVEL=INTERVAL | NOMINAL
specifies the variables in the input data table. You can specify the following methods:

INTERVAL	specifies that the *variables* are interval variables, which must be numeric.
NOMINAL	specifies that the *variables* are nominal variables, also known as classification variables, which can be numeric or character.

By default, LEVEL=INTERVAL.

MITIGATEBIAS Statement

MITIGATEBIAS SENSITIVEVAR=*variable* **PREDICTEDVARS=(***variable-names***)** **PREDICTEDEVENTS=***"event-list"* < *options* > **;**

The MITIGATEBIAS statement enables PROC NNET to iteratively train a model while minimizing the bias metric that you specify. To use this statement, you must also specify the SAVESTATE statement and a categorical response (target) variable. You cannot also specify the AUTOTUNE statement or the CROSSVALIDATION statement.

You must specify the following options:

PREDICTEDEVENTS="*event-list***"**

PEVENTS="*event-list***"**

specifies the events that correspond to each variable in the PREDICTEDVARS= option. You must specify the PREDICTEDEVENTS= and PREDICTEDVARS= options with the same number of items in the same order. Items in the *event-list* must be separated by the delimiter that you specify in the DELIMETER= option.

PREDICTEDVARS=(*variable-names***)**

PVARS=(*variable-names***)**

specifies the names of the variables that contain the posterior probability for each level in model prediction that corresponds to the response (target) variable. You must specify the PREDICTEDVARS= and PREDICTEDEVENTS= options with the same number of items in the same order. Duplicate variables are not allowed.

SENSITIVEVAR=*variable*

specifies the sensitive variable to use in order to reduce the value of the bias measurement that you specify in the BIASMETRIC= option.

You can also specify the following *options*:

BIASMETRIC=*bias-metric*

specifies the type of bias measurement. You can specify the following *bias-metrics*:

DEMOGRAPHICPARITY specifies demographic parity as the bias measurement.

EQUALIZEDODDS specifies equalized odds as the bias measurement.

EQUALOPPORTUNITY specifies equal opportunity as the bias measurement.

PREDICTIVEPARITY specifies predictive parity as the bias measurement.

By default, BIASMETRIC=DEMOGRAPHICPARITY.

BOUND=*number*

specifies the bound value for the exponentiated gradient reduction algorithm.

By default, BOUND=100.

DELIMITER="*character***"**
DLM="*character***"**

specifies the delimiter to be used to separate events that you specify in the PREDICTEDEVENTS= option. A delimiter is used when event names contain embedded spaces (such as "Fuel Oil") or special characters (such as ";" or "*"). You must use quotation marks around *character*. You can specify only one delimiter; combinations of delimiters are not supported. The valid values of the DELIMITER= option are " " (space), ";" (semicolon), "*" (asterisk), "." (period), and "," (comma).

By default, DELIMITER=" " (space).

If you specify a delimiter other than " " (space), PROC NNET treats any spaces (including empty spaces) as valid characters. You are responsible for mapping events correctly in order to obtain correct results.

LEARNINGRATE=*number*

specifies the step size to use in updating the exponentiated gradient reduction algorithm.

By default, LEARNINGRATE=0.01.

LOGLEVEL=0 | 1 | 2

specifies the level of log information to print.

0 prints only warnings and errors.

1 prints some notes.

2 prints notes for each iteration.

By default, LOGLEVEL=1.

MAXITER=*number*

specifies the maximum number of iterations to run the exponentiated gradient reduction algorithm.

By default, MAXITER=10.

SEED=*number*

specifies the seed for the pseudorandom number generator. By default, the random number stream is based on the time of day from the computer's clock. A 0 or negative seed value also results in random number streams that are based on the time of day from the computer's clock. If you want a reproducible random number sequence between runs, specify a value greater than 0.

By default, SEED=0.

TARGETEVENT="*event***"**

specifies the formatted value of the response (target) variable that represents the event of interest.

By default, *event* is the first level that you specify in the PREDICTEDEVENTS= option.

TOLERANCE=*number*

specifies the parity constraint violation tolerance. If you set *number* to 0, PROC NNET trains for the maximum number of iterations, which you specify in the MAXITER= option.

By default, TOLERANCE=0.005.

TUNEBOUND
specifies that the bound value must be tuned.

OPTIMIZATION Statement

OPTIMIZATION < *options* > **;**

The OPTIMIZATION statement specifies options for the optimization method that is used to train your model. When you are training your model, the objective function to be minimized is

$$f(w) = \frac{1}{n}\sum_{i=0}^{n} L(w; x_i, y_i) + R(w)$$

where $L(w; x_i, y_i)$ is the loss associated with observation i having data x_i and correct classification y_i, and $R(w)$ is a regularization term defined by

$$R(w) = \lambda_1 \|w\|_1 + \frac{\lambda_2}{2}\|w\|_2^2$$

You can specify the following *options*:

ALGORITHM=ADAM < *sgd-options* > | **LBFGS** < *lbfgs-options* > | **SGD** < *sgd-options* > | **HF**

ALG=ADAM < *sgd-options* > | **LBFGS** < *lbfgs-options* > | **SGD** < *sgd-options* > | **HF**
specifies the optimization algorithm to use during training. You can specify the following optimization algorithms:

ADAM < *sgd-options* > specifies the adaptive moments (ADAM) algorithm, which is one of the variations of the stochastic gradient descent algorithm. It keeps track of the decaying averages of the past gradients and past squared gradients.

You can specify these additional *sgd-options*:

ANNEALINGRATE=*number*
specifies the annealing parameter, β. Annealing is a way to automatically reduce the learning rate as the algorithm progresses, causing smaller steps as the algorithm approaches a solution. Effectively, it replaces the learning rate parameter, η, with

$$\eta' = \frac{\eta}{1 + \beta t}$$

where t is the number of iterations that the algorithm has performed.

By default, ANNEALINGRATE=1.0E–6. The *number* must be a nonnegative double.

COMMFREQ=*number*
specifies the number of minibatches that each computational thread processes before weights are synchronized across all threads and nodes.

LEARNINGRATE=*number*
specifies the learning rate parameter, η, for the algorithm. New iterates for the algorithm are found by using

$$w_{k+1} = w_k - \frac{\eta}{\|I_k\|} \sum_{(x_i, y_i) \in I_k} \nabla L(w_k; x_i, y_i)$$

where w_k is the current weight vector, w_{k+1} is the new weight vector, I_k is the minibatch used during iteration k, and $L(w_k; x_i, y_i)$ is the loss associated with the ith observation.

If you see a huge objective value from the algorithm, especially for a small data set, it is likely that the learning rate is set too high.

By default, LEARNINGRATE=0.001. The *number* must be a nonnegative double.

MINIBATCHSIZE=*number*
specifies the size of the minibatches to use in the algorithm.

By default, MINIBATCHSIZE=10.

MOMENTUM=*number*
specifies the value for momentum. The *number* must be greater than or equal to 0 and less than 1. By default, MOMENTUM=0.

SEED=*number*
specifies the seed for random access of observations on each thread for the algorithm. If *number* is less than or equal to 0 or not specified, a random seed is generated by reading the time of day from the computer's clock.

USELOCKING
specifies that computational threads share a common weight vector and update weight vector without race conditions. If you do not specify this option, computational threads update a single weight vector simultaneously. This causes intentional race conditions and nondeterministic behavior but increases performance significantly.

HF (*Experimental*) specifies the Hessian-free algorithm, a nonlinear optimization algorithm that uses the Hessian vector product to build the second-order information. This algorithm is based on a modified conjugate gradient method. Because it does not use the Hessian directly, it can work for large-scale problems.

LBFGS < *lbfgs-options* > specifies the limited-memory Broyden-Fletcher-Goldfarb-Shanno algorithm.

You can specify the following *lbfgs-option*:

NUMCORRECTIONS=*number*
specifies a memory size limit for the LBFGS algorithm, where *number* is an integer between 1 and 200, inclusive.

By default, NUMCORRECTIONS=6.

SGD < *sgd-options* > specifies the plain stochastic gradient descent (SGD) algorithm.

You can specify the following *sgd-options*:

ANNEALINGRATE=*number*

specifies the annealing parameter, β. Annealing is a way to automatically reduce the learning rate as the algorithm progresses, causing smaller steps as the algorithm approaches a solution. Effectively, it replaces the learning rate parameter, η, with

$$\eta' = \frac{\eta}{1 + \beta t}$$

where t is the number of iterations that the algorithm has performed.

By default, ANNEALINGRATE=1.0E–6. The *number* must be a nonnegative double.

COMMFREQ=*number*

specifies the number of minibatches that each computational thread processes before weights are synchronized across all threads and nodes.

LEARNINGRATE=*number*

specifies the learning rate parameter, η, for the algorithm. New iterates for the algorithm are found by using

$$w_{k+1} = w_k - \frac{\eta}{\|I_k\|} \sum_{(x_i, y_i) \in I_k} \nabla L(w_k; x_i, y_i)$$

where w_k is the current weight vector, w_{k+1} is the new weight vector, I_k is the minibatch used during iteration k, and $L(w_k; x_i, y_i)$ is the loss associated with the ith observation.

If you see a huge objective value from the algorithm, especially for a small data set, it is likely that the learning rate is set too high.

By default, LEARNINGRATE=0.001. The *number* must be a nonnegative double.

MINIBATCHSIZE=*number*

specifies the size of the minibatches to use in the algorithm.

By default, MINIBATCHSIZE=10.

MOMENTUM=*number*

specifies the value for momentum. The *number* must be greater than or equal to 0 and less than 1. By default, MOMENTUM=0.

SEED=*number*

specifies the seed for random access of observations on each thread for the algorithm. If *number* is less than or equal to 0 or not specified, a random seed is generated by reading the time of day from the computer's clock.

USELOCKING

specifies that computational threads share a common weight vector and update weight vector without race conditions. If you do not specify this option, computational threads update a single weight vector simultaneously. This causes intentional race conditions and nondeterministic behavior but increases performance significantly.

By default, ALGORITHM=LBFGS.

MAXITER=*number*
specifies the iteration budget for training. When ALGORITHM=LBFGS, the algorithm stops after MAXITER= iterations if convergence has not been achieved. When ALGORITHM=SGD or ADAM, *number* specifies the desired number of training epochs.

By default, MAXITER=250.

MAXTIME=*number*
specifies the maximum time (in seconds) allowed for optimization, where *number* must be greater than or equal to 1. When this value is reached, the optimization terminates the search and returns results. When MAXTIME=0, no maximum time is set.

By default, MAXTIME=0.

REGL1=*number*
specifies the L1 regularization parameter λ_1 for the model loss function. The *number* must be nonnegative. Note that this value is autotuned when you specify the AUTOTUNE statement.

By default, REGL1=0.

REGL2=*number*
specifies the L2 regularization parameter λ_2. The *number* must be nonnegative. Note that this value is autotuned when you specify the AUTOTUNE statement.

By default, REGL2=0.

OUTPUT Statement

OUTPUT OUT=*libref.data-table* < *option* > **;**

SCORE OUT=*libref.data-table* < *option* > **;**

The OUTPUT statement creates a new data table that contains the prediction results from using the input data and the model.

You must specify the following option:

OUT=*libref.data-table*
names the output data table for PROC NNET to use. You must specify this option before any other options. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to where the data table is to be stored. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 623.

data-table specifies the name of the output data table.

You can also specify the following *option*:

COPYVAR=*variable*

COPYVARS=(*variables***)**

lists one or more *variables* from the input data table to be transferred to the output data table.

ROLE=*variable*

specifies the variable name of the partition indicator when the FRACTION option in the PARTITION statement is specified. The default name is _Fraction_PartInd_. Table 24.2 shows how this *variable* is interpreted for each observation.

Table 24.2 Role Interpretation

Value	Observation Role
1	Training
2	Validation
3	Testing

PARTITION Statement

PARTITION < *partition-options* > **;**

The PARTITION statement specifies how observations in the input data table are logically partitioned into disjoint subsets for model training, validation, and testing. Either you can designate a variable in the input data table and a set of formatted values of that variable to determine the role of each observation, or you can specify proportions to use for random assignment of observations for each role. Alternatively, you can use a separate validation data table in the TRAIN statement to do validation.

You can specify the following mutually exclusive *partition-options*:

ROLEVAR=*variable***(TRAIN=***value* **VALIDATE=***value* **< TEST=***value* **>)**

names the *variable* in the input data table whose values are used to assign roles to each observation. The formatted values of this variable, which are used to assign observations roles, are specified in the TEST=, TRAIN=, and VALIDATION= suboptions. The VALIDATE= suboptions is required; the TRAIN= and TEST=suboptions are optional. If you do not specify the TRAIN= suboption, the training subset that PROC NNET uses is the complement set of the VALIDATE= suboption, or the complement set of the VALIDATE= and TEST= suboptions if they are both specified.

FRACTION(VALIDATE=*fraction* **TEST=***fraction* **< SEED=***random-seed* **>)**

randomly assigns the specified proportions of the observations in the input data table to training and validation roles. You specify the proportions for testing and validation by using the TEST= and VALIDATE= suboptions. The VALIDATE= suboption is required, and the TEST= suboption is optional. If you specify both the TEST= and VALIDATE= suboptions, then the sum of the specified fractions must be less than 1 and the remaining fraction of the observations are assigned to the training role. Otherwise, the PARTITION statement is ignored. The range of the VALIDATE= and TEST= suboptions is from 1E–5 to 1 – (1E–5), inclusive. You can specify the ROLE option to name the partition indicator.

NOTE: The split between training, validation, and test observations can only approximate the requested fraction, because the fraction is used as a cutoff value for a random number generator to determine the

actual split. If you require a more accurate split, you must use the ROLEVAR= option to specify the split explicitly.

You cannot use the PARTITION statement along with the CROSSVALIDATION statement.

SAVESTATE Statement

SAVESTATE RSTORE=*libref.data-table* **;**

The SAVESTATE statement creates an analytic store for the model and saves it as a binary object in a data table. You can use the analytic store in the ASTORE procedure to score new data. For more information, see Chapter 4, "ASTORE Procedure."

You must specify the following option:

RSTORE=*libref.data-table*
specifies a data table in which to save the analytic store for the model. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 623.

TARGET Statement

TARGET *variable* < */options* > **;**

The TARGET statement specifies the target variable for the neural network. If you do not specify a TARGET statement, PROC NNET trains an autoencoder. For more information, see the section "Autoencoder" on page 649.

You can specify the following *options*:

ACT=EXP | IDENTITY | SIN | SOFTMAX | TANH
specifies the activation function for the target.

You can specify the following values:

EXP	specifies the exponential function. You can use ACT=EXP only with ERROR=GAMMA or ERROR=POISSON.
IDENTITY	specifies the identity function.
SIN	specifies the sine function.
SOFTMAX	specifies the softmax function.
TANH	specifies the hyperbolic tangent function.

For the GLIM architecture, you can only specify ACT=IDENTITY for an interval target and ACT=SOFTMAX for a nominal target, which are the same by default. For the MLP or MLP DIRECT architecture, the SOFTMAX method is used only with a nominal target, whereas the other methods are used only with an interval target. By default, ACT=IDENTITY for the interval target and ACT=SOFTMAX for the nominal target.

COMB=ADD | LINEAR

specifies the combination function for the target layer. You can specify the following combination functions:

ADD specifies the additive combination function.

LINEAR specifies the linear combination function.

By default, COMB=LINEAR.

ERROR=ENTROPY | GAMMA | NORMAL | POISSON

specifies the error function. The entropy error function is used only when LEVEL=NOM. You can specify the following error functions:

ENTROPY specifies the cross-entropy function.

GAMMA specifies the gamma error function. This function is usually used when you want to predict the time between events. Only ACT=EXP is valid when ERROR=GAMMA.

NORMAL specifies the normal error function, which is the sum of the squared differences between the network output and the target value.

POISSON specifies the Poisson error function. This function is usually used when you want to predict the number of events per unit time. Only ACT=EXP is valid when ERROR=POISSON.

By default, ERROR=NORMAL when LEVEL=INT, and ERROR=ENTROPY when LEVEL=NOM.

LEVEL=INTERVAL | NOMINAL

specifies the *variable* type. You can specify the following values:

INTERVAL specifies that the *variable* is interval, which must be numeric.

NOMINAL specifies that the *variable* is nominal, also known as a classification variable, which can be numeric or character.

By default, LEVEL=INTERVAL.

TRAIN Statement

TRAIN OUTMODEL=*libref.data-table* < *options* > **;**

The TRAIN statement causes the NNET procedure to use the training data that are specified in the PROC NNET statement to train a neural network model whose structure is specified in the ARCHITECTURE, INPUT, TARGET, and HIDDEN statements. The goal of training is to determine a set of network weights that best predicts the targets in the training data while still doing a good job of predicting targets of unseen data (that is, generalizing well and not overfitting).

Training starts with a pseudorandomly generated set of initial weights. PROC NNET then computes the objective function for the training partition, and the optimization algorithm adjusts the weights. This process is repeated until any one of the following conditions is met:

- The objective function that is computed using the training partition stops improving.
- The objective function that is computed using the validation partition stops improving.
- The process has been repeated the *number* of times specified in the MAXITER= and MAXTIME= options in the OPTIMIZATION statement.

When you are training, you must include exactly one TRAIN statement. The TRAIN statement is not allowed when you are doing stand-alone scoring.

You must specify the following option:

OUTMODEL=*libref.data-table*
specifies the final model from training. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 623.

You can use the model data table later to score a different input data table as long as the variable names and types of the variables in the new input data table match those in the training data table.

You can also specify the following *options*:

DROPOUTHIDDEN=*ratio*
specifies the dropout *ratio* of hidden layers. This option is valid only when you specify ALGORITHM=SGD or ADAM in the OPTIMIZATION statement and when all the connections use the linear combination function. The *ratio* must be between 0 and 1, inclusive.

By default, DROPOUTHIDDEN=0.

DROPOUTINPUT=*ratio*
specifies the dropout *ratio* of input layers. This option is valid only when you specify ALGORITHM=SGD or ADAM in the OPTIMIZATION statement and when all the connections use the linear combination function. The *ratio* must be between 0 and 1, inclusive.

By default, DROPOUTINPUT=0.

NUMTRIES=*number*
specifies the *number* of times the network is to be trained using a different starting point. Specifying this option helps ensure that the optimizer finds the table of weights that truly minimizes the objective function and does not return a local minimum. The value of *number* must be an integer between 1 and 20,000, inclusive. By default, NUMTRIES=1.

NOTE: When NUMTRIES > 1, the ODS tables "OptIterHistory" and "ConvergenceStatus" are suppressed.

RESUME
trains with the initial weight that is specified in the INMODEL= option in the PROC NNET statement. If you specify the RESUME option, you must also specify the INMODEL= option.

STAGNATION=*number*

specifies the *number* of iterations that result in no improvement for the validation subset before early stopping takes effect during training. This option is valid only when the VALIDATION= option or the PARTITION statement is specified.

By default, STAGNATION=3.

VALIDATION=*libref.data-table*

specifies a separate data table for validation during training. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the input data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 623.

If you specify both the VALIDATION= option and the PARTITION statement, the PARTITION statement is ignored. The VALIDATION= data table must have the same variables that you specify in the DATA= option in the PROC NNET statement.

VALIDGOAL=*number*

specifies the *number* targeted goal of validation error before early stopping takes effect during training. This option is valid only when the VALIDATION= option or the PARTITION statement is specified.

By default, VALIDGOAL=0.

WSEED=*random-seed*

SEED=*random-seed*

specifies the seed for generating initial random weights. If you do not specify a seed or you specify a value less than or equal to 0, the seed is generated by reading the time of day from the computer's clock. This option enables you to reproduce the same sample output.

WEIGHT Statement

WEIGHT *variable* **;**

If you specify a WEIGHT statement, *variable* identifies a numeric *variable* in the input data table that contains the weight to be placed on the prediction error (the difference between the output of the network and the target value specified in the input data table) for each observation during training.

If the *variable* is less than or equal to 0 or is missing, the observation is not used for training. When you perform scoring, PROC NNET scores the observation even if the weight is less than or equal to 0 or missing.

The WEIGHT statement is optional. If a WEIGHT statement is not included, all observations are assigned a weight of 1.

Details: NNET Procedure

Computational Method

PROC NNET trains a multilayer perceptron neural network that contains one or more hidden layers. For more information about multilayer perceptron neural networks, see Bishop (1995).

The NNET procedure does not have many parameters that you must specify for training. You must specify where the training data are (in the DATA= option in the PROC NNET statement), the names and types of the input variables (in the INPUT statement), the names and types of the target variables (in the TARGET statement), the number of hidden layers (in the HIDDEN statement), and the number of neurons in each hidden layer (in the HIDDEN statement).

Optionally, you can also specify where to write the score file that contains targets from the input file and predicted target variables from the trained network and where to write the model file that contains the parameters of the trained network (in the OUTPUT statement). In addition, you can specify where to write the SAS DATA step statements that you can use to score new data tables (in the CODE statement).

The optimization algorithms available in PROC NNET are the limited-memory Broyden-Fletcher-Goldfarb-Shanno algorithm (LBFGS), the stochastic gradient descent algorithm (SGD), the adaptive moments algorithm (ADAM), and the Hessian-free algorithm (HF), which are popular methods of solving large-scale nonlinear optimization problems. The algorithms terminate based on criteria such as convergence tolerance and maximum iterations. In addition, PROC NNET stops if the validation error (which is calculated after each line search) lacks improvement a certain number of times in a row.

The most important parameters that you can specify are the number of hidden layers in the network and the number of neurons in each hidden layer. A good strategy is to start with a single hidden layer by specifying a single HIDDEN statement with a small number of neurons, and slowly increase the number until the validation error stops improving.

The next most important parameter that you can specify is the number of times the network is to be retrained using different sets of initial weights (in the NUMTRIES= option in the TRAIN statement). Finally, unless your training data table is very large, you should set the MAXITER= option in the OPTIMIZATION statement to a large number, say 1,000 or more to prevent the optimization algorithm from stopping prematurely. The value of the MAXITER= option is only a limit; for example, specifying MAXITER=1000 does not mean that the algorithm runs for 1,000 iterations. Most training runs use far fewer iterations. If you have a large data table, you can start with MAXITER=1 to see how long a single iteration takes, and then increase the MAXITER= value.

Hyperparameter Tuning

For more information about hyperparameter tuning, see the section "Hyperparameter Tuning" on page 24 in Chapter 2, "Shared Concepts."

You can tune the following hyperparameter values when you specify the AUTOTUNE statement:

- number of hidden layers
- number of hidden units in each hidden layer
- REGL1= option for the L1 regularization parameter
- REGL2= option for the L2 regularization parameter
- ANNEALINGRATE= option for the annealing rate parameter, which is used by the training algorithm of the SGD or ADAM optimizer when you specify ALGORITHM=SGD or ADAM in the OPTIMIZATION statement
- LEARNINGRATE= option for the learning rate parameter, which is used by the training algorithm of the SGD or ADAM optimizer when you specify ALGORITHM=SGD or ADAM in the OPTIMIZATION statement

k-fold Cross Validation

The CROSSVALIDATION statement performs *k*-fold cross validation to assess the accuracy of a model. During cross validation, all data are divided into *k* subsets (folds), where *k* is the value of the KFOLD= option. For each fold, a new model is trained on the (k–1) folds and then validated using the selected (holdout) fold. The assessment metrics are then averaged across all the holdout folds. The CROSSVALIDATION statement creates a table that has k+1 rows. The first *k* rows contain the assessment metrics for each holdout fold, and the last row contains the average across all the holdout folds. Which metrics the table includes depends on whether the target is nominal or interval. For nominal targets, the table includes the values of square error, mean consequential error, and multiclass log loss. For interval targets, the table includes the values of square error, absolute error, and square logarithmic error.

Autoencoder

An autoencoder is a neural network that is used for efficient codings and widely used for feature extraction and nonlinear principal component analysis. Architecturally, an autoencoder is similar to a multilayer perceptron neural network because it has an input layer, hidden layers, and an output layer. However, it differs in that the output layer is duplicated from the input layer. Therefore, autoencoders are unsupervised learning models. You can train autoencoder in PROC NNET by omitting the TARGET and ARCHITECTURE statements. The score output of the autoencoder contains the learned features of hidden layers.

Bias Mitigation

With the widespread use of AI and machine learning models in decision-making processes, there is concern about the social impacts of these models. Bias can be embedded in both the training data and the trained model, either of which can lead to unfair or biased predictions for certain groups or populations. To reduce bias in the data and models, researchers have developed three main categories of bias mitigation methods:

- Preprocessing methods, which transform the data prior to model training. These methods try to eliminate the correlations between the input features and the sensitive feature while retaining as much information from the input data as possible.
- In-process methods, which consider fairness constraints during the model training process. These methods actively adjust the model parameters during training to create a model that produces fair predictions and classifications.
- Postprocessing methods, which do not directly alter the input data or the model training process. These methods adjust the predictive outputs from the learned models to compensate for the bias in the model.

You can reduce bias in the trained model by specifying the MITIGATEBIAS statement. This statement uses the exponentiated gradient reduction (EGR) algorithm, which is an in-process method of mitigating bias.

Exponentiated Gradient Reduction (EGR) Algorithm

The exponentiated gradient reduction (EGR) algorithm was introduced at ICML 2018 (Agarwal et al. 2018). As an in-process algorithm, it considers fairness constraints during the model training process. The EGR algorithm has three main advantages. First, it is model-agnostic, which means that you can apply it to a wide variety of machine learning models. The EGR algorithm works like a wrapper that is built around existing machine learning models. The wrapper iteratively reweights and possibly relabels the data, and then it trains a new classifier model on the basis of reweighted data in each iteration. By doing so, it reduces the complex optimization problem of training a machine learning model with fairness constraints to a series of standard model training problems without the fairness constraints. Second, the EGR algorithm supports various fairness measurements, such as demographic parity, equal opportunity, and equalized odds. Finally, the algorithm has strong theoretical support in terms of its optimization trade-off between fairness and accuracy. In theory, the EGR algorithm returns solutions that satisfy all the fairness constraints and also achieve the optimal error rate of the best fair solution.

Training a fair machine learning model is in general a nonlinear programming problem. In nonlinear optimization tasks, it is not uncommon to get suboptimal solutions because the constraints are infeasible

or because the algorithm is stuck in a local minimum. As with many other machine learning algorithms, the effectiveness of the EGR algorithm depends on many factors, such as the quality of the input data, the selection of fairness constraints, the quality of the machine learning model that it wraps around, and the values of the hyperparameters.

PROC NNET supports bias mitigation for multilevel nominal target variables and multilevel nominal sensitive variables. Note that when you include a large number of levels in the sensitive variable, the EGR algorithm searches more complex optimization scenarios; this can result in suboptimal solutions. Therefore, the solutions from this algorithm might not always be desirable when the number of levels in the sensitive variable is high. You should review and validate your sensitive variable, training models, and fairness constraints before applying the EGR algorithm.

Fairness Constraints

PROC NNET supports the following fairness measurements: demographic parity, predictive parity, equal opportunity, and equalized odds. The goal of the EGR algorithm is to bring the fairness metrics of groups as close together as possible, as well as to convert fair machine learning to a sequence of standard cost-sensitive machine learning problems whose solutions yield a randomized classifier with the lowest error rate, subject to the desired constraints. As a result, the EGR algorithm iteratively trains the machine learning models by using different weights to reduce the fairness disparities without significantly increasing the loss of the final machine learning model. For example, when there are two protected groups in the sensitive variable, the fairness constraints that the EGR algorithm requires for demographic parity are as follows:

$$\mu_1^t - \mu_0^t \leq c$$
$$\mu_1^t - \mu_0^t \geq -c$$
$$\mu_2^t - \mu_0^t \leq c$$
$$\mu_2^t - \mu_0^t \geq -c$$

where μ_1^t is the demographic parity for the first protected group during iteration t; μ_2^t is the demographic parity for the second protected group during iteration t; μ_0^t is the demographic parity for the entire population during iteration t; and c is the predefined violation tolerance threshold.

Exponentiated Gradient Reduction Algorithm Workflow

Two key hyperparameters in the EGR algorithm are specified by the BOUND= option (denoted by B) and the LEARNINGRATE= option (denoted by lr); both are available in the MITIGATEBIAS statement. To derive the fair machine learning algorithm, the fairness constraints (see the section "Fairness Constraints" on page 650) are added to the loss function of the cost-sensitive machine learning model by introducing the Lagrange multipliers $\lambda_{j,t}$, where j is the index in the constraint set. The multipliers $\lambda_{j,t}$ are updated on the basis of the degree of constraint violations $\gamma_{j,t}$, B, and lr. Then $\lambda_{j,t}$ is used to derive the new weights for each observation in the input data set for the next round of training. Note that the values of B and lr play important roles in the performance of the EGR algorithm, based on experiments. The next section discusses how to come up with appropriate values for B and lr.

Bound Value Recommendation

Because the EGR algorithm is designed to strike a balance between model accuracy and fairness constraint violations, it does not perform well unless you specify appropriate values for the BOUND= and LEARNINGRATE= options in the MITIGATEBIAS statement. The BOUND= option is important for the sample reweighting and relabeling process. It controls how heavily the EGR algorithm penalizes fairness constraint violations, and therefore it determines the trade-off between the fairness constraints and the model accuracy.

When the bound is too small, there is not enough momentum to drive the model close enough to the saddle points in the solution sets. In this case, the bias metric values of the different subgroups do not approach each other.

The LEARNINGRATE= option controls how quickly the algorithm reduces fairness constraint violations. If the learning rate is too small, the algorithm converges very slowly, resulting in a long training process; if the learning rate is too large, the algorithm becomes unstable and might not converge. In general, the learning rate and the bound value are coupled, and changing one of these values changes the optimal value of the other.

The MITIGATEBIAS statement offers a built-in bound value recommendation function to automatically search for a good candidate bound value when you request it. To eliminate the coupling effect of the bound and the learning rate, PROC NNET fixes the initial learning rate at 0.01 during the tuning process. The procedure runs the bound value recommendation function if you specify the TUNEBOUND option. This function finds the best bound value through multiple iterations by analyzing the convergence patterns of the fairness metrics. PROC NNET first runs the EGR algorithm with some initial bound value for a few iterations, while monitoring the convergence pattern for signs of stagnation or overshooting to determine whether the bound value is too small or too large. It then adjusts the bound value until it finds an appropriate value or reaches the maximum number of iterations that are allowed. After the recommendation process, PROC NNET applies the recommended bound value to the EGR algorithm to train a final model. Note that because of the nonlinear nature of the optimization process, the bound value recommendation is also subject to the problem of local minima. This recommendation should be used as a general guideline for setting the bound value; you might find a better solution by further fine-tuning.

Displayed Output

PROC NNET displays basic fit statistics in the SAS log and more detailed information in several ODS tables.

When you are scoring, PROC NNET displays the mean square error for an interval target and the misclassification rate for a nominal target. The scoring summary is based on the entire input data table if there is no validation subset. If a validation or testing subset is used, the scoring is performed for each partition. In addition, PROC NNET generates ODS tables that display detailed information about the model structure, input data, iteration history, and status for optimization solver.

The following sections describe the output that PROC NNET produces.

Optimization Iteration History

The “Optimization Iteration History” table contains the iteration history from the optimization solver. This table is suppressed if the value of the NUMTRIES= option in the TRAIN statement is greater than 1.

Convergence Status

The “Convergence Status” table contains the returning status of optimization solver. This table is suppressed if the value of the NUMTRIES= option in the TRAIN statement is greater than 1.

Model Information

The “Model Information” table contains information about the neural network model.

Score Information

The “Score Information” table contains the misclassification rate or mean square error for training, validation, or testing partition sets.

Cross Validation Results

The “Cross Validation Results” table contains the average error rate (misclassification error or average square error) of *k*-fold cross validation.

PredProbName

The “PredProbName” table contains the name mapping between the score variables and the corresponding events in the nominal target.

PredIntoName

The “PredIntoName” table contains the name mapping between the predicted target variable and the original target variable when the target is nominal.

PredName

The “PredName” table contains the name mapping between the predicted target variable and the original target variable when the target is interval.

Constraint History

The “Constraint History” table lists the values of the various constraints at each iteration of the EGR algorithm. This table is produced by the MITIGATEBIAS statement.

Group Metrics History

The “Group Metrics History” table displays the metrics for accuracy and fairness for each subpopulation over each iteration of the EGR algorithm. This table is produced by the MITIGATEBIAS statement.

Iteration History

The “Iteration History” table displays the values of accuracy and bias metric at each iteration of the EGR algorithm. This table is produced by the MITIGATEBIAS statement.

Mitigation Information

The “Mitigation Information” table displays information about the options that you use for bias mitigation. This table is produced by the MITIGATEBIAS statement.

Timing

The “Timing” table displays the breakdown of how long bias mitigation spent on the various tasks of training models. This table is produced by the MITIGATEBIAS statement.

Weight History

The “Weight History” table displays the weights that are assigned by the EGR algorithm for each level of the response (target) variable and for each subpopulation. This table is produced by the MITIGATEBIAS statement.

ODS Table Names

Each table that the NNET procedure creates has a name associated with it. You must use this name to refer to the table when you use ODS statements. The name of each table and a short description of the contents are listed in Table 24.3.

Table 24.3 ODS Tables Produced by PROC NNET

Table Name	Description	Statement	Option
BestConfiguration	Hyperparameters and objective function values for the best configuration	AUTOTUNE	Default
ConstraintHistory	Values of the various constraints at each iteration of the EGR algorithm	MITIGATEBIAS	Default
ConvergenceStatus	Convergence status	PROC NNET	Default
CrossValidateMLFitStat	Error metrics for *k*-fold cross validation	CROSSVALIDATION	Default
EvaluationHistory	Values of the hyperparameters, the objective function for the all model configurations evaluated by the tuner	AUTOTUNE	Default
FitStat	Error metrics for supervised learning models	AUTOTUNE, TARGET	Default
GroupMetricsHistory	Metrics for accuracy and fairness for each subpopulation over each iteration of the EGR algorithm	MITIGATEBIAS	Default

Table 24.3 *continued*

Table Name	Description	Statement	Option
HyperparameterImportance	Relative importance of all hyperparameters	AUTOTUNE	Default
IterationHistory	Values of the model accuracy and bias metric over each iteration of the EGR algorithm	MITIGATEBIAS	Default
MitigationInformation	Information about the options that are used for bias mitigation	MITIGATEBIAS	Default
ModelInfo	Information about the modeling environment	PROC NNET	Default
OptIterHistory	Optimization iteration history information	PROC NNET	Default
PredIntoName	The name mapping between the predicted target variable and the original target variable when the target is nominal	TARGET	PRINTTARGET LEVEL=NOMINAL
PredName	The name mapping between the predicted target variable and the original target variable when the target is interval	TARGET	PRINTTARGET LEVEL=INTERVAL
PredProbName	The name mapping between the score variables and the corresponding events in the nominal target	TARGET	PRINTTARGET LEVEL=NOMINAL
ROCInfo	ROC curve information for the specified event in the nominal target	AUTOTUNE, TARGET	LEVEL=NOMINAL
ScoreInfo	Misclassification rate or mean square error for partition sets	SCORE, PARTITION	VALIDATION=
Timing	Breakdown of how long bias mitigation spent on the various tasks of training models	MITIGATEBIAS	Default
TunerInfo	Setup values used by the tuner	AUTOTUNE	Default

Table 24.3 *continued*

Table Name	Description	Statement	Option
TunerResults	Values of the hyperparameters, the objective function for the default configuration (Iteration 0), and up to 10 best configurations found	AUTOTUNE	Default
TunerSummary	Statistics about the tuning process	AUTOTUNE	Default
TunerTiming	Total time spent on different tasks during tuning	AUTOTUNE	Default
WeightHistory	Weights that are assigned by the EGR algorithm for each level of the response (target) variable and for each subpopulation	MITIGATEBIAS	Default

ODS Graphics

You can refer to every graph that is produced through ODS Graphics by name. The name of the graph that PROC NNET generates is listed in Table 24.4.

Table 24.4 Graph Produced by PROC NNET

ODS Graph Name	Plot Description	Statement	Option
IterationHistoryPlot	Iteration history of the model accuracy and bias metric	MITIGATEBIAS	Default

Examples: NNET Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

Example 24.1: Binary Target Classification with Partition

This example demonstrates how to use PROC NNET to predict whether a mortgage applicant will default on a loan. The data table Hmeq, which is in the Sampsio library that SAS provides, contains observations for 5,960 mortgage applicants. A variable named Bad indicates whether the applicant, after being approved for a loan, paid off or defaulted on the loan.

The following DATA steps load the Hemq data set into a SAS library in the first statement of each step:

```
data mylib.hmeq;
   set sampsio.hmeq;
run;
```

These statements assume that the SAS library is named mylib, but you can substitute any appropriately defined SAS library.

The following statements run PROC NNET and output the results to ODS tables. Based on the specified percentages, the PARTITION statement randomly splits the hemq data set into three partitions: training, validation, and testing. The error rates of the validation data set are computed during the training process as one of the stopping criteria in order to prevent overfitting.

```
proc nnet data=mylib.hmeq standardize=midrange missing=mean;
   architecture mlp;
   input job reason / level=nominal;
   input debtinc delinq loan mortdue value yoj derog clage clno;
   hidden 7;
   target bad / level=nominal;
   optimization algorithm=lbfgs maxiter=500;
   train outmodel=mylib.nnetmodel1 seed=12345;
   partition fraction(validate=0.2 test=0.1 seed=54321);
run;
```

The PROC NNET call creates the model, nnetModel1, from the training data; this model contains all the weight of the neural network. The example is run on one controller node.

Output 24.1.1 shows the model information for the neural network.

Output 24.1.1 Model Information

The NNET Procedure

Model Information	
Model	Neural Net
Number of Observations Used	4139
Number of Observations Read	4139
Target/Response Variable	BAD
Number of Nodes	28
Number of Input Nodes	19
Number of Output Nodes	2
Number of Hidden Nodes	7
Number of Hidden Layers	1
Number of Weight Parameters	140
Number of Bias Parameters	9
Architecture	MLP
Seed for Initial Weight	12345
Optimization Technique	LBFGS
Number of Neural Nets	1
Objective Value	2.0061305595
Misclassification Rate for Validation	0.2055

Output 24.1.2 shows the misclassification rate of the training data.

Output 24.1.2 Score Information for Training Data

Score Information for Training	
Number of Observations Read	4139
Number of Observations Used	4139
Misclassification Rate	0.1942

Output 24.1.3 shows the misclassification rate of the validation data.

Output 24.1.3 Score Information for Validation Data

Score Information for Validation	
Number of Observations Read	1241
Number of Observations Used	1241
Misclassification Rate	0.2055

Output 24.1.4 shows the misclassification rate of the testing data.

Output 24.1.4 Score Information for Testing Data

Score Information for Testing	
Number of Observations Read	580
Number of Observations Used	580
Misclassification Rate	0.2241

Example 24.2: Finding the Best Neural Network Configuration

This example illustrates how to use the AUTOTUNE statement to search for the best set of hyperparameters within the domains that you specify. The data set (iris) is the same data set that is used in the section "Getting Started: NNET Procedure" on page 623. The AUTOTUNE statement searches for the best network for iris within two hidden layers (each of which has specified ranges), and it also searches for the best L1 and L2 regularization values based on the specified ranges. Only one controller node is used in the example.

You can load the sashelp.iris data set into your SAS library by naming your SAS library in the first statement of the following DATA step. These statements assume that your SAS library is named mylib, but you can substitute any appropriately defined SAS library.

```
data mylib.iris;
   set sashelp.iris;
run;
```

The following statements run PROC NNET and output the results to ODS tables. The AUTOTUNE statement activates the tuning optimization algorithm, which applies the specified ranges in the local searching process.

```
proc nnet data=mylib.iris;
   input SepalLength SepalWidth PetalLength PetalWidth;
   target Species / level=nominal;
   train outmodel=mylib.nnetModel2 seed=1517878693;
   autotune useparameters=custom objective=MCE searchmethod=GA
            tuningparameters=(nhidden(LB=1 UB=2 INIT=1)
                              nunits1(LB=1 UB=10 INIT=1)
                              nunits2(LB=1 UB=15 INIT=2)
                              regl1(LB=1e-04 UB=1e-02 INIT=1e-03)
                              regl2(LB=1e-04 UB=1e-02 INIT=1e-03)
                              );
   optimization algorithm=LBFGS maxiter=100;
run;
```

Output 24.2.1 shows the setup values that are used by the tuner.

Output 24.2.1 Tuner Information

The NNET Procedure

Tuner Information	
Model Type	Neural Net
Tuner Objective Function	Misclassification
Search Method	GA
Population Size	10
Maximum Iterations	5
Maximum Tuning Time in Seconds	36000
Validation Type	Single Partition
Validation Partition Fraction	0.30
Log Level	2
Seed	1517878693
Number of Parallel Evaluations	19
Number of Workers per Subsession	1

Output 24.2.2 shows the results that are reported by the NNET procedure. The first row displays results from the default settings, the second row displays the best results that are found by the tuner, and the third row displays the second-best results that are found.

Output 24.2.2 Tuner Results

Tuner Results
Default and Best Configurations

Evaluation	Hidden Layers	Neurons in Hidden Layer 1	Neurons in Hidden Layer 2	L1 Regularization	L2 Regularization	Misclassification	Evaluation Time in Seconds
0	1	1	0	0.001000	0.001000	0.3333	2.01
23	1	6	0	0.001179	0.000231	0.0000	1.46
51	1	6	0	0.001179	0.000231	0.0000	1.66
52	1	6	0	0.001179	0.000231	0.0000	1.62
65	1	6	0	0.001179	0.000231	0.0000	1.45
67	1	6	0	0.001179	0.000231	0.0000	1.55
81	1	6	0	0.001179	0.000231	0.0000	1.46
83	1	6	0	0.001179	0.000231	0.0000	1.71
12	1	6	0	0.001000	0.001000	0.0222	23.28
14	2	1	8	0.001000	0.001000	0.0222	24.02
17	2	6	8	0.010000	0.001000	0.0222	24.02

Output 24.2.3 shows the best values of the tuning parameters from the tuning process.

Output 24.2.3 Best Configuration

Best Configuration	
Evaluation	23
Hidden Layers	1
Neurons in Hidden Layer 1	6
Neurons in Hidden Layer 2	0
L1 Regularization	0.00117874
L2 Regularization	0.00023089
Misclassification	0

Output 24.2.4 shows the tuner summary.

Output 24.2.4 Tuner Summary

Tuner Summary	
Initial Configuration Objective Value	0.3333
Best Configuration Objective Value	0
Worst Configuration Objective Value	0.3333
Initial Configuration Evaluation Time in Seconds	2.0096
Best Configuration Evaluation Time in Seconds	1.4565
Number of Improved Configurations	4
Number of Evaluated Configurations	86
Total Tuning Time in Seconds	34.5837
Parallel Tuning Speedup	3.9902

Output 24.2.5 shows the run time for each task during the searching process. It is evident that the tuner spent the vast majority of its time on training; this behavior is similar to most tuner runs. Therefore, it is important to understand that tuning might take a very long time by nature if the training time is long. Typically, networks that have more neurons or larger training samples take more time; also, if the value of the MAXITER= option is very large and the nonlinear objective function converges slowly, the run time could be very long. In general, tuner performance should not be a concern, because you typically use tuning only once in a while.

Output 24.2.5 Tuner Timing

Tuner Task Timing		
Task	**Seconds**	**Percent**
Model Training	71.80	52.03
Model Scoring	42.24	30.61
Total Objective Evaluations	114.06	82.66
Tuner	23.93	17.34
Total CPU Time	138.00	100.00

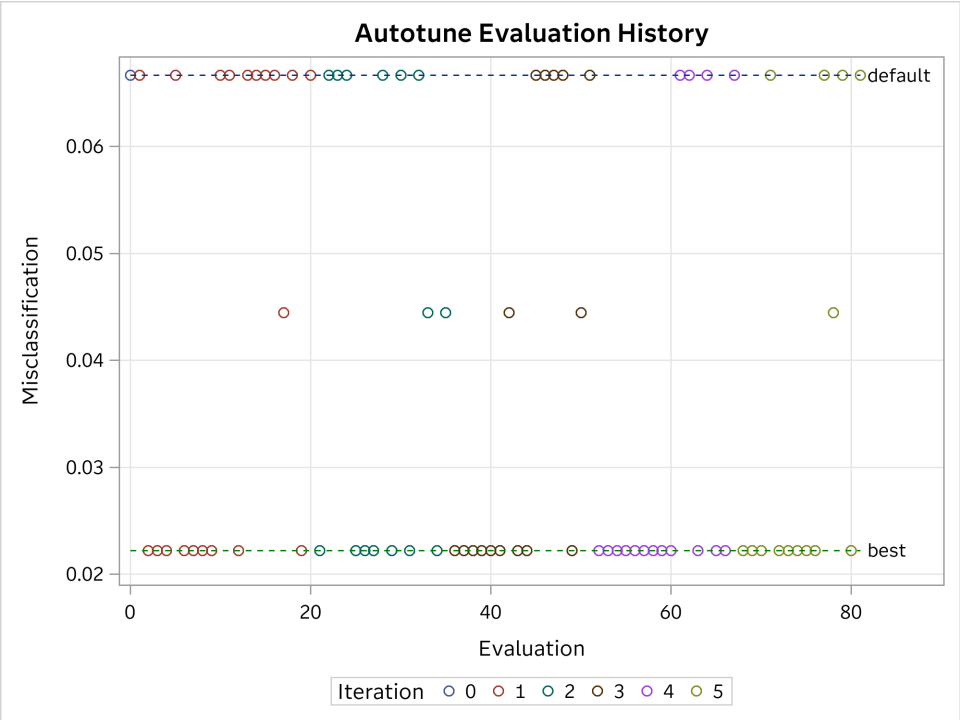
Output 24.2.6 shows the relative importance of each hyperparameter.

Output 24.2.6 Hyperparameter Importance

Hyperparameter Importance	
Hyperparameter	**Relative Importance**
REGL1	1.0000
NUNITS1	0.9351
REGL2	0.4244
NUNITS2	0.0297
NHIDDEN	0.0192

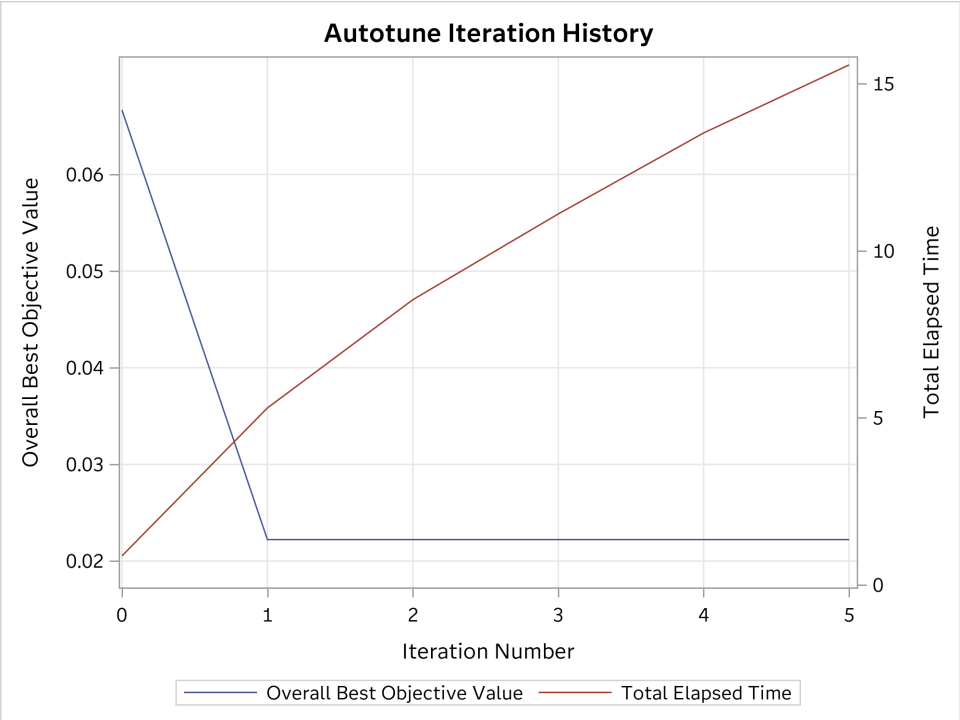
Output 24.2.7 displays a scatter plot of all configurations that the tuner tried. The objective values are shown on the Y axis, and the evaluation numbers are shown on the X axis.

Output 24.2.7 Evaluation History Plot



The plot in Output 24.2.8 displays how the best found objective value and the elapsed time changed with each iteration of the tuner.

Output 24.2.8 Iteration History Plot



Example 24.3: Mitigating Bias of a Classification Neural Network

This example uses the NNET procedure to create a neural network model while minimizing the demographic parity for the Job variable. The response variable for the neural network model is Bad, a classification variable that has two values (0 for payment of loan and 1 for default). The other variables are predictor variables for the model.

The following statements load the data table mylib.hmeq into your SAS library. For this example, the statements assume that your SAS library is named mylib, but you can substitute any appropriately defined SAS library.

```
data mylib.hmeq;
   set sampsio.hmeq;
run;
```

The following statements use PROC NNET to create a neural network model without minimizing the demographic parity:

```
proc nnet data=mylib.hmeq missing=mean;
   input CLAge CLNo DebtInc Derog
         Loan MortDue Value / level=interval;
   input Reason / level=nominal;
   target Bad / level=nominal;
   hidden 50;
   train seed=54321 outmodel=mylib.nnetModel3;
   output out=mylib.scored copyvars=(_ALL_);
   saveState rstore=mylib.model;
run;
```

The MISSING= option specifies that missing values are replaced by the mean value of the corresponding interval input variable. The two INPUT statements specify CLAge, CLNo, DebtInc, Derog, Loan, MortDue, Value, and Reason as the input variables. The TARGET statement specifies Bad as the target variable. The HIDDEN statement specifies one hidden layer with 50 neurons. The TRAIN statement specifies the name of the final model and seed for initial random weight generation. The OUTPUT statement generates the data table mylib.scored, which contains the predicted mortgage default from the model for each observation and contains all variables from the input data table as copy variables. The SAVESTATE statement creates an analytic store for the model and saves it as a binary object in the data table mylib.model.

Output 24.3.1 shows the misclassification rate of the training data.

Output 24.3.1 Score Information for Training Data

The NNET Procedure

Score Information	
Number of Observations Read	5960
Number of Observations Used	5960
Misclassification Rate	0.1475

After you train a classification neural network model, you might want to view the demographic parity of the trained model. You can do this by running the ASSESSBIAS procedure. For more information about PROC ASSESSBIAS, see Chapter 3, "ASSESSBIAS Procedure."

The following statements use PROC ASSESSBIAS to assess the bias measurements of the model:

```
proc assessbias data=mylib.scored;
   target bad / event="1" level=nominal;
   var P_Bad1;
   fitstat pvar=P_BAD0 / pevent="0";
   sensitiveVar Job;
run;
```

Output 24.3.2 displays the bias metric values of the model. PROC ASSESSBIAS also produces other output not shown here.

Output 24.3.2 Bias Metrics Table

The ASSESSBIAS Procedure

Bias Metrics					
Bias Statistic	**Bias Statistic Label**	**Bias Statistic Value**	**Base Level**	**Compare Level**	**Note**
DemographicParity	Demographic Parity (Statistical Parity)	0.1236	Self	Office	
PredictiveParity	Predictive Parity	0.1384	Sales	ProfExe	
EqualAccuracy	Equal Accuracy	0.2071		Sales	
EqualizedOdds	Equalized Odds	0.2063		Office	The maximum TPR difference is greater than the maximum FPR difference.
EqualOpportunity	Equal Opportunity	0.2063		Office	

In order to use the MITIGATEBIAS statement to reduce the demographic parity, you must specify the levels of the response (target) variable and the corresponding predicted variables. You can use the DESCRIBE statement in the ASTORE procedure to obtain this information. The following statements use PROC ASTORE to describe the model:

```
proc astore;
   describe rstore=mylib.model;
run;
```

Output 24.3.3 lists the output variables that the model creates. The predicted variables are P_BAD1 and P_BAD0. Their corresponding response (target) levels are 1 and 0, respectively.

Output 24.3.3 Output Variables Table

The ASTORE Procedure

Output Variables			
Name	**Length**	**Type**	**Label**
I_BAD	12	Character	
P_BAD1	8	Num	Predicted: BAD=1
P_BAD0	8	Num	Predicted: BAD=0

The following statements use PROC NNET to create a neural network model, while minimizing the demographic parity for Job by using the MITIGATEBIAS statement:

```
ods graphics on;

proc nnet data=mylib.hmeq missing=mean;
   input CLAge CLNo DebtInc Derog
         Loan MortDue Value / level=interval;
   input Reason / level=nominal;
   hidden 50;
   target Bad / level=nominal;
   train seed=54321 outmodel=mylib.nnetModel4;
   output out=mylib.scored;
   saveState rstore=mylib.model;
   mitigateBias pevents="1 0"
                pvars=(P_BAD1 P_BAD0)
                seed=12345
                sensitivevar=Job
                targetEvent="1";
run;
```

The PEVENTS= option lists the levels of the response (target) variable Bad. The predicted variables correspond to the levels that are listed in the PVARS= option. The PVARS= option lists the predicted variables that contain the posterior probability for each level in model prediction that corresponds to the response (target) variable Bad. The SEED= option specifies the seed for the pseudorandom number generator to be 12345. The SENSITIVEVAR= option names the variable Job as the sensitive variable for the purpose of reducing the value of the bias measurement. The TARGETEVENT= option specifies the value 1 as the event of interest.

Output 24.3.4 displays the information about the options that are used for bias mitigation. Output 24.3.5 displays the model accuracy and bias metric values over each iteration of the EGR algorithm. Output 24.3.6 shows the iteration history, which is a graphical representation of the iteration history table. The graph displays the bias metric and the misclassification rate for each iteration. PROC NNET selects the iteration that has the lowest bias metric. In this case, the last iteration is selected. From the iteration history, you can observe that bias mitigation has reduced the gap of demographic parity from 0.12309 to 0.037904, whereas the overall misclassification rate increases only by 0.0077. The MITIGATEBIAS statement also creates other tables, which are not shown here but are listed in the section "ODS Table Names" on page 653.

Output 24.3.4 Mitigation Information Table

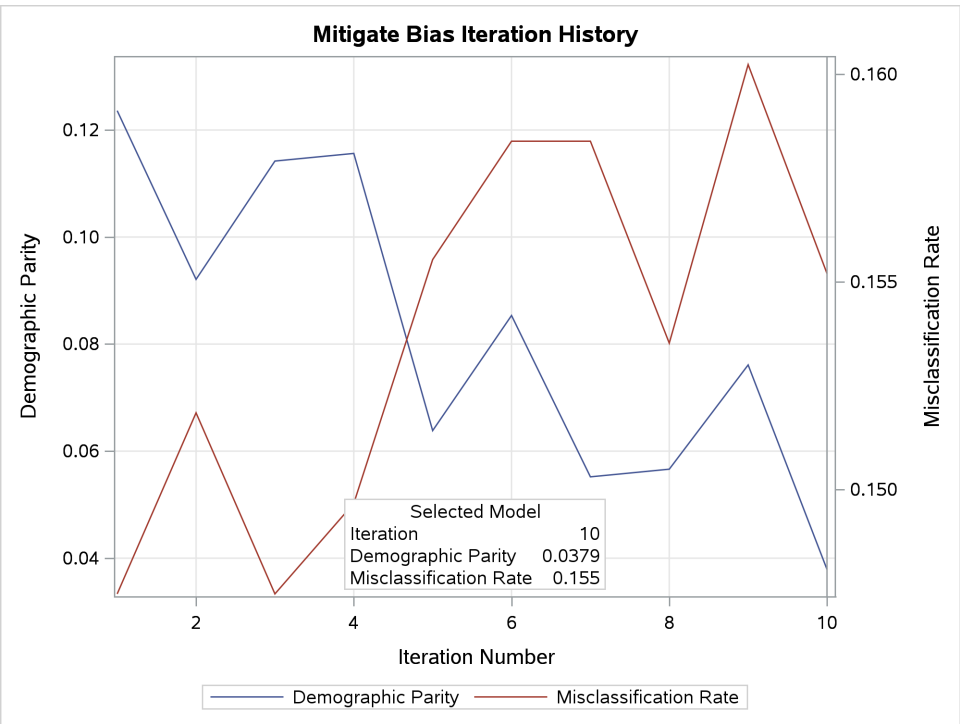
The NNET Procedure

Mitigation Information		
Description	**Character Value**	**Value**
Bias Metric	DEMOGRAPHICPARITY	.
Sensitive Variable	JOB	.
Response	BAD	.
Bound Value	100	100
Learning Rate	0.01	0.01
Maximum Iteration	10	10
Tolerance	0.005	0.005
Event Level	1	.

Output 24.3.5 Iteration History Table

EGR Iteration History			
Iteration	Demographic Parity (Statistical Parity)	Misclassification Rate	Multi-Class Log Loss
1	0.123609	0.1475	0.3782
2	0.092091	0.1518	0.3841
3	0.114197	0.1475	0.3767
4	0.115624	0.1497	0.3780
5	0.063833	0.1555	0.3872
6	0.085342	0.1584	0.3919
7	0.055197	0.1584	0.3940
8	0.056627	0.1535	0.3912
9	0.076104	0.1602	0.3943
10	0.037904	0.1552	0.3986

Output 24.3.6 Iteration History Graph



References

Agarwal, A., Beygelzimer, A., Dudik, M., Langford, J., and Wallach, H. (2018). "A Reductions Approach to Fair Classification." In *Proceedings of the 35th International Conference on Machine Learning*, 60–69. PMLR.

Bishop, C. M. (1995). *Neural Networks for Pattern Recognition*. Oxford: Oxford University Press.

Fisher, R. A. (1936). "The Use of Multiple Measurements in Taxonomic Problems." *Annals of Eugenics* 7:179–188.

Chapter 25
NOMINALDR Procedure

Contents

Overview: NOMINALDR Procedure **668**
- PROC NOMINALDR Features 668
- Using SAS Viya Workbench 669

Getting Started: NOMINALDR Procedure **669**

Syntax: NOMINALDR Procedure **671**
- PROC NOMINALDR Statement 671
- DISPLAY Statement 673
- DISPLAYOUT Statement 674
- FREQ Statement 675
- ID Statement 675
- INPUT Statement 675
- OUTPUT Statement 676
- SAVESTATE Statement 676

Details: NOMINALDR Procedure **677**
- Multiple Correspondence Analysis (MCA) 677
- Logistic Principal Component Analysis (LPCA) 677
- Displayed Output 678
- ODS Table Names 679
- Output Data Tables 680

Example: NOMINALDR Procedure **680**
- Example 25.1: Dimensionality Reduction with PROC NOMINALDR for Classification 681

References **684**

Overview: NOMINALDR Procedure

The proliferation of advanced technologies and large-scale data collection has made high-dimensional data more prevalent than ever. However, handling such data comes with challenges. High-dimensional data sets require significant storage space and memory to load and process. Even when they are successfully loaded, they demand machines with high computational power and require long processing times. Additionally, high dimensionality can lead to issues such as the curse of dimensionality and overfitting.

To address these challenges, dimension-reduction techniques are used to reduce the size of the data set while preserving its key patterns. These methods help reduce storage requirements and can also serve as preprocessing steps for more efficient subsequent analysis. Principal component analysis is one of the most widely used dimension-reduction methods, but it is limited to continuous data sets. However, many real-world data sets, especially in fields like social sciences, marketing, bioinformatics, and survey analysis, contain nominal data.

The NOMINALDR procedure implements dimensionality reduction methods for nominal data. For more information about these models, see the section "Details: NOMINALDR Procedure" on page 677.

PROC NOMINALDR Features

PROC NOMINALDR has the following features:

- reads input data in parallel when the data source is on a distributed system
- is multithreaded during all phases of analytic execution
- supports large-scale training data
- enables you to perform analysis on selected columns of data
- enables you to reduce the dimensionality of nominal data
- enables you to save the dimensionality reduction model for later application to new nominal data with the ASTORE procedure

Using SAS Viya Workbench

SAS Viya Workbench has greatly simplified the SAS programming experience by offering a syntax style nearly identical to that of SAS 9 and earlier releases. However, there are minor exceptions. In the past, SAS Viya procedures required the use of a CAS engine libref. This engine served multiple purposes, but it was primarily used to upload data to the CAS server and manage the data. Its additional purposes included support of new data types and improved performance for large data sets. No CAS server is needed for SAS Viya Workbench, because all computations are done in place for you in a single location.

In the following program, the LIBNAME statement creates a mylib libref that the examples in this chapter use to refer to your current or desired work directory:

```
libname mylib ".";
```

For more information, see *SAS Viya Workbench: Getting Started*.

Getting Started: NOMINALDR Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

This example demonstrates how to use the NOMINALDR procedure to reduce the dimensionality of a nominal data table.

The molecularBiology data set, which is derived from molecular biology research, is used in the example. The data set is available from the UCI Machine Learning Repository (1991). You can get the data set from the website at **https://archive.ics.uci.edu/dataset/69/molecular+biology+splice+junction+gene+sequences**.[1] The molecularBiology data set includes 60 nominal variables: Base1, Base2, . . . , Base60. This example demonstrates how to reduce these 60 nominal variables to 10 reduced variables.

Assume that the data set is stored as a comma-separated-value (CSV) file named *molecularBiology.csv* in the current working directory. The following DATA step loads the data table from the directory into your SAS library by using your libref:

```
proc casutil;
    load file="molecularBiology.csv" /*or other user-defined location*/
    casout="molecularBiology"
    importoptions=(filetype="CSV" getnames="TRUE");
run;
```

[1] Disclaimer: SAS may reference other websites or content or resources for use at Customer's sole discretion. SAS has no control over any websites or resources that are provided by companies or persons other than SAS. Customer acknowledges and agrees that SAS is not responsible for the availability or use of any such external sites or resources, and does not endorse any advertising, products, or other materials on or available from such websites or resources. Customer acknowledges and agrees that SAS is not liable for any loss or damage that may be incurred by Customer or its end users as a result of the availability or use of those external sites or resources, or as a result of any reliance placed by Customer or its end users on the completeness, accuracy, or existence of any advertising, products, or other materials on, or available from, such websites or resources.

The following statement uses PROC NOMINALDR to reduce the 60 nominal input variables to 10, as specified by the DIMENSION= option:

```
proc NOMINALDR data=mylib.molecularBiology dimension=10 method=MCA;
    input Base1-Base60 /LEVEL=NOMINAL;
    output out=mylib.mcaMolecularBiology;
run;
```

The METHOD= option specifies that the procedure is to use the MCA method.

When the training is completed, the procedure produces an output data table named mcaMolecularBiology. This table contains the variables that are reduced by the MCA method for all observations.

The following statements print the reduced variables for the first five observations in the mcaMolecularBiology table. The statements assume that your libref is named mylib, but you can substitute any appropriately defined SAS libref. The output is shown in Figure 25.1.

```
proc print data=mylib.mcaMolecularBiology(obs=5);
run;
```

Figure 25.1 Variables of the First Five Observations Reduced by the MCA Method

Obs	rv1	rv2	rv3	rv4	rv5	rv6	rv7	rv8	rv9	rv10
1	-0.017147	-0.25936	-0.10532	0.04206	0.09260	-0.015206	-0.00429	-0.09158	0.08327	0.09266
2	-0.016186	-0.53011	-0.35331	-0.04692	0.29795	0.030010	-0.11183	0.05915	0.01256	0.28951
3	-0.012715	-0.33034	-0.29113	0.33252	-0.13742	-0.001791	0.33437	-0.16953	0.00947	-0.02099
4	-0.019876	-0.33007	-0.01995	0.27982	0.02057	0.021704	-0.05290	-0.14793	0.15868	0.44207
5	-0.018986	-0.31486	-0.23000	0.01996	0.46719	0.061501	-0.05140	0.08693	0.03574	-0.39435

Similarly, you can use the LPCA method to reduce the dimensionality of the nominal variables in the molecularBiology data table by using the following statements. The METHOD= option specifies that the procedure is to use the LPCA method. The M= option specifies the approximation of $\mathrm{logit}(1)$ and $-\mathrm{logit}(0)$.

```
proc NOMINALDR data=mylib.molecularBiology dimension=10 method=LPCA m=1;
    input Base1-Base60 /LEVEL=NOMINAL;
    output out=mylib.lpcaMolecularBiology;
run;
```

The variables that you reduce by using the LPCA method are contained in the lpcaMolecularBiology data table. The following statements print the variables that are reduced by the LPCA method for the first five observations. The results are shown in Figure 25.2.

```
proc print data=mylib.lpcaMolecularBiology(obs=5);
run;
```

Figure 25.2 Variables of the First Five Observations Reduced by the LPCA Method

Obs	rv1	rv2	rv3	rv4	rv5	rv6	rv7	rv8	rv9	rv10
1	2.33976	0.44913	1.15030	0.32812	0.55270	0.22815	-0.28483	0.02045	0.78380	3.18420
2	3.55188	0.52928	3.39016	-0.89492	2.61275	0.06044	0.72973	0.44475	2.31191	-0.41688
3	2.75207	0.41207	2.97719	1.66522	-2.02575	2.18676	-1.86302	-0.82404	-0.15550	-0.07708
4	2.96910	0.37273	0.62074	1.94794	-0.35877	0.07342	-0.25042	-0.66327	1.68980	0.73484
5	2.26516	0.51982	2.46082	0.13719	4.05922	0.31809	-0.01384	-0.31078	-2.53159	1.34748

Syntax: NOMINALDR Procedure

The following statements are available in the NOMINALDR procedure:

```
PROC NOMINALDR CUMVARIANCE=number | DIMENSION=number < options > ;
   DISPLAY < table-list > < /options > ;
   DISPLAYOUT table-spec-list < /options > ;
   FREQ variable ;
   ID variables ;
   INPUT variables < / LEVEL=INTERVAL | NOMINAL > ;
   OUTPUT OUT=libref.data-table < options > ;
   SAVESTATE RSTORE=libref.data-table ;
```

The PROC NOMINALDR and INPUT statements are required. The following sections describe the PROC NOMINALDR statement and then describe the other statements in alphabetical order.

PROC NOMINALDR Statement

```
PROC NOMINALDR CUMVARIANCE=number | DIMENSION=number < options > ;
```

The PROC NOMINALDR statement invokes the procedure. Table 25.1 summarizes the options available in this statement.

Table 25.1 PROC NOMINALDR Statement Options

Option	Description
Input Data Table Option	
DATA=	Specifies the input data table
Other Options	
CUMVARIANCE=	Specifies the minimum cumulative proportion of variance that must be explained by the reduced variables
DIMENSION=	Specifies the number of reduced variables
EPSILON=	Specifies the tolerance that is used in determining the convergence of the iterative algorithm
M=	Specifies a finite positive value m to approximate the logit function's infinite limits: $\mathrm{logit}(1)$ is approximated by m, and $\mathrm{logit}(0)$ is approximated by $-m$
MAXITER=	Specifies the maximum number of iterations of the iterative algorithm
METHOD=	Specifies the method that the dimensionality reduction model uses for nominal variables
NTHREADS=	Specifies the number of threads to use on each computation node
PREFIX=	Specifies a prefix to apply to the names of the reduced output variables
RANDSVD	Uses the randomized SVD method instead of the exact SVD method

You must specify one of the following options. If you specify both options, the CUMVARIANCE= option is ignored.

CUMVARIANCE=*number*
specifies the minimum cumulative proportion of variance that must be explained by the reduced variables. The *number* must be in the range $(0, 1]$.

DIMENSION=*number*
specifies the number of reduced variables. The *number* must be an integer greater than or equal to 1.

You can also specify the following *options*:

DATA=*libref.data-table*
names the input data table for PROC NOMINALDR to use. The default is the most recently created data table. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to the data. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 669.

data-table specifies the name of the input data table.

EPSILON=*number*
specifies the tolerance that is used in determining the convergence of the iterative algorithm when METHOD=LPCA. The *number* must be in the range $[10^{-8}, 0.1]$. The default value is 0.0001.

M=*number*
specifies a finite positive value m to approximate the logit function's infinite limits in the natural parameters of the saturated Bernoulli distribution for the observations: $\mathrm{logit}(1)$ is approximated by m, and $\mathrm{logit}(0)$ is approximated by $-m$. This option is used when METHOD=LPCA. For more information, see the section "Logistic Principal Component Analysis (LPCA)" on page 677. The *number* must be in the range $(0, 10]$. By default, M=4.

MAXITER=*number*
specifies the maximum number of iterations of the iterative algorithm when METHOD=LPCA. The *number* must be an integer in the range $[1, 10^8]$. By default, MAXITER=100.

METHOD=LPCA | MCA
specifies the method that the dimensionality reduction model uses for nominal variables. You can specify the following values:

LPCA uses the logistic principle component analysis (LPCA) method. For more information about this method, see the section "Logistic Principal Component Analysis (LPCA)" on page 677.

MCA uses the multiple correspondence analysis (MCA) method. For more information about this method, see the section "Multiple Correspondence Analysis (MCA)" on page 677.

By default, METHOD=MCA.

NTHREADS=*number-of-threads*
specifies the number of threads per computation node. The default value is the number of threads available per computation node.

PREFIX=*name*
specifies a prefix to apply to the names of the reduced output variables. By default, the reduced variables are named rv1, rv2, and so on. If you specify PREFIX=Abc, the reduced variables are named Abc1, Abc2, Abc3, and so on. The number of characters in the prefix plus the number of digits that are required to designate the variables should not exceed the current name length that is defined by the VALIDVARNAME= system option. For more information about the VALIDVARNAME= system option, see *SAS System Options: Reference*.

RANDSVD < (POWER=*number***) >**
uses the randomized singular value decomposition (SVD) method when you specify the DIMENSION= and METHOD=MCA options. By default, the exact SVD method is used. You should use this option only when the total number of levels is so large that it takes a long time for the exact SVD method to run.

You can specify the following suboption:

POWER=*number*
specifies the parameter power (number of power iterations) for the randomized SVD method to use when you specify the RANDSVD option. The POWER= suboption is ignored when you omit the RANDSVD option. The *number* must be a positive integer. A larger *number* results in greater accuracy but slower execution. Although there is no maximum value, you should specify a *number* no greater than 10. By default, POWER=3.

DISPLAY Statement

DISPLAY < *table-list* > < / *options* > **;**

The DISPLAY statement enables you to specify a list of display tables to display or exclude. This statement is similar to the ODS SELECT, ODS EXCLUDE, and ODS TRACE statements. However, the DISPLAY statement can improve performance when a large number of tables could be generated (such as in BY-group processing). The DISPLAY statement produces only the requested tables, but ODS statements generate all the tables and then keep the requested tables.

If you use both DISPLAY and ODS statements together, the DISPLAY statement takes precedence over the ODS statements. Note that the ODS EXCLUDE statement processes tables after they have been filtered by the DISPLAY statement. In some cases, it might appear that the ODS EXCLUDE statement is taking precedence because it can further filter the tables. For more information about ODS, see *SAS Output Delivery System: Procedures Guide*.

You can specify the *table-list* as a list of table names, paths, partial pathnames, and regular expressions.

The table names that you can specify are listed in the section "ODS Table Names" on page 679. A path is a table name that is prefixed with dot-separated grouping information. For example, a SelectionSummary table that a procedure produces during a selection routine might have the path *Bygroup1.Summary.SelectionSummary*. A partial pathname does not include all groups; for example, *SelectionSummary* and *Summary.SelectionSummary* are partial pathnames for *Bygroup1.Summary.SelectionSummary*.

When you specify a table name or partial pathname, all display tables whose paths end in the specified name are selected for display or exclusion. For example, both *SelectionSummary* and *Summary.SelectionSummary* select *Bygroup1.Summary.SelectionSummary*.

A regular expression is enclosed in forward slashes (/). For example, specifying "/tions/" selects all pathnames that contain the substring "tions"; in particular, the *Bygroup1.Summary.SelectionSummary* table is selected. Specifying "!/tions/" selects all pathnames that do not contain the substring "tions"; in particular, the *Bygroup1.Summary.SelectionSummary* table is not selected.

You can specify the following *options* after a slash (/):

CASESENSITIVE
performs a case-sensitive comparison of table names in the *table-list* to display table names when tables are subsetted for display. To preserve case, you must enclose table names in the *table-list* in quotation marks.

EXCLUDE
displays all display tables except those that you specify in the *table-list*.

EXCLUDEALL
suppresses display of all tables. This option takes precedence over the other options.

TRACE
displays the display table names, labels, and paths.

DISPLAYOUT Statement

DISPLAYOUT *table-spec-list* < / *options* > **;**

The DISPLAYOUT statement enables you to create output tables from your displayed output. This statement is similar to the ODS OUTPUT statement. For more information about ODS, see *SAS Output Delivery System: Procedures Guide*.

The *table-spec-list* specifies a list of output tables to create. Each entry in the list has either a *key*=*value* format or a *key* format:

key=*value* specifies *key* as the ODS table name, path, or partial pathname, and specifies *value* as the output table name.

key specifies *key* as the ODS table name and also as the output table name.

The ODS table names that you can specify are listed in the section "ODS Table Names" on page 679. You cannot specify the ODS table named OutputCasTables in the *table-spec-list*.

Table names and partial pathnames are discussed under the DISPLAY statement. The DISPLAYOUT statement does not support regular expressions.

You can specify the following *options* after a slash (/):

INCLUDEALL
creates output tables for all display tables. The name of the created output table is the same as the name of the corresponding display table. If you specify this option, the *table-spec-list* specification is ignored.

NOREPLACE
does not replace any existing output table of the same name.

FREQ Statement

FREQ *variable* **;**

The *variable* in the FREQ statement identifies a numeric variable in the input data table that contains the frequency of occurrence of each observation. PROC NOMINALDR treats each observation as if it appears f times, where f is the value of the FREQ *variable* for the observation. If f is not an integer, it is truncated to an integer. If f is less than 1 or missing, the observation is not used in the analysis. When you omit the FREQ statement, each observation is assigned a frequency of 1.

ID Statement

ID *variables* **;**

The ID statement lists one or more *variables* to copy from the input data table to the output data tables that you specify in the OUT= option in the OUTPUT statement and the RSTORE= option in the SAVESTATE statement.

INPUT Statement

INPUT *variables* < / **LEVEL=INTERVAL | NOMINAL** > **;**

The INPUT statement specifies the names of the *variables* whose dimensionality you want to reduce. It names one or more input *variables* that use common options. If you want to use different options for different *variables*, you can specify multiple INPUT statements.

You can specify the following option in each INPUT statement after a slash:

LEVEL=INTERVAL | NOMINAL
specifies the level of measurement of the *variables*. You can specify the following values:

INTERVAL specifies that the level of measurement of the *variables* is interval.

NOMINAL specifies that the level of measurement of the *variables* is nominal.

By default, LEVEL=INTERVAL for numeric variables and LEVEL=NOMINAL for categorical variables.

OUTPUT Statement

OUTPUT OUT=*libref.data-table* < *options* > **;**

The OUTPUT statement creates an output data table to contain the results of the procedure run.

You must specify the following option:

OUT=*libref.data-table*
names the output data table for PROC NOMINALDR to use. You must specify this option before any other options. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to where the data table is to be stored. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 669.

data-table specifies the name of the output data table.

You can also specify the following *option*:

COPYVAR=*variable*
COPYVARS=(*variables***)**
lists one or more *variables* from the input data table to be transferred to the output data table.

SAVESTATE Statement

SAVESTATE RSTORE=*libref.data-table* **;**

The SAVESTATE statement creates an analytic store for the model and saves it as a binary object in a data table. You can subsequently use the analytic store in the ASTORE procedure to score new data. For more information, see Chapter 4, "ASTORE Procedure."

You must specify the following option:

RSTORE=*libref.data-table*
specifies a data table in which to save the analytic store for the model. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 669.

Details: NOMINALDR Procedure

Multiple Correspondence Analysis (MCA)

Conceptually based on PCA, multiple correspondence analysis (MCA) is specifically designed for nominal data. It reduces dimensionality by analyzing the relationships between categories of all the nominal variables. Through this process, MCA projects high-dimensional nominal data sets into a smaller set of dimensions while retaining as much variability and structure as possible.

You can use MCA by specifying METHOD=MCA in the PROC NOMINALDR statement to reduce the dimensionality of nominal data. The implementation of this method in this procedure is a distributed, multithreaded implementation of the MCA algorithm in SAS Viya. The method accepts as its input a table, where each row is a training sample and each column is a nominal variable. The nominal variables can be either numeric or character. Typically, all observations for a given variable that have the same formatted value are placed in the same level, and observations that have different values are placed in different levels.

The MCA method begins by transforming the nominal data set into an indicator matrix, where each unique level of a nominal variable is represented as a binary column. The indicator matrix is then normalized to probabilities and subsequently standardized to balance contributions from different observations and levels. Next, singular value decomposition (SVD) is applied to the standardized matrix, and the top k singular values and their associated singular vectors are selected to produce the reduced-dimension data.

Logistic Principal Component Analysis (LPCA)

Principal component analysis (PCA), which reduces the dimensionality of continuous data, can be interpreted as maximizing the likelihood under the assumption that the data follow a Gaussian distribution. However, this assumption is incorrect for nominal or binary data. Binary data are more naturally modeled by using the Bernoulli distribution. Logistic principal component analysis (LPCA) extends PCA to binary data by maximizing the Bernoulli log likelihood, under the assumption that each binary value follows a Bernoulli distribution.

You can use LPCA by specifying the METHOD=LPCA option in the PROC NOMINALDR statement to reduce the dimensionality of nominal data. The implementation of this method in this procedure is a distributed, multithreaded implementation of the LPCA algorithm in SAS Viya. The method accepts as its input a table, where each row is a training sample and each column is a nominal variable. The nominal variables can be either numeric or character. Typically, all observations for a given variable that have the same formatted value are placed in the same level, and observations that have different values are placed in different levels.

The LPCA method treats each level of all nominal variables as separated binary variables. Let x_{ij} be a binary indicator that represents whether the observation i belongs to the level j. LPCA assumes that x_{ij} follows a Bernoulli distribution with parameter p_{ij}, and its natural parameter is $\theta_{ij} = \mathrm{logit}(p_{ij})$. For all observations $i = 1, 2, \ldots, I$ and all levels $j = 1, 2, \ldots, J$, the loss function is defined as

$$\mathrm{loss} = -\frac{2}{IJ} \sum_{i=1}^{I} \sum_{j=1}^{J} \log P(x_{ij} | \theta_{ij})$$

where $P(x_{ij}|\theta_{ij})$ denotes the probability of observing x_{ij} under a Bernoulli distribution with natural parameter θ_{ij}. LPCA minimizes this loss function, which is equivalent to maximizing the Bernoulli log likelihood.

LPCA solves the optimization problem subject to a low-rank structure of the $I \times J$ natural parameter matrix Θ, where the (i, j) element is θ_{ij}. Different formulations of this low-rank constraint have been proposed, including those in Schein, Saul, and Ungar (2003); De Leeuw (2006); Landgraf and Lee (2020).

The LPCA method in the NOMINALDR procedure enforces the low-rank structure as in Landgraf and Lee (2020):

$$\Theta = \mathbf{1}_n \mu^\top + (\tilde{\Theta} - \mathbf{1}_n \mu^\top) U U^\top, \quad U^\top U = I_k$$

Here $\tilde{\Theta}$ is the $I \times J$ matrix whose (i, j) element $\tilde{\theta}_{ij}$ is the saturated natural parameter of x_{ij}, $\mathbf{1}_n$ is an I-dimensional vectors of ones, I_k is the $k \times k$ identity matrix, and the J-dimensional vector μ and $J \times k$ orthonormal matrix U are estimated during the optimization. The scalar k is the dimension of the reduced variables.

For each binary indicator x_{ij} that is assumed to follow a Bernoulli distribution, the saturated distribution, which provides the best possible fit to the data, occurs when $\tilde{p}_{ij} = x_{ij}$. This implies that $\tilde{\theta}_{ij} = \mathrm{logit}(0) = -\infty$ if $x_{ij} = 0$, and $\tilde{\theta}_{ij} = \mathrm{logit}(1) = \infty$ if $x_{ij} = 1$. To make computation feasible, these infinite limits are approximated by using a finite positive constant m: $\mathrm{logit}(1)$ is approximated by m, and $\mathrm{logit}(0)$ is approximated by $-m$. In practice, m does not need to be very large, because the inverse logit function is 0.9933 at 5 and 0.9999 at 10.

In LPCA, the constrained optimization problem is solved iteratively by using the majorization-minimization algorithm. For more information, see Landgraf and Lee (2020).

When the optimization is complete, LPCA computes the reduced variables as $(\tilde{\Theta} - \mathbf{1}_n \mu^\top) U$.

Displayed Output

PROC NOMINALDR displays various tables for analysis. The following sections describe the output in the order of its appearance when you specify the related options.

Model Information

The "Model Information" table displays the number of nominal variables, the number of encoded dummy variables (which is the total number of levels for the nominal variables), and the number of reduced variables. When you specify the CUMVARIANCE= option in the PROC NOMINALDR statement, the table also displays the specified minimum cumulative proportion of variance that must be explained by the reduced variables. When you specify the METHOD=LPCA option in the PROC NOMINALDR statement, the "Model Information" table also displays the finite approximation of the logit function's infinite limits, convergence tolerance, maximum number of iterations, actual number of iterations, and best loss value that is achieved.

In addition, the "Model Information" table displays the actual number of threads that are used in the computation and the type of SVD method that is used. When you specify the DIMENSION=, METHOD=MCA, and RANDSVD options in the PROC NOMINALDR statement, the table also displays the parameter power that the randomized SVD method uses.

Number of Observations

The "Number of Observations" table displays the number of observations that are read from the input data table and the number that are used.

Variable Information

The "Variable Information" table shows the number of levels for the nominal variables.

Explained Variance Summary

The "Explained Variance Summary" table shows the variance that is explained by each reduced variable, along with the proportion of variance that is explained by each reduced variable and the cumulative proportion of explained variance.

Iteration History

When you specify the METHOD=LPCA option in the PROC NOMINALDR statement, the "Iteration History" table displays the iteration number, the number of reduced variables, the cumulative proportion of explained variance by the reduced variables, and the loss value for each iteration.

OutputCasTables Table

The OutputCasTables table is a special table that has information about each table that is created. The information for each table consists of the table name, the library in which the table resides, and the number of columns and rows in the table. Because this table is not a typical ODS table that contains analytical results, you cannot include it in the *table-spec-list* in the DISPLAYOUT statement.

ODS Table Names

Each table that PROC NOMINALDR creates has a name associated with it. You must use this name to refer to the table when you use ODS statements. The name of each table and a short description of the contents are listed in Table 25.2.

Table 25.2 ODS Tables Produced by PROC NOMINALDR

Table Name	Description	Statement	Option
IterHistory	Iteration history	PROC NOMINALDR	METHOD=LPCA
ModelInfo	Model information	PROC NOMINALDR	Default
NObs	Number of observations	PROC NOMINALDR	Default
OutputCasTables	See the section "OutputCasTables Table" on page 679	PROC NOMINALDR	Default
Variance	Explained variance summary	PROC NOMINALDR	Default
VarInfo	Number of levels for each nominal variable	PROC NOMINALDR	Default

Output Data Tables

PROC NOMINALDR creates one data table to which it writes the reduced variables for all input observations. The table's dimensions are determined by the number of observations that are read from the input data table and the reduced dimension that you specify in the DIMENSION= option. You specify the names of this data table in the OUTPUT statement.

Information about the output data table is displayed in the section "OutputCasTables Table" on page 679.

Example: NOMINALDR Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

Example 25.1: Dimensionality Reduction with PROC NOMINALDR for Classification

This example illustrates how you can use the NOMINALDR procedure as a preprocessing step to reduce the dimensionality of nominal data for downstream analysis. By transforming high-dimensional nominal data into a lower-dimensional representation, PROC NOMINALDR helps improve computational efficiency while retaining essential patterns in the data. Using a logistic regression model, the example shows that applying this procedure significantly reduces model training time while maintaining comparable accuracy.

The molecularBiology data set, which is derived from molecular biology research, is used in this example. The data set is available from the UCI Machine Learning Repository (1991). You can get the data set from the website at **https://archive.ics.uci.edu/dataset/69/molecular+biology+splice+junction+gene+sequences**.[2] The data set is split into 80% training and 20% testing sets, which are stored as the comma-separated-value (CSV) files *molecularBiologyTrain.csv* and *molecularBiologyTest.csv*, respectively, in the working directory. The following DATA step loads these data sets into your SAS library as Train and Test, respectively:

```
proc casutil;
    load file="molecularBiologyTrain.csv" /*or other user-defined location*/
    casout="Train" importoptions=(filetype="CSV" getnames="TRUE");
    load file="molecularBiologyTest.csv" /*or other user-defined location*/
    casout="Test" importoptions=(filetype="CSV" getnames="TRUE");
run;
```

The Train and Test data tables consist of 60 nominal input variables (Base1, Base2, . . . , Base60) and one nominal target variable (class).

In the following statements, PROC NOMINALDR is first applied to the Train data table to reduce the 60 nominal input variables to 10, as specified by the DIMENSION= option. The METHOD= option specifies that the MCA method is to be used in the procedure. The trained model, which is saved as mcaSTORE, is then applied to score the Test data table by using the ASTORE procedure. By default, the statements assume that your libref is named mylib, but you can replace it with any appropriately defined SAS libref.

```
proc NOMINALDR data=mylib.Train dimension=10 method=MCA;
    input Base1-Base60 /LEVEL=NOMINAL;
    output out=mylib.mcaTrain copyVars=class;
    savestate RSTORE=mylib.mcaSTORE;
run;
proc astore;
    score data=mylib.Test rstore=mylib.mcaSTORE
        out=mylib.mcaTest copyVars=class;
quit;
```

[2]Disclaimer: SAS may reference other websites or content or resources for use at Customer's sole discretion. SAS has no control over any websites or resources that are provided by companies or persons other than SAS. Customer acknowledges and agrees that SAS is not responsible for the availability or use of any such external sites or resources, and does not endorse any advertising, products, or other materials on or available from such websites or resources. Customer acknowledges and agrees that SAS is not liable for any loss or damage that may be incurred by Customer or its end users as a result of the availability or use of those external sites or resources, or as a result of any reliance placed by Customer or its end users on the completeness, accuracy, or existence of any advertising, products, or other materials on, or available from, such websites or resources.

Upon completion, the output data tables mcaTrain and mcaTest are generated. They contain the reduced variables along with the target variable (class), as specified by the COPYVARS= option in the OUTPUT statement in PROC NOMINALDR and the SCORE statement in PROC ASTORE.

Similarly, you apply the LPCA method to the Train data table as specified by the METHOD= option and apply the trained LPCA model to score the Test data table, as in the following statements:

```
proc NOMINALDR data=mylib.Train dimension=10 method=LPCA m=1;
    input Base1-Base60 /LEVEL=NOMINAL;
    output out=mylib.lpcaTrain copyVars=class;
    savestate RSTORE=mylib.lpcaSTORE;
run;

proc astore;
    score data=mylib.Test rstore=mylib.lpcaSTORE
        out=mylib.lpcaTest copyVars=class;
quit;
```

The output data tables lpcaTrain and lpcaTest contain the variables that are reduced by the LPCA method, along with the target variable (class).

Logistic regression models are trained on both the original and reduced-dimension data tables to compare their classification performance.

The following LOGISTIC procedure statements train a logistic regression model by using the original nominal data table Train and then evaluate it by using the Test data table. The error rate for the Test data table is shown in Output 25.1.1.

```
proc logistic data=mylib.Train;
    class Base1-Base60;
    model class= Base1-Base60 / LINK=GLOGIT;
    score data=mylib.Test fitstat;
    ods output ScoreFitStat=ScoreFitStat;
run;

proc print data=ScoreFitStat(keep=MisClass) NOOBS LABEL;
run;
```

Output 25.1.1 Error Rate for Model Trained on Original Nominal Data

Error Rate
0.0787

The following statements use PROC LOGISTIC to train a logistic regression model by using the reduced-dimension data table mcaTrain and then evaluate it by using the mcaTest data table. The error rate for the mcaTest data table is shown in Output 25.1.2.

```
proc logistic data=mylib.mcaTrain;
    model class=rv1-rv10 / LINK=GLOGIT;
    score data=mylib.mcaTest fitstat;
    ods output ScoreFitStat=ScoreFitStatRVMCA;
run;
```

```
proc print data=ScoreFitStatRVMCA(keep=MisClass) NOOBS LABEL;
run;
```

Output 25.1.2 Error Rate for Model Trained on Reduced-Dimension Data by the MCA Method

Error Rate
0.0677

Another logistic regression model is trained by using the reduced-dimension data table lpcaTrain and then evaluated by using the lpcaTest data table, as in the following PROC LOGISTIC statements. The error rate for the lpcaTest data table is shown in Output 25.1.3.

```
proc logistic data=mylib.lpcaTrain;
    model class=rv1-rv10 / LINK=GLOGIT;
    score data=mylib.lpcaTest fitstat;
    ods output ScoreFitStat=ScoreFitStatRVLPCA;
run;

proc print data=ScoreFitStatRVLPCA(keep=MisClass) NOOBS LABEL;
run;
```

Output 25.1.3 Error Rate for Model Trained on Reduced-Dimension Data by the LPCA Method

Error Rate
0.0551

The original data table contains 60 nominal input variables, and the logistic regression model that is trained on this data table achieves an error rate of 0.0787 on the test set. In contrast, the reduced-dimension data tables, each of which contains only 10 variables, yield lower error rates: 0.0677 when you use the MCA method and 0.0551 when you use the LPCA method. This improvement suggests that eliminating unimportant nominal variables or levels can enhance classification accuracy, because some of them might degrade model performance. The reduced-dimension data table that the LPCA method produces achieves better classification performance (that is, a lower error rate) than the one that the MCA method produces. This might be attributed to the iterative nature of LPCA, which used three iterations on the molecularBiology data set, whereas MCA is noniterative. The iterations enable LPCA to capture more informative structure in the data. As a result, the MCA method is faster, requiring only 0.60 seconds; the LPCA method requires 0.99 seconds.

Regarding training time for the logistic regression models, the training on the original data table, which has 60 nominal input variables, takes 31.65 seconds. In contrast, training on the reduced-dimension data tables, each of which has only 10 variables, is significantly faster: 0.70 seconds for the data table that was reduced by the MCA method, and 0.71 seconds for the data table that was reduced by the LPCA method.

These results demonstrate that both MCA and LPCA can significantly enhance the efficiency of subsequent analysis. This makes each of them a valuable preprocessing technique for working with high-dimensional nominal data, despite a possible minor trade-off in predictive accuracy.

References

De Leeuw, J. (2006). "Principal Component Analysis of Binary Data by Iterated Singular Value Decomposition." *Computational Statistics and Data Analysis* 50:21–39.

Landgraf, A. J., and Lee, Y. (2020). "Dimensionality Reduction for Binary Data through the Projection of Natural Parameters." *Journal of Multivariate Analysis* 180:104668.

Schein, A. I., Saul, L. K., and Ungar, L. H. (2003). "A Generalized Linear Model for Principal Component Analysis of Binary Data." In *Proceedings of the Ninth International Workshop on Artificial Intelligence and Statistics*, 240–247. PMLR. http://proceedings.mlr.press/r4/schein03a/schein03a.pdf.

UCI Machine Learning Repository (1991). "Molecular Biology (Splice-Junction Gene Sequences)." UCI MLR. https://doi.org/10.24432/C5M888.

Chapter 26
PARTIALDEPEND Procedure

Contents

Overview: PARTIALDEPEND Procedure **686**
- PROC PARTIALDEPEND Features 686
- Using SAS Viya Workbench 687

Getting Started: PARTIALDEPEND Procedure **687**

Syntax: PARTIALDEPEND Procedure **689**
- PROC PARTIALDEPEND Statement 690
- ANALYSISVARIABLE Statement 691
- ASTOREMODEL Statement 692
- CODE Statement 692
- DISPLAY Statement 693
- DISPLAYOUT Statement 694
- FREQ Statement 694
- INPUT Statement 695
- OUTPUT Statement 695
- PREDICTEDTARGET Statement 696
- WEIGHT Statement 696

Details: PARTIALDEPEND Procedure **696**
- Required Specifications 696
- Two-Way PD Functions 697
- Performance 697
- Bin and Level Counts 697
- Missing Values 697
- Frequency and Weight Variables 698
- Displayed Output 698
- ODS Table Names 698
- ODS Graphics 699

Examples: PARTIALDEPEND Procedure **699**
- Example 26.1: Compute Partial Dependence for an Interval Target Variable 699
- Example 26.2: Compute Partial Dependence for a Nominal Target Variable 701
- Example 26.3: Compute a Two-Way Partial Dependence Function 703

Overview: PARTIALDEPEND Procedure

The PARTIALDEPEND procedure computes partial dependence (PD) functions for machine learning models that are built using algorithms such as gradient boosting, forest, and neural networks and for many other types of models that are created using SAS software. You can compute partial dependence functions for any model that produces an analytic store or DATA step code.

For a given model, PD values represent the model's average prediction for each value of a particular input variable. Thus, PD plots show how changes to the model's input values affect the model's predictions. For example, a PD plot can show whether predicted automobile price increases linearly with horsepower or whether the relationship is of another type, such as a step function, curvilinear, and so on. You can also use PD plots to show interactions between two model inputs and to evaluate competing machine learning models. PROC PARTIALDEPEND produces plots of partial dependence only if you specify just one analysis variable. You can plot the output from a two-way PD by using SAS graphics procedures like PROC SGPLOT.

PROC PARTIALDEPEND also produces individual conditional expectation (ICE) tables. Whereas a partial dependence function represents the average model prediction, an ICE table represents the variation of individual model predictions around the overall average. ICE curves therefore help you identify interactions between model variables as well as identify subgroups that diverge from the overall average.

PROC PARTIALDEPEND Features

PROC PARTIALDEPEND provides the following features:

- computes partial dependence (PD) for machine learning models that produce an analytic store or DATA step code
- produces a PD plot via ODS Graphics if you specify only one analysis variable
- computes two-way PD values if you specify two analysis variables
- provides an OUTPUT statement to produce a data table that contains individual conditional expectation (ICE) values
- provides a FREQ statement for grouped analysis
- provides a WEIGHT statement for weighted analysis

Using SAS Viya Workbench

SAS Viya Workbench has greatly simplified the SAS programming experience by offering a syntax style nearly identical to that of SAS 9 and earlier releases. However, there are minor exceptions. In the past, SAS Viya procedures required the use of a CAS engine libref. This engine served multiple purposes, but it was primarily used to upload data to the CAS server and manage the data. Its additional purposes included support of new data types and improved performance for large data sets. No CAS server is needed for SAS Viya Workbench, because all computations are done in place for you in a single location.

In the following program, the LIBNAME statement creates a mylib libref that the examples in this chapter use to refer to your current or desired work directory:

```
libname mylib ".";
```

For more information, see *SAS Viya Workbench: Getting Started*.

Getting Started: PARTIALDEPEND Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

This example illustrates some of the basic features of the PARTIALDEPEND procedure by analyzing a gradient boosting model that is trained on the cars data table.

The following DATA step creates the input data table mylib.cars:

```
data mylib.cars;
    set sashelp.cars;
run;
```

The following statements run the GRADBOOST procedure to build a gradient boosting model to predict city gas mileage for each automobile in the mylib.cars data table. The procedure also outputs an analytic store named gbStore_cars.

```
proc gradboost data = mylib.cars seed = 12345;
   input Cylinders Horsepower MSRP / level = interval;
   input Origin Type / level = nominal;
   target MPG_City / level = interval;
   saveState rstore = mylib.gbStore_cars;
run;
```

The following statements run PROC PARTIALDEPEND and produce a PD plot with ODS Graphics enabled:

```
ods graphics on; /* Enables ODS Graphics */
ods output PartialDependence = pd_table; /* Saves the ODS table */

proc PartialDepend data = mylib.cars seed = 12345;
   input Cylinders Horsepower MSRP / level = interval;
   input Origin Type / level = nominal;
   predictedTarget P_MPG_City;
```

```
    analysisVariable Type;
    astoreModel rstore = mylib.gbStore_cars;
run;
```

The DATA= option specifies the data table to use for PD analysis. The SEED= option specifies the seed to use for pseudorandom number generation. The two INPUT statements specify that the Cylinders, Horsepower, MSRP, Origin, and Type variables be used as inputs. The PREDICTEDTARGET statement specifies that the P_MPG_City variable be used as the predicted target. The ANALYSISVARIABLE statement specifies that the Type variable be used as the analysis variable. The ASTOREMODEL statement with the RSTORE= option specifies that the analytic store saved in the data table mylib.gbStore_cars be used as the model for analysis.

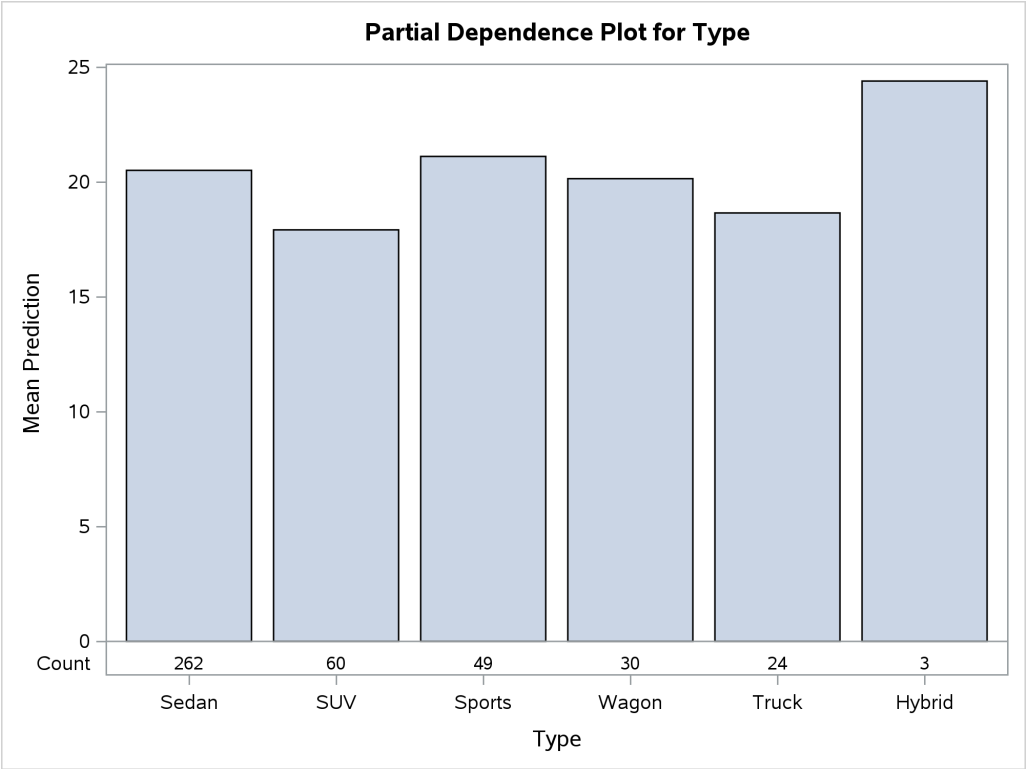
Figure 26.1 displays the average predicted city mileage by each level of the Type variable. Figure 26.2 displays the average predicted city mileage by Type level as a plot. It also displays the number of observations for each level of the analysis variable in the sampled data (labeled as "Count") at the bottom of the plot.

Figure 26.1 Partial Dependence Table

The PARTIALDEPEND Procedure

Partial Dependence				
Bin (Type)	Type	Mean Prediction (P_MPG_City)	Standard Error (P_MPG_City)	Count
1	Sedan	20.50570829	0.1910949728	262
2	SUV	17.914748769	0.2014259473	60
3	Sports	21.112282551	0.2227940166	49
4	Wagon	20.143071169	0.2220553717	30
5	Truck	18.651004329	0.2489319097	24
6	Hybrid	24.392705139	0.3452386778	3

Figure 26.2 Partial Dependence Plot



Syntax: PARTIALDEPEND Procedure

The following statements are available in the PARTIALDEPEND procedure:

PROC PARTIALDEPEND < *options* > **;**
  ANALYSISVARIABLE *variable* < / *options* > **;**
  ASTOREMODEL RSTORE=*libref.data-table* **;**
  CODE FILE=*filename* | **TABLE=***libref.data-table* **;**
  DISPLAY < *table-list* > < / *options* > **;**
  DISPLAYOUT *table-spec-list* < / *options* > **;**
  FREQ *variable* **;**
  INPUT *variables* < / **LEVEL=INTERVAL** | **NOMINAL** > **;**
  OUTPUT OUT=*libref.data-table* < *options* > **;**
  PREDICTEDTARGET *variable* **;**
  WEIGHT *variable* **;**

The PROC PARTIALDEPEND, ANALYSISVARIABLE, INPUT, and PREDICTEDTARGET statements are required. At least one ASTOREMODEL or CODE statement is required. You can specify up to two ANALYSISVARIABLE statements. The ASTOREMODEL and INPUT statements can appear multiple times. The rest of this section provides detailed syntax information about each of the preceding statements, beginning with the PROC PARTIALDEPEND statement. The remaining statements are described in alphabetical order.

PROC PARTIALDEPEND Statement

PROC PARTIALDEPEND < *options* > **;**

The PROC PARTIALDEPEND statement invokes the procedure. Table 26.1 summarizes the options in this statement.

Table 26.1 PROC PARTIALDEPEND Statement Options

Option	Description
Basic Options	
DATA=	Specifies the input data table
LOGLEVEL=	Specifies a value to control the quantity and type of notes to print to the client log
NTHREADS=	Specifies the number of threads to use in the computation
REPLICATETYPE=	Specifies whether to replicate bin cutpoints or midpoints for an interval analysis variable
SAMPLESIZE=	Specifies the approximate number of observations to sample from the input data table
SEED=	Specifies the seed value for pseudorandom number generation

You can specify the following *options*:

DATA=*libref.data-table*

names the input data table for PROC PARTIALDEPEND to use. The default is the most recently created data table. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to the data. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 687.

data-table specifies the name of the input data table.

LOGLEVEL=*number*

controls the quantity and type of notes to print to the client log. A value of 0 prints only warnings and errors; a value of 1 prints some notes; and a value of 2 prints many notes.

By default, LOGLEVEL=1.

NTHREADS=*number*

specifies the number of threads to use in the computation. The default value is the number of CPUs available on the machine.

REPLICATETYPE=CUTPOINTS | MIDPOINTS

specifies whether to replicate bin cutpoints or midpoints for an interval analysis variable.

By default, REPLICATETYPE=CUTPOINTS.

SAMPLESIZE=*number*
specifies the approximate number of observations to sample from the input data table. The value of *number* must be between 100 and 5000, inclusive.

By default, SAMPLESIZE=1000.

SEED=*number*
specifies the seed value for pseudorandom number generation. If you do not specify a seed or you specify a value less than or equal to 0, the seed is generated by reading the time of day from the computer's clock.

ANALYSISVARIABLE Statement

ANALYSISVARIABLE *variable* < / *options* > **;**

The ANALYSISVARIABLE statement names the analysis variable for which to compute the partial dependence. You can specify one or two ANALYSISVARIABLE statements. If you specify only one statement, PROC PARTIALDEPEND produces a PD plot of the analysis variable. If you specify two statements, the procedure computes the two-way PD for the two analysis variables and does not produce a PD plot.

You can specify the following *options*:

LEVELINDEXVARIABLE=*variable*
specifies a variable that defines how to order the levels that are read from the nominal levels table.

LEVELSTABLE=*libref.data-table*
specifies a table of levels of the nominal analysis variable(s). *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 687.

MAX=*number*
specifies the maximum value of an interval analysis variable.

MAXLEVELS=*number*
specifies the maximum number of levels to use for a nominal analysis variable. The value of *number* must be between 2 and 100, inclusive.

By default, MAXLEVELS=10.

MIN=*number*
specifies the minimum value of an interval analysis variable.

NBINS=*number*
specifies the number of bins to use for an interval analysis variable. The value of *number* must be between 2 and 100, inclusive.

By default, NBINS=10.

REPORTMISSING
reports missing values of the analysis variable.

ASTOREMODEL Statement

ASTOREMODEL RSTORE=*libref.data-table* **;**

The ASTOREMODEL statement names an analytic store to be explained. If you specify multiple ASTOREMODEL statements, you must also specify the CODE statement to name the DS2 code that is used along with the analytic stores. For more information about DS2 code and using DS2 code with analytic stores, see *SAS DS2 Language Reference*, *SAS DS2 Programmer's Guide*, and Chapter 4, "ASTORE Procedure."

You must specify the following option:

RSTORE=*libref.data-table*
specifies the analytic store that contains the model to explain. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 687.

CODE Statement

CODE FILE=*filename* | **TABLE=***libref.data-table* **;**

The CODE statement names the DATA step code that describes the model or the DS2 code that is used along with an analytic store that you specify in the ASTOREMODEL statement. If you specify multiple ASTOREMODEL statements, you must also specify the CODE statement to name the DS2 code that is used along with the analytic stores. For more information about DS2 code and using DS2 code with analytic stores, see *SAS DS2 Language Reference*, *SAS DS2 Programmer's Guide*, and Chapter 4, "ASTORE Procedure."

You must specify one of the following options:

FILE=*filename*
names the external file that contains the DATA step code that describes the model or the DS2 code that is used along with an analytic store that you specify in the ASTOREMODEL statement. When you enclose the *filename* in a quoted string (for example, FILE="c:\mydir\scorecode.sas"), this option specifies the path and file name for reading the code from an external file. You can also specify an unquoted *filename* of no more than eight characters. If the *filename* is assigned as a fileref in a Base SAS FILENAME statement, the file that is specified in the FILENAME statement is used.

TABLE=*libref.data-table*
names the data table that contains the DATA step code that describes the model or the DS2 code that is used along with an analytic store that you specify in the ASTOREMODEL statement. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 687.

DISPLAY Statement

DISPLAY < *table-list* > < / *options* > **;**

The DISPLAY statement enables you to specify a list of display tables to display or exclude. This statement is similar to the ODS SELECT, ODS EXCLUDE, and ODS TRACE statements. However, the DISPLAY statement can improve performance when a large number of tables could be generated (such as in BY-group processing). The DISPLAY statement produces only the requested tables, but ODS statements generate all the tables and then keep the requested tables.

If you use both DISPLAY and ODS statements together, the DISPLAY statement takes precedence over the ODS statements. Note that the ODS EXCLUDE statement processes tables after they have been filtered by the DISPLAY statement. In some cases, it might appear that the ODS EXCLUDE statement is taking precedence because it can further filter the tables. For more information about ODS, see *SAS Output Delivery System: Procedures Guide*.

You can specify the *table-list* as a list of table names, paths, partial pathnames, and regular expressions.

The table names that you can specify are listed in the section "ODS Table Names" on page 698. A path is a table name that is prefixed with dot-separated grouping information. For example, a SelectionSummary table that a procedure produces during a selection routine might have the path *Bygroup1.Summary.SelectionSummary*. A partial pathname does not include all groups; for example, *SelectionSummary* and *Summary.SelectionSummary* are partial pathnames for *Bygroup1.Summary.SelectionSummary*.

When you specify a table name or partial pathname, all display tables whose paths end in the specified name are selected for display or exclusion. For example, both *SelectionSummary* and *Summary.SelectionSummary* select *Bygroup1.Summary.SelectionSummary*.

A regular expression is enclosed in forward slashes (/). For example, specifying "/tions/" selects all pathnames that contain the substring "tions"; in particular, the *Bygroup1.Summary.SelectionSummary* table is selected. Specifying "!/tions/" selects all pathnames that do not contain the substring "tions"; in particular, the *Bygroup1.Summary.SelectionSummary* table is not selected.

You can specify the following *options* after a slash (/):

CASESENSITIVE
performs a case-sensitive comparison of table names in the *table-list* to display table names when tables are subsetted for display. To preserve case, you must enclose table names in the *table-list* in quotation marks.

EXCLUDE
displays all display tables except those that you specify in the *table-list*.

EXCLUDEALL
suppresses display of all tables. This option takes precedence over the other options.

TRACE
displays the display table names, labels, and paths.

DISPLAYOUT Statement

DISPLAYOUT *table-spec-list* < / *options* > **;**

The DISPLAYOUT statement enables you to create output tables from your displayed output. This statement is similar to the ODS OUTPUT statement. For more information about ODS, see *SAS Output Delivery System: Procedures Guide*.

The *table-spec-list* specifies a list of output tables to create. Each entry in the list has either a *key*=*value* format or a *key* format:

key=*value* specifies *key* as the ODS table name, path, or partial pathname, and specifies *value* as the output table name.

key specifies *key* as the ODS table name and also as the output table name.

The ODS table names that you can specify are listed in the section "ODS Table Names" on page 698. You cannot specify the ODS table named OutputCasTables in the *table-spec-list*.

Table names and partial pathnames are discussed under the DISPLAY statement. The DISPLAYOUT statement does not support regular expressions.

You can specify the following *options* after a slash (/):

INCLUDEALL
creates output tables for all display tables. The name of the created output table is the same as the name of the corresponding display table. If you specify this option, the *table-spec-list* specification is ignored.

NOREPLACE
does not replace any existing output table of the same name.

FREQ Statement

FREQ *variable* **;**

The *variable* in the FREQ statement identifies a numeric variable in the input data table that contains the frequency of occurrence of each observation. PROC PARTIALDEPEND treats each observation as if it appears f times, where f is the value of the FREQ *variable* for the observation. If f is not an integer, it is truncated to an integer. If f is less than 1 or missing, the observation is not used in the analysis. When you omit the FREQ statement, each observation is assigned a frequency of 1.

INPUT Statement

INPUT *variables* < / **LEVEL=INTERVAL | NOMINAL** > **;**

The INPUT statement specifies one or more *variables* as input variables. You can specify the INPUT statement multiple times.

You can specify the following option:

LEVEL=INTERVAL | NOMINAL
specifies the type of all the variables in the statement. You can specify the following values:

INTERVAL treats all the variables in the INPUT statement as interval variables.

NOMINAL treats all the variables in the statement as nominal variables.

By default, LEVEL=INTERVAL for numeric variables and LEVEL=NOMINAL for character variables.

OUTPUT Statement

OUTPUT OUT=*libref.output-table* < *options* > **;**

The OUTPUT statement creates an output data table that contains the individual conditional expectation (ICE) values.

You must specify the following option:

OUT=*libref.data-table*
names the output data table for PROC PARTIALDEPEND to use. You must specify this option before any other options. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to where the data table is to be stored. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 687.

data-table specifies the name of the output data table.

You can also specify the following *option*:

COPYVAR=*variable*

COPYVARS=(*variables***)**
lists one or more *variables* from the input data table to be transferred to the output data table.

PREDICTEDTARGET Statement

PREDICTEDTARGET *variable* **;**

The PREDICTEDTARGET statement names the variable that contains the model's predictions. The specified *variable* typically has a P_ prefix.

WEIGHT Statement

WEIGHT *variable* **;**

The *variable* in the WEIGHT statement is used as a weight to perform a weighted analysis of the data. Observations that have nonpositive or missing weights are not included in the analysis. The values of the WEIGHT variable can be nonintegral and are not truncated. When you omit the WEIGHT statement, all observations that are used in the analysis are assigned a weight of 1.

Details: PARTIALDEPEND Procedure

Required Specifications

The following specifications are required in order to run the PARTIALDEPEND procedure:

- An input data table. Typically, this is the data table that is used to train the model.
- A list of input variables. These are the model inputs that are used to train the model. Inputs can be interval or nominal.
- One or two analysis variables. These are the input variables that you want to analyze. The analysis variables can be interval or nominal, but both must be model inputs.
- A scoring function. You must provide either an analytic store, several analytic stores and DS2 code, or DATA step scoring code that is created by the modeling algorithm.
- A predicted target variable. This is the output variable that contains the model's predictions when the scoring function is applied to the input data table. The values of the variable can represent numeric predictions or predicted probabilities for an interval or nominal target, respectively. In many SAS Viya machine learning models, the name of the variable is constructed as follows: for an interval target, "P_" is prefixed to the name of the target variable (for example, "P_MSRP"); for a nominal target, "P_" is prefixed to the name of the target variable, and the value that corresponds to the target event is appended (for example, "P_Bad1").

Two-Way PD Functions

If you specify a single analysis variable, PROC PARTIALDEPEND computes the relationship between the values of that model input and the model's predicted values. If you specify two analysis variables, the partial dependence (PD) function shows how the model's predictions change as the values of one or both variables vary. Thus, two-way PD functions can help identify and characterize interactions between model inputs.

Performance

Traditional PD functions use all observations in the training table and plot each unique value (or level) of the analysis variable. However, this approach is impractical for large data tables and high-cardinality analysis variables, because the number of scoring operations that are required in order to compute the PD function grows with both the number of observations and number of values of the analysis variables. Thus, to ensure reasonable performance, PROC PARTIALDEPEND samples observations from the input data and bins values of interval analysis variables before computing the PD function. For nominal analysis variables, only the most frequent levels are used by default. You can specify the number of samples (up to a limit), the number of bins, and the maximum number of nominal levels to keep. Decreasing any of these values might reduce computation time by reducing the number of scoring operations, whereas increasing any of them increases the fidelity of the PD function that the procedure produces.

Bin and Level Counts

Output from PROC PARTIALDEPEND always includes level counts for analyses that involve nominal analysis variables only—a single nominal analysis variable or two nominal analysis variables.

For analyses that involve interval analysis variables or a combination of interval and nominal analysis variables, the REPLICATETYPE= option controls whether counts are displayed. To display counts for such analyses, specify REPLICATETYPE=MIDPOINTS in the PROC PARTIALDEPEND statement.

When you specify the FREQ statement, the procedure calculates the counts by using the frequency variable. When you specify the WEIGHT statement, the procedure generates a weighted count column. Calculation of the weighted counts uses both the weight variable and the frequency variable, if both are specified.

Missing Values

By default, the PARTIALDEPEND procedure excludes missing values of the analysis variables when computing the PD function. You can use the REPORTMISSING option in the PROC PARTIALDEPEND statement to control whether to include missing values of the analysis variables in the output.

How the procedure handles missing values of the other model input variables depends on the modeling algorithm that it uses to generate the scoring function. For example, for tree-based algorithms, missing input values might be treated as a valid level. However, other modeling algorithms, such as neural networks, might exclude observations that contain missing values, and such observations would therefore be excluded when the PD function is computed.

Frequency and Weight Variables

The FREQ statement names a numeric variable that provides the frequency of each observation in the input data set. If you do not specify a frequency variable, then PROC PARTIALDEPEND assumes that the frequency of all observations is 1. The procedure expects nonnegative integer frequency values. If the frequency of an observation is not an integer, the procedure truncates the frequency value to the largest integer that is less than or equal to the value. The procedure treats negative and missing frequencies as 0.

The WEIGHT statement names a numeric variable that provides the weight of each observation in the input data set. If you do not specify a weight variable, then PROC PARTIALDEPEND assumes that the weight of all observations is 1. The procedure expects nonnegative weight values. It treats negative and missing weights as 0.

PROC PARTIALDEPEND uses both the frequency and weight variables to calculate the mean prediction and the standard error. The procedure uses the sum of weights minus 1 (WDF) as the variance divisor to calculate the standard error.

Displayed Output

The following sections describe the output that PROC PARTIALDEPEND produces.

Partial Dependence

The "Partial Dependence" table contains the partial dependence values and corresponding standard errors of the analysis variable(s). This table can also include bin and level count information, which is explained in the section "Bin and Level Counts" on page 697. This table is produced by default.

OutputCasTables Table

The OutputCasTables table is a special table that has information about each table that is created. The information for each table consists of the table name, the library in which the table resides, and the number of columns and rows in the table.

ODS Table Names

Each table that PROC PARTIALDEPEND creates has a name that is associated with it. You must use this name to refer to the table when you use ODS statements. The name of each table and a short description of the contents are listed in Table 26.2.

Table 26.2 ODS Tables Produced by PROC PARTIALDEPEND

Table Name	Description	Statement	Option
OutputCasTables	See the section "OutputCasTables Table" on page 698	OUTPUT	Default
PartialDependence	Partial dependence for analysis variable(s)	PROC PARTIALDEPEND	Default

ODS Graphics

You can refer to every graph that is produced through ODS Graphics by name. The name of the graph that PROC PARTIALDEPEND generates is listed in Table 26.3. The PDPlot graph is produced when only one analysis variable is specified.

Table 26.3 Graph Produced by PROC PARTIALDEPEND

ODS Graph Name	Plot Description
PDPlot	Partial dependence plot for the analysis variable

Examples: PARTIALDEPEND Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

Example 26.1: Compute Partial Dependence for an Interval Target Variable

This example shows how to use the PARTIALDEPEND procedure to estimate the partial dependence (PD) function for an interval target variable.

The following DATA step creates the input data table mylib.cars:

```
data mylib.cars;
    set sashelp.cars;
run;
```

The following statements run the GRADBOOST procedure to build a gradient boosting model to predict city gas mileage for each automobile in the mylib.cars data table. The procedure also outputs an analytic store named gbStore_cars.

```
proc gradboost data = mylib.cars seed = 12345;
   input Cylinders Horsepower MSRP / level = interval;
   input Origin Type / level = nominal;
   target MPG_City / level = interval;
   saveState rstore = mylib.gbStore_cars;
run;
```

The following statements run PROC PARTIALDEPEND and produce the PD plot with ODS Graphics enabled:

```
ods graphics on; /* Enables ODS Graphics */

proc PartialDepend data = mylib.cars replicateType = MIDPOINTS seed = 12345;
   input Cylinders Horsepower MSRP / level = interval;
   input Origin Type / level = nominal;
   predictedTarget P_MPG_City;
   analysisVariable Horsepower;
   astoreModel rstore = mylib.gbStore_cars;
run;
```

The DATA= option specifies the data table to use for PD analysis. The REPLICATETYPE= option replicates bin midpoints for the analysis variable. The SEED= option specifies the seed to use for pseudorandom number generation. The two INPUT statements specify that the Cylinders, Horsepower, MSRP, Origin, and Type variables be used as inputs. The PREDICTEDTARGET statement specifies that the P_MPG_City variable be used as the predicted target. The ANALYSISVARIABLE statement specifies that the Horsepower variable be used as the analysis variable. The ASTOREMODEL statement with the RSTORE= option specifies that the analytic store saved in the data table mylib.gbStore_cars be used as the model for analysis.

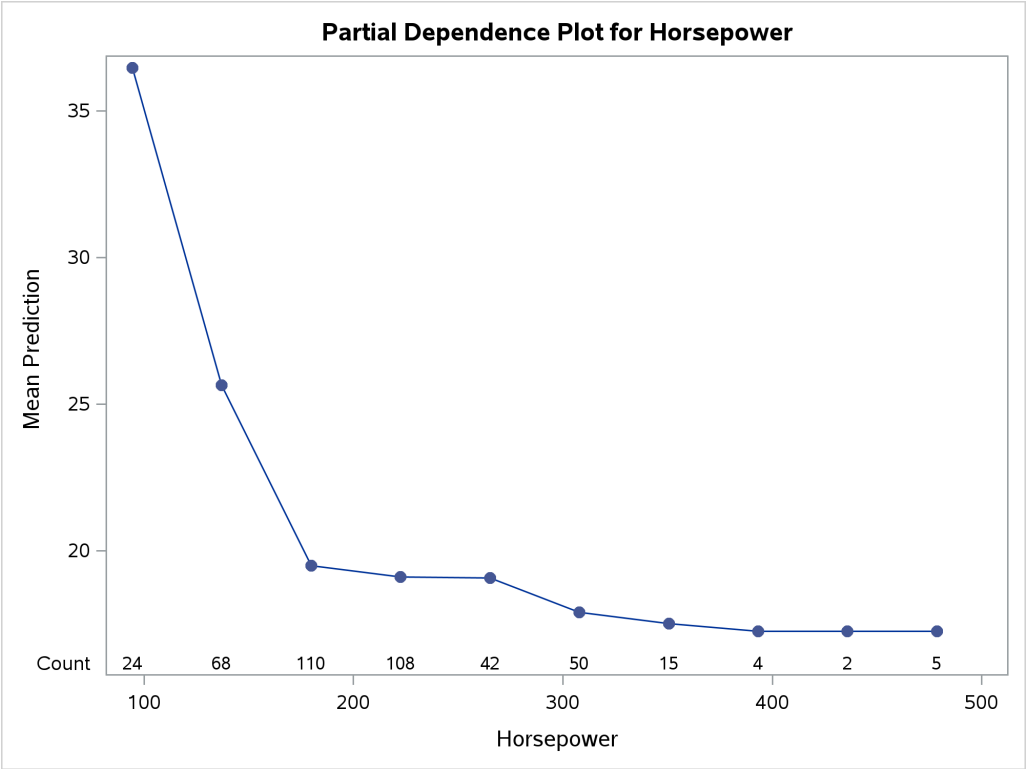
Output 26.1.1 displays the average predicted city mileage by each binned Horsepower value. Output 26.1.2 displays the average predicted city mileage by Horsepower as a plot. It also displays the number of observations for each bin of the analysis variable in the sampled data (labeled as "Count") at the bottom of the plot.

Output 26.1.1 Partial Dependence Table

The PARTIALDEPEND Procedure

	Partial Dependence			
Bin (Horsepower)	Horsepower	Mean Prediction (P_MPG_City)	Standard Error (P_MPG_City)	Count
1	94.35	36.4692088	0.1233019178	24
2	137.05	25.644018663	0.1181638873	68
3	179.75	19.489612639	0.1110720415	110
4	222.45	19.10767521	0.1118359726	108
5	265.15	19.079890007	0.1103457652	42
6	307.85	17.900404766	0.1099332326	50
7	350.55	17.518976898	0.1136050818	15
8	393.25	17.256772898	0.1199480112	4
9	435.95	17.256772898	0.1199480112	2
10	478.65	17.256772898	0.1199480112	5

Output 26.1.2 Partial Dependence Plot



Example 26.2: Compute Partial Dependence for a Nominal Target Variable

This example shows how to use the PARTIALDEPEND procedure to estimate the partial dependence (PD) function for a nominal target variable. It also shows how to generate an ICE table.

The following DATA step creates the input data table mylib.hmeq. In this example, an observation ID variable, _ID_, is added to help index the ICE curves for individuals.

```
data mylib.hmeq;
    set sampsio.hmeq;
    _ID_ =_n_;
run;
```

The following statements run the FOREST procedure to build a forest model to estimate the probability that mortgage loan applicants will default on repayment of their loan. The procedure also outputs an analytic store named forestStore_hmeq.

```
proc forest data = mylib.hmeq seed = 1234;
   input DebtInc Value Loan Mortdue / level = interval;
   input Job Reason / level = nominal;
   target Bad / level = nominal;
   saveState rstore = mylib.forestStore_hmeq;
run;
```

Note that the target variable Bad, which has two numeric levels (0 and 1), is defined as a nominal variable. The following statements run PROC PARTIALDEPEND and produce the PD plot with ODS Graphics enabled:

```
ods graphics on; /* Enables ODS Graphics */

proc PartialDepend data = mylib.hmeq seed = 1234;
   input Job Reason / level = nominal;
   input DebtInc Value Loan Mortdue / level = interval;
   predictedTarget P_Bad1;
   analysisVariable Job;
   astoreModel rstore = mylib.forestStore_hmeq;
   output out = mylib.ICE_out copyvars = (_ID_);
run;
```

The DATA= option specifies the data table to use for PD analysis. The SEED= option specifies the seed to use for pseudorandom number generation. The two INPUT statements specify that the Job, Reason, DebtInc, Value, Loan, and Mortdue variables be used as inputs. The PREDICTEDTARGET statement specifies that the P_Bad1 variable be used as the predicted target. The ANALYSISVARIABLE statement specifies that the Job variable be used as the analysis variable. The ASTOREMODEL statement with the RSTORE= option specifies that the analytic store saved in the data table mylib.forestStore_hmeq be used as the model for analysis. The OUTPUT statement with the OUT= and COPYVARS= options specifies that the ICE values for each sampled individual be saved in the data table mylib.ICE_out, with the variable _ID_ copied to the output data table.

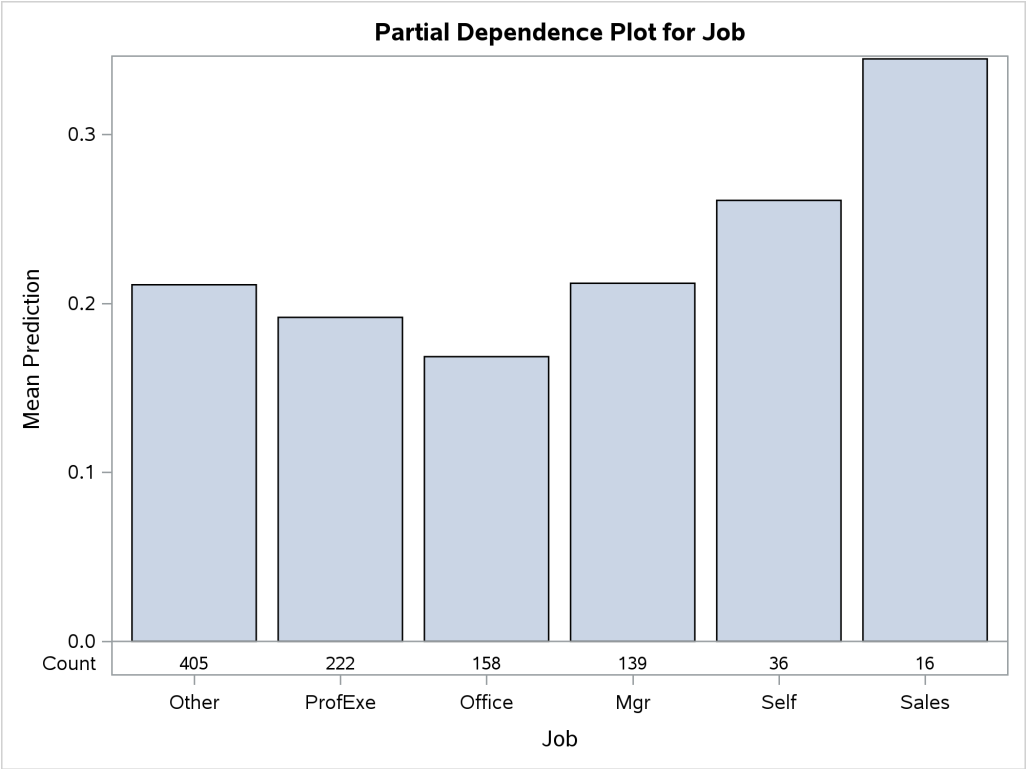
Output 26.2.1 displays the average predicted probability of loan default (BAD=1) by each Job level. Output 26.2.2 displays the average predicted probability of loan default by Job as a plot. It also displays the number of observations for each level of the analysis variable in the sampled data (labeled as "Count") at the bottom of the graph.

Output 26.2.1 Partial Dependence Table

The PARTIALDEPEND Procedure

		Partial Dependence		
Bin (Job)	JOB	Mean Prediction (P_Bad1)	Standard Error (P_Bad1)	Count
1	Other	0.210981092	0.0083166049	405
2	ProfExe	0.1916962576	0.0079161127	222
3	Office	0.1684452274	0.0068102209	158
4	Mgr	0.211865001	0.0083646163	139
5	Self	0.260902705	0.0092112979	36
6	Sales	0.3445961044	0.0078446872	16

Output 26.2.2 Partial Dependence Plot



Example 26.3: Compute a Two-Way Partial Dependence Function

This example shows how to use the PARTIALDEPEND procedure to estimate the partial dependence (PD) function for two analysis variables.

The following DATA step creates the input data table mylib.cars:

```
data mylib.cars;
    set sashelp.cars;
run;
```

The following statements run the GRADBOOST procedure to build a gradient boosting model to predict city gas mileage for each automobile in the mylib.cars data table. The procedure also outputs an analytic store named gbStore_cars.

```
proc gradboost data = mylib.cars seed = 12345;
   input Cylinders Horsepower MSRP / level = interval;
   input Origin Type / level = nominal;
   target MPG_City / level = interval;
   saveState rstore = mylib.gbStore_cars;
run;
```

The following statements run PROC PARTIALDEPEND and output the PD plot with ODS Graphics enabled:

```
ods graphics on; /* Enables ODS Graphics */

proc PartialDepend data = mylib.cars replicateType = MIDPOINTS seed = 12345;
   input Cylinders Horsepower MSRP / level = interval;
   input Origin Type / level = nominal;
   predictedTarget P_MPG_City;
   analysisVariable Type;
   analysisVariable Cylinders / min = 3 max = 9 nBins = 3;
   astoreModel rstore = mylib.gbStore_cars;
run;
```

The DATA= option specifies the data table to use for PD analysis. The REPLICATETYPE= option replicates bin midpoints for any numeric analysis variable. The SEED= option specifies the seed to use for pseudorandom number generation. The two INPUT statements specify that the Cylinders, Horsepower, MSRP, Origin, and Type variables be used as inputs. The PREDICTEDTARGET statement specifies that the P_MPG_City variable be used as the predicted target. The first ANALYSISVARIABLE statement specifies that the Type variable be used as an analysis variable. The second ANALYSISVARIABLE statement specifies that the Cylinders variable be used as the second analysis variable. Because Cylinders is an interval variable, its values are binned. The MIN= and MAX= options specify the range of values to bin. The NBINS= option requests three bins. The ASTOREMODEL statement with the RSTORE= option specifies that the analytic store saved in the data table mylib.gbStore_cars be used as the model for analysis.

Output 26.3.1 displays the average predicted mileage for each combination of values of the two analysis variables. It also displays the counts of observations for each combination of values in the sampled data table. Some counts are 0 because the specific combinations do not exist in the sampled training data.

Output 26.3.1 Partial Dependence Table

The PARTIALDEPEND Procedure

			Partial Dependence			
Bin (Type)	Bin2 (Cylinders)	Type	Cylinders	Mean Prediction (P_MPG_City)	Standard Error (P_MPG_City)	Count
1	1	Sedan	4	21.789041565	0.1542755927	96
1	2	Sedan	6	20.23581313	0.1617475483	126
1	3	Sedan	8	19.259887895	0.1745958163	40
2	1	SUV	4	19.472021089	0.1562924052	7
2	2	SUV	6	17.810132347	0.1711337882	30
2	3	SUV	8	16.320553188	0.1884769942	23
3	1	Sports	4	21.990765184	0.1983879824	11
3	2	Sports	6	21.004352955	0.2023398463	20
3	3	Sports	8	19.973304191	0.2147457093	16
4	1	Wagon	4	21.228180189	0.1884482201	14
4	2	Wagon	6	20.080857321	0.1950523097	12
4	3	Wagon	8	18.840263506	0.2130501856	4
5	1	Truck	4	20.754898963	0.1951855984	6
5	2	Truck	6	17.933089025	0.2085177833	9
5	3	Truck	8	17.211059707	0.2155960847	9
6	1	Hybrid	4	26.598580576	0.2859271912	3
6	2	Hybrid	6	23.693327973	0.2936767864	0
6	3	Hybrid	8	22.507343623	0.3084082644	0

Chapter 27
RECASSESS Procedure

Contents

Overview: RECASSESS Procedure **705**
PROC RECASSESS Features 706
Using SAS Viya Workbench 706
Getting Started: RECASSESS Procedure **706**
Syntax: RECASSESS Procedure **709**
PROC RECASSESS Statement 709
DISPLAY Statement 710
DISPLAYOUT Statement 711
INPUT Statement 712
RANK Statement 712
RECOMITEMID Statement 712
TESTITEMID Statement 712
USERID Statement 713
Details: RECASSESS Procedure **713**
Displayed Output 715
ODS Table Names 715

Overview: RECASSESS Procedure

The RECASSESS procedure calculates several different metrics to assess the quality of recommendations from a recommender system. You can use this procedure together with the RECENGINE procedure to access the quality of recommendation models that PROC RECASSESS generates.

PROC RECASSESS Features

The RECASSESS procedure has the following features:

- is multithreaded during all phases of analytic execution
- supports large-scale assessment data
- enables you to access recommendation system methods
- enables you to perform analysis on selected columns of data

Using SAS Viya Workbench

SAS Viya Workbench has greatly simplified the SAS programming experience by offering a syntax style nearly identical to that of SAS 9 and earlier releases. However, there are minor exceptions. In the past, SAS Viya procedures required the use of a CAS engine libref. This engine served multiple purposes, but it was primarily used to upload data to the CAS server and manage the data. Its additional purposes included support of new data types and improved performance for large data sets. No CAS server is needed for SAS Viya Workbench, because all computations are done in place for you in a single location.

In the following program, the LIBNAME statement creates a mylib libref that the examples in this chapter use to refer to your current or desired work directory:

```
libname mylib ".";
```

For more information, see *SAS Viya Workbench: Getting Started*.

Getting Started: RECASSESS Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

This example shows how to use the RECASSESS procedure to assess the quality of item recommendations to users.

You can construct the input table by training and scoring a recommender engine model, or you can generate recommendations by using your own method. This example first generates an input table for PROC RECASSESS by using the RECENGINE procedure to train a *k*-nearest neighbors recommender model. Then it scores the trained model to generate recommendations by using the ASTORE procedure.

This example uses the usersCommunity data set, which is available here:

https://support.sas.com/documentation/onlinedoc/viya/examples.htm.

The data set consists of click events over a period of time on a peer-to-peer support community website. Each web page (viewed as an item in the procedure) on the site belongs to a board that includes a number of topics

of specific interest. The same user might have viewed the same web page multiple times. Also, in this data set, it is indicated whether or not the questions within the topic have been resolved. For each click event, the user ID and item ID are recorded. The data set includes 40,000 click events (user feedback).

You can load the usersCommunity data table into your SAS library by using your libref as in the following statements:

```
   data mylib.usersCommunity;
input   itemid    boardid    userid    isSolvedTopic;
datalines;

   ... more lines ...
```

The following statements use PROC RECENGINE to train a *k*-nearest neighbors model by using the usersCommunity data:

```
proc recengine data=mylib.usersCommunity method=KNN outmodel=mylib.nearestNeighbors;
  input userid boardid / level=nominal;
  userid userid;
  itemid boardid;
  savestate rstore=mylib.astoreModel;
  knn k=15 outSimilarity=mylib.nearestNeighborValues;
run;
```

When the training is completed, PROC RECENGINE produces a data table named mylib.astoreModel for generating recommendations from the *k*-nearest neighbors model. Next, you use PROC ASTORE as in the following code to generate recommendations for the users in the usersCommunity data by using the mylib.astoreModel table. **NOTE:** When you use PROC ASTORE to score the *k*-nearest neighbors model, specify the COPYVARS= option in the SCORE statement to write the test items to the output table. These test items are used to assess the quality of recommendations that the *k*-nearest neighbors model generates.

```
proc astore;
   setoption REC_TOP_N 10;
   score data     = mylib.usersCommunity
         rstore   = mylib.astoreModel
         out      = mylib.recommendations
         copyvars = (boardid);
   run;
quit;
```

The following statements print the first 15 rows of the mylib.recommendations table. The results are shown in Figure 27.1.

```
proc print data=mylib.recommendations(obs=15);
run;
```

Figure 27.1 User Recommendations Produced by PROC ASTORE

Obs	userid	_RECOMMENDED_boardid	_SCORE	_RANK	boardid
1	6313	4926	13.3612	1	4817
2	6313	38398	13.3612	2	4817
3	6313	26088	13.3612	3	4817
4	6313	213078	13.0077	4	4817
5	6313	212700	13.0077	5	4817
6	6313	20822	13.0077	6	4817
7	6313	217513	13.0077	7	4817
8	6313	168258	12.4303	8	4817
9	6313	129505	12.4303	9	4817
10	6313	210408	9.6665	10	4817
11	1355	46715	0.7359	1	7589
12	1355	35181	0.7359	2	7589
13	1355	85302	0.7071	3	7589
14	1355	3800	0.7071	4	7589
15	1355	23384	0.7071	5	7589

NOTE: Although the mylib.recommendations table is sorted by user ID and rank in this example, sorting is not required when you use PROC RECASSESS. This procedure also does not require prior filtering of the input table for duplicate item recommendations for a user. If the input table for this procedure contains duplicate item recommendations or duplicate ranks for a user, the duplicate values are filtered out to ensure that the assessment metrics are calculated accurately.

The following statements use PROC RECASSESS to generate assessment metrics for the recommendations data that are generated by PROC ASTORE on the training data. The results are shown in Figure 27.2.

```
proc recassess data = mylib.recommendations nrec=5;
   input userid _RECOMMENDED_boardid boardid / level = nominal;
   input _rank / level = interval;
   userId userId;
   recomitemid _RECOMMENDED_boardid;
   testitemid boardid;
   rank _rank;
run;
```

Figure 27.2 Assessment Metrics for mylib.recommendations Data Generated by PROC ASTORE

Assessment Metrics	
Metric	**Value**
Mean Average Precision	0.2825914486
Recall	0.213889599
Hit Rate	0.3531976744
Mean Reciprocal Rank	0.3074460998
Weighted Normalized Discounted Cumulative Gain	0.2774431441

Syntax: RECASSESS Procedure

The following statements are available in the RECASSESS procedure:

PROC RECASSESS < *options* > **;**
  DISPLAY < *table-list* > < */options* > **;**
  DISPLAYOUT *table-spec-list* < */options* > **;**
  INPUT *variables* < **LEVEL=INTERVAL** | **NOMINAL** > **;**
  RANK *variable* **;**
  RECOMITEMID *variable* **;**
  TESTITEMID *variable* **;**
  USERID *variable* **;**

The PROC RECASSESS, INPUT, ITEMID, TESTITEMID, and USERID statements are required. The following sections describe the PROC RECASSESS statement and then describe the other statements in alphabetical order.

PROC RECASSESS Statement

PROC RECASSESS < *options* > **;**

The PROC RECASSESS statement invokes the procedure. Table 27.1 summarizes the *options* available in this statement.

Table 27.1 PROC RECASSESS Statement Options

Option	Description
Input Data Table Options	
DATA=	Specifies the input data table
NREC=	Specifies how many recommendations to use in calculating assessment metrics
Other Option	
NTHREADS=	Specifies the number of threads to use on each computation node

You can specify the following *options*:

DATA=*libref.data-table*
names the input data table for PROC RECASSESS to use. The default is the most recently created data table. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to the data. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 706.

data-table specifies the name of the input data table.

NREC=*number-of-recommendations*
specifies how many recommendations to use in calculating assessment metrics.

By default, NREC=1.

NTHREADS=*number-of-threads*
specifies the number of threads per computation node. The default value is the number of threads available per computation node.

DISPLAY Statement

DISPLAY < *table-list* > < / *options* > **;**

The DISPLAY statement enables you to specify a list of display tables to display or exclude. This statement is similar to the ODS SELECT, ODS EXCLUDE, and ODS TRACE statements. However, the DISPLAY statement can improve performance when a large number of tables could be generated (such as in BY-group processing). The DISPLAY statement produces only the requested tables, but ODS statements generate all the tables and then keep the requested tables.

If you use both DISPLAY and ODS statements together, the DISPLAY statement takes precedence over the ODS statements. Note that the ODS EXCLUDE statement processes tables after they have been filtered by the DISPLAY statement. In some cases, it might appear that the ODS EXCLUDE statement is taking precedence because it can further filter the tables. For more information about ODS, see *SAS Output Delivery System: Procedures Guide*.

You can specify the *table-list* as a list of table names, paths, partial pathnames, and regular expressions.

The table names that you can specify are listed in the section "ODS Table Names" on page 715. A path is a table name that is prefixed with dot-separated grouping information. For example, a SelectionSummary table that a procedure produces during a selection routine might have the path *Bygroup1.Summary.SelectionSummary*. A partial pathname does not include all groups; for example, *SelectionSummary* and *Summary.SelectionSummary* are partial pathnames for *Bygroup1.Summary.SelectionSummary*.

When you specify a table name or partial pathname, all display tables whose paths end in the specified name are selected for display or exclusion. For example, both *SelectionSummary* and *Summary.SelectionSummary* select *Bygroup1.Summary.SelectionSummary*.

A regular expression is enclosed in forward slashes (/). For example, specifying "/tions/" selects all pathnames that contain the substring "tions"; in particular, the *Bygroup1.Summary.SelectionSummary* table is selected. Specifying "!/tions/" selects all pathnames that do not contain the substring "tions"; in particular, the *Bygroup1.Summary.SelectionSummary* table is not selected.

You can specify the following *options* after a slash (/):

CASESENSITIVE
performs a case-sensitive comparison of table names in the *table-list* to display table names when tables are subsetted for display. To preserve case, you must enclose table names in the *table-list* in quotation marks.

EXCLUDE
displays all display tables except those that you specify in the *table-list*.

EXCLUDEALL
suppresses display of all tables. This option takes precedence over the other options.

TRACE
displays the display table names, labels, and paths.

DISPLAYOUT Statement

DISPLAYOUT *table-spec-list* < / *options* > **;**

The DISPLAYOUT statement enables you to create output tables from your displayed output. This statement is similar to the ODS OUTPUT statement. For more information about ODS, see *SAS Output Delivery System: Procedures Guide*.

The *table-spec-list* specifies a list of output tables to create. Each entry in the list has either a *key*=*value* format or a *key* format:

key=*value* specifies *key* as the ODS table name, path, or partial pathname, and specifies *value* as the output table name.

key specifies *key* as the ODS table name and also as the output table name.

The ODS table names that you can specify are listed in the section "ODS Table Names" on page 715. You cannot specify the ODS table named OutputCasTables in the *table-spec-list*.

Table names and partial pathnames are discussed under the DISPLAY statement. The DISPLAYOUT statement does not support regular expressions.

You can specify the following *options* after a slash (/):

INCLUDEALL
creates output tables for all display tables. The name of the created output table is the same as the name of the corresponding display table. If you specify this option, the *table-spec-list* specification is ignored.

NOREPLACE
does not replace any existing output table of the same name.

INPUT Statement

INPUT *variables* < **LEVEL=INTERVAL | NOMINAL** > **;**

The INPUT statement specifies the names of the variables to be used in the recommendation assessment. It names one or more input variables that use common options. If you want to use different options for different variables, you can specify multiple INPUT statements.

You can include the following option in each INPUT statement:

LEVEL=INTERVAL | NOMINAL
specifies the level of measurement of the *variables*. By default, LEVEL=INTERVAL for numeric variables and LEVEL=NOMINAL for categorical variables.

RANK Statement

RANK *variable* **;**

The RANK statement lists one *variable* to use as the rank of the recommended item ID to be assessed. It must be one of the nominal variables that you specify in the INPUT statement.

RECOMITEMID Statement

RECOMITEMID *variable* **;**

The RECOMITEMID statement lists one *variable* to use as the recommended item ID to be assessed. It must be one of the nominal variables that you specify in the INPUT statement.

TESTITEMID Statement

TESTITEMID *variable* **;**

The TESTITEMID statement lists one *variable* for the recommender system to use as an item ID to identify unique items that the user actually selected. It must be one of the nominal variables that you specify in the INPUT statement.

USERID Statement

USERID *variable* **;**

The USERID statement lists a *variable* for the recommender system to use as a user ID to identify unique users. It must be one of the nominal variables that you specify in the INPUT statement.

Details: RECASSESS Procedure

The RECASSESS procedure calculates metrics to assess the quality of user recommendations on the basis of historical item interactions. This procedure accepts as its input a table in which each row is a scoring sample that contains the following information: user ID, recommended item ID, recommended item ID rank, and test item ID. Using this input table, the procedure calculates five metrics: mean average precision, hit rate, recall, mean reciprocal rank, and weighted binomial normalized discounted cumulative gain.

If you have a user u from a set of users $\mathcal{U}$, the set $\mathrm{rel}(u)$ is the set of items that are relevant to user u. The set $\mathrm{rec}_k(u)$ is the set of k items that are recommended to user u. And $\mathrm{rank}(u, i)$ is the rank of item i for user u in the rank-ordered list of recommendations $\mathrm{rec}_k(u)$. Given these definitions, the metrics that PROC RECASSESS calculates are defined by the following equations.

Mean average precision is defined as follows:

$$\text{mean average precision} = \frac{1}{|\mathcal{U}|} \sum_{u \in \mathcal{U}} \frac{|\mathrm{rel}(u) \cap \mathrm{rec}_k(u)|}{\min(|\mathrm{rel}(u)|, k)}$$

Mean average precision at k is the proportion of top k recommendations that are relevant, when averaged across users. For example, a mean average precision at $k = 10$ of 30% means that on average, 30% of the top k user recommendations are found in the user's relevant items.

Recall is defined as follows:

$$\text{recall} = \frac{1}{|\mathcal{U}|} \sum_{u \in \mathcal{U}} \frac{|\mathrm{rel}(u) \cap \mathrm{rec}_k(u)|}{|\mathrm{rel}(u)|}$$

Recall at k is the proportion of relevant items that are contained in the top k recommendations, when averaged across users. For example, a recall at $k = 10$ of 30% means that on average, 30% of the user's relevant items are found in the top k user recommendations. Note that the recall metric is sensitive to the number of relevant items for each user; for a given user, if the number of relevant items is larger than the number of recommendations, then the user's contribution to the recall metric is less than 1 even if all the recommended items are in the user's relevant item set.

Hit rate is defined as follows:

$$\text{hit rate} = \frac{1}{|\mathcal{U}|} \sum_{u \in \mathcal{U}} I\{|\text{rel}(u) \cap \text{rec}_k(u)| > 0\}$$

Hit rate at k is the proportion of users for whom a relevant item is found in their top k recommendations. For example, a hit rate at $k = 10$ of 30% means that 30% of users had at least one recommendation that was also in their relevant items.

Mean reciprocal rank is defined as follows:

$$\text{mean reciprocal rank} = \frac{1}{|\mathcal{U}|} \sum_{u \in \mathcal{U}} \max_{i \in \text{rel}(u) \cap \text{rec}_k(u)} \text{rank}(u, i)^{-1}$$

Mean reciprocal rank at k is the average multiplicative inverse of the rank of the first relevant item in the ranked list of top k recommendations. This metric measures how close the relevant user recommendations are to the top of the rank-ordered recommendation list. For example, a mean reciprocal rank at $k = 10$ of 50% means that on average, the highest rank of a relevant item for a user is rank 2. A mean reciprocal rank at $k = 10$ of 25% means that on average, the highest rank of a relevant item for a user is rank 4.

Weighted binomial normalized discounted cumulative gain (WBNDCG) is another measure of ranking quality. WBNDCG is the average ratio of the binomial normalized discounted cumulative gain (BNDCG) and the ideal binomial normalized discounted cumulative gain (IBNDCG) across users:

$$\text{WBNDCG} = \frac{1}{|\mathcal{U}|} \sum_{u \in \mathcal{U}} \frac{\text{BNDCG}_u}{\text{IBNDCG}_u}$$

where BNDCG_u is given by

$$\text{BNDCG}_u = \sum_{i \in \text{rec}_k(u)} \frac{I\{i \in \text{rel}(u)\}}{\log_2(\text{rank}(u, i) + 1)}$$

and IBNDCG_u is the ideal BNDCG for user u if all relevant recommendations were at the top of the user's rank-ordered recommendation list. Unlike mean reciprocal rank, which uses only the top relevant item in the rank-ordered recommendation list, WBNDCG uses the rank ordering of all relevant items in the set of user recommendations. Relevant items contribute to the metric on a discounted basis of their rank. Because each user's contribution to WBNDCG is weighted by the user's IBNDCG, WBNDCG is bounded between 0 and 1, where a value of 1 indicates that all recommendations relevant to the user are at the top of the rank-ordered recommendation list.

Displayed Output

The RECASSESS procedure displays various tables for analysis. The following sections describe the output in the order of its appearance when you specify the related options.

Model Information

The ModelInfo table displays basic information about the parameters that are used in the analysis. This information includes the number of recommendations that are used to calculate assessment metrics.

Number of Observations

The NObs table displays the number of observations that are read from the input data table and the number that are used in the analysis.

Nominal Variables

The DescStatsNom table displays the number of levels for each nominal variable.

Assessment Metrics

The AssessmentMetrics table displays the assessment metrics that are calculated using the input data table.

ODS Table Names

Each table that the RECASSESS procedure creates has a name associated with it. You must use this name to refer to the table when you use ODS statements. The name of each table and a short description of the contents are listed in Table 27.2.

Table 27.2 ODS Tables Produced by PROC RECASSESS

Table Name	Description	Statement	Option
AssessmentMetrics	Calculated assessment metrics	PROC RECASSESS	Default
DescStatsNom	Number of levels of each nominal variable	PROC RECASSESS	Default
ModelInfo	Model information	PROC RECASSESS	Default
NObs	Number of observations	PROC RECASSESS	Default

Chapter 28
RECENGINE Procedure

Contents

Overview: RECENGINE Procedure **718**
- PROC RECENGINE Features 718
- Using SAS Viya Workbench 718

Getting Started: RECENGINE Procedure **719**

Syntax: RECENGINE Procedure **721**
- PROC RECENGINE Statement 722
- AUTOTUNE Statement 724
- BPR Statement 732
- CONTEXT Statement 733
- DISPLAY Statement 734
- DISPLAYOUT Statement 735
- DTOS Statement 735
- INPUT Statement 737
- ITEMFEATURE Statement 737
- ITEMID Statement 738
- KNN Statement 738
- SAVESTATE Statement 739
- TARGET Statement 739
- USERFEATURE Statement 740
- USERID Statement 740

Details: RECENGINE Procedure **741**
- Bayesian Personalized Ranking (BPR) 741
- Data Translation with Optimal Step Size (DTOS) 742
- K-Nearest Neighbors (KNN) 743
- Displayed Output 744
- ODS Table Names 745

Examples: RECENGINE Procedure **746**
- Example 28.1: Recommending SAS Users Community Pages to SAS Users 746
- Example 28.2: Running the KNN Method of the Recommender Engine Model on Explicit Feedback 747
- Example 28.3: Tuning Hyperparameters of an MF-PDF Model 750
- Example 28.4: Tuning Hyperparameters of a Bayesian Personalized Ranking Model . 752
- Example 28.5: Tuning Hyperparameters of a k-Nearest Neighbors Model 753

References **756**

Overview: RECENGINE Procedure

Recommender systems are information filters that predict and suggest items or content that users might like, based on their previous behaviors, preferences, and interactions, in order to provide tailored suggestions to individual users. By using data-driven algorithms, recommender systems help users discover relevant content in an environment that inundates them with information. Recommender systems play a crucial role in enhancing user experience, increasing content or product consumption, and driving sales by presenting users with items that they are more likely to purchase, watch, or engage with. By personalizing user experiences, they also help businesses optimize their marketing strategies and understand their customers better.

Applications of recommender systems are diverse and continually expanding, with new use cases being identified in the research literature every year. Business domains that use recommender systems include e-commerce, content delivery, retail, banking, health care, oil and gas, automotive, industrial automation, and many others. Recommender system applications have also been identified for other service use cases, including in the context of municipal government and electrical grid management.

The RECENGINE procedure implements several different recommender system models that perform the task of recommending items to users on the basis of historical iterations between users and items. For more information about these models, see the section "Details: RECENGINE Procedure" on page 741.

PROC RECENGINE Features

The RECENGINE procedure has the following features:

- is multithreaded during all phases of analytic execution
- supports large-scale training data
- enables you to take advantage of methods that are designed specifically for implicit interactions between users and items
- enables you to perform analysis on selected columns of data

Using SAS Viya Workbench

SAS Viya Workbench has greatly simplified the SAS programming experience by offering a syntax style nearly identical to that of SAS 9 and earlier releases. However, there are minor exceptions. In the past, SAS Viya procedures required the use of a CAS engine libref. This engine served multiple purposes, but it was primarily used to upload data to the CAS server and manage the data. Its additional purposes included support of new data types and improved performance for large data sets. No CAS server is needed for SAS Viya Workbench, because all computations are done in place for you in a single location.

In the following program, the LIBNAME statement creates a mylib libref that the examples in this chapter use to refer to your current or desired work directory:

```
libname mylib ".";
```

For more information, see *SAS Viya Workbench: Getting Started*.

Getting Started: RECENGINE Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

This example shows how to use the RECENGINE procedure to train a recommender system model on observations in a data table. The usersCommunity data set includes a subset of the click events over a period of time on a peer-to-peer-support community website. Each web page (item) on the site belongs to a message board that includes a number of topics of specific interest. The same user might have viewed the same web page multiple times. Also, in this data set, it is indicated whether or not the questions within the topic have been resolved. For each click event, the user ID and item ID are recorded. This data set includes 40,000 click events (observations).

The following code includes the DATA step that generates the data table mylib.usersCommunity. You can find the complete DATA set here:

https://support.sas.com/documentation/onlinedoc/viya/examples.htm

You can load the usersCommunity data table into your SAS library by using your libref as in the following statements:

```
data mylib.usersCommunity;
   input itemid boardid userid isSolvedTopic;
   datalines;
   38 4817 6313 0
   15 7589 1355 1
   46 9560 2926 1
   16 11635 3985 1
   38 13023 4163 0

... more lines ...
```

The following statements train a recommender system model on the usersCommunity data by using PROC RECENGINE:

```
proc recengine data=mylib.usersCommunity method=bpr nthreads=1 outmodel=mylib.factors;
   input userid itemid/level=nominal;
   userid userid;
   itemid itemid;
   bpr maxIter=20 nFactors=10 learnStep=0.01 regl2=0.01 seed=1;
   savestate rstore=mylib.state;
run;
```

The following statements print the first 10 observations in the mylib.factors data table, which includes some of the rows that are related to the latent factors for users. The output is shown in Figure 28.1.

```
proc print data=mylib.factors(obs=10);
run;
```

Figure 28.1 User Factors

Obs	Variable	Level	Bias	Factor1	Factor2	Factor3	Factor4	Factor5	Factor6	Factor7	Factor8	Factor9	Factor10
1	_GLOBAL_		0	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000	0.00000
2	itemid	8	0	-1.12485	4.45192	2.19208	-0.60274	-1.77642	2.14684	-0.59482	-0.39089	1.39398	2.05710
3	itemid	9	0	0.03624	3.21847	-3.67782	4.29316	-1.04534	3.83772	4.33494	-1.09129	0.34840	-2.91478
4	itemid	10	0	-2.78328	1.86587	-2.04741	-2.00398	-1.13518	0.82657	-1.10613	-2.48457	-1.70573	0.79410
5	itemid	11	0	0.12972	0.73663	1.27758	2.48206	-1.89837	-0.45707	2.19836	-3.74934	-2.91389	-3.74224
6	itemid	15	0	6.11635	-2.34953	-2.74044	-4.51393	-0.15203	0.50069	-2.16138	1.98882	-3.95932	1.74290
7	itemid	16	0	1.31643	1.50721	1.40146	-1.93326	-2.93120	7.88654	0.02650	-1.51039	-2.15252	-3.79505
8	itemid	17	0	-1.97416	3.98234	-0.82852	-1.04245	-1.02738	0.37425	-0.77400	-1.92542	-4.39585	-0.70957
9	itemid	18	0	4.59144	0.40213	-1.29228	-2.76491	-2.50467	-1.08777	1.55707	0.42976	-2.94995	1.10726
10	itemid	22	0	1.71007	-2.33381	-2.07336	-4.04132	-2.21438	2.65438	-0.57034	-1.43373	-2.61990	-2.00019

The following statements print the last 10 observations in the mylib.factors data table, which includes some of the rows that are related to the latent factors for items. The output is shown in Figure 28.2.

```
data _null_;
   if 0 then set mylib.factors nobs=n;
   call symputx("n",left(put(n,best.)),'l');
run;

proc print data=mylib.factors(firstobs=%eval(&n-10));
run;
```

Figure 28.2 Item Factors

Obs	Variable	Level	Bias	Factor1	Factor2	Factor3	Factor4	Factor5	Factor6	Factor7	Factor8	Factor9	Factor10
5530	userid	6493	0	-2.05324	-0.74745	-0.43335	-0.14207	1.27943	-1.48819	-0.50519	3.75641	1.75114	0.92086
5531	userid	6494	0	-0.99664	-1.52075	0.15280	0.74807	2.66734	-0.66143	-1.11315	0.71371	1.50215	2.22022
5532	userid	6495	0	-1.40196	0.28836	-1.22295	1.40126	1.68950	0.66801	-2.45429	1.66835	0.68817	-0.89730
5533	userid	6496	0	-1.74555	-0.90889	0.53717	2.17322	-1.58115	-2.26984	-1.04863	1.60068	-0.18941	-0.89444
5534	userid	6497	0	-1.33361	0.22481	-1.37731	0.35985	-1.27575	-1.15299	-1.01390	-0.48978	2.41042	1.13173
5535	userid	6498	0	-1.22752	-0.13601	-0.94915	-0.38083	2.77113	-2.06585	-1.59524	-0.20439	2.10712	1.84687
5536	userid	6499	0	-2.44133	-0.20218	-1.28871	1.65504	1.48795	-1.32646	-1.47451	1.66530	1.57993	1.82965
5537	userid	6500	0	-0.81307	1.30038	-0.71154	-0.00072	1.57934	-1.71881	-0.65367	2.54162	0.44767	0.77166
5538	userid	6501	0	-1.50311	-0.84557	-0.60752	0.90024	2.62379	-1.63011	-0.52599	2.06889	1.77451	2.02186
5539	userid	6502	0	-0.27437	-1.01527	-0.03764	1.40936	0.53061	1.07937	0.90707	0.13427	-1.68125	-0.64236
5540	userid	6503	0	-1.75542	-0.40282	-1.86557	1.61692	0.57108	-2.03655	0.70714	1.12797	-1.05233	1.49678

You can use the mylib.factors output table to rank items for each user. The following code shows how to find the top two items for a sample user (that is, the one whose user ID is 6313).

First, you create the scoring data table that includes the ID of the sample user of interest:

```
data mylib.user6313;
input   userid;
datalines;
6313
;
run;
```

Next, you use the ASTORE procedure to score this user:

```
proc astore;
   setoption REC_TOP_N 2;
   score data=mylib.user6313 rstore=mylib.state out=mylib.rankedItems;
   run;
quit;
```

The following statements print the top two recommended items for user 6313. The output is shown in Figure 28.3.

```
proc print data=mylib.rankedItems;
run;
```

Figure 28.3 Top Two Items for User 6313

Obs	userid	_RECOMMENDED_itemid	_RANK	_SCORE
1	6313	16	1	12.1376
2	6313	15	2	10.2563

Syntax: RECENGINE Procedure

The following statements are available in the RECENGINE procedure:

PROC RECENGINE METHOD=*method*< *options*> ;
- **AUTOTUNE** < *options*> ;
- **BPR** < *options*> ;
- **CONTEXT** < *variables*> ;
- **DISPLAY** < *table-list*> < */options*> ;
- **DISPLAYOUT** *table-spec-list* < */options*> ;
- **DTOS** < *options*> ;
- **INPUT** *variables* < / **LEVEL=INTERVAL** | **NOMINAL** > ;
- **ITEMFEATURE DATA=***libref.data-table* **INPUT=(***variables***)** < */option*> ;
- **ITEMID** *variable* ;
- **KNN** < *options*> ;
- **SAVESTATE RSTORE=***libref.data-table* ;
- **TARGET** *variable* ;
- **USERFEATURE DATA=***libref.data-table* **INPUT=(***variables***)** < */option*> ;
- **USERID** *variable* ;

The PROC RECENGINE, INPUT, ITEMID, and USERID statements are required. The following sections describe the PROC RECENGINE statement and then describe the other statements in alphabetical order.

PROC RECENGINE Statement

PROC RECENGINE METHOD=*method*< *options*> **;**

The PROC RECENGINE statement invokes the procedure. Table 28.1 summarizes the options available in this statement.

Table 28.1 PROC RECENGINE Statement Options

Option	Description
Data Table Options	
DATA=	Specifies the input data table
ITEMMODEL=	Specifies the output table in which to save the item model output parameters
OUTMODEL=	Specifies the output table in which to save the model output parameters
USERMODEL=	Specifies the output table in which to save the user model output parameters
Other Options	
METHOD=	Specifies the recommender system method to be used
NGPUS=	Specifies the number of GPUs to use
NTHREADS=	Specifies the number of threads to use on each computation node

You must specify the following option:

METHOD=*method*

specifies the recommender system *method* to use. You can specify the following *methods*:

BPR uses the Bayesian personalized ranking (BPR) method. For more information about this method, see the section "Bayesian Personalized Ranking (BPR)" on page 741.

DTOS uses the data translation with optimal step size (DTOS) method. For more information about this method, see the section "Data Translation with Optimal Step Size (DTOS)" on page 742.

KNN uses a sparse k-nearest neighbors (KNN) method. For more information about this method, see the section "Details: RECENGINE Procedure" on page 741.

You can also specify the following *options*:

DATA=*libref.data-table*

names the input data table for PROC RECENGINE to use. The default is the most recently created data table. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to the data. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 718.

data-table specifies the name of the input data table.

ITEMMODEL=*libref.data-table*

specifies the name of the output table that contains the model parameters for items. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 718.

NGPUS=*number-of-gpus*

specifies the number of GPUs to use in order to accelerate computation on at least one node, where *number-of-gpus* can be 0 or 1. Although it is not required, it is recommended that all nodes have equivalent or similar GPU resources to achieve maximum performance. This option is available when METHOD=DTOS.

By default, NGPUS=0.

NTHREADS=*number-of-threads*

specifies the number of threads per computation node. The default value is the number of threads available per computation node.

OUTMODEL=*libref.data-table*

specifies the name of the output table that contains the model outputs. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 718.

USERMODEL=*libref.data-table*

specifies the name of the output table that contains the model parameters for users. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 718.

AUTOTUNE Statement

AUTOTUNE < *options* > **;**

The AUTOTUNE statement searches for the best combination of some option values in PROC RECENGINE.

Table 28.2 summarizes the *options* that you can specify in the AUTOTUNE statement. For more information about all options except the AGGREGATIONMETHOD=, ITEMFILTER, TOPN=, and TUNINGPARAMETERS= options, see the option's description in the section "AUTOTUNE Statement" on page 6 in Chapter 2, "Shared Concepts." These four options are described following Table 28.2. Note that the hyperparameter that you specify in the TUNINGPARAMETERS= option for tuning must be related to the METHOD= option value that you specify in the PROC RECENGINE statement.

Table 28.2 AUTOTUNE Statement Options

Option	Description
AGGREGATIONMETHOD=	Specifies the type of aggregation to apply in pooling similarity scores of nearest neighbors
APPENDLOOKUP	Specifies that the table specified in the HISTORYTABLE= option contain the rows from the table specified in the LOOKUPTABLE= option
CONSTRAINTS=	Specifies the constraints on the output metrics
EVALHISTORY=	Specifies how to report the evaluation history of the tuner
FOLDCOLUMN=	Specifies the column in the data table in which the cross validation fold value is indicated
FRACTION=	Specifies the fraction of observations to use for validation
HISTORYTABLE=	Specifies the table that contains the evaluation history
KFOLD=	Specifies the number of folds for *k*-fold cross validation
ITEMFILTER	Specifies that only items that the user has not previously observed be recommended during the tuning process
LIVEUPDATE	Specifies that the table specified in the HISTORYTABLE= option be updated at every evaluation
LOOKUPTABLE=	Specifies the table to use for evaluation lookup
MAXBAYES=	Specifies the maximum number of points in the kriging model
MAXEVALS=	Specifies the maximum number of evaluations
MAXITER=	Specifies the maximum number of iterations when SEARCHMETHOD=GA or SEARCHMETHOD=BAYESIAN
MAXTIME=	Specifies the maximum time for all iterations
MAXTRAINTIME=	Specifies the maximum time for a model training
NCONVITER=	Specifies the number of convergence iterations
NOGRIDSHUFFLE	Requests that the grid points not be shuffled
NOLOCALSEARCH	Disables local search optimization
NPARALLEL=	Specifies the number of parallel sessions
NSUBSESSIONWORKERS=	Specifies the number of workers in parallel sessions
OBJECTIVE=	Specifies the objective function
POPSIZE=	Specifies the population size when SEARCHMETHOD=GA or SEARCHMETHOD=BAYESIAN

Table 28.2 *continued*

Option	Description
SAMPLESIZE=	Specifies the sample size when SEARCHMETHOD=LHS or SEARCHMETHOD=RANDOM
SEARCHMETHOD=	Specifies the search method that the optimizer uses
SECONDOBJECTIVE=	Specifies the second objective to use for tuning
SELECTINITPOINT	Specifies that the tuner select the best evaluation from the lookup table
TARGETEVENT=	Specifies the target event for ROC-based calculations
TOPN=	Specifies the number of items to be recommended to each user in the scoring input data during the tuning process
TRAINFRACTION=	Specifies the fraction of observations to use for training
TUNINGPARAMETERS=	Specifies the custom tuning parameters
USEPARAMETERS=	Specifies how to handle the TUNINGPARAMETERS= option

AGGREGATIONMETHOD=0 | 1 | 2 | 3

specifies the type of aggregation to apply in pooling similarity scores of nearest neighbors when METHOD=KNN. You can specify the following values:

0 sums scores for repeated nearest neighbors.

1 averages scores for repeated nearest neighbors.

2 sums scores for repeated nearest neighbors with mean adjustment.

3 sums scores for repeated nearest neighbors with Z-score adjustment.

By default, AGGREGATIONMETHOD=0.

ITEMFILTER

specifies that only items that the user has not previously observed be recommended during the tuning process.

TOPN=*number*

specifies the number of items to be recommended to each user in the scoring input data during the tuning process.

By default, TOPN=50.

TUNINGPARAMETERS=(*suboption* **| . . . | <** *suboption***>)**

TUNEPARMS=(*suboption* **| . . . | <** *suboption***>)**

specifies which parameters to tune and which ranges to tune over. If USEPARAMETERS=STANDARD, this option is ignored.

You can specify one or more of the following *suboptions*:

DISTANCEMETRIC (LB=*number* UB=*number* VALUES=*value-list* INIT=*number* EXCLUDE)
specifies the distance metric to be used during a KNN search in the tuning process. This option is relevant only when METHOD=KNN.

By default, DISTANCEMETRIC=COSINE.

FEEDBACKCONFIDENCESLOPE (LB=*number* UB=*number* VALUES=*value-list* INIT=*number* EXCLUDE)

specifies the slope that converts confidence to weight. This option is relevant only when METHOD=DTOS.

You can specify the following additional *suboptions*:

LB=*number*
specifies the minimum value for the slope that converts confidence to weight to use in the tuning process. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=0.

UB=*number*
specifies the maximum value for the slope that converts confidence to weight to use in the tuning process. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=100.

VALUES=*value-list*
specifies a list of values for the slope that converts confidence to weight to use in the tuning process, where *value-list* is a space-separated list. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial value for the slope that converts confidence to weight to use in the tuning process.

By default, INIT=50.

EXCLUDE
excludes the slope that converts confidence to weight from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

K (LB=*number* UB=*number* VALUES=*value-list* INIT=*number* EXCLUDE)
specifies the number of neighbors in a KNN search. This option is relevant only when METHOD=KNN.

You can specify the following additional *suboptions*:

LB=*number*
specifies the minimum number of neighbors in a KNN search to use in the tuning process, where *number* is a nonnegative integer. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=1.

UB=*number*
specifies the maximum number of neighbors in a KNN search to use in the tuning process, where *number* is a nonnegative integer. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=30.

VALUES=*value-list*
specifies a list of values for the number of neighbors in a KNN search to use in the tuning process, where *value-list* is a space-separated list of nonnegative integer values. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial value for the number of neighbors in a KNN search to use in the tuning process.

By default, INIT=5.

EXCLUDE
excludes the number of neighbors in a KNN search from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

LEARNSTEP (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**
specifies the learning step size for the stochastic gradient descent (SGD) algorithm. This option is relevant only when METHOD=BPR.

You can specify the following additional *suboptions*:

LB=*number*
specifies the minimum learning step size to use in the tuning process,. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=0.001.

UB=*number*
specifies the maximum learning step size to use in the tuning process. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=0.5.

VALUES=*value-list*
specifies a list of learning step size values to use in the tuning process, where *value-list* is a space-separated list. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial learning step size to use in the tuning process.

By default, INIT=0.1.

EXCLUDE
excludes the learning step size from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

MAXITER (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**
specifies the maximum number of iterations for the algorithm to perform. This option is relevant only when METHOD=BPR or METHOD=DTOS.

You can specify the following additional *suboptions*:

LB=*number*
specifies the minimum number of iterations to use in the tuning process, where *number* is a nonnegative integer. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=10 when METHOD=BPR and LB=1 when METHOD=DTOS.

UB=*number*
specifies the maximum number of iterations to use in the tuning process, where *number* is a nonnegative integer. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=100 when METHOD=BPR and UB=1000 when METHOD=DTOS.

VALUES=*value-list*
specifies a list of maximum iterations to use in the tuning process, where *value-list* is a space-separated list of nonnegative integers. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial maximum iterations in the tuning process.

By default, INIT=20 when METHOD=BPR and INIT=10 when METHOD=DTOS.

EXCLUDE
excludes the maximum iterations from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

MINIBATCHSIZE (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**
specifies the number of observation samples to be used in each step of the stochastic gradient descent (SGD) algorithm. This option is relevant only when METHOD=BPR.

You can specify the following additional *suboptions*:

LB=*number*
specifies the minimum minibatch size to use in the tuning process, where *number* is a nonnegative integer. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=1.

UB=*number*
specifies the maximum minibatch size to use in the tuning process, where *number* is a nonnegative integer. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=100.

VALUES=*value-list*
specifies a list of minibatch size values to use in the tuning process, where *value-list* is a space-separated list. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial minibatch size to use in the tuning process.

By default, INIT=1.

EXCLUDE
excludes the minibatch size from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

MINNORMOBJCHANGE (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**

specifies the minimum normalized objective change over two consecutive training iterations that is required in order to stop training before the maximum number of iterations. This option is relevant only when METHOD=DTOS.

You can specify the following additional *suboptions*:

LB=*number*
specifies the minimum value for the minimum normalized objective change parameter to use in the tuning process. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=0.0.

UB=*number*
specifies the maximum value for the minimum normalized objective change to use in the tuning process. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=1.0.

VALUES=*value-list*
specifies a list of values for the minimum normalized objective change parameter to use in the tuning process, where *value-list* is a space-separated list. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial value for the minimum normalized objective change parameter to use in the tuning process.

By default, INIT=0.01.

EXCLUDE
excludes the minimum normalized objective change parameter from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

NFACTORS (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**
specifies the number of latent factors to be estimated. This option is relevant only when METHOD=BPR or METHOD=DTOS.

You can specify the following additional *suboptions*:

LB=*number*
specifies the minimum number of latent factors to use in the tuning process, where *number* is a nonnegative integer. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=5 when METHOD=BPR and LB=1 when METHOD=DTOS.

UB=*number*
specifies the maximum number of latent factors to use in the tuning process, where *number* is a nonnegative integer. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=30 when METHOD=BPR and UB=50 when METHOD=DTOS.

VALUES=*value-list*
specifies a list of number of latent factors to use in the tuning process, where *value-list* is a space-separated list of nonnegative integers. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial number of latent factors to use in the tuning process.

By default, INIT=10 when METHOD=BPR and INIT=1 when METHOD=DTOS.

EXCLUDE
excludes the number of latent factors from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

NFEATFACTORS (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**
specifies the number of latent factors for each nominal feature of users or items to be estimated. This option is relevant only when METHOD=DTOS and when the NODENSIFYNOMFEAT option is omitted.

You can specify the following additional *suboptions*:

LB=*number*
specifies the minimum number of latent factors for each nominal feature of users or items to use in the tuning process, where *number* is a nonnegative integer. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=1.

UB=*number*
specifies the maximum number of latent factors for each nominal feature of users or items to use in the tuning process, where *number* is a nonnegative integer. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=50.

VALUES=*value-list*
specifies a list of number of latent factors for each nominal feature of users or items to use in the tuning process, where *value-list* is a space-separated list of nonnegative integers. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial number of latent factors for each nominal feature of users or items to use in the tuning process.

By default, INIT=3.

EXCLUDE
excludes the number of latent factors for each nominal feature of users or items from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

NODENSIFYNOMFEAT
specifies that the nominal features are represented by sparse one-hot vectors in the tuning process. This option is relevant only when METHOD=DTOS.

NOITEMBIAS
specifies that the model not include item bias in the tuning process. This option is relevant only when METHOD=DTOS.

By default, this option is off.

NOUSERBIAS
specifies that the model not include user bias in the tuning process. This option is relevant only when METHOD=DTOS.

By default, this option is off.

REGULARIZATION (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**
specifies the regularization parameter. This option is relevant only when METHOD=BPR or METHOD=DTOS.

You can specify the following additional *suboptions*:

LB=*number*
specifies the minimum value for the regularization parameter to use in the tuning process. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=0.001.

UB=*number*
specifies the maximum value for the regularization parameter to use in the tuning process. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=0.5 when METHOD=BPR and UB=1000 when METHOD=DTOS.

VALUES=*value-list*
specifies a list of regularization values to use in the tuning process, where *value-list* is a space-separated list of nonnegative integers. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial value for the regularization parameter to use in the tuning process.

By default, INIT=0.1 when METHOD=BPR and INIT=0.01 when METHOD=DTOS.

EXCLUDE
excludes the regularization parameter from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

BPR Statement

BPR < *options* > **;**

The BPR statement specifies the parameters of the Bayesian personalized ranking method.

You can specify the following *options*:

CONVTOL=*number*
specifies the convergence tolerance, which is the minimum change in loss value that is required in order to be considered a meaningful improvement in the optimization process.

By default, CONVTOL=0.0001.

CONVWINSIZE=*number*
specifies the number of consecutive iterations without improvement in loss before the optimization process stops. If you specify CONVWINSIZE=0, the convergence criterion is disabled.

By default, CONVWINSIZE=3.

LEARNSTEP=*number*
specifies the learning step size for the stochastic gradient descent (SGD) algorithm, where *number* is a positive real number. The learning step size controls the amount by which the factors are updated at each iteration.

By default, LEARNSTEP=0.01. This value can be tuned with the AUTOTUNE statement.

MAXITER=*number*
specifies the maximum number of epochs for the algorithm to perform, where *number* is an integer greater than or equal to 1. In each iteration of the SGD method, the factors are recomputed.

By default, MAXITER=30. This value can be tuned with the AUTOTUNE statement.

MINIBATCHSIZE=*number*
specifies the number of observation samples to be used in each step of the stochastic gradient descent (SGD) algorithm. By default, the minibatch size is calculated automatically as a function of relevant factors, including data size and gradient sparsity.

NFACTOR=*number*
specifies the number of latent factors to be estimated, where *number* is an integer.

By default, NFACTOR=3. This value can be tuned with the AUTOTUNE statement.

OPTIMIZER=ADAM | ADAMW | SGD
specifies the optimization algorithm to use. You can specify one of the following values:

ADAM	uses the adaptive moments (Adam) variant of the stochastic gradient descent solver.
ADAMW	uses the Adam variant with decoupled weight decoy.
SGD	uses the stochastic gradient descent solver.

By default, OPTIMIZER=ADAMW.

REGL2=*number*
specifies the regularization parameter for the BPR optimization.

By default, REGL2=0.01. This value can be tuned with the AUTOTUNE statement.

SEED=*number*
specifies an integer that is used to start the pseudorandom number generator. This option enables you to reproduce the same output, but only when you specify NTHREADS=1 in the PROC RECENGINE statement. If you do not specify a seed or you specify a value less than or equal to 0, the seed is generated by reading the time of day from the computer's clock.

CONTEXT Statement

CONTEXT < *variables* > **;**

The CONTEXT statement specifies the names of the variables in the INPUT= statement that are used as context features. Context features are transaction features such as time or amount that is spent; their type can be either interval or categorical. However, for the best model performance, it is recommended that you convert interval variables to categorical variables through binning or other methods.

Context features are used only when you specify METHOD=BPR in the PROC RECENGINE statement.

DISPLAY Statement

DISPLAY < *table-list* > < / *options* > **;**

The DISPLAY statement enables you to specify a list of display tables to display or exclude. This statement is similar to the ODS SELECT, ODS EXCLUDE, and ODS TRACE statements. However, the DISPLAY statement can improve performance when a large number of tables could be generated (such as in BY-group processing). The DISPLAY statement produces only the requested tables, but ODS statements generate all the tables and then keep the requested tables.

If you use both DISPLAY and ODS statements together, the DISPLAY statement takes precedence over the ODS statements. Note that the ODS EXCLUDE statement processes tables after they have been filtered by the DISPLAY statement. In some cases, it might appear that the ODS EXCLUDE statement is taking precedence because it can further filter the tables. For more information about ODS, see *SAS Output Delivery System: Procedures Guide*.

You can specify the *table-list* as a list of table names, paths, partial pathnames, and regular expressions.

The table names that you can specify are listed in the section "ODS Table Names" on page 745. A path is a table name that is prefixed with dot-separated grouping information. For example, a SelectionSummary table that a procedure produces during a selection routine might have the path *Bygroup1.Summary.SelectionSummary*. A partial pathname does not include all groups; for example, *SelectionSummary* and *Summary.SelectionSummary* are partial pathnames for *Bygroup1.Summary.SelectionSummary*.

When you specify a table name or partial pathname, all display tables whose paths end in the specified name are selected for display or exclusion. For example, both *SelectionSummary* and *Summary.SelectionSummary* select *Bygroup1.Summary.SelectionSummary*.

A regular expression is enclosed in forward slashes (/). For example, specifying "/tions/" selects all pathnames that contain the substring "tions"; in particular, the *Bygroup1.Summary.SelectionSummary* table is selected. Specifying "!/tions/" selects all pathnames that do not contain the substring "tions"; in particular, the *Bygroup1.Summary.SelectionSummary* table is not selected.

You can specify the following *options* after a slash (/):

CASESENSITIVE
performs a case-sensitive comparison of table names in the *table-list* to display table names when tables are subsetted for display. To preserve case, you must enclose table names in the *table-list* in quotation marks.

EXCLUDE
displays all display tables except those that you specify in the *table-list*.

EXCLUDEALL
suppresses display of all tables. This option takes precedence over the other options.

TRACE
displays the display table names, labels, and paths.

DISPLAYOUT Statement

DISPLAYOUT *table-spec-list* < / *options* > **;**

The DISPLAYOUT statement enables you to create output tables from your displayed output. This statement is similar to the ODS OUTPUT statement. For more information about ODS, see *SAS Output Delivery System: Procedures Guide*.

The *table-spec-list* specifies a list of output tables to create. Each entry in the list has either a *key*=*value* format or a *key* format:

key=*value* specifies *key* as the ODS table name, path, or partial pathname, and specifies *value* as the output table name.

key specifies *key* as the ODS table name and also as the output table name.

The ODS table names that you can specify are listed in the section "ODS Table Names" on page 745. You cannot specify the ODS table named OutputCasTables in the *table-spec-list*.

Table names and partial pathnames are discussed under the DISPLAY statement. The DISPLAYOUT statement does not support regular expressions.

You can specify the following *options* after a slash (/):

INCLUDEALL
creates output tables for all display tables. The name of the created output table is the same as the name of the corresponding display table. If you specify this option, the *table-spec-list* specification is ignored.

NOREPLACE
does not replace any existing output table of the same name.

DTOS Statement

DTOS < *options* > **;**

The DTOS statement specifies the parameters of the data translation with optimal step size method.

You can specify the following *options*:

CONFIDENCE=*variable*
specifies the variable in the input data set that represents the confidence in the interaction between the user and the item in each observation.

BOUNDOBJ
specifies that the local approximation of the objective bounds the objective from above.

FEEDBACKCONFIDENCESLOPE=*number*
specifies the slope that converts confidence to weight.

By default, FEEDBACKCONFIDENCESLOPE=100. This value can be tuned with the AUTOTUNE statement.

MAXITER=*number*
specifies the maximum number of iterations for the algorithm to perform, where *number* is an integer greater than or equal to 1.

By default, MAXITER=30. This value can be tuned with the AUTOTUNE statement.

MINNORMOBJCHANGE=*number*
specifies the minimum normalized objective change over two consecutive training iterations that is required in order to stop training before the maximum number of iterations, as specified by the MAXITER= option, is reached.

By default, MINNORMOBJCHANGE=0. This value can be tuned with the AUTOTUNE statement.

NFACTOR=*number*
specifies the number of factors to be estimated, where *number* is an integer.

By default, NFACTOR=3. This value can be tuned with the AUTOTUNE statement.

NFEATFACTORS=*number*
specifies the number of latent factors for each nominal feature of users or items, where *number* is an integer. This option is not available when you specify the NODENSIFYNOMFEAT option.

By default, NFEATFACTORS=3.

NODENSIFYNOMFEAT
specifies that the nominal features are represented by sparse one-hot vectors. If you omit this option, the nominal features are represented by low-dimensional dense vectors.

NOITEMBIAS
specifies that the model not include item bias. This option can be tuned with the AUTOTUNE statement.

NOUSERBIAS
specifies that the model not include user bias. This option can be tuned with the AUTOTUNE statement.

REGL2=*number*
specifies the regularization parameter for the DTOS method.

By default, REGL2=0.01. This value can be tuned with the AUTOTUNE statement.

SEED=*number*
specifies an integer that is used to start the pseudorandom number generator. If you do not specify a seed or you specify a value less than or equal to 0, the seed is generated by reading the time of day from the computer's clock.

INPUT Statement

INPUT *variables* < / **LEVEL=INTERVAL | NOMINAL** > **;**

The INPUT statement specifies the names of the variables to be used in the recommender system. It names one or more input variables that use common options. If you want to use different options for different variables, you can specify multiple INPUT statements.

You can include the following option in each INPUT statement after a slash:

LEVEL=INTERVAL | NOMINAL
specifies the level of measurement of the *variables*. You can specify the following values:

INTERVAL specifies that the level of measurement of the *variables* is interval.

NOMINAL specifies that the level of measurement of the *variables* is nominal.

By default, LEVEL=INTERVAL for numeric variables and LEVEL=NOMINAL for categorical variables.

ITEMFEATURE Statement

ITEMFEATURE DATA=*libref.data-table* **INPUT=(***variables***)** < */option* > **;**

The ITEMFEATURE statement specifies the table that includes item features to be used in the recommender engine models. **NOTE:** You cannot currently use this statement when you specify the METHOD=KNN option in the PROC RECENGINE statement.

You must specify the following options:

DATA=*libref.data-table*
names the input data table for PROC RECENGINE to use. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to the data. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 718.

data-table specifies the name of the input data table.

Note that the item ID variable that you specify in the ITEMID statement must be one of the columns of the item feature data set.

INPUT=(*variables***)**
specifies the names of the item feature variables that are candidates for the model. You must enclose the *variables* in parentheses. Again, note that the item ID variable that you specify in the ITEMID statement must be one of the INPUT= option *variables*.

You can also specify the following *option* after a slash (/):

NOMINAL=(*variables*)
specifies the names of the variables in the INPUT= option that are nominal.

ITEMID Statement

ITEMID *variable* **;**

The ITEMID statement lists one variable to be used as an item ID by the recommender system to identify unique items. This variable must be one of the nominal variables that you specify in the INPUT statement.

KNN Statement

KNN < *options* > **;**

The KNN statement specifies the parameters of the sparse *k*-nearest neighbors method.

You can specify the following *options*:

DISTANCEMETRIC=COSINE | DICE
specifies the distance metric to be used during a KNN search. You can specify the following values:

COSINE	uses the cosine similarity measure during a KNN search.
DICE	uses the Dice similarity measure during a KNN search.

By default, DISTANCEMETRIC= COSINE.

K=*number*
specifies the number of neighbors in a KNN search.

MEMORYEFFICIENT
specifies that the model should use memory efficient hashing structures during a KNN search.

OUTSIMILARITY=*libref.data-table*
specifies the name of the output table to save the nearest-neighbor similarity scores to. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 718.

SAVESTATE Statement

SAVESTATE RSTORE=*libref.data-table* **;**

The SAVESTATE statement creates an analytic store for the model and saves it as a binary object in a data table. You can subsequently use the analytic store in the ASTORE procedure to score new data. For more information, see Chapter 4, "ASTORE Procedure."

You must specify the following option:

RSTORE=*libref.data-table*
specifies a data table in which to save the analytic store for the model. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 718.

The input scoring data table for PROC ASTORE must include the user ID column that you specify in the USERID statement. Also, this table can have some or all of the user features from the input user feature table that you specify in the DATA= option of the USERFEATURE statement, as well as some or all of the context features that you specify in the CONTEXT statement. Furthermore, the values of user features can be missing in the input scoring table.

NOTE: An existing user is a user who has a record in the input table that you specify in the DATA= option of the PROC RECENGINE statement. If the value of a feature for an existing user in the input scoring table is missing, its value is retrieved from the file that is specified in the RSTORE= option at the time of scoring. Also, for an existing user, if the value of a user feature in the input scoring table is different from the value of that feature in the user feature table, the value in the scoring table is used for scoring.

TARGET Statement

TARGET *variable* **;**

The TARGET statement names the *variable* that includes the explicit ratings. This *variable* should be an interval variable. The value of the target variable in each observation is the rating that the user assigns to the item for the transaction in that observation.

NOTE: This statement is currently valid only if you specify the METHOD=KNN option in the PROC RECENGINE statement.

USERFEATURE Statement

USERFEATURE DATA=*libref.data-table* **INPUT=(***variables***)** < */option* > **;**

The USERFEATURE statement specifies the table that includes user features to be used in the recommender engine models. **NOTE:** You cannot currently use this statement when you specify the METHOD=KNN option in the PROC RECENGINE statement.

You must specify the following options:

DATA=*libref.data-table*
names the input data table for PROC RECENGINE to use. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to the data. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 718.

data-table specifies the name of the input data table.

Note that the user ID variable that you specify in the USERID statement must be one of the columns of the user feature data set.

INPUT=(*variables***)**
specifies the names of the user feature variables that are candidates for the model. You must enclose the *variables* in parentheses. Again, note that the user ID variable that you specify in the USERID statement must be one of the INPUT= option *variables*.

You can also specify the following *option* after a slash (/):

NOMINAL=(*variables***)**
specifies the names of the variables in the INPUT= option that are nominal.

USERID Statement

USERID *variable* **;**

The USERID statement lists a variable to be used as a user ID by the recommender system to identify unique users. This variable must be one of the nominal variables that you specify in the INPUT statement.

Details: RECENGINE Procedure

This section discusses the recommender system methods that the RECENGINE procedure implements.

Bayesian Personalized Ranking (BPR)

Bayesian personalized ranking (BPR) is a common method that is designed specifically to optimize ranking. It has been shown to perform better than the standard learning techniques that are widely used to analyze explicit feedback (Rendle et al. 2012). The RECENGINE procedure accepts data that include the interactions of users with items, and it trains a matrix factorization model that is based on BPR.

You can use this method by specifying METHOD=BPR in the PROC RECENGINE statement. You can specify the options for this method in the BPR statement. The implementation of this method in this procedure is a distributed, multithreaded implementation of the BPR algorithm in SAS Viya. The method accepts as its input a table, where each row is a training sample in the following form: user ID, item ID. The resulting output of the BPR model (that is, factors) can be stored in an output table that you specify in the OUTMODEL= option in the PROC RECENGINE statement.

In an implicit feedback scenario, an observation or event $(u, i, \boldsymbol{x}_z)$ consists of a user (where $u \in U$), an item (where $i \in I$), and a vector of auxiliary features (where $\boldsymbol{x}_z, z \in Z$), where U, I, and Z denote the index set of all users, the index set of all items, and the index set of all possible auxiliary features, respectively. The auxiliary features can include user features, item features, and possibly context features. Assume that $S := \{(u, i, \boldsymbol{x}_z) | u \in U, i \in I, z \in Z\}$ is the observed implicit feedback data set. Let $I_u^+ := \{i \in I : (u, i, \boldsymbol{x}_z)) \in S\}$ denote the index set of items that user u has observed. Then, the training data set that is derived from the observed data set is defined as $D_S := \{(u, i, j) | i \in I_u^+, j \in I \backslash I_u^+, (u, i, \boldsymbol{x}_z)) \in S\}$. The task of the recommender system is to provide a user u with a personalized ranking of all pairs of items.

Bayesian personalized ranking that uses matrix factorization has been introduced by Rendle et al. (2012). This method can be generalized using the model parameters, $\boldsymbol{\theta}$, and the score function, $f(\boldsymbol{x}|\boldsymbol{\theta})$. The objective function of maximizing the posterior density of $\boldsymbol{\theta}$ is proposed as follows:

$$\sum_{(u,i,j) \in D_S} \ln \sigma(g(u, i, j|\boldsymbol{\theta})) - \frac{\lambda}{2} ||\boldsymbol{\theta}||_2^2$$

where $g(u, i, j|\boldsymbol{\theta}) := f(\boldsymbol{x}_{u,i}|\boldsymbol{\theta}) - f(\boldsymbol{x}_{u,j}|\boldsymbol{\theta})$ and $\sigma(g) := 1/(1 + \exp(-g))$ is the sigmoid function. The method of stochastic gradient descent (SGD) or the variants Adam and AdamW are applied to solve the optimization problem.

In the typical matrix factorization, a target matrix is approximated by the matrix product of two low-rank matrices, $V_U : |U| \times k$ and $V_I : |I| \times k$, where k is the rank of approximation, also called the number of latent factors. Each row of V_U, v_u, can be considered a k-dimensional latent vector for a user u, and similarly, each row of V_I, v_i, can be considered a k-dimensional latent feature vector for an item i. Therefore, the model parameters for matrix factorization are $\boldsymbol{\theta} := (V_U, V_I)$. Assume that $\boldsymbol{x}_{u,i}$ is the input vector that corresponds to the event (u, i). Then the score function can be written as

$$f(\boldsymbol{x}_{u,i}|\boldsymbol{\theta}) = \sum_{f=1}^{k} v_{u,f} v_{i,f}$$

You can enhance the score function by using the factorization machine (FM) model (Rendle 2012) when you take account of the auxiliary features. For simplicity, let $\boldsymbol{x}_{u,i,z} := (\boldsymbol{x}_u, \boldsymbol{x}_i, \boldsymbol{x}_z)$ be a feature vector for the event $(u, i, \boldsymbol{x}_z)$, where $\boldsymbol{x}_u = (0, \ldots, 0, 1, 0, \ldots, 0) \in \mathbb{R}^{|U|}$ is a one-hot vector for user u and $\boldsymbol{x}_i = (0, \ldots, 0, 1, 0, \ldots, 0) \in \mathbb{R}^{|I|}$ is a one-hot vector for item i. The corresponding model parameters are represented by $\boldsymbol{\theta} = (w_0, w_1, \ldots, w_n, v_{1,1}, \ldots, v_{n,k})$. The score function of the FM model is formed as

$$
\begin{aligned}
f(\boldsymbol{x}_{u,i,z}|\theta) =& w_0 + w_u + w_i + \sum_{z \in Z} w_z x_z + \sum_{f=1}^{k} v_{u,f} v_{i,f} + \sum_{z \in Z} x_z \sum_{f=1}^{k} v_{u,f} v_{z,f} \\
&+ \sum_{z \in Z} x_z \sum_{f=1}^{k} v_{i,f} v_{z,f} + \sum_{z_1 < z_2} x_{z_1} x_{z_2} \sum_{f=1}^{k} v_{z_1,f} v_{z_2,f}
\end{aligned}
$$

Data Translation with Optimal Step Size (DTOS)

Data translation with optimal step size (DTOS) is a method that uses a fast algorithm to train recommender systems on implicit feedback (Liao and Koch 2021). You can use this method by specifying METHOD=DTOS in the PROC RECENGINE statement. You can specify the options for this method in the DTOS statement. The implementation of this method in the procedure is a distributed, multithreaded implementation in SAS Viya. This method accepts as its input a table, where each row is a training sample in the following form: user ID, item ID, confidence. The recommender model that the DTOS algorithm trains is represented by matrix factorization with partially defined factors (MF-PDF), which is a model that generalizes matrix factorization (MF) to include predefined factors (PDF) of users and/or items. The resulting output of the DTOS model (that is, factors) can be stored in an output table that you specify in the OUTMODEL= option in the PROC RECENGINE statement.

For m users and n items, the procedure stores the user-item interactions in the $m \times n$ sparse matrix R, where $R_{ui} = 1$ if user u provides feedback for item i and $R_{ui} = 0$ otherwise. The MF-PDF model represents R as a product of two factor matrices, $R = XY$, where each factor matrix is partitioned into three submatrices,

$$
X = [X_{:\mathbb{F}_1}, X_{:\mathbb{F}_2}, X_{:\mathbb{F}_3}] \quad \text{and} \quad Y = \begin{bmatrix} Y_{\mathbb{F}_1:} \\ Y_{\mathbb{F}_2:} \\ Y_{\mathbb{F}_3:} \end{bmatrix}
$$

where $\mathbb{F}_1$, $\mathbb{F}_2$, and $\mathbb{F}_3$ form a partition of $\{1, 2, \ldots, |\mathbb{F}_1| + |\mathbb{F}_2| + |\mathbb{F}_3|\}$ (that is, they are mutually exclusive and $\mathbb{F}_1 \cup \mathbb{F}_2 \cup \mathbb{F}_1 = \{1, 2, \ldots, |\mathbb{F}_1| + |\mathbb{F}_2| + |\mathbb{F}_3|\}$). The partition yields a three-term representation of the user-item interactions:

$$
R = X_{:\mathbb{F}_1} Y_{\mathbb{F}_1:} + X_{:\mathbb{F}_2} Y_{\mathbb{F}_2:} + X_{:\mathbb{F}_3} Y_{\mathbb{F}_3:}
$$

Each term in this equation is a submodel that is detailed as follows:

1. The first term is a linear regression model, where $X_{:\mathbb{F}_1}$ are the predefined (fixed) factors (PDF) of users and $Y_{\mathbb{F}_1:}$ are the associated regression parameters.

2. The second term is a standard matrix factorization model, where $X_{:\mathbb{F}_2}$ are the latent user factors and $Y_{\mathbb{F}_2:}$ are the latent item factors.

3. The third term is a linear regression model, where $Y_{\mathbb{F}_3:}$ are the predefined (fixed) factors of items and $X_{:\mathbb{F}_3}$ are the associated regression parameters.

In the current implementation, $X_{:\mathbb{F}_1}$ is frozen as a column vector of ones and $Y_{\mathbb{F}_3:}$ is frozen as a row vector of ones; therefore $|\mathbb{F}_1| = |\mathbb{F}_3| \equiv 1$. In this special case, $X_{:\mathbb{F}_3}$ are the biases of users and $Y_{\mathbb{F}_1:}$ are the biases of items.

Because $X_{:\mathbb{F}_1}$ and $Y_{\mathbb{F}_3:}$ are fixed and nonadjustable, the parameters of MF-PDF are $X_{:\mathbb{P}}$ and $Y_{\mathbb{Q}:}$, where $\mathbb{P} = \mathbb{F}_2 \cup \mathbb{F}_3$ and $\mathbb{Q} = \mathbb{F}_1 \cup \mathbb{F}_2$.

The RECENGINE procedure minimizes the following objective (Hu, Koren, and Volinsky 2008):

$$g(X_{:\mathbb{P}}, Y_{\mathbb{Q}:}) = \sum_{u,i} W_{ui}(R_{ui} - X_{u:}Y_{:i})^2 + \lambda\,(\|X_{:\mathbb{P}}\|^2 + \|Y_{\mathbb{Q}:}\|^2)$$

where $\|\cdot\|$ is the Frobenius norm, λ (≥ 0) is a tuning parameter, and W_{ui} is a weight that indicates the importance of R_{ui}. The weights are further described as follows.

Each weight is defined as $W_{ui} = 1 + \gamma C_{ui}$, where γ (> 0) is a constant normalizer and C_{ui} is the confidence about R_{ui}. For any missing feedback—that is, any (u, i) not available in the training data—C_{ui} is set to 0. For any (u, i) in the training data, if the confidence is not explicitly specified by using the CONFIDENCE= option in the DTOS statement, an optimal calculated feedback confidence is used for all C_{ui}. The value of γ is the value that you specify in the FEEDBACKCONFIDENCESLOPE= option in the DTOS statement.

K-Nearest Neighbors (KNN)

The k-nearest neighbors method in PROC RECENGINE is an implementation of the inverted index base (IIB) k-nearest neighbors (KNN) algorithm (Wang et al. 2010). This method accepts as input a table in which each row is a training sample in the following form: user ID, item ID, and user response (optional). The recommender model that this method generates consists of the k-nearest neighbors for each unique item in the ITEMID statement variable, based on the shared user ID interactions within the item set. If a user response column is provided, the k-nearest neighbors are generated on the basis of the provided explicit user response. If a user response column is not provided, then the user response is assumed to be implicit, and the k-nearest neighbors are generated by treating the user response as equal to 1 for each user-item interaction.

For m users and n items, the KNN method stores the user-item interactions in the $m \times n$ sparse matrix R, where $R_{ui} = 1$ if user u has interacted with item i and $R_{ui} = 0$ otherwise. For each row R_w in R, the k rows most similar to row w are calculated using the IIB algorithm, where similarity ϕ between rows w and v is given by one of the following metrics:

$$\phi(R_w, R_v) = \frac{\sum_{i=1}^{m} R_{wi} R_{vi}}{\|R_w\|_2 \|R_v\|_2} \text{ (cosine similarity)}$$

$$\phi(R_w, R_v) = \frac{2\sum_{i=1}^{m} R_{wi} R_{vi}}{\|R_w\|_2^2 + \|R_v\|_2^2} \text{ (Dice similarity)}$$

The IIB algorithm uses sparse representations of R in order to efficiently identify nearest neighbors for ultra-high-dimensional sparse data.

An optional memory-efficient setting is also available when you are working with extremely large data sets. This setting uses memory-efficient hash tables during nearest-neighbor calculations to lower the memory cost when you run the action. Because hashed structures require increased lookup time, it is recommended that you use the memory-efficient option only when necessary. You can invoke this version of the method by specifying the MEMORYEFFICIENT option in the KNN statement.

Displayed Output

The RECENGINE procedure displays various tables for analysis. The following sections describe the output in the order of its appearance when you specify the related options.

Model Information

The "Model Information" table displays basic information about the parameters that are used in the analysis. This information includes the maximum number of iterations, number of factors, learning step size, and seed value for random number generation.

Number of Observations

The "Number of Observations" table displays the number of observations that are read from the input data table and the number that are used.

Nominal Variables

The "Nominal Variables" table shows the number of levels for the nominal variables.

Iteration History

The "Iteration History" table displays the iteration number and the loss value that is computed in each iteration. When METHOD=BPR, the loss is the approximate value of the loss function. When METHOD=DTOS, the loss is the exact value of the loss function, and the optimal step size is also displayed for each iteration. The "displayed loss" indicates computational efficiency.

OutputCasTables Table

The OutputCasTables table is a special table that has information about each table that is created. The information for each table consists of the table name, the library in which the table resides, and the number of columns and rows in the table. Because this table is not a typical ODS table that contains analytical results, you cannot include it in the *table-spec-list* in the DISPLAYOUT statement.

ODS Table Names

Each table that the RECENGINE procedure creates has a name associated with it. You must use this name to refer to the table when you use ODS statements. The name of each table and a short description of the contents are listed in Table 28.3.

Table 28.3 ODS Tables Produced by PROC RECENGINE

Table Name	Description	Statement	Option
BestConfiguration	Hyperparameters and objective function values for the best configuration	AUTOTUNE	Default
DescStatsNom	Number of levels of each nominal variable	PROC RECENGINE	Default
EvaluationHistory	Values of the hyperparameters and the objective function for all configurations	AUTOTUNE	Default
HyperparameterImportance	Relative importance of hyperparameters	AUTOTUNE	Default
IterationHistory	Tuner iteration history	AUTOTUNE	Default
ModelInfo	Model information	PROC RECENGINE	Default
NObs	Number of observations	PROC RECENGINE	Default
OptIterHistory	Iteration history	PROC RECENGINE	Default
OutputTables	See the section "OutputCasTables Table" on page 744	PROC RECENGINE	Default
TunerCasOutputTables	Tables created during tuning	AUTOTUNE	Default
TunerInfo	Setup values that are used by the tuner	AUTOTUNE	Default
TunerResults	Values of the hyperparameters, the objective function for the default configuration (iteration 0), and up to 10 best configurations found	AUTOTUNE	Default
TunerSummary	Statistics about the tuning process	AUTOTUNE	Default
TunerTiming	Total time spent on different tasks during tuning	AUTOTUNE	Default

Examples: RECENGINE Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

Example 28.1: Recommending SAS Users Community Pages to SAS Users

This example shows how to use the RECENGINE procedure to train a matrix factorization with partially defined factors (MF-PDF) model on the usersCommunity data set (which is available here: **https://support.sas.com/documentation/onlinedoc/viya/examples.htm**). The data set consists of click events over a period of time on a peer-to-peer-support community website. Each web page (viewed as an item in the action) on the site belongs to a message board that includes a number of topics of specific interest. The same user might have viewed the same web page multiple times. Also, in this data set, it is indicated whether or not the questions within the topic have been resolved. For each click event, the user ID and item ID are recorded. The data set includes 40,000 click events (user feedback).

The following code includes the DATA step that generates the data table mylib.usersCommunity.

```
data mylib.usersCommunity;
   input itemid boardid userid isSolvedTopic;
   datalines;
   38 4817 6313 0
   15 7589 1355 1
   46 9560 2926 1
   16 11635 3985 1
   38 13023 4163 0

... more lines ...
```

The following statements use PROC RECENGINE to train an MF-PDF model on the usersCommunity data:

```
proc recengine data=mylib.usersCommunity method=dtos outmodel=mylib.factors;
  input userid itemid /level=nominal;
  input isSolvedTopic/level=interval;
  userid userid;
  itemid itemid;
  dtos nFactors=12 maxIter=20 regL2=0.1
       feedbackConfidenceSlope=50 /*confidence=isSolvedTopic*/;
  savestate rstore=mylib.astoreModel;
run;
```

NOTE: When the option CONFIDENCE=ISSOLVEDTOPIC is commented out, the confidence takes a constant value that is optimally determined by training data. When this option is not commented out, the confidence takes its value from the isSolvedTopic column of the usersCommunity table.

When the training is completed, PROC RECENGINE produces the mylib.factors table, which contains the factors and biases of both items and users.

The following code shows how to find the top five items for a sample user, whose user ID is 2572, by using the built-in analytic store scoring function.

First, you need to create the scoring data table that includes user ID 2572:

```
data mylib.user2572;
input   userid;
datalines;
2572
;
run;
```

Next, you use the ASTORE procedure to score the binary-format model mylib.astoreModel for this user:

```
proc astore;
   setoption nTopRecoms 5;
   score data=mylib.user2572 rstore=mylib.astoreModel out=mylib.rankedItemsDtos;
   run;
quit;
```

The following statements print the top five recommended items for user 2572. The output is shown in Output 28.1.1.

```
proc print data=mylib.rankedItemsDtos;
run;
```

Output 28.1.1 Top Five Items for User 2572

Obs	userid	_RECOMMENDED_itemid	_SCORE	_RANK
1	2572	16	0.71173	1
2	2572	15	0.34203	2
3	2572	38	0.27307	3
4	2572	25	0.22279	4
5	2572	50	0.11351	5

Example 28.2: Running the KNN Method of the Recommender Engine Model on Explicit Feedback

This example shows how to use PROC RECENGINE to train a *k*-nearest-neighbors model on an explicit feedback data set, the MovieLens data set. The MovieLens data set was developed by the GroupLens project at the University of Minnesota and is available at **http://grouplens.org/datasets/movielens**. This example uses the MovieLens 1M version. You can download the compressed archive file from the website at **http://files.grouplens.org/datasets/movielens/ml-1m.zip** and use any third-party unzip tool to extract all the files in the archive to the destination directory of your choice.[1] The file that contains the ratings is *ratings.dat*. The data set consists of movie ratings that are provided by a collection

[1]Disclaimer: SAS may reference other websites or content or resources for use at Customer's sole discretion. SAS has no control over any websites or resources that are provided by companies or persons other than SAS. Customer acknowledges and agrees that SAS is not responsible for the availability or use of any such external sites or resources, and does not endorse any advertising, products, or other materials on or available from such websites or resources. Customer acknowledges and agrees that SAS is not liable for any loss or damage that may be incurred by Customer or its end users as a result of the availability or use of those external

of users. For each observation, a user (indicated by the user ID column) has provided a rating from 1 to 5, as indicated by the rating column for a movie (indicated by the item ID column). The data set includes 1,000,209 rating events (user feedback). The following DATA step loads the data table from the directory into your session:

```
data mylib.movieRatings;
   infile 'ratings.dat' DLMSTR = '::';
   input userid itemid rating timestamp;
run;
```

The following statements use the KNN method that is implemented in PROC RECENGINE to train a *k*-nearest-neighbors model on the mylib.movieRatings data:

```
proc recengine data=mylib.movieRatings method =KNN outModel=mylib.nearestNeighbors;
     input     rating / level=interval;
     input     userid itemid / level=nominal;
     userid    userid;
     itemid    itemid;
     knn       k=5 distanceMetric=cosine outSimilarity=mylib.nearestNeighborValues;
     target    rating;
     savestate rstore=mylib.astoreModel;
run;
```

When the training is completed, PROC RECENGINE produces three output data tables: nearestNeighbors, nearestNeighborValues, and astoreModel. The nearestNeighbors table contains the nearest neighbors for each item; the nearestNeighborValues table contains the similarity scores for each item's nearest neighbors; and the astoreModel table contains the nearest neighbors, similarity scores, and training data.

The following statements print the nearest neighbors of the first 10 items in the nearestNeighbors table. The results are shown in Output 28.2.1.

```
proc print data=mylib.nearestNeighbors(obs=10);
run;
```

sites or resources, or as a result of any reliance placed by Customer or its end users on the completeness, accuracy, or existence of any advertising, products, or other materials on, or available from, such websites or resources.

Output 28.2.1 Nearest Neighbors of the First 10 Items

Obs	Variable	Level	Neighbor1	Neighbor2	Neighbor3	Neighbor4	Neighbor5
1	itemid	1	3114	1265	588	2355	1270
2	itemid	10	1722	1370	733	648	349
3	itemid	100	832	1589	454	225	1404
4	itemid	1000	1470	3065	2955	847	1351
5	itemid	1002	106	821	1850	3228	633
6	itemid	1003	1598	280	22	61	1518
7	itemid	1004	1626	204	1687	1497	1382
8	itemid	1005	2042	2082	374	1021	2953
9	itemid	1006	1003	1672	805	280	350
10	itemid	1007	2016	2095	1010	1011	1009

The following statements print the nearest-neighbor similarity scores of the first 10 items in the nearestNeighborValues table. The results are shown in Output 28.2.2.

```
proc print data=mylib.nearestNeighborValues(obs=10);
run;
```

Output 28.2.2 Nearest-Neighbor Similarity Scores of the First 10 Items

Obs	Variable	Level	Similarity1	Similarity2	Similarity3	Similarity4	Similarity5
1	itemid	1	0.63310	0.61083	0.60585	0.57938	0.57013
2	itemid	10	0.67199	0.57320	0.56666	0.56045	0.55527
3	itemid	100	0.30563	0.29816	0.29713	0.29549	0.28901
4	itemid	1000	0.27280	0.27280	0.23264	0.22796	0.21858
5	itemid	1002	0.37227	0.30984	0.29691	0.28868	0.27031
6	itemid	1003	0.45325	0.38697	0.36452	0.34244	0.33584
7	itemid	1004	0.55141	0.43995	0.39007	0.38128	0.36717
8	itemid	1005	0.67295	0.44130	0.39629	0.38543	0.38217
9	itemid	1006	0.26997	0.26703	0.25305	0.24894	0.24633
10	itemid	1007	0.47901	0.41869	0.39998	0.37202	0.36637

The following two code blocks show how to find the top five items for a sample user, whose user ID is 6040, by using the built-in analytic store scoring function. First, you create the scoring data table that includes the user ID of interest:

```
data mylib.user6040;
input   userid;
datalines;
6040
;
run;
```

Next, you use the ASTORE procedure to score the binary-format output data table astoreModel for this user. In this scoring example, the following parameters are specified in SETOPTION statements:

- **REC_TOP_N** is set to 5, resulting in five recommendations for user 6040.
- **AGGREGATION_METHOD** is set to 0, indicating that shared similarity scores are to be summed.
- **REC_ITEM_FILTER** is set to 1, indicating that the user's recommendations cannot include items that the user has interacted with during model training. This is the default behavior during scoring.

```
proc astore;
   setoption REC_TOP_N            5;
   setoption AGGREGATION_METHOD   0;
   setoption REC_ITEM_FILTER      1;
   score data=mylib.user6040 rstore=mylib.astoreModel out=mylib.rankedItemsKnn;
   run;
quit;
```

The following statements print the five most recommended items for user 6040. The results are shown in Output 28.2.3.

```
proc print data=mylib.rankedItemsKnn;
run;
```

Output 28.2.3 Top Five Recommended Items for User 6040

Obs	userid	_RECOMMENDED_itemid	_SCORE	_RANK
1	6040	2360	5	1
2	6040	942	5	2
3	6040	926	5	3
4	6040	915	5	4
5	6040	911	5	5

Example 28.3: Tuning Hyperparameters of an MF-PDF Model

This example shows how to use the AUTOTUNE statement in the RECENGINE procedure to tune a matrix factorization with partially defined factors (MF-PDF) model on the usersCommunity data set (which is available here: **https://support.sas.com/documentation/onlinedoc/viya/examples.htm**). The data set consists of click events over a period of time on a peer-to-peer-support community website. Each web page (viewed as an item by PROC RECENGINE) on the site belongs to a message board that includes a number of topics of specific interest. The same user might have viewed the same web page multiple times. Also, in this data set, it is indicated whether or not the questions within the topic have been resolved. For each click event, the user ID and item ID are recorded. The data set includes 40,000 click events (user feedback).

The following code includes the DATA step that generates the data table usersCommunity. These statements assume that your libref is named mylib, but you can substitute any appropriately named libref.

```
data mylib.usersCommunity;
   input itemid boardid userid isSolvedTopic;
   datalines;
   38 4817 6313 0
   15 7589 1355 1
   46 9560 2926 1
   16 11635 3985 1
   38 13023 4163 0

... more lines ...
```

The following statements use PROC RECENGINE with the AUTOTUNE statement to tune an MF-PDF model on the usersCommunity data:

```
proc recengine
   data=mylib.userscommunity
   method=dtos
   outmodel=mylib.dtosmodel;
   input userid itemid / level=nominal;
   input issolvedtopic / level=interval;
   userid userid;
   itemid itemid;
   autotune
      /* Tuning Parameters
      You do not need to specify any tuning parameters for the default
      tuning process. If you want to make adjustments to the default
      tuning process, uncomment the following block of code and change
      any of the tuning parameters' attributes.

      tuningparameters=(
         MAXITER                 ( init=10   lb=1     ub=1000   )
         MINNORMOBJCHANGE        ( init=0.01 lb=1E-10 ub=1.0    )
         NFACTORS                ( init=1    lb=1     ub=50     )
         REGL2                   ( init=0.01 lb=0.001 ub=1000   )
         FEEDBACKCONFIDENCESLOPE ( init=50   lb=1E-10 ub=100    )
         NOUSERBIAS
         NOITEMBIAS
      )
      */
   ;
   ods select BestConfiguration; /* Remove to see all results tables */
run;
```

These statements produce the table shown in Output 28.3.1. The table displays the evaluation number, the values of the tuning parameters, and the error metric value for the best MF-PDF model that the tuner found. Note that the ODS SELECT statement limits the displayed results to a single table. You can remove this statement to see all results tables. For the full list of ODS tables that PROC RECENGINE produces, see Table 28.3.

Output 28.3.1 Best Configuration Table

Best Configuration	
Evaluation	50
Maximum Number of Iterations	1000
Minimum Normalized Objective Change	0
Number of Factors	26
L2 Regularization	500.0005
Confidence Slope	100
Has User Bias	TRUE
Has User Bias	TRUE
Mean Average Precision	0.9237302087

Example 28.4: Tuning Hyperparameters of a Bayesian Personalized Ranking Model

This example shows how to use the AUTOTUNE statement in the RECENGINE procedure to tune a Bayesian personalized ranking model on the usersCommunity data set (which is available here: **https://support.sas.com/documentation/onlinedoc/viya/examples.htm**). The data set consists of click events over a period of time on a peer-to-peer-support community website. Each web page (viewed as an item by PROC RECENGINE) on the site belongs to a message board that includes a number of topics of specific interest. The same user might have viewed the same web page multiple times. Also, in this data set, it is indicated whether or not the questions within the topic have been resolved. For each click event, the user ID and item ID are recorded. The data set includes 40,000 click events (user feedback).

The following code includes the DATA step that generates the data table usersCommunity. These statements assume that your libref is named mylib, but you can substitute any appropriately named libref.

```
data mylib.usersCommunity;
   input itemid boardid userid isSolvedTopic;
   datalines;
   38 4817 6313 0
   15 7589 1355 1
   46 9560 2926 1
   16 11635 3985 1
   38 13023 4163 0

... more lines ...
```

The following statements use PROC RECENGINE with the AUTOTUNE statement to tune a Bayesian personalized ranking model on the usersCommunity data:

```
proc recengine
   data=mylib.userscommunity
   method=bpr
   outmodel=mylib.bprmodel;
   input userid itemid / level=nominal;
   input issolvedtopic / level=interval;
   userid userid;
```

```
   itemid itemid;
   autotune
      /* Tuning Parameters
      You do not need to specify any tuning parameters for the default
      tuning process. If you want to make adjustments to the default
      tuning process, uncomment the following block of code and change
      any of the tuning parameters' attributes.

      tuningparameters=(
         MAXITER          ( init=20    lb=10     ub=100   )
         MINIBATCHSIZE    ( init=10000 lb=400    ub=40000 )
         NFACTORS         ( init=10    lb=5      ub=30    )
         LEARNSTEP        ( init=0.1   lb=0.001  ub=0.5   )
         REGULARIZATION   ( init=0.1   lb=0.001  ub=0.5   )
      )
      */
   ;
   ods select BestConfiguration; /* Remove to see all results tables */
run;
```

These statements produce the table shown in Output 28.4.1. The table displays the evaluation number, the values of the tuning parameters, and the error metric value for the best Bayesian personalized ranking model that the tuner found. Note that the ODS SELECT statement limits the displayed results to a single table. You can remove this statement to see all results tables. For the full list of ODS tables that PROC RECENGINE produces, see Table 28.3.

Output 28.4.1 Best Configuration Table

Best Configuration	
Evaluation	34
Maximum Number of Iterations	60
Number of Factors	27
Minibatch Size	40000
Learning Step Size	0.22277778
Regularization	0.33366667
Mean Average Precision	0.8943986667

Example 28.5: Tuning Hyperparameters of a k-Nearest Neighbors Model

This example shows how to use the AUTOTUNE statement in the RECENGINE procedure to tune a k-nearest neighbors model on the usersCommunity data set (which is available here: **https://support.sas.com/documentation/onlinedoc/viya/examples.htm**). The data set consists of click events over a period of time on a peer-to-peer-support community website. Each web page (viewed as an item by PROC RECENGINE) on the site belongs to a message board that includes a number of topics of specific interest. The same user might have viewed the same web page multiple times. Also, in this data set, it is indicated whether or not the questions within the topic have been resolved. For each click event, the user ID and item ID are recorded. The data set includes 40,000 click events (user feedback).

The following code includes the DATA step that generates the data table usersCommunity. These statements assume that your libref is named mylib, but you can substitute any appropriately named libref.

```
data mylib.usersCommunity;
   input itemid boardid userid isSolvedTopic;
   datalines;
   38 4817 6313 0
   15 7589 1355 1
   46 9560 2926 1
   16 11635 3985 1
   38 13023 4163 0

... more lines ...
```

The following statements use PROC RECENGINE with the AUTOTUNE statement to tune a *k*-nearest neighbors model on the usersCommunity data:

```
proc recengine
   data=mylib.userscommunity
   method=knn
   outmodel=mylib.knnmodel;
   input userid itemid / level=nominal;
   input issolvedtopic / level=interval;
   userid userid;
   itemid itemid;
   autotune
      /* Tuning Parameters
      You do not need to specify any tuning parameters for the default
      tuning process. If you want to make adjustments to the default
      tuning process, uncomment the following block of code and change
      any of the tuning parameters' attributes.

      tuningparameters=(
         K ( init=5  lb=1 ub=30 )
         DISTANCEMETRIC
      )
      */
   ;
   /* Remove this line to see all results */
   ods select BestConfiguration EvaluationHistoryPlot IterationHistoryPlot;
run;
```

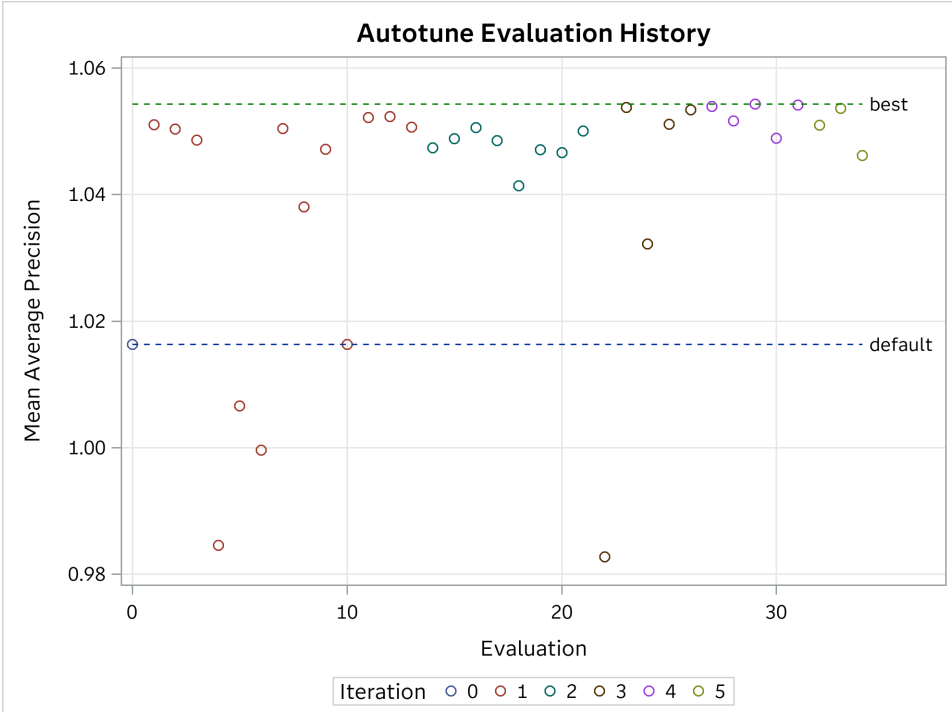
The preceding statements produce the table and the plots shown in Output 28.5.1 through Output 28.5.3. The table in Output 28.5.1 displays the evaluation number, the values of the tuning parameters, and the error metric value for the best *k*-nearest neighbors model that the tuner found. Note that the ODS SELECT statement limits the displayed results to a single table and two plots. You can remove this statement to display all tables and plots. For the full list of ODS tables that PROC RECENGINE produces, see Table 28.3.

Output 28.5.1 Best Configuration Table

Best Configuration	
Evaluation	29
Nearest Neighbors (K)	24
Distance Metric	DICE
Mean Average Precision	1.0542803093

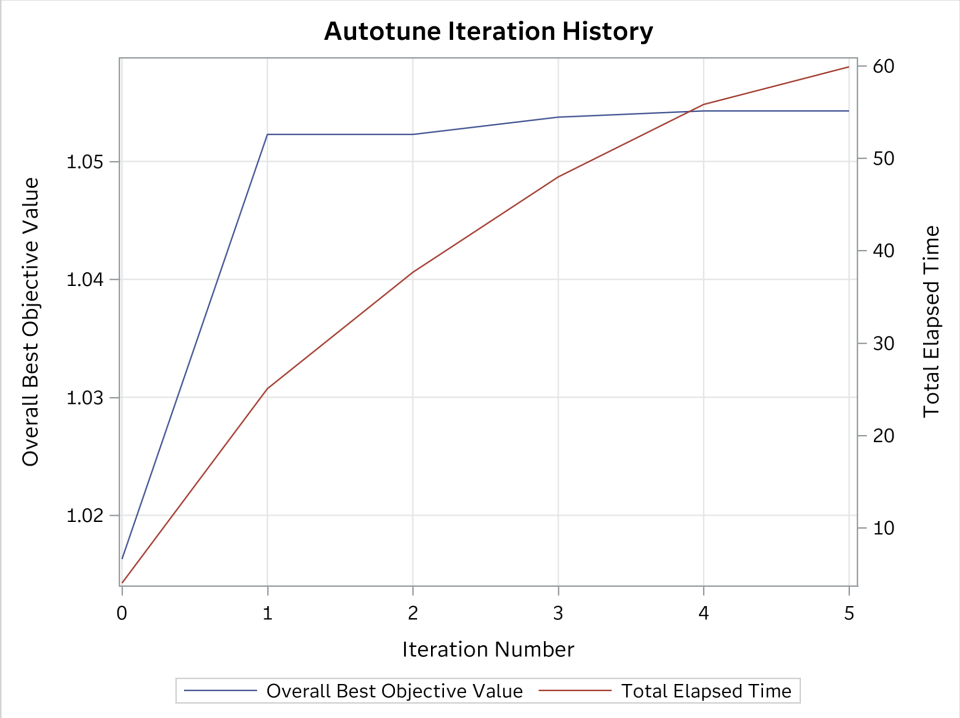
Output 28.5.2 displays a scatter plot of all configurations that the tuner tried. The objective values are shown on the Y axis, and the evaluation numbers are shown on the X axis.

Output 28.5.2 Evaluation History Plot



The plot in Output 28.5.3 displays how the best found objective value and the elapsed time changed with each iteration of the tuner.

Output 28.5.3 Iteration History Plot



References

Hu, Y., Koren, Y., and Volinsky, C. (2008). "Collaborative Filtering for Implicit Feedback Datasets." In *Proceedings of the 2008 Eighth IEEE International Conference on Data Mining*, 263–272. Washington, DC: IEEE Computer Society.

Liao, X., and Koch, P. (2021). *DTOS: Efficient Collaborative Filtering with Implicit Feedback*. Technical report, SAS Institute Inc.

Rendle, S. (2012). "Factorization Machines with libFM." *ACM Transactions on Intelligent Systems and Technology* 3:1–22.

Rendle, S., Freudenthaler, C., Gantner, Z., and Schmidt-Thieme, L. (2012). "BPR: Bayesian Personalized Ranking from Implicit Feedback." https://arxiv.org/abs/1205.2618.

Wang, J., Lin, L., Huang, T., Wang, J., and He, Z. (2010). "Efficient K-Nearest Neighbor Join Algorithms for High Dimensional Sparse Data." https://arxiv.org/abs/1011.2807.

Chapter 29
REGISTERMODEL Procedure

Contents

Overview: REGISTERMODEL Procedure **758**
PROC REGISTERMODEL Features 758
Using SAS Viya Workbench 758
Getting Started: REGISTERMODEL Procedure **759**
Syntax: REGISTERMODEL Procedure **760**
PROC REGISTERMODEL Statement 761
ASSESSMENT Statement 764
ASTOREMODEL Statement 765
CODE Statement 765
PROJECT Statement 765
TARGET Statement 766
Details: REGISTERMODEL Procedure **767**
Required Specifications 767
Input Data Table 767
Project Information 768
Target Information 769
Assessment Requirements 769
Authentication 770
ODS Table Names 772
Examples: REGISTERMODEL Procedure **772**
Example 29.1: Register an Analytic Store with Assessment Information 772
Example 29.2: Register a DATA Step Model with Assessment Information 774
Example 29.3: Register an Analytic Store with Prescored Data 775

Overview: REGISTERMODEL Procedure

The REGISTERMODEL procedure registers DATA step and analytic store models in SAS Model Manager.

PROC REGISTERMODEL Features

PROC REGISTERMODEL provides the following features:

- registers an analytic store model
- registers a DATA step model
- calculates and registers assessment statistics for prediction and classification models

Using SAS Viya Workbench

SAS Viya Workbench has greatly simplified the SAS programming experience by offering a syntax style nearly identical to that of SAS 9 and earlier releases. However, there are minor exceptions. In the past, SAS Viya procedures required the use of a CAS engine libref. This engine served multiple purposes, but it was primarily used to upload data to the CAS server and manage the data. Its additional purposes included support of new data types and improved performance for large data sets. No CAS server is needed for SAS Viya Workbench, because all computations are done in place for you in a single location.

In the following program, the LIBNAME statement creates a mylib libref that the examples in this chapter use to refer to your current or desired work directory:

```
libname mylib ".";
```

For more information, see *SAS Viya Workbench: Getting Started*.

Getting Started: REGISTERMODEL Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

This example illustrates some of the basic features of the REGISTERMODEL procedure by registering a decision tree analytic store model that is trained on the HMEQ data table.

The following DATA step creates the input data table mylib.hmeq in your SAS library. These statements assume that your SAS libref is named mylib, but you can substitute any appropriately defined SAS libref.

```
data mylib.hmeq;
    set sampsio.hmeq;
run;
```

The following statements run the TREESPLIT procedure to build a decision tree to estimate the variable bad, which indicates whether mortgage loan applicants defaulted on repayment of their loan. The procedure also outputs an analytic store named treeStore_hmeq.

```
proc treesplit data = mylib.hmeq;
   input loan value mortdue debtinc derog / level = interval;
   input job / level = nominal;
   target bad / level = nominal;
   prune none;
   saveState rstore = mylib.treeStore_hmeq;
run;
```

The following statements run the ASTORE procedure to download the analytic store treeStore_hmeq as a binary file:

```
proc astore;
   download rstore = mylib.treeStore_hmeq
            store = "treeAstore.sasast";
run;
```

The following statements run PROC REGISTERMODEL to register the analytic store model in SAS Model Manager:

```
proc registermodel
      name = "Tree Astore"
      description = "Decision Tree Astore Model"
      algorithm = TREE
      function = CLASSIFICATION
      server = "http://myserver.com"
      oauthtoken = "myTokenName"
      replace;
   project name="myProject" folder="myFolder";
   astoremodel store = "treeAstore.sasast";
run;
```

The NAME= option specifies the name to register the model under in SAS Model Manager. The DESCRIPTION= option specifies the description of the model. The ALGORITHM= option specifies that the

decision tree algorithm was used to train the model. The FUNCTION= option specifies that the function of the model is classification. The SERVER= option specifies the root URL for your SAS Viya server. The OAUTHTOKEN= option specifies the name of the authentication token. The REPLACE option indicates that you want to replace an existing model if a model that has the same name already exists in the same project.

The PROJECT statement specifies the project to register the model in. The NAME= option specifies the name of the project. The FOLDER= option specifies the folder where the project resides. The ASTOREMODEL statement with the STORE= option specifies the analytic store file to be uploaded to and registered in SAS Model Manager.

Figure 29.1 displays the model registration information.

Figure 29.1 Model Registration Information

Registration Information	
Creation Time	2024-02-19T15:51:07-04:00
Model Name	Tree Astore
Model ID	bc608dfa-b220-447f-a5e3-97d3a297fc03
Project Name	myProject
Project ID	7de86372-654f-4501-a817-2d36a9cb178c
Project Version ID	c4a68493-cdd2-41c3-b09f-9ae886a74f45
Folder ID	fd174b10-b616-4b99-af64-e56b1fca6fdf
Repository ID	f0e58396-8633-4a92-8004-d07b7676283a

Syntax: REGISTERMODEL Procedure

The following statements are available in the REGISTERMODEL procedure:

PROC REGISTERMODEL NAME='*model-name*' **SERVER=**'*server-address*' **OAUTHTOKEN=**'*oAuthToken-name*' < *options* > **;**
ASSESSMENT < *options* > **;**
ASTOREMODEL STORE=*filename* **;**
CODE FILE=*filename* **;**
PROJECT < *options* > **;**
TARGET *variable* / **LEVEL=BINARY** | **INTERVAL** | **NOMINAL** < **EVENT=**'*event*' > **;**

The PROC REGISTERMODEL statement and either the ASTOREMODEL statement or the CODE statement are required. You cannot specify both the ASTOREMODEL and CODE statements. The rest of this section provides detailed syntax information about each of the preceding statements, beginning with the PROC REGISTERMODEL statement. The remaining statements are described in alphabetical order.

PROC REGISTERMODEL Statement

PROC REGISTERMODEL NAME='*model-name*' **SERVER=**'*server-address*' **OAUTHTOKEN=**'*oAuthToken-name*' < *options* > **;**

The PROC REGISTERMODEL statement invokes the procedure. Table 29.1 summarizes the options in this statement.

Table 29.1 PROC REGISTERMODEL Statement Options

Option	Description
ALGORITHM=	Specifies the algorithm that was used to train the model
DATA=	Specifies the name of the input table that was used to train the model
DESCRIPTION=	Specifies the description of the model
FUNCTION=	Specifies the model function type
MODELER=	Specified the name of the modeler
NAME=	Specifies the name of the model
OAUTHTOKEN=	Specifies the name of the authentication token for connecting to SAS Model Manager
REPLACE	Specifies that you want to replace existing models
SERVER=	Specifies the root URL for your SAS Viya server where SAS Model Manager is located

You must specify the following options:

NAME='*model-name*'
specifies the name to register the model under in SAS Model Manager. The '*model-name*' must be a quoted string.

OAUTHTOKEN='*oAuthToken-name*'
specifies the name of the authentication token for connecting to SAS Model Manager. The '*oAuthToken-name*' must be a quoted string.

SERVER='*server-address*'
specifies the root URL for your SAS Viya server where SAS Model Manager is located. The '*server-address*' must be a quoted string.

You can also specify the following *options*:

ALGORITHM=*algorithm-key-value* | '*algorithm-name*'
specifies the algorithm that was used to train the model. This can be specified as a key value from Table 29.2 or as a quoted string. When you specify the algorithm by using one of the listed key values, SAS Model Manager displays the algorithm name that is associated with the key value text in the language of the browser locale. When you specify it as a quoted string that does not match any of the listed key values, the algorithm name is treated as a user-defined value, and the value is not mapped to the display name.

Table 29.2 PROC REGISTERMODEL *algorithm-key-values*

algorithm-key-value	**English Display Name**
BINNING	Binning
BNET	Bayesian network
BOOLRULE	Boolean rules
ENSEMBLE	Ensemble
FACTMAC	Factorization machine
FOREST	Forest
GAM	Generalized additive model
GLM	Generalized linear model
GRADBOOST	Gradient boosting
IMPUTE	Imputation
LINEARREG	Linear regression
LOGISTIC	Logistic regression
MWPCA	Moving window principal component analysis
NEURAL	Neural networks
NPLR	Nonparametric logistic regression
OTHER	Other
PCA	Principal component analysis
PLSREG	Partial least squares regression
QUANTREG	Quantile regression
REPLACE	Replacement
RPCA	Robust principal component analysis
RULEBASED	Rule-based
SVD	SVD
SVDD	Support vector data description
SVM	Support vector machine
TEXTTOPIC	Text topic (discovery)
TREE	Decision tree

DATA=*libref.data-table*

names the input data table for PROC REGISTERMODEL to use. The default is the most recently created data table. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to the data. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 758.

data-table specifies the name of the input data table.

For a DATA step model, you must provide an input data table containing all the input variables that the model uses. You also must provide an input data table if you specify the ASSESSMENT statement.

DESCRIPTION='*model-description*'
specifies the description of the model. The '*model-description*' must be a quoted string.

FUNCTION=*function-key-value*
specifies the model function type. You can specify one of the key values that are listed in Table 29.3, which also lists the corresponding display names. SAS Model Manager displays the function name that is associated with the key value text in the language of the browser locale.

By default, FUNCTION=ANALYTICAL.

Table 29.3 PROC REGISTERMODEL *function-key-values*

function-key-value	**English Display Name**
ANALYTICAL	Analytical
CLASSIFICATION	Classification
CLUSTERING	Clustering
IMAGE_CLASSIFICATION	Computer vision – image classification
IMAGE_SEGMENTATION	Computer vision – image segmentation
KEYPOINTS_DETECTION	Computer vision – keypoints detection
OBJECT_DETECTION	Computer vision – object detection
PREDICTION	Prediction
SEGMENTATION	Segmentation
SINGLE_SERIES_FORECASTING	Single-series forecasting
TEXT_CATEGORIZATION	Text categories
TEXT_CONCEPTS	Text concepts
TEXT_SENTIMENT	Text sentiment
TEXT_TOPICS	Text topics
TRANSFORMATION	Transformation

Only the classification and prediction functions are supported for assessment.

MODELER='*modeler*'
specifies the name of the modeler. The '*modeler*' must be a quoted string.

By default, this option is set to the value of the SYSUSERID automatic macro variable. For more information about the SYSUSERID automatic macro variable, see *SAS Macro Language: Reference*.

REPLACE
specifies that you want to replace an existing model with a new one if the new model would have the same model name in the same project.

ASSESSMENT Statement

ASSESSMENT < *options* > **;**

ASSESS < *options* > **;**

The ASSESSMENT statement enables you to assess the model and register assessment statistics in SAS Model Manager. To enable assessment, the FUNCTION= option value in the PROC REGISTERMODEL statement must be PREDICTION or CLASSIFICATION. You must also specify the TARGET statement and provide the training data table by specifying the DATA= option in the PROC REGISTERMODEL statement.

You can specify the following *options*:

FREQ=*variable*

identifies a numeric variable in the input data table and the holdout data table that contains the frequency of occurrence of each observation. PROC REGISTERMODEL treats each observation as if it appears f times, where f is the value of the frequency *variable* for the observation. If f is not an integer, it is truncated to an integer. If f is less than 1 or missing, the observation is not used in the analysis. When you omit the FREQ= option, each observation is assigned a frequency of 1. This variable is optional in the holdout data table. If the holdout data table does not contain this variable, each observation in the holdout data table is assigned a frequency of 1.

HOLDOUT=*libref.data-table*

names the holdout data table for assessment. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the input data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 758.

NCUTS=10 | 20 | 50 | 100 | 500 | 1000

specifies the number of cuts to use in the ROC calculation. This option is ignored when the level of the target variable is interval.

By default, NCUTS=100.

PARTITIONVAR=*variable***(< TEST=**'*value*' **> < TRAIN=**'*value*' **> < VALIDATE=**'*value*' **>)**

specifies the variable in the input data set to be used to decide whether an observation is used for one of the three roles: training, validation, or testing. You must specify at least one of the roles and its corresponding *value*. An observation is used for a role only if the value of the *variable* equals the *value* that you specify for that role. For example, if you specify TRAIN='*value*', then the observation is used for training only if the value of *variable* equals *value*. This variable is ignored in the holdout data table.

ASTOREMODEL Statement

ASTOREMODEL STORE=*filename* **;**

The ASTOREMODEL statement names an analytic store to be registered. For more information about analytic stores, see Chapter 4, "ASTORE Procedure."

You must specify the following option:

STORE=*filename*
names the external file that contains the analytic store that holds the model to be registered. When you enclose the *filename* in a quoted string (for example, FILE="c:\mydir\astore.sasast"), this option specifies the path and file name for reading the code from an external file. You can also specify an unquoted *filename* of no more than eight characters. If the *filename* is assigned as a fileref in a Base SAS FILENAME statement, the file that is specified in the FILENAME statement is used. If you have an analytic store saved in a table, you can create a file by using the DOWNLOAD statement in PROC ASTORE to download the table.

CODE Statement

CODE FILE=*filename* **;**

The CODE statement names the DATA step code to be registered.

You must specify the following option:

FILE=*filename*
names the external file that contains the DATA step code to be registered. When you enclose the *filename* in a quoted string (for example, FILE="c:\mydir\scorecode.sas"), this option specifies the path and file name for reading the code from an external file. You can also specify an unquoted *filename* of no more than eight characters. If the *filename* is assigned as a fileref in a Base SAS FILENAME statement, the file that is specified in the FILENAME statement is used.

PROJECT Statement

PROJECT < *options* > **;**

The PROJECT statement specifies the project in which to register the model.

You can specify the following *options*:

FOLDER='*folder-name*'
specifies the folder name where the SAS Model Manager project resides. The '*folder-name*' must be a quoted string.

ID='*project-ID*'
specifies the ID of an existing SAS Model Manager project to register the model in. The FOLDER=, NAME=, and REPOSITORY= options in the PROJECT statement are ignored if you specify the ID= option. The '*project-ID*' must be a quoted string.

NAME='*project-name*'
specifies the name of the SAS Model Manager project to register the model in. The '*project-name*' must be a quoted string.

REPOSITORY='*repository-name*'
specifies the name of the repository where the SAS Model Manager project folder resides. The '*repository-name*' must be a quoted string.

By default, REPOSITORY='PUBLIC'.

TARGET Statement

TARGET *variable* / **LEVEL=BINARY | INTERVAL | NOMINAL** < **EVENT=**'*event*' > **;**

The TARGET statement names the *variable* whose values the model tries to predict. You must specify a TARGET statement if you specify the ASSESSMENT statement.

You must specify the following option:

LEVEL=BINARY | INTERVAL | NOMINAL
specifies the level of measurement of the target variable. You can specify the following values:

BINARY	specifies that the measurement level is binary.
INTERVAL	specifies that the measurement level is interval.
NOMINAL	specifies that the measurement level is nominal.

If LEVEL=BINARY or NOMINAL, then you must also specify the following option:

EVENT='*event*'
specifies the event level of the target. If the target variable is formatted, you must specify the formatted event level. The '*event*' must be a quoted string.

Details: REGISTERMODEL Procedure

Required Specifications

The following specifications are required in order to run the REGISTERMODEL procedure:

- A model name to register the model under in SAS Model Manager.
- A model file that contains either an analytic store or DATA step code.
- If the model file contains DATA step code, then an input data table that contains all the input variables is required. Typically, this is the data table that is used to train the model. If you do not specify the ASSESSMENT statement, then PROC REGISTERMODEL uses only the first row of the data table to gather information about the variables in the model. For more information about data requirements for assessment, see the sections "Input Data Table" on page 767 and "Assessment Requirements" on page 769.
- The root URL for your SAS Viya server where SAS Model Manager is located. For more information about the URL, see the section "Authentication" on page 770.
- An authentication token for your SAS Viya server where SAS Model Manager is located. For more information about how to get the authentication token, see the section "Authentication" on page 770.

Input Data Table

The DATA= option in the PROC REGISTERMODEL statement specifies the input data table. This option is required if the model consists of DATA step code or if you specify the ASSESSMENT statement. In most cases, this input data table should be the same input data table that you use to train the model. The procedure uses the input data table in the following ways:

- *If the model consists of DATA step code, and you omit the ASSESSMENT statement*: The input data table must include all the input variables that the model uses. PROC REGISTERMODEL uses the first row of the data table to get information about variables that the model uses.
- *If the model consists of DATA step code, and you specify the ASSESSMENT statement*: The input data table should be either the original training data table or a prescored version of that data table.
  - If you use the original training data table, the table must include all the input variables and the target variable that the model uses. PROC REGISTERMODEL uses the entire data table to get information about variables that the model uses, scores the input data table by using the model that you specify, and uses the scored results to calculate assessment statistics.
  - If you use a prescored data table, the table must include all the input variables, the target variable, and all the output prediction variables that the model calculates. PROC REGISTERMODEL uses the entire data table to get information about variables that the model uses and to calculate assessment statistics. It does not score the input table in this case.

- *If the model is an analytic store, and you omit the ASSESSMENT statement*: The input data table is not used.
- *If the model is an analytic store, and you specify the ASSESSMENT statement*: The input data table should be either the original training data table or a prescored version of that table.
  - If you use the original training data table, the table must include all the input variables and the target variable that the model uses. PROC REGISTERMODEL scores the input data table by using the model that you specify, and it uses the scored results to calculate assessment statistics.
  - If you use a prescored data table, the table must include the target variable that the model uses and all the prediction variables that the model outputs. PROC REGISTERMODEL uses the entire data table to calculate assessment statistics. It does not score the input table in this case.

If you specify the FREQ= or PARTITIONVAR= option in the ASSESSMENT statement, the input data table must also include the variable(s) that you specify. If you want to send additional assessment statistics that are calculated by using a holdout data table, you can specify the HOLDOUT= option in the ASSESSMENT statement. For more information about the requirements for the holdout data table, see the section "Assessment Requirements" on page 769.

Project Information

PROC REGISTERMODEL registers the model that you specify in a project in SAS Model Manager. If you want to register the model in an existing project, you can provide the project ID by specifying the ID= option in the PROJECT statement. You can find the project ID in SAS Model Manager, in the General Properties of the project, listed under UUID. When you specify a project ID, the NAME=, FOLDER=, and REPOSITORY= options in the PROJECT statement are ignored. For more information about SAS Model Manager projects, see *SAS Model Manager: User's Guide*.

If you do not specify a project ID, PROC REGISTERMODEL locates the project by using the project name, repository name, and folder name. If you do not specify a project name, the procedure assigns the project name to be the same as the model name. If you do not specify a folder name, it assigns a folder name that matches the project name. It first searches for a project under the project name in */Model Repositories/<repository name>/<folder name>*. If a project with the project name that you specify exists in that location, PROC REGISTERMODEL registers the model in this existing project. If such a project cannot be found, it first creates a new project at */Model Repositories/<repository name>/<folder name>* and then registers the model in the new project. By default, the repository name is PUBLIC.

NOTE: Only SAS administrators and other authorized users can create a new repository.

Target Information

The TARGET statement is optional. However, if you specify the ASSESSMENT statement, then you must also specify the TARGET statement. If you specify the TARGET statement, PROC REGISTERMODEL sends the following target-related information to SAS Model Manager: the target variable name, target measurement level, and output prediction variable name. If the target measurement level is BINARY or NOMINAL, it also sends the output event probability variable name to SAS Model Manager.

For binary and nominal targets, you must specify the event level by using the EVENT= option. PROC REGISTERMODEL assumes that the model outputs a probability variable for each target level. If the model that you want to register does not output a probability variable for each level, you can omit the TARGET statement. Note that in this case you cannot assess the model. You can add the target information to the Properties tab of the model in SAS Model Manager after the model is registered. Some procedures have options that enable you to generate probability variables for all target levels. For example, in the GENSELECT and LOGSELECT procedures, you can set the PCATALL option in the CODE statement to generate probabilities for all levels of categorical targets.

Assessment Requirements

The ASSESSMENT statement is optional. However, if you specify this statement, then you must also specify the DATA= option in the PROC REGISTERMODEL statement and specify the TARGET statement. The procedure supports assessment only of classification and prediction models.

When you specify the ASSESSMENT statement, the procedure runs an assessment on the input data table that you specify in the DATA= option in the PROC REGISTERMODEL statement, and then it sends the assessment statistics to SAS Model Manager. If you specify a holdout data table by using the HOLDOUT= option in the ASSESSMENT statement, the procedure also runs an assessment on the holdout data table and sends the assessment statistics to SAS Model Manager. The PARTITIONVAR= option applies only to the input data table and not to the holdout data table. The FREQ= option applies to both the input data table and the holdout data table. However, the frequency variable is optional in the holdout data table. If the holdout data table does not contain this variable, each observation in the holdout data table is assigned a frequency of 1.

Typically, the input data table should be the original training input data table. Both the input data table and the holdout data table should include all the input variables and the target variable that the model uses. Alternatively, you can use the scored training data as the input data table or a prescored holdout data table. In this case, the procedure does not score the prescored data table(s), and it calculates assessment statistics directly on the basis of the output prediction variables already in the data table. If you use a prescored data table, it must include the following variables:

- The target variable
- All the output prediction variables that the model outputs. For binary and nominal targets, the model must output a probability variable for each target level. For more information, see the section "Target Information" on page 769.

- If the model consists of DATA step code, the prescored input data table must also include all the input variables that the model uses. The prescored holdout data table does not need to include the input variables.

Authentication

In order to connect to SAS Model Manager, your SAS Viya server where SAS Model Manager is located must have the IP address of your SAS Viya Workbench in its allowed IP ranges. If you need to change the allowed IP ranges, please contact your SAS administrator. Additionally, PROC REGISTERMODEL requires the root URL for your SAS Viya server where SAS Model Manager is located and the name of the authentication token that allows authentication in SAS Model Manager.

You can retrieve the root URL by using PROC OPTIONS or the GETOPTION function in SAS Studio from the same SAS Viya deployment where SAS Model Manager is located. For example, the following statements print the root URL to the SAS log by using PROC OPTIONS:

```
proc options option = SERVICESBASEURL;
run;
```

For more information about PROC OPTIONS, see *Base SAS Procedures Guide*.

The following statements save the root URL to a macro variable named myServerURL and prints its value to the log:

```
%let myServerURL=%sysfunc(getoption(SERVICESBASEURL));
%put &=myServerURL;
```

You can use the SAS Viya platform command-line interface (CLI) to obtain the authentication token. For more information about the SAS Viya platform CLI, see *SAS Viya Platform: Using the Command-Line Interface*. You can follow these steps from SAS Viya Workbench:

1. From the Visual Studio Code interface, open a terminal window. You can use one of the following methods:
   - Use the keyboard shortcut **Ctrl+Shift+`**.
   - Click the menu icon ☰ at the top left of the window, select **Terminal**, and then select **New Terminal**.
2. Create a SAS Viya profile:
   a) In the terminal window, enter the following command to initialize a default profile:
      ```
      sas-viya profile init
      ```
   b) When prompted for the service endpoint, specify the root URL for your SAS Viya server.
   c) When prompted for the output type, follow the prompt to specify your preferred format.
   d) When prompted for the ANSI colored output, follow the prompt to specify whether to enable colored output.
3. Sign in to the SAS Viya server and obtain an access token:

a) In the terminal window, enter the following command:

```
sas-viya -k auth loginCode
```

b) Follow the prompt to open the link in a web browser. You can press **Ctrl** and click the link to open it in a new browser tab.

c) Sign in to the SAS Viya server by using your user name and password.

d) Copy the authorization code, paste it into the terminal window in SAS Viya Workbench, and click **Enter** to continue.

You should see a message in the terminal that indicates that the login was successful and that the token has been saved.

4. Define a SAS macro that contains the token:

a) In the terminal window, enter the following command to print the access token string:

```
sas-viya profile show -key access-token
```

b) Open a new SAS code file as follows:

i. Click the menu icon ☰ at the top left of the window, select **File**, and then select **New Text File**.

ii. Click **Select a language**, and choose **SAS (sas)** as the language.

c) In the SAS code file, copy and paste the following code, and replace the **<access-token-string>** with the access token string from the terminal window:

```
%macro myTokenName() / secure;
"Bearer <access-token-string>"
%mend;
```

d) Run the SAS macro code to define the macro myTokenName.

You can now use myTokenName as the token name in PROC REGISTERMODEL. You can also replace the macro name myTokenName with another name that you prefer.

When you use the token, if you encounter SSL certificate issues, you might need to set the SSLREQCERT= environment variable to “ALLOW” as follows before you run PROC REGISTERMODEL:

```
options set=SSLREQCERT="allow";
```

ODS Table Names

Each table that PROC REGISTERMODEL creates has a name that is associated with it. You must use this name to refer to the table when you use ODS statements. The name of the table and a short description of the contents are listed in Table 29.4.

Table 29.4 ODS Table Produced by PROC REGISTERMODEL

Table Name	Description	Statement	Option
RegisterInfo	Registration information	PROC REGISTERMODEL	Default

Examples: REGISTERMODEL Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

Example 29.1: Register an Analytic Store with Assessment Information

This example shows how to use the REGISTERMODEL procedure to register an analytic store model together with information about its assessment.

The following DATA step creates the input data table mylib.hmeq in your SAS library. These statements assume that your SAS libref is named mylib, but you can substitute any appropriately defined SAS libref.

```
data mylib.hmeq;
    set sampsio.hmeq;
run;
```

The following statements run the TREESPLIT procedure to build a decision tree to estimate the variable bad, which indicates whether the mortgage loan applicants defaulted on repayment of their loan. The procedure also outputs an analytic store named treeStore_hmeq.

```
proc treesplit data = mylib.hmeq;
   input loan value mortdue debtinc derog / level = interval;
   input job / level = nominal;
   target bad / level = nominal;
   prune none;
   saveState rstore = mylib.treeStore_hmeq;
run;
```

The following statements run the ASTORE procedure to download the analytic store treeStore_hmeq as a binary file:

```
proc astore;
   download rstore = mylib.treeStore_hmeq
            store = "treeAstore.sasast";
run;
```

The following statements run PROC REGISTERMODEL to register the analytic store model in SAS Model Manager:

```
proc registermodel
      name = "Tree Astore"
      description = "Decision Tree Astore Model"
      data = mylib.hmeq
      algorithm = TREE
      function = CLASSIFICATION
      server = "http://myserver.com"
      oauthtoken = "myTokenName"
      replace;
   project name="myProject" folder="myFolder";
   astoremodel store = "treeAstore.sasast";
   target bad / level=binary event="1";
   assessment;
run;
```

The NAME= option specifies the name to register the model under in SAS Model Manager. The DESCRIPTION= option specifies the description of the model. The DATA= option specifies the data table to use for assessment. The ALGORITHM= option specifies that the decision tree algorithm was used to train the model. The FUNCTION= option specifies that the function of the model is classification. The SERVER= option specifies the root URL for your SAS Viya server. The OAUTHTOKEN= option specifies the name of the authentication token. The REPLACE option indicates that you want to replace an existing model if a model that has the same name already exists in the same project.

The PROJECT statement specifies the project to register the model in. The NAME= option specifies the name of the project. The FOLDER= option specifies the folder where the project resides. The ASTOREMODEL statement with the STORE= option specifies the analytic store file to be uploaded to and registered in SAS Model Manager.

The TARGET statement specifies that the bad variable was used as the target variable to train the model. The LEVEL= option specifies that the measurement level of the target variable is BINARY. The EVENT= option specifies that the value 1 be used as the event level. The ASSESSMENT statement specifies that you want to calculate the assessment statistics and register the assessment information in SAS Model Manager.

Output 29.1.1 displays the model registration information.

Output 29.1.1 Model Registration Information

Registration Information	
Creation Time	2024-02-19T15:52:42-04:00
Model Name	Tree Astore
Model ID	59d8a25b-3059-479d-b52f-9055d457ea08
Project Name	myProject
Project ID	7de86372-654f-4501-a817-2d36a9cb178c
Project Version ID	c4a68493-cdd2-41c3-b09f-9ae886a74f45
Folder ID	fd174b10-b616-4b99-af64-e56b1fca6fdf
Repository ID	f0e58396-8633-4a92-8004-d07b7676283a

Example 29.2: Register a DATA Step Model with Assessment Information

This example shows how to use the REGISTERMODEL procedure to register a DATA step model together with information about its assessment.

The following DATA step creates the input data table mylib.hmeq in your SAS library. These statements assume that your SAS libref is named mylib, but you can substitute any appropriately defined SAS libref.

```
data mylib.hmeq;
    set sampsio.hmeq;
run;
```

The following statements run the LOGISTIC procedure to build a logistic regression model to estimate the variable bad, which indicates whether the mortgage loan applicants defaulted on repayment of their loan. The procedure also outputs a DATA step code file named *logisticCode.sas*.

```
proc logistic data = mylib.hmeq;
   class job;
   model bad = loan value mortdue debtinc derog job;
   code file="logisticCode_hmeq.sas";
run;
```

The following statements run PROC REGISTERMODEL to register the DATA step model in SAS Model Manager:

```
proc registermodel
      name = "Logistic DATA Step"
      description = "Logistic DATA Step Code Model"
      data = mylib.hmeq
      algorithm = LOGISTIC
      function = CLASSIFICATION
      server = "http://myserver.com"
      oauthtoken = "myTokenName"
      replace;
   project name="myProject" folder="myFolder";
   code file = "logisticCode_hmeq.sas";
   target bad / level=binary event="1";
   assessment;
run;
```

The NAME= option specifies the name to register the model under in SAS Model Manager. The DESCRIPTION= option specifies the description of the model. The DATA= option specifies the data table to use for input variable registration and assessment. The ALGORITHM= option specifies that the logistic regression algorithm was used to train the model. The FUNCTION= option specifies that the function of the model is classification. The SERVER= option specifies the root URL for your SAS Viya server. The OAUTHTOKEN= option specifies the name of the authentication token. The REPLACE option indicates that you want to replace an existing model if a model that has the same name already exists in the same project.

The PROJECT statement specifies the project to register the model in. The NAME= option specifies the name of the project. The FOLDER= option specifies the folder where the project resides. The CODE statement with the FILE= option specifies the DATA step code file to be uploaded to and registered in SAS Model Manager.

The TARGET statement specifies that the bad variable was used as the target variable to train the model. The LEVEL= option specifies that the measurement level of the target variable is BINARY. The EVENT= option specifies that the value 1 be used as the event level. The ASSESSMENT statement specifies that you want to calculate the assessment statistics and register the assessment information in SAS Model Manager.

Output 29.2.1 displays the model registration information.

Output 29.2.1 Model Registration Information

Registration Information	
Creation Time	2024-02-19T15:53:37-04:00
Model Name	Logistic DATA Step
Model ID	0035e2dd-9a67-41f0-87a9-da57f87782ac
Project Name	myProject
Project ID	7de86372-654f-4501-a817-2d36a9cb178c
Project Version ID	c4a68493-cdd2-41c3-b09f-9ae886a74f45
Folder ID	fd174b10-b616-4b99-af64-e56b1fca6fdf
Repository ID	f0e58396-8633-4a92-8004-d07b7676283a

Example 29.3: Register an Analytic Store with Prescored Data

This example shows how to use the REGISTERMODEL procedure to register an analytic store model by using prescored data with a partitioning variable for assessment.

The following DATA step creates the data table mylib.hmeq in your SAS library. These statements assume that your SAS libref is named mylib, but you can substitute any appropriately defined SAS libref. The statements also replace the missing values of the REASON variable with a new value, Unknown.

```
data mylib.hmeq;
    set sampsio.hmeq;
    if reason = "" then reason = "Unknown";
run;
```

The following statements run the PARTITION procedure to create the partitioned data table mylib.hmeq_part. These statements divide the mylib.hmeq data table into three partitions and save the partition index variable _PartInd_ in the mylib.hmeq_part data table.

```
proc partition
      data = mylib.hmeq
      samppct = 60
      samppct2 = 10
      seed = 123
      nThreads = 1
      partind;
   output out = mylib.hmeq_part;
run;
```

The following statements run the FOREST procedure to build a forest to predict the variable reason, which represents the reason for the mortgage loan application. The procedure uses the partition variable _PartInd_ to indicate the training, validation, and testing partitions. It also outputs an analytic store named mylib.forestStore_hmeq and a scored output table named mylib.scored_hmeq. The target variable and the partition variables are both copied to the scored output table.

```
proc forest data = mylib.hmeq_part seed = 123;
  input loan value mortdue debtinc derog / level = interval;
  input job / level = nominal;
  target reason / level = nominal;
  saveState rstore = mylib.forestStore_hmeq;
  partition rolevar = _PartInd_(train = '1' validate = '0' test = '2');
  output out = mylib.scored_hmeq copyvars = (reason _PartInd_);
run;
```

The following statements run the ASTORE procedure to download the analytic store mylib.forestStore_hmeq as a binary file:

```
proc astore;
   download rstore = mylib.forestStore_hmeq
            store = "forestAstore.sasast";
run;
```

The following statements run PROC REGISTERMODEL to register the analytic store model in SAS Model Manager and use the scored output table for assessment:

```
proc registermodel
      name = "Forest Astore"
      description = "Forest Astore Model"
      data = mylib.scored_hmeq
      algorithm = FOREST
      function = CLASSIFICATION
      server = "http://myserver.com"
      oauthtoken = "myTokenName"
      replace;
   project name="myProject" folder="myFolder";
   astoremodel store = "forestAstore.sasast";
   target reason / level=nominal event="HomeImp";
   assessment partitionvar=_PartInd_(train="0" valid="1" test="2");
run;
```

The NAME= option specifies the name to register the model under in SAS Model Manager. The DESCRIPTION= option specifies the description of the model. The DATA= option specifies the data table to use for assessment. The ALGORITHM= option specifies that the forest algorithm was used to train the model.

The FUNCTION= option specifies that the function of the model is classification. The SERVER= option specifies the root URL for your SAS Viya server. The OAUTHTOKEN= option specifies the name of the authentication token. The REPLACE option indicates that you want to replace an existing model if a model that has the same name already exists in the same project.

The PROJECT statement specifies the project to register the model in. The NAME= option specifies the name of the project. The FOLDER= option specifies the folder where the project resides. The ASTOREMODEL statement with the STORE= option specifies the analytic store file to be uploaded to and registered in SAS Model Manager.

The TARGET statement specifies that the reason variable was used as the target variable to train the model. The LEVEL= option specifies that the measurement level of the target variable is NOMINAL. The EVENT= option specifies that the value HomeImp (for "home improvements") be used as the event level. The ASSESSMENT statement specifies that you want to calculate the assessment statistics and register the assessment information in SAS Model Manager. The PARTITIONVAR= option specifies that the _PartInd_ variable be used as the partition variable in assessment; the variable value of 1 indicates the training partition, 0 indicates the validation partition, and 2 indicates the testing partition.

Output 29.3.1 displays the model registration information.

Output 29.3.1 Model Registration Information

Registration Information	
Creation Time	2024-02-19T15:54:32-04:00
Model Name	Forest Astore
Model ID	96908bf3-6b9b-44f2-8ad2-a758810fdf97
Project Name	myProject
Project ID	7de86372-654f-4501-a817-2d36a9cb178c
Project Version ID	c4a68493-cdd2-41c3-b09f-9ae886a74f45
Folder ID	fd174b10-b616-4b99-af64-e56b1fca6fdf
Repository ID	f0e58396-8633-4a92-8004-d07b7676283a

Chapter 30
RPCA Procedure

Contents

Overview: RPCA Procedure **780**
PROC RPCA Features 780
Using SAS Viya Workbench 781
Getting Started: RPCA Procedure **781**
Syntax: RPCA Procedure **785**
PROC RPCA Statement 785
ANOMALYDETECTION Statement 789
CODE Statement 790
DISPLAY Statement 791
DISPLAYOUT Statement 792
FREQ Statement 792
ID Statement 793
IMAGE Statement 793
INPUT Statement 793
OUTDECOMP Statement 793
SAVESTATE Statement 794
SVD Statement 795
Details: RPCA Procedure **795**
Augmented Lagrange Multiplier Method 796
Accelerated Proximal Gradient Method 796
Scaling and Centering 797
Cumulative Eigenvalue Percentage Tolerance 797
Anomaly Detection 797
Displayed Output 799
ODS Table Names 799
Output Data Tables 800
Examples: RPCA Procedure **801**
Example 30.1: Analyzing the Energy Output of the SAS Solar Farm 801
Example 30.2: Detecting Anomalies 804
References **804**

Overview: RPCA Procedure

The RPCA procedure implements robust principal component analysis (RPCA) in SAS Viya Workbench. RPCA can be used in many areas, including image processing, latent semantic indexing, ranking, and matrix completion (Candès et al. 2011).

The RPCA procedure decomposes an input matrix into a sum of two matrices: a low-rank matrix and a sparse matrix. You can use the low-rank matrix to do feature extraction and use the sparse matrix to detect anomalies.

Robustness in RPCA comes from the property that the principal components are computed from observations after removing the outliers—that is, from the low-rank matrix. Many applications of RPCA focus on the sparse matrix. One example is the extraction of moving objects from the background in surveillance videos.

PROC RPCA stores the results of the input matrix decomposition in output tables that are produced by the OUTLOWRANK=, OUTSPARSE=, and OUTERROR= options. Also, PROC RPCA stores the results of the low-rank matrix decomposition in output tables that are produced by the OUTDECOMP statement.

PROC RPCA Features

The RPCA procedure has the following features:

- is multithreaded during all phases of analytic execution
- supports large-scale training data
- enables you to do the analysis on selected columns of the data
- enables you to generate multiple decomposition outputs
- enables you to project new data on principal components as well as to project new data in the low-rank space
- enables you to do anomaly detection
- enables you to input and process image data

Using SAS Viya Workbench

SAS Viya Workbench has greatly simplified the SAS programming experience by offering a syntax style nearly identical to that of SAS 9 and earlier releases. However, there are minor exceptions. In the past, SAS Viya procedures required the use of a CAS engine libref. This engine served multiple purposes, but it was primarily used to upload data to the CAS server and manage the data. Its additional purposes included support of new data types and improved performance for large data sets. No CAS server is needed for SAS Viya Workbench, because all computations are done in place for you in a single location.

In the following program, the LIBNAME statement creates a mylib libref that the examples in this chapter use to refer to your current or desired work directory:

```
libname mylib ".";
```

For more information, see *SAS Viya Workbench: Getting Started*.

Getting Started: RPCA Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

This example shows how to use the RPCA procedure to decompose the input data set into a low-rank matrix and a sparse matrix. It also demonstrates how to decompose the low-rank matrix to obtain the principal components. In this example, the input data set rpcaData has 50 observations and three variables: Index, X, and Y. Index is simply an observation number. X is a randomly generated number between 0 and 1. Y is X plus a random term between −0.1 and 0.1. This data set also has three outliers for variable Y in observations 7, 20, and 33.

The following DATA step creates the rpcaData data set:

```
data rpcaData;
input index X Y;
datalines;
1    0.522    0.510
2    0.642    0.583
3    0.628    0.543
4    0.826    0.875
5    0.101    0.031
6    0.310    0.311
7    0.447    4.421
8    0.419    0.481
9    0.861    0.874
10    0.418    0.334
11    0.929    1.020
12    0.946    0.946
13    0.548    0.567
14    0.626    0.643
15    0.616    0.581
```

```
16    0.684    0.622
17    0.438    0.450
18    0.264    0.174
19    0.705    0.607
20    0.932    3.024
21    0.866    0.836
22    0.145    0.138
23    0.225    0.133
24    0.577    0.515
25    0.815    0.832
26    0.678    0.706
27    0.844    0.892
28    0.996    1.033
29    0.695    0.676
30    0.988    0.930
31    0.684    0.614
32    0.582    0.660
33    0.048    2.004
34    0.230    0.276
35    0.928    0.885
36    0.509    0.539
37    0.187    0.224
38    0.339    0.405
39    0.726    0.782
40    0.459    0.463
41    0.938    1.038
42    0.435    0.451
43    0.520    0.535
44    0.051    0.146
45    0.951    1.017
46    0.173    0.199
47    0.164    0.119
48    0.773    0.834
49    0.428    0.336
50    0.864    0.918
;
run;
```

You can load rpcaData into your SAS library by using your libref in the first statement of the following DATA step:

```
data mylib.rpcaData;
set rpcaData;
run;
```

The following code runs PROC RPCA and outputs the decomposition results:

```
proc rpca data=mylib.rpcaData
          decomp=svd
          outlowrank=mylib.lowrankmat
          outsparse=mylib.sparsemat;
id index;
outdecomp svdleft=mylib.svdleft
          svddiag=mylib.svddiag
          svdright=mylib.svdright;
```

```
run;
```

These statements produce the mylib.lowrankmat and mylib.sparsemat tables, which are the decompositions of mylib.rpcaData. They also produce the mylib.svdleft, mylib.svddiag, and mylib.svdright tables, which are the decompositions of the mylib.lowrankmat table. The DECOMP= option produces the SVD decomposition of the low-rank matrix.

Figure 30.1 ODS Tables

Model Information	
Data Source	RPCADATA
RPCA Method	Augmented Lagrange Multiplier
SVD Method	Eigenvalue Decomposition
Lambda	0.1414214
Lambda Weight	1
Cumulative Eigenvalue Percent Tolerance Specified	1
Cumulative Eigenvalue Percent Tolerance Applied	1

Number of Observations	50
Number of Variables	2
Number of Observations Used	50
Number of Observations with Missing Values	0

Results Summary	
Sparsity of Sparse Matrix	0.1600
Rank of Low-Rank Matrix	1
Number of Iterations	29
Solution Status	Optimal
Run Time (Seconds)	0.18
Total SVD Time (Seconds)	0.11

Figure 30.1 displays the "Model Information," "Dimensions," and "Results Summary" tables. The "Model Information" table shows the default parameters for the RPCA method, the SVD method, Lambda, and LambdaWeight. The "Dimensions" table shows the number of observations and variables in the input table and the number of observations that have missing values. (PROC RPCA ignores the observations that have missing values.) The "Results Summary" table shows that the solution status is optimal; that is, the RPCA algorithm converged based on the tolerance value (this example uses the default value of 10^{-7}) within the maximum number of iterations (this example uses the default value of 1,000). In the "Results Summary" table, you can also see that the algorithm converged after 29 iterations. Furthermore, you can observe in this table that the rank of the low-rank matrix is 1, as expected from the fact that variable Y is highly correlated with variable X. Note that the sparsity value is 0.16, which indicates that the sparse matrix contains many nonzero values. As you increase the value of the LambdaWeight parameter, the sparsity of the sparse matrix increases.

The default value of the LambdaWeight parameter is 1. You can use the following PROC RPCA statement to change this parameter to 3.5. Figure 30.2 shows that the sparsity of the sparse matrix increases to 0.97 and the rank of the low-rank matrix increases to 2.

```
proc rpca data=mylib.rpcaData
          lambdaweight = 3.5
          outsparse=mylib.sparsemat2;
id index;
run;
```

```
proc print data=mylib.sparsemat2;
run;
```

Figure 30.2 Results Summary When LambdaWeight Is 3.5

Results Summary	
Sparsity of Sparse Matrix	0.9700
Rank of Low-Rank Matrix	2
Number of Iterations	17
Solution Status	Optimal
Run Time (Seconds)	0.10
Total SVD Time (Seconds)	0.06

If you look at mylib.sparsemat2, you can see that the only nonzero values in the sparse matrix are related to the outlier values that were introduced for variable Y (in observations 7, 20, and 33).

Alternatively, you can use the ANOMALYDETECTION statement in PROC RPCA and then use the ASTORE procedure to detect the outliers:

```
proc rpca data=mylib.rpcaData
          scale center;
id index;
anomalydetection;
savestate rstore=mylib.store;
run;

proc astore;
setoption rpca_projection_type 2;
score rstore=mylib.store data=mylib.rpcaData out=mylib.scored;
run;

proc print data=mylib.scored;
run;
```

If you look at mylib.scored, you can see that the same outliers (observations 7, 20, and 33) are detected. The two methods that are described in this section are not exactly the same, but with proper settings they both detect outliers.

Syntax: RPCA Procedure

The following statements are available in the RPCA procedure:

```
PROC RPCA <options> ;
   ID variables ;
   INPUT variables ;
   FREQ variable ;
   SVD <options> ;
   OUTDECOMP <options> ;
   ANOMALYDETECTION <options> ;
   CODE FILE=filename ;
   DISPLAY <table-list> </ options> ;
   DISPLAYOUT table-spec-list </ options> ;
   IMAGE variable ;
   SAVESTATE RSTORE=libref.data-table ;
```

The following sections describe the PROC RPCA statement and then describe the other statements in alphabetical order.

PROC RPCA Statement

```
PROC RPCA <options> ;
```

The PROC RPCA statement invokes the procedure. Table 30.1 summarizes the *options* available in the PROC RPCA statement.

Table 30.1 PROC RPCA Statement Options

Option	Description
Input Data Table Option	
DATA=	Specifies the input data table
RPCA Options	
CENTER	Centers the input data
CUMEIGPCTTOL=	Specifies the significance level of eigenvalues that determine the rank of the low-rank matrix
DECOMP=	Specifies the low-rank decomposition method
FIXEDMU	Uses a fixed value for μ
LAMBDA=	Specifies the parameter λ
LAMBDAWEIGHT=	Specifies the parameter λ_{weight}
MAXITER=	Specifies the maximum number of iterations
METHOD=	Specifies the method of solving RPCA
MU=	Specifies the parameter μ
NTHREADS=	Specifies the number of threads to use on each computation node
PCPREFIX=	Specifies a prefix for naming the principal components

Table 30.1 *continued*

Option	Description
SCALE	Scales the input data
TOLERANCE=	Specifies the convergence tolerance
Decomposition Output Table Options	
OUTERROR=	Specifies the name of the error output table
OUTLOWRANK=	Specifies the name of the low-rank output table
OUTSPARSE=	Specifies the name of the sparse output table
Simple Statistics Table Option	
COLSTATISTICS=	Specifies the name of the table that contains simple statistics for the variables in the input data set

You can specify the following *options*:

CENTER

centers the observations by the mean of each column. For more information, see the section "Scaling and Centering" on page 797.

COLSTATISTICS=*libref.data-table*

specifies the name of the output table to contain simple statistics for the variables of the input data table.

libref.data-table is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 781.

NOTE: This option is disabled for image input.

CUMEIGPCTTOL=*number*

specifies the significance level of eigenvalues that determine the rank of the low-rank matrix. This value must be between 0 and 1. By default, CUMEIGPCTTOL=1. For more information, see the section "Cumulative Eigenvalue Percentage Tolerance" on page 797.

DATA=*libref.data-table*

names the input data table for PROC RPCA to use. The default is the most recently created data table. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to the data. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 781.

data-table specifies the name of the input data table.

DECOMP=SVD | PCA | NONE

specifies the type of analysis. If MAXITER=0, this step is applied to the original input data; otherwise, it is applied to the low-rank matrix. You can specify the following values:

SVD	performs singular value decomposition.
PCA	performs principal component analysis.
NONE	performs neither of the two preceding analyses.

By default, DECOMP=NONE. **NOTE:** This option is disabled for image input.

FIXEDMU

uses a fixed value for μ in each iteration of the RPCA algorithm when METHOD=APG. Otherwise, the value of μ is dynamically updated in each iteration.

LAMBDA=*number*

specifies a value for the parameter λ, where *number* is a positive real number. The default value is computed by $\frac{1}{\sqrt{n}}$, where n is the greater of the number of observations and the number of variables in the input data set. This parameter affects the sparsity of the sparse matrix. For more information about this setting, see Candès et al. (2011).

LAMBDAWEIGHT=*number*

specifies the value of the parameter λ_{weight} to be applied on the l_1 norm of the sparse matrix. The final value for the λ that is used in the RPCA algorithm is calculated by multiplying λ_{weight} by λ. You can use this value to control the sparsity of the sparse matrix. A larger *number* produces a more sparse matrix. For more information, see the section "Details: RPCA Procedure" on page 795.

By default, LAMBDAWEIGHT=1.

MAXITER=*number*

specifies the maximum number of iterations before the process stops, where *number* is a positive integer. If MAXITER=0, PROC RPCA runs no iterations. If you also specify the DECOMP= option, the analysis is performed on the original matrix (instead of the low-rank matrix)..

If the RPCA procedure does not converge in *number* of iterations, a solution status of "Failed to Converge" is written to the "Results Summary" table.

By default, MAXITER=1000.

METHOD=ALM | APG

specifies the method to solve RPCA. You can specify the following values:

ALM	uses the augmented Lagrange multiplier method.
APG	uses the accelerated proximal gradient method.

By default, METHOD=ALM.

MU=*number*

specifies an initial value of μ when METHOD=APG. By default, MU=0.001.

NTHREADS=*number-of-threads*

specifies the number of threads per computation node. The default value is the lesser of 16 and the number of threads available per computation node.

OUTERROR=*libref.data-table*

specifies the name of the output table for the error matrix, which contains the noise in the input data.

libref.data-table is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 781.

OUTLOWRANK=*libref.data-table*

specifies the name of the output table for the low-rank matrix.

libref.data-table is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 781.

OUTSPARSE=*libref.data-table*

specifies the name of the output table for the sparse matrix.

libref.data-table is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 781.

PCPREFIX=*name*

specifies a prefix for naming the principal components. By default, the components are named Prin1, Prin2, ..., Prin$n$. If you specify PCPREFIX=Abc, the components are named Abc1, Abc2, Abc3, and so on. The number of characters in the prefix plus the number of digits required to designate the variables should not exceed the current name length that is defined by the VALIDVARNAME= system option. For more information about the VALIDVARNAME= system option, see *SAS System Options: Reference*.

SCALE

scales the observations by the standard deviation of each column. If a constant variable exists (where all observations have the same value), the observations of this column are not scaled. For more information, see the section "Scaling and Centering" on page 797.

TOLERANCE=*number*

specifies the convergence criterion for the RPCA algorithm. The default value is 10^{-7}.

ANOMALYDETECTION Statement

ANOMALYDETECTION < *options* > **;**

ANOMALY < *options* > **;**

The ANOMALYDETECTION statement specifies the parameters of the anomaly detection methods that are used in the subsequent ASTORE procedure run to detect anomalies in the scoring observations.

You can specify the following *options*:

ICAMAXITER=*number*
specifies the maximum number of iterations for infomax independent component analysis (ICA), where *number* is a positive integer. By default, MAXITER=100.

ICAMETHOD=FOBI | GEOD | SYMM | UNCON
specifies the method of ICA to use during training. You can specify the following values:

FOBI	performs the fourth-order blind identification method of ICA.
GEOD	performs the geodesic infomax method of ICA.
SYMM	performs the symmetric orthogonalized infomax method of ICA.
UNCON	performs the unconstrained infomax method of ICA.

By default, ICAMETHOD=FOBI.

METHOD=ICANORMS | ICASIGVARS | R4S | SIGVARS
specifies the method of anomaly detection. You can specify the following values:

ICANORMS	uses the ICANORMS method of anomaly detection.
ICASIGVARS	uses the ICASIGVARS method of anomaly detection.
R4S	uses the R4S method of anomaly detection.
SIGVARS	uses the SIGVARS method of anomaly detection.

For more information about the ICANORMS, ICASIGVARS, R4S, and SIGVARS methods, see the section "Anomaly Detection" on page 797. You can override this parameter by specifying the RPCA_ANOMALYDETECTION_METHOD option and the override value in the SETOPTION statement in the ASTORE procedure. By default, METHOD=SIGVARS.

NUMSIGVARS=*number*
specifies the minimum number of significant variables in an observation for it to be considered as an anomaly by SIGVARS method, where *number* must be a positive integer value. This option is used when METHOD=SIGVARS. You can override this parameter by specifying the RPCA_NUMSIGVAR option and the override value in the SETOPTION statement in the ASTORE procedure. By default, NUMSIGVARS=1.

SIGMACOEF=*number*
specifies the threshold on the standardized sparse value in the SIGVARS method or a coefficient that is applied on the threshold in the R4S method. For more information see the section "Anomaly Detection" on page 797. You can override this parameter by specifying the RPCA_SIGMACOEF option and the override value in the SETOPTION statement in the ASTORE procedure. By default, SIGMACOEF=1.

USEMATRIX=INPUT | SPARSE
specifies which matrix to use to standardize the sparse part of the scoring observations in the anomaly detection methods:

INPUT uses the standard deviation of the columns of the original input data set.

SPARSE uses the standard deviation of the columns of the output matrix that is specified in the OUTSPARSE= option in the PROC RPCA statement.

You can override this parameter by specifying the RPCA_USEMATRIX option and the override value in the SETOPTION statement in the ASTORE procedure. By default, USEMATRIX=INPUT.

CODE Statement

CODE FILE=*filename* < **PROJECTIONTYPE=PCA | LRS** > **;**

The CODE statement generates SAS DATA step code that mimics the computations that are performed. The generated SAS DATA step code can be used to score new observations—that is, to project the new observations onto the principal component space or the low-rank subspace.

The CODE statement is optional. Only one CODE statement is processed. If you specify multiple CODE statements, only the first one is used. If you do not include a CODE statement, no score code is generated.

You must specify the following option:

FILE=*filename*
specifies the name of the file to write the SAS score code to.

You can also specify the following option:

PROJECTIONTYPE=PCA | LRS
specifies the type of projection. You can specify the following values:

PCA projects the scoring observations onto the principal component space.

LRS projects the scoring observations onto the low-rank subspace.

By default, PROJECTIONTYPE=PCA.

NOTE: The CODE statement is disabled for image input.

DISPLAY Statement

DISPLAY < *table-list* > < / *options* > **;**

The DISPLAY statement enables you to specify a list of display tables to display or exclude. This statement is similar to the ODS SELECT, ODS EXCLUDE, and ODS TRACE statements. However, the DISPLAY statement can improve performance when a large number of tables could be generated (such as in BY-group processing). The DISPLAY statement produces only the requested tables, but ODS statements generate all the tables and then keep the requested tables.

If you use both DISPLAY and ODS statements together, the DISPLAY statement takes precedence over the ODS statements. Note that the ODS EXCLUDE statement processes tables after they have been filtered by the DISPLAY statement. In some cases, it might appear that the ODS EXCLUDE statement is taking precedence because it can further filter the tables. For more information about ODS, see *SAS Output Delivery System: Procedures Guide*.

You can specify the *table-list* as a list of table names, paths, partial pathnames, and regular expressions.

The table names that you can specify are listed in the section "ODS Table Names" on page 799. A path is a table name that is prefixed with dot-separated grouping information. For example, a SelectionSummary table that a procedure produces during a selection routine might have the path *Bygroup1.Summary.SelectionSummary*. A partial pathname does not include all groups; for example, *SelectionSummary* and *Summary.SelectionSummary* are partial pathnames for *Bygroup1.Summary.SelectionSummary*.

When you specify a table name or partial pathname, all display tables whose paths end in the specified name are selected for display or exclusion. For example, both *SelectionSummary* and *Summary.SelectionSummary* select *Bygroup1.Summary.SelectionSummary*.

A regular expression is enclosed in forward slashes (/). For example, specifying "/tions/" selects all pathnames that contain the substring "tions"; in particular, the *Bygroup1.Summary.SelectionSummary* table is selected. Specifying "!/tions/" selects all pathnames that do not contain the substring "tions"; in particular, the *Bygroup1.Summary.SelectionSummary* table is not selected.

You can specify the following *options* after a slash (/):

CASESENSITIVE
performs a case-sensitive comparison of table names in the *table-list* to display table names when tables are subsetted for display. To preserve case, you must enclose table names in the *table-list* in quotation marks.

EXCLUDE
displays all display tables except those that you specify in the *table-list*.

EXCLUDEALL
suppresses display of all tables. This option takes precedence over the other options.

TRACE
displays the display table names, labels, and paths.

DISPLAYOUT Statement

DISPLAYOUT *table-spec-list* < / *options* > **;**

The DISPLAYOUT statement enables you to create output tables from your displayed output. This statement is similar to the ODS OUTPUT statement. For more information about ODS, see *SAS Output Delivery System: Procedures Guide*.

The *table-spec-list* specifies a list of output tables to create. Each entry in the list has either a *key*=*value* format or a *key* format:

key=*value* specifies *key* as the ODS table name, path, or partial pathname, and specifies *value* as the output table name.

key specifies *key* as the ODS table name and also as the output table name.

The ODS table names that you can specify are listed in the section "ODS Table Names" on page 799. You cannot specify the ODS table named OutputCasTables in the *table-spec-list*.

Table names and partial pathnames are discussed under the DISPLAY statement. The DISPLAYOUT statement does not support regular expressions.

You can specify the following *options* after a slash (/):

INCLUDEALL
creates output tables for all display tables. The name of the created output table is the same as the name of the corresponding display table. If you specify this option, the *table-spec-list* specification is ignored.

NOREPLACE
does not replace any existing output table of the same name.

FREQ Statement

FREQ *variable* **;**

The *variable* in the FREQ statement identifies a numeric variable in the input data table that contains the frequency of occurrence of each observation. If the value of the *variable* for an observation is not an integer, it is truncated to an integer. If the value is less than 1 or missing, the observation is not used in the analysis. If you omit this statement, each observation is assigned a frequency of 1.

ID Statement

ID *variables* **;**

The ID statement lists one or more variables to be copied from the input data table to the OUTLOWRANK=, OUTSPARSE=, OUTERROR=, PCSCORES=, and SVDLEFT= output data tables, and to the file that is specified in the RSTORE= option in the SAVESTATE statement. If any of the *variables* in the ID statement appears in the INPUT statement, that variable is considered to be an ID variable and is not used in the analysis (it is only copied to the output tables and the file that you specify in the RSTORE= option).

IMAGE Statement

IMAGE *variable* **;**

The IMAGE statement specifies the name of a *variable* that consists of image binaries that are encoded as JPG, PNG, TIF, or WIDE files. **NOTE:** You cannot specify this statement together with the INPUT statement.

INPUT Statement

INPUT *variables* **;**

The INPUT statement specifies the names of *variables* to be considered in the RPCA procedure. Only numeric *variables* are accepted. If you do not specify this statement, PROC RPCA considers all numeric *variables* in the input data table. **NOTE:** You cannot specify this statement with image input.

OUTDECOMP Statement

OUTDECOMP < *options* > **;**

The OUTDECOMP statement specifies output filenames for the low-rank matrix decomposition tables.

If you specify DECOMP=SVD in the PROC RPCA statement, then you can specify the following three *options*:

SVDDIAG=*libref.data-table*
specifies the name of the output table to contain the SVD diagonal vector. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 781.

SVDLEFT=*libref.data-table*
specifies the name of the output table to contain the SVD left matrix. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 781.

SVDRIGHT=*libref.data-table*
specifies the name of the output table to contain the SVD right matrix. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 781.

If you specify DECOMP=PCA in the PROC RPCA statement, then you can specify the following two *options*:

PCLOADINGS=*libref.data-table*
specifies the name of the output table to contain the matrix of principal component loadings. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 781.

PCSCORES=*libref.data-table*
specifies the name of the output table to contain the matrix of principal component scores. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 781.

NOTE: The OUTDECOMP statement is disabled for image input.

SAVESTATE Statement

SAVESTATE RSTORE=*libref.data-table* **;**

The SAVESTATE statement creates an analytic store for the model and saves it as a binary object in a data table. You can use the analytic store in the ASTORE procedure to score new data. For more information, see Chapter 4, "ASTORE Procedure."

You must specify the following option:

RSTORE=*libref.data-table*
specifies the name of the table in which to save the analytic store. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 781.

You can use the ASTORE procedure to specify the projection type. Specify the option RPCA_PROJECTION_TYPE in the SETOPTION statement in PROC ASTORE: the value 0 projects the scoring observations onto the principal component space, the value 1 projects the scoring observations onto the low-rank subspace, and the value 2 projects the scoring observations onto the low-rank subspace but stores the sparse part of the scoring data in the scoring results table.

SVD Statement

SVD < *options* > **;**

The SVD statement specifies the parameters of the singular value decomposition (SVD) algorithm that are used in RPCA iterations. You can specify the following *options*:

MAXRANK=*number*
specifies the maximum value for the rank that the SVD solver considers. The default value is the number of input variables.

METHOD=EIGEN | ITERATIVE | RANDOM
specifies the type of the SVD solver. You can specify the following values:

EIGEN	uses the eigenvalue decomposition method.
ITERATIVE	uses the iterative SVD method.
RANDOM	uses the randomized SVD method.

When the number of variables in the input data set is very large, it is recommended that you use METHOD=RANDOM. By default, METHOD=EIGEN.

POWER=*number*
specifies the parameter power for the SVD solver if METHOD=RANDOM. By default POWER=0.

SEED=*number*
specifies the parameter seed for the SVD solver if METHOD=RANDOM. The default value is the current local time in SAS datetime format.

NOTE: The SVD statement is disabled for image input.

Details: RPCA Procedure

Robust principal component analysis (RPCA) is a matrix decomposition algorithm that decomposes an input matrix M into a low-rank matrix L_0 and a sparse matrix S_0, where $M = L_0 + S_0$. This decomposition is obtained by solving a convex programming problem called principal component pursuit (PCP).

Let $||L||_* := \sum_i \rho_i(L)$ denote the nuclear norm of the matrix L (that is, the sum of the singular values of L), and let $||S||_1 = \sum_{ij} |S_{ij}|$ denote the l_1 norm of S. In this case, PCP can be formulated as

$$\begin{array}{ll} \text{minimize} & ||L||_* + \lambda ||S||_1 \\ \text{subject to} & L + S = M \end{array}$$

For more information about this formulation, see Candès et al. (2011).

RPCA is robust because the principal components are computed from the low-rank matrix. The sparse matrix includes the outlier values.[1] As a result, many applications of RPCA focus on the sparse matrix.

Two algorithms are implemented in the RPCA procedure: the augmented Lagrange multiplier algorithm and the accelerated proximal gradient algorithm.

Augmented Lagrange Multiplier Method

In general, the augmented Lagrange method is used to solve nonlinear constrained optimization problems. In the case of PCP, an augmented Lagrange function is used to reformulate the PCP problem as the following nonlinear unconstrained optimization problem:

$$\text{minimize} \quad l(L,S,Y) = ||L||_* + \lambda||S||_1 + < Y, M - L - S > + \frac{\mu}{2}||M - L - S||_F^2$$

Candès et al. (2011) use the augmented Lagrange multiplier (ALM) method to find the solution to the preceding optimization problem. The basic idea is to update S, L, and Y iteratively. At iteration k, given L_k and Y_k, the first step is to find S_{k+1} by minimizing $l(L_k, S, Y_k)$. In the second step, L_{k+1} is obtained by the singular value thresholding operator, which minimizes $l(L, S_{k+1}, Y_k)$.[2] Next, the Lagrange multiplier Y_{k+1} is updated. For more information, see Candès et al. (2011).

Accelerated Proximal Gradient Method

Another common relaxation of PCP is the following unconstrained optimization formulation, which contains two terms in the objective function. The first term is the original objective function weighted by μ, and the second term is the penalty term of the constraint:

$$\text{minimize} \quad \mu||L||_* + \mu\lambda||S||_1 + \frac{1}{2}||M - L - S||_F^2$$

Lin et al. (2009) applied the accelerated proximal gradient (APG) algorithm to solve the preceding relaxed model. In each iteration of their proposed algorithm, L_{k+1} and S_{k+1} are obtained by applying singular value decomposition to the updated matrix, which is computed based on the values of L and S in the last two iterations—that is, L_{k-1}, L_k, S_{k-1}, and S_k.

The APG algorithm is generally slower than the ALM algorithm. Zhou et al. (2010) suggest that APG be used when the observations are significantly corrupted by noise.

[1] The sparse matrix can also include noise. The sparsity of the sparse matrix depends highly on the settings of the λ and λ_{weight} parameters.

[2] The singular value thresholding operator is based on the singular value decomposition.

Scaling and Centering

If you specify the CENTER or SCALE option (or both), the original data M are standardized as

$$M(i,j) = \frac{M(i,j) - \mu_j}{\sigma_j}$$

where μ_j and σ_j are the mean and standard deviation of the jth column in M, respectively. Therefore, the low-rank matrix (L), the sparse matrix (S), and the error matrix (N) are updated as follows:

$$L(i,j) = L(i,j) * (\sigma_j) + \mu_j$$

$$S(i,j) = S(i,j) * (\sigma_j)$$

$$N(i,j) = N(i,j) * (\sigma_j)$$

Cumulative Eigenvalue Percentage Tolerance

Sometimes when the input data set has noise, the rank that is reported in the ODS summary table is higher than the true rank. This occurs because the existence of noise in the input data set can increase the number of eigenvalues that is reported in the table specified in the SVDDIAG= option. To address this issue, you can eliminate the insignificant eigenvalues by specifying a tolerance on the cumulative percentage of eigenvalues.

Suppose that the eigenvalues that are computed for the low-rank matrix are 1,000, 100, and 1. If CUMEIGPCTTOL=1, then the rank of the low-rank matrix is reported as 3. However, the third eigenvalue is much smaller than the other two eigenvalues. If CUMEIGPCTTOL=0.99, then the first two eigenvalues cumulatively contribute to more than 99.90% of the sum of the eigenvalues. In this case, the smallest eigenvalue can be ignored, and the rank of the low-rank matrix is 2.

Anomaly Detection

The RPCA procedure implements two anomaly detection methods, the ICANORMS method and the ICASIGVARS method, that use independent component analysis (ICA) in postprocessing to recover latent sources of the sparse part. For more information about these formulations, see Kolay et al. (2024). For general background on ICA, see Hyvärinen, Karhunen, and Oja (2001).

Each observation is projected onto the orthogonal complement of the low-rank space, which is referred to as the S_{pr} vector. This vector is then centered; after that, it is referred to as the CS_{pr} vector. Finally, the unmixing matrix W from the independent component analysis in the training step is used to obtain the latent sources vector as follows:

$$LS = W \cdot CS_{pr}$$

ICANORMS Method

In the ICANORMS method, the ℓ_1, ℓ_2, and ℓ_3 norms of the latent sources vector LS are calculated for each scoring observation. *LS* is declared to be anomalous if for any $i \in \{1, 2, 3\}$, the distance of this norm from the corresponding mean from training is greater than the corresponding standard deviation from training multiplied by the value of the SIGMACOEF= option.

ICASIGVARS Method

In the ICASIGVARS method, the *j*th variable LS_j of LS is considered to be significant if its absolute value is greater than the value of the SIGMACOEF= option. If the number of significant values in the scoring observation is greater than or equal to the proportional value of the NUMSIGVARS= option, the scoring observation is considered to be anomalous.

PROC RPCA implements two additional anomaly detetion methods, the R4S method and the SIGVARS method, where the scoring observations are projected onto the low-rank space. The resulting projection is subtracted from the scoring observation to identify the sparse part, which is referred to as the S_{sc} vector.

R4S Method

In the R4S method, the values of the S_{sc} vector are standardized as

$$SS_{sc_j} = \frac{S_{sc_j}}{\sigma_j}$$

where *j* refers to variable *j* in the input data set and σ_j is the standard deviation of either column *j* of the input data set or column *j* of the OUTSPARSE= output matrix, depending on the value of the USEMATRIX= option.

The root of the sum of squares of the values on SS_{sc}, denoted as $RSSSS_{sc}$, is calculated for each scoring observation:

$$RSSSS_{sc} = \sqrt{\sum_{j \in input\ vars} (SS_{sc_j})^2}$$

This $RSSSS_{sc}$ is compared to the following value: the standard deviation of the same measures calculated for the OUTSPARSE= output matrix multiplied by the value of the SIGMACOEF= option in the ANOMALYDETECTION statement. The scoring observation is considered anomalous if $RSSSS_{sc}$ is greater than or equal to this value.

SIGVARS Method

In the SIGVARS method, as in the R4S method, the values of S_{sc} are standardized. However, a different criterion is used to detect whether the scoring observation is anomalous. The absolute value of SS_{sc_j} is considered significant if it is greater than the value of the SIGMACOEF= option. If the number of significant values in the scoring observation is greater than or equal to the value of the NUMSIGVARS= option, the scoring observation is considered to be anomalous.

Displayed Output

The RPCA procedure displays various tables that are related to input and results. The following sections describe the output tables in the order of their appearance.

Model Information

The "Model Information" table displays basic information about the parameters that are used in the RPCA procedure. This information includes the data source, the RPCA method, the SVD method used in RPCA, and the value of parameters λ and λ_{weight}. Also, if the SVD method is the randomized method (that is, METHOD=RANDOM in the SVD statement), the seed that is used in the SVD method is displayed.

Dimensions

The "Dimensions" table displays the number of observations and variables in the input data set, the number of missing observations, and the number of observations from the input data set that are considered for the RPCA analysis.

Results Summary

The "Results Summary" table displays the summary of the RPCA results, including the sparsity of the sparse matrix, the rank of the low-rank matrix, the number of iterations, the solution status, and the run time.

The solution status takes one of these values:

Optimal RPCA converges and decompositions are successful.

SVD Failure Internal SVD call failure. This can be caused by running out of memory or SVD not converging.

Failed to Converge RPCA does not converge with the specified MAXITER= option.

Failed RPCA failed because it ran out of memory or because of other problems.

ODS Table Names

Each table that the RPCA procedure creates has a name associated with it. You must use this name to refer to the table when you use ODS statements. The names of each table and a short description of the contents are listed in Table 30.2.

Table 30.2 ODS Tables Produced by PROC RPCA

Table Name	Description	Statement	Option
Dimensions	Dimensions of the input table	PROC RPCA	Default
ModelInfo	Model information	PROC RPCA	Default
Summary	Summary of the results	PROC RPCA	Default

Output Data Tables

The RPCA procedure creates output tables for the low-rank matrix (if the OUTLOWRANK= option is specified), the sparse matrix (if the OUTSPARSE= option is specified), the error matrix (if the OUTERROR= option is specified), and column statistics (if the COLSTATISTICS= option is specified).

Also, depending on the value of the DECOMP=option in the PROC RPCA statement, the low-rank matrix is decomposed in SVD tables (which are specified in the SVDLEFT=, SVDDIAG=, and SVDRIGHT= options) or PCA tables (which are specified in the PCLOADINGS= and PCSCORES= options). Table 30.3 lists details about these data tables.

Table 30.3 Output Data Tables Produced by PROC RPCA

Data Table	Content	Dimensions
COLSTATISTICS	Mean and standard deviation for each variable of the input data table	Number of variables × 3. Not available for image input.
OUTERROR	Error matrix	Number of observations × number of variables, or number of observations × image (for images)
OUTLOWRANK	Low-rank matrix	Number of observations × number of variables, or number of observations × image(for images)
OUTSPARSE	Sparse matrix	Number of observations × number of variables, or number of observations × image (for images)
PCLOADINGS	Right-singular vectors (and an extra column for variable names)	Number of variables × (number of singular values + 1)
PCSCORES	Product of matrix of left-singular vectors and diagonal matrix of singular values	Number of observations × number of singular values
SVDDIAG	Singular values in decreasing order	Number of singular values
SVDLEFT	Left-singular vectors	Number of observations × number of singular values
SVDRIGHT	Right-singular vectors (and an extra column for variable names)	Number of variables × (number of singular values + 1)

Examples: RPCA Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

Example 30.1: Analyzing the Energy Output of the SAS Solar Farm

This example applies the RPCA procedure to data from the solar farm at SAS headquarters in Cary, North Carolina. The data set consists of data from many sensors, at different aggregations, as well as weather data. All together, it contains 80 variables. The goal of the example is to predict the daytime hourly energy output (KWH_Delivered) from the solar farm by using robust principal components.

The following code includes the DATA step that generates the mylib.SolarFarm data table and the PROC RPCA statement that is applied to these data.

```
   data mylib.solarfarm;
input
date : date9. hour KWH_Delivered A_B_Avg_Volt App_Energy_Tot_Delv App_Pwr_Total
App_Pwr_Total_Max B_C_Avg_Volt C_A_Avg_Volt Current_Avg Current_Pct_Unblanc
Day_Energy_created Energy_created_10mi Frequency Hr_Energy_created L_L_Avg_Volt

   ... more lines ...
```

Of the observations in the data table, 60% are used for training and 40% are used for testing. You can use the following PARTITION procedure code to randomly divide the observations in the mylib.SolarFarm data table into training and testing data tables:

```
proc partition data = mylib.solarfarm partind samppct=60;
   output out=mylib.solarfarmpart;
run;

data mylib.solarfarmtrain(drop = _partind_);
   set mylib.solarfarmpart;
   if _partind_ eq 1 then output;
run;

data mylib.solarfarmtest(drop = _partind_);
   set mylib.solarfarmpart;
   if _partind_ eq 0 then output;
run;

proc contents data=mylib.solarfarm out=varlist(keep=NAME) noprint;
run;
proc sql noprint;
   select NAME into:inputvars separated by ' '
   from varlist where NAME not in ("date","hour","KWH_Delivered");
quit;
```

```
proc rpca data=mylib.solarfarmtrain
          outlowrank=mylib.lowrank
          decomp=svd method = APG
          scale center;
    id date hour;
    input &inputvars.;
    outdecomp svddiag=mylib.svddiag;
    code file='rpcascore.sas' projectiontype=PCA;
run;
```

In the svddiag output table (not shown), the first 13 eigenvalues explain more than 95% of the variability in the data. So you can project the training data on the first 13 principal components and build the regression model to predict the generated energy from these 13 robust components by using the following statements:

```
data mylib.rpcaprojectedtrain
     (keep=Prin1 Prin2 Prin3 Prin4 Prin5 Prin6 Prin7 Prin8
           Prin9 Prin10 Prin11 Prin12 Prin13 KWH_Delivered);
   set mylib.solarfarmtrain;
   %inc 'rpcascore.sas';
run;

proc regselect data=mylib.rpcaprojectedtrain;
   model KWH_Delivered= Prin1 Prin2 Prin3 Prin4 Prin5 Prin6 Prin7 Prin8 Prin9
                        Prin10 Prin11 Prin12 Prin13;
   code file='rpcareg.sas';
run;
```

Next, you can use the following statements to project the test data into the principal component space and to predict the energy output of the test data by using the regression model that is obtained in the previous step:

```
data mylib.rpcaprojectedtest
    (keep= date hour Prin1 Prin2 Prin3 Prin4 Prin5 Prin6
           Prin7 Prin8 Prin9 Prin10 Prin11 Prin12 Prin13 KWH_Delivered);
   set mylib.solarfarmtest;
   %inc 'rpcascore.sas';
run;

data mylib.EnergyforecastOnTestDataRPCA;
   set mylib.rpcaprojectedtest;
    %inc 'rpcareg.sas';
run;
```

You can repeat the same steps by using standard PCA (with the PCA procedure) as well:

```
proc pca data=mylib.solarfarmtrain;
   var &inputvars.;
   code file='pcascore.sas';
run;
```

In the resulting “Eigenvalues” table (not shown), 25 principal components explain 95% of the variability in the data. So you can project the training data on these 25 principal components and build a regression model

by using the following statements:

```
data mylib.pcaprojectedtrain
     (keep=Prin1 Prin2 Prin3 Prin4 Prin5 Prin6 Prin7 Prin8 Prin9 Prin10
           Prin11 Prin12 Prin13 Prin14 Prin15 Prin16 Prin17 Prin18 Prin19
           Prin20 Prin21 Prin22 Prin23 Prin24 Prin25 KWH_Delivered);
   set mylib.solarfarmtrain;
   %inc 'pcascore.sas';
run;

proc regselect data=mylib.pcaprojectedtrain;
   model KWH_Delivered= Prin1 Prin2 Prin3 Prin4 Prin5 Prin6 Prin7 Prin8 Prin9
                        Prin10 Prin11 Prin12 Prin13 Prin14 Prin15 Prin16 Prin17
                        Prin18 Prin19 Prin20 Prin21 Prin22 Prin23 Prin24 Prin25;
   code file='pcareg.sas';
run;
```

Next, you can use the following statements to project the test data on these 25 principal components and get a prediction of the energy that is generated:

```
data mylib.pcaprojectedtest
     (keep= date hour Prin1 Prin2 Prin3 Prin4 Prin5 Prin6 Prin7 Prin8 Prin9
            Prin10 Prin11 Prin12 Prin13 Prin14 Prin15 Prin16 Prin17 Prin18
            Prin19 Prin20 Prin21 Prin22 Prin23 Prin24 Prin25 KWH_Delivered);
   set mylib.solarfarmtest;
   %inc 'pcascore.sas';
run;

data mylib.EnergyforecastOnTestDataPCA;
   set mylib.pcaprojectedtest;
   %inc 'pcareg.sas';
run;
```

Now you can calculate the RMSE to compare the model that is built using RPCA to the model that is built using regular PCA, by using the following statements:

```
data mylib.rpcaresiduals(keep=date hour rpcaresidual);
   set  mylib.EnergyforecastOnTestDataRPCA;
   rpcaresidual= KWH_Delivered - P_KWH_Delivered;
run;
 data mylib.pcaresiduals(keep=date hour pcaresidual);
   set  mylib.EnergyforecastOnTestDataPCA;
   pcaresidual= KWH_Delivered - P_KWH_Delivered;
run;

data mylib.residuals;
   merge   mylib.rpcaresiduals  mylib.pcaresiduals;
   by date hour;
run;

proc sql;
   select sqrt(sum(rpcaresidual*rpcaresidual)/count(rpcaresidual))
```

```
            as rpcaRMSE from mylib.rpcaresiduals;
    select sqrt(sum(pcaresidual*pcaresidual)/count(pcaresidual))
            as pcaRMSE  from mylib.pcaresiduals;
quit;
```

The results from the last step (not shown) demonstrate that the RMSE of the regression model that is built using robust principal components is lower than the RMSE of the regression model that is built using standard principal components. This is no surprise, because the principal components that you obtain from standard PCA are affected by the outliers in the data.

Example 30.2: Detecting Anomalies

This example uses the RPCA procedure to apply anomaly detection to the Wisconsin-Breast Cancer data set. You can find the data in the files *wbctr.csv* for training and *wbcan.csv* for scoring on the website https://support.sas.com/documentation/onlinedoc/viya/examples.htm. The *wbctr.csv* file contains 284 normal observations, and the *wbcan.csv* file contains 21 anomalous observations; both have 30 variables.

The following statements run PROC RPCA on the training data set:

```
proc rpca data=mylib.wbctr
decomp=svd method = ALM;
anomalydetection icamethod=SYMM icamaxiter=100;
savestate rstore=mylib.store ;
run;
```

Next, you can use the following statements to score the anomalous data by using the ICANORMS method:

```
proc astore;
setoption rpca_projection_type 2;
setoption rpca_anomalydetection_method 3;
score data=mylib.wbcan
rstore=mylib.store
out=mylib.out;
quit;

proc means data=mylib.out;
var _outlier_detection_score_;
run;
```

This method detects more than 95% of the anomalies.

References

Candès, E. J., Li, X., Ma, Y., and Wright, J. (2011). "Robust Principal Component Analysis?" *Journal of the Association for Computing Machinery* 58:1–37.

Hyvärinen, A., Karhunen, J., and Oja, E. (2001). *Independent Component Analysis*. New York: John Wiley & Sons.

Kolay, S., Xu, S. G., Shen, K., and Talebi, Z. A. (2024). "Anomaly Detection Using RPCA and ICA." US Patent Number 11887012.

Lin, Z., Ganesh, A., Wright, J., Wu, L., Chen, M., and Ma, Y. (2009). "Fast Convex Optimization Algorithms for Exact Recovery of a Corrupted Low-Rank Matrix." In *IEEE International Workshop on Computational Advances in Multi-sensor Adaptive Processing*. Piscataway, NJ: Institute of Electrical and Electronics Engineers.

Zhou, Z., Li, X., Wright, J., Candès, E. J., and Ma, Y. (2010). "Stable Principal Component Pursuit." In *IEEE International Symposium on Information Theory Proceedings*, 1518–1522. Piscataway, NJ: Institute of Electrical and Electronics Engineers.

Chapter 31
SHAPLEY Procedure

Contents

Overview: SHAPLEY Procedure **808**
- PROC SHAPLEY Features 808
- Using SAS Viya Workbench 808

Getting Started: SHAPLEY Procedure **809**

Syntax: SHAPLEY Procedure **810**
- PROC SHAPLEY Statement 811
- ASTOREMODEL Statement 812
- CODE Statement 812
- FREQ Statement 813
- INPUT Statement 813
- METHOD Statement 814
- PREDICTEDTARGET Statement 815
- WEIGHT Statement 815

Details: SHAPLEY Procedure **815**
- Required Specifications 815
- Shapley Values 816
- HyperSHAP Method 817
- Kernel SHAP Method 818
- Frequency and Weight Variables 819
- Results 819
- ODS Table Names 820

Examples: SHAPLEY Procedure **820**
- Example 31.1: Compute Shapley Values by Using the Kernel SHAP Method 820
- Example 31.2: Compute Shapley Values (Kernel SHAP Method) Using Prescored Data 822

References **824**

Overview: SHAPLEY Procedure

The SHAPLEY procedure estimates Shapley values of a query on the basis of the information that is provided in a reference table. Shapley values are a measure of variable importance for a single observation. You can use them to explain why a machine learning model yields a specific prediction for a query observation. They serve as an additive measure of importance whose sum equals the predicted output from the model that you are explaining. PROC SHAPLEY uses two methods to approximate the same Shapley values. The first method is kernel Shapley additive explanations (Kernel SHAP), which fits linear regression models to the scored output from machine learning models. The second method is hyperparameterized Shapley value estimation (HyperSHAP), which calculates the Shapley values for predictive models.

PROC SHAPLEY Features

PROC SHAPLEY has the following features:

- provides two methods, Kernel SHAP and HyperSHAP, for model explanations
- computes Shapley values for model explanations
- computes an explainer fidelity value when you use the Kernel SHAP method

Using SAS Viya Workbench

SAS Viya Workbench has greatly simplified the SAS programming experience by offering a syntax style nearly identical to that of SAS 9 and earlier releases. However, there are minor exceptions. In the past, SAS Viya procedures required the use of a CAS engine libref. This engine served multiple purposes, but it was primarily used to upload data to the CAS server and manage the data. Its additional purposes included support of new data types and improved performance for large data sets. No CAS server is needed for SAS Viya Workbench, because all computations are done in place for you in a single location.

In the following program, the LIBNAME statement creates a mylib libref that the examples in this chapter use to refer to your current or desired work directory:

```
libname mylib ".";
```

For more information, see *SAS Viya Workbench: Getting Started*.

Getting Started: SHAPLEY Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

This example uses the SHAPLEY procedure to explain a prediction that is made by a forest model by using the HyperSHAP method with a depth of 1.

The following DATA step creates the input data table mylib.dmagecr:

```
data mylib.dmagecr;
   set sampsio.dmagecr;
   keep age amount coapp duration foreign job good_bad;
run;
```

The following DATA step creates the query data table mylib.query in your SAS library:

```
data mylib.query;
   set sampsio.dmagecr;
   keep age amount coapp duration foreign job good_bad;
   if _N_ = 20;
run;
```

The following statements use the FOREST procedure to build a forest model to predict whether the credit rating of each individual in the mylib.dmagecr data table is good or bad. The procedure also outputs an analytic store named forest_astore.

```
proc forest
   data=mylib.dmagecr
   seed=1234;
   input age amount duration / level=interval;
   input coapp foreign job /level=nominal;
   target good_bad;
   savestate rstore=mylib.forest_astore;
run;
```

The following statements run PROC SHAPLEY and produce ODS tables of the results:

```
proc shapley
   data=mylib.query
   referenceData=mylib.dmagecr;
   input age amount duration / level=interval;
   input coapp foreign job /level=nominal;
   predictedTarget P_good_badbad;
   method hypershap(depth=1);
   astoreModel rstore=mylib.forest_astore;
run;
```

The DATA= option names the input query data table, which specifies the data to use for Shapley analysis. The REFERENCEDATA= option names the input reference data table. The two INPUT statements specify that age, amount, and duration are the interval variables and coapp, foreign, and job are the nominal variables. The PREDICTEDTARGET statement specifies that P_good_badbad be used as the predicted target variable. The METHOD statement specifies that the HyperSHAP method be used in the explanation. The DEPTH=

option specifies the level of the HyperSHAP approximation to calculate. The ASTOREMODEL statement specifies the analytic store that contains the model to explain.

Figure 31.1 displays the Shapley values for the single observation in the mylib.query table.

Figure 31.1 Shapley Values

The SHAPLEY Procedure

Shapley Values	
Variable	**Shapley Value**
Intercept	0.2647776386
age	0.044926239
amount	-0.025196656
duration	0.0534664932
coapp	-0.000781549
foreign	0.0050053966
job	-0.000528767

Syntax: SHAPLEY Procedure

The following statements are available in the SHAPLEY procedure:

```
PROC SHAPLEY DATA=libref.data-table REFERENCEDATA=libref.data-table < options > ;
    ASTOREMODEL RSTORE=libref.data-table ;
    CODE FILE=filename | TABLE=libref.data-table ;
    FREQ variable ;
    INPUT variables < / LEVEL=INTERVAL | NOMINAL > ;
    METHOD method-type < (options) > ;
    PREDICTEDTARGET variable ;
    WEIGHT variable ;
```

The PROC SHAPLEY, INPUT, and PREDICTEDTARGET statements are required. The ASTOREMODEL and INPUT statements can appear multiple times. The rest of this section provides detailed syntax information about each statement, beginning with the PROC SHAPLEY statement. The remaining statements are described in alphabetical order.

PROC SHAPLEY Statement

PROC SHAPLEY DATA=*libref.data-table* **REFERENCEDATA=***libref.data-table* < *options* > **;**

The PROC SHAPLEY statement invokes the procedure. Table 31.1 summarizes the options in this statement.

Table 31.1 PROC SHAPLEY Statement Options

Option	Description
DATA=	Specifies the input query data table, which contains the query observation
IMPUTE=	Specifies how to impute missing values
LOGLEVEL=	Specifies a value to control the quantity and type of notes to print to the client log
REFERENCEDATA=	Specifies the input reference data table

You must specify the following options:

DATA=*libref.data-table*
names the input data table for PROC SHAPLEY to use. The default is the most recently created data table. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to the data. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 808.

data-table specifies the name of the input data table.

REFERENCEDATA=*libref.data-table*
specifies the input reference data table. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the input data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 808.

You can also specify the following *options*:

IMPUTE=MEAN | NONE | WEIGHTEDMEAN
specifies the method to use for imputing the missing observations. You can specify the following methods:

MEAN uses the mean to impute missing values of interval variables, and uses the mode to impute missing values of nominal variables.

NONE does not impute missing values.

WEIGHTEDMEAN uses the weighted mean to impute missing values of interval variables, and uses the mode to impute missing values of nominal variables.

By default, IMPUTE=NONE.

LOGLEVEL=0 | 1 | 2
controls the quantity and type of notes to print to the client log.

0 prints only warnings and errors.

1 prints some notes.

2 prints many notes.

By default, LOGLEVEL=1.

ASTOREMODEL Statement

ASTOREMODEL RSTORE=*libref.data-table* **;**

The ASTOREMODEL statement names an analytic store that contains the model to explain. If you specify multiple ASTOREMODEL statements, you must also specify the CODE statement to name the DS2 code that is used along with the analytic stores. For more information about DS2 code and using DS2 code with analytic stores, see *SAS DS2 Language Reference*, *SAS DS2 Programmer's Guide*, and Chapter 4, "ASTORE Procedure."

You must specify the following option:

RSTORE=*libref.data-table*
specifies the analytic store that contains the model to explain. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the input data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 808.

CODE Statement

CODE FILE=*filename* **| TABLE=***libref.data-table* **;**

The CODE statement names the DATA step code that describes the model or the DS2 code that is used along with an analytic store that you specify in the ASTOREMODEL statement. If you specify multiple ASTOREMODEL statements, you must also specify the CODE statement to name the DS2 code that is used along with the analytic stores. For more information about DS2 code and using DS2 code with analytic stores, see *SAS DS2 Language Reference*, *SAS DS2 Programmer's Guide*, and Chapter 4, "ASTORE Procedure."

You must specify one of the following options:

FILE=*filename*
names the external file that contains the DATA step code that describes the model or the DS2 code to be used along with an analytic store that you specify in the ASTOREMODEL statement. When you enclose the *filename* in a quoted string (for example, FILE="c:\mydir\scorecode.sas"), this option specifies the path and file name for reading the code from an external file. You can also specify an unquoted *filename* of no more than eight characters. If the *filename* is assigned as a fileref in a Base SAS FILENAME statement, the file that you specify in the FILENAME statement is used.

TABLE=*libref.data-table*
names the data table that contains the DATA step code that describes the model or the DS2 code to be used along with an analytic store that you specify in the ASTOREMODEL statement. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the input data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 808.

FREQ Statement

FREQ *variable* **;**

The *variable* in the FREQ statement identifies a numeric variable in the input data table that contains the frequency of occurrence of each observation. PROC SHAPLEY treats each observation as if it appears f times, where f is the value of the FREQ *variable* for the observation. If f is not an integer, it is truncated to an integer. If f is less than 1 or missing, the observation is not used in the analysis. When you omit the FREQ statement, each observation is assigned a frequency of 1.

INPUT Statement

INPUT *variables* < / **LEVEL=INTERVAL | NOMINAL** > **;**

The INPUT statement specifies one or more *variables* as input variables. You can specify this statement multiple times.

You can specify the following option after a slash:

LEVEL=INTERVAL | NOMINAL
specifies the type of all the *variables* in the statement. You can specify the following values:

INTERVAL treats all the *variables* in the statement as interval variables.

NOMINAL treats all the *variables* in the statement as nominal variables.

By default, LEVEL=INTERVAL for numeric variables and LEVEL=NOMINAL for character variables.

METHOD Statement

METHOD *method-type* < **(***options***)** > **;**

The METHOD statement specifies the method to use in the model explanations.

You can specify the following *method-types*:

HYPERSHAP uses the hyperparameterized Shapley value estimation (HyperSHAP) method.

You can specify the following *option*:

DEPTH=1 | 2
specifies the level of the HyperSHAP approximation to calculate.

By default, DEPTH=1.

KERNELSHAP uses the kernel Shapley value estimation (Kernel SHAP) method.

You can specify the following *options*:

BINWIDTH=*number*
specifies the bin width (relative to the standard deviation) to use in creating the binary encoding for interval variables, where *number* is greater than or equal to 1E–6.

By default, BINWIDTH=0.1.

INCLUDEMISSING
treats missing values of nominal variables as a valid level.

SAMPLESIZE=*number*
specifies the approximate number of observations to generate, where *number* is an integer greater than or equal to 2. The default is 500 times the number of variables that are listed in the INPUT statement or 1,000,000, whichever value is smaller.

SEED=*number*
specifies the seed value for pseudorandom number generation. If you do not specify a seed, or if you specify a value less than or equal to 0, the seed is generated by reading the time of day from the computer's clock.

USERAWDATA
uses the raw reference data to calculate Shapley values and skip generating data. If you omit this option, the procedure generates data that have variable values from the variable distributions of the reference data table.

By default, the *method-type* is HYPERSHAP.

PREDICTEDTARGET Statement

PREDICTEDTARGET *variable* **;**

The PREDICTEDTARGET statement names the variable that contains the model's predictions. The specified *variable* typically has a P_ prefix.

WEIGHT Statement

WEIGHT *variable* **;**

The *variable* in the WEIGHT statement is used as a weight to perform a weighted analysis of the data. Observations that have nonpositive or missing weights are not included in the analysis. The values of the WEIGHT variable can be nonintegral and are not truncated. When you omit the WEIGHT statement, all observations that are used in the analysis are assigned a weight of 1.

Details: SHAPLEY Procedure

Required Specifications

The following specifications are required in order to run the SHAPLEY procedure:

- A reference data table. Typically, this is the data table that you use to train the model.
- A list of input variables. These are the model inputs that are used to train the model. Input variables can be interval or nominal, but you must identify any numeric inputs that you want to treat as nominal by using the LEVEL=NOMINAL option in the INPUT statement. String variables are always treated as nominal.
- An input data table that has a single row, which is known as the query table. This row is the observation that acts as the focus of the model explanation.
- A predicted target variable. This is the output variable that contains the model's predictions when the scoring function is applied to the input data table. The values of the variable can represent numeric predictions or predicted probabilities for an interval or nominal target, respectively. In many SAS Viya machine learning models, the name of the variable is constructed as follows: for an interval target, "P_" is prefixed to the name of the target variable (for example, "P_MSRP"); for a nominal target, "P_" is prefixed to the name of the target variable, and the value that corresponds to the target event is appended (for example, "P_Bad1").
- For the HyperSHAP method, you must provide a scoring function, which must be one of the following: an analytic store; several analytic stores and DS2 code; or DATA step scoring code that is created by the modeling algorithm.

- For the Kernel SHAP method, you must provide either a scoring function or prescored data. If you omit the scoring function by omitting the ASTOREMODEL and CODE statements, you must specify the USERAWDATA option in the METHOD statement and provide prescored query and reference data tables.

Shapley Values

Shapley values are used to explain the contribution of each variable to the prediction of a single observation, which is called a query (Molnar 2019). The Shapley value for the ith variable of a model $f : \mathbb{R}^m \rightarrow \mathbb{R}$ for the query, $\boldsymbol{q} \in \mathbb{R}^m$, is defined as

$$\phi_i(f, \boldsymbol{q}) = \sum_{S \subset V \setminus \{i\}} \frac{(m-s-1)!s!}{m!} [f_{S \cup \{i\}}(\boldsymbol{q}_{S \cup \{i\}}) - f_S(\boldsymbol{q}_S)]$$

where

- $V \setminus \{i\}$ is the set of variables that excludes the ith variable.
- $s = |S|$ is the number of variables in subset S.
- f_S is the model that is trained over the set of variables in subset S.
- $\boldsymbol{q}_S$ is the subset of the query that includes only the variables in subset S.

Shapley values are calculated by retraining a model on different subsets of variables. However, retraining a model requires knowledge of the model and its hyperparameters; this requirement conflicts with the goal of model-agnostic explanations.

To avoid model retraining, you approximate Shapley values by considering the change in a model's prediction when a subset of the variable values in the observations are selectively set equal to a query's variable values. You can write these changes in model prediction as expected differences in variable subsets, yielding the following equation for the Shapley values:

$$\phi_i(f, \boldsymbol{q}) = \sum_{\boldsymbol{z} \in [0,1]^m, z_i = 0} \frac{(m - |\boldsymbol{z}| - 1)!|\boldsymbol{z}|!}{m!} \left(\mathbb{E}_{\boldsymbol{z}+\boldsymbol{e}_i}[(f(\boldsymbol{x})] - \mathbb{E}_{\boldsymbol{z}}[f(\boldsymbol{x})] \right)$$

where

- $\boldsymbol{z} \in [0, 1]^m$, $z_i = 0$ is a vector of length m whose ith element is set to 0.
- $\boldsymbol{z} + \boldsymbol{e}_i$ is the vector $\boldsymbol{z}$, whose ith element has been set to 1.

The selection vector, $\boldsymbol{z}$, is defined as follows with regard to a subset of variables S:

$$z_j = \begin{cases} 1 & \text{variable } j \in S \\ 0 & \text{variable } j \notin S \end{cases}$$

An important feature of Shapley values for predictive models is that

$$f(\boldsymbol{q}) = \phi_0 + \sum_i \phi_i$$

where $\phi_0 = \mathbb{E}[f(\boldsymbol{x})]$ is the expected value of the model over the set of observations $\boldsymbol{x} \in X$. Thus, the Shapley values tell you which variable values influence a prediction the most, and in what way, on a query-by-query basis.

HyperSHAP Method

The HyperSHAP method uses a hyperparameter χ, known as the depth, that controls the allowable subsets that are used to approximate the Shapley values. Instead of computing expected differences for all variable subset combinations, the HyperSHAP method restricts calculations to the following subsets, based on the definition of the selection vector:

$$\mathcal{S}_i^{\chi+} = \{\boldsymbol{z} : z_i = 0, 0 \le |\boldsymbol{z}| \le \chi - 1\}$$
$$\mathcal{S}_i^{\chi-} = \{\boldsymbol{z} : z_i = 0, m - \chi \le |\boldsymbol{z}| \le m - 1\}$$

Thus, for a given value of χ, the HyperSHAP method considers only subsets of variables that have at most $\chi - 1$ selected variables or at least $m - \chi$ selected variables. The calculation of the Shapley value for the ith variable is

$$\phi_i^{\chi}(f, \boldsymbol{q}) = \sum_{\boldsymbol{z} \in \mathcal{S}_i^{\chi-} \cup \mathcal{S}_i^{\chi+}} w_\chi(|\boldsymbol{z}|) \left(\mathbb{E}_{\boldsymbol{z}+\boldsymbol{e}_i}[f(\boldsymbol{x})] - \mathbb{E}_{\boldsymbol{z}}[f(\boldsymbol{x})]\right)$$

where the weight function w_χ is

$$w_\chi(k) = \begin{cases} \frac{k!(m-k-1)!}{m!} \cdot \frac{m}{2\chi}, & \chi < \lceil m/2 \rceil \\ \frac{k!(m-k-1)!}{m!}, & \text{otherwise} \end{cases}$$

The depth hyperparameter enables you to achieve balance between the speed and level of approximation.

Depth

The DEPTH= option in the METHOD statement specifies the level χ in the HyperSHAP approximation. Currently, PROC SHAPLEY supports a depth of 1 or 2.

Kernel SHAP Method

The Kernel SHAP method approximates the Shapley values by using a linear regression in which the observations are weighted according to their distance from the query observation. The observations that you use in this regression have either of two sources: they come from generated samples that are based on the data table that you specify in the REFERENCEDATA= option in the PROC SHAPLEY statement, or they are the raw observations that the data table itself includes. You can use the SAMPLESIZE= and SEED= options in the METHOD statement to control the generation of data that fit the univariate reference distribution of your input data. Alternatively, you can specify the USERAWDATA option in the METHOD statement to indicate that the procedure should use the reference data directly instead of generating sample data points.

The weight calculation for the regression is a two-step process. The specification of the weights in the regression follows Lundberg and Lee (2017). First, for each variable, you calculate a derived variable that has a value of 0 or 1, based on the following rules:

- For nominal variables, the derived variable has a value of 1 if the original variable has the same value as the same variable in the query observation. Otherwise, the derived value is 0.
- For interval variables, the derived variable has a value of 1 if the absolute difference between the value of the original variable in the observation and the value of the original variable in the query is less than the standard deviation of that variable times the bin width, which you specify in the BINWIDTH= option in the METHOD statement. Otherwise, the derived variable has a value of 0.

The weight is then calculated as

$$w = \frac{M - 1}{C(M, |z|)|z|(M - |z|)}$$

where M is the number of variables and $|z|$ is the number of variables for which a value of 1 was calculated in the first step. For the case where $|z| = 0$ or $|z| = M$, the weight is set to 1E6.

In the second step, you train the weighted regression by using the predictor variables as the derived variables and the response variable as the predicted target that you specify in the PREDICTEDTARGET statement.

The parameter estimates from the weighted regression are the approximate Shapley values.

Frequency and Weight Variables

The optional FREQ statement names a numeric variable that provides the frequency of each observation in the reference data set. If you do not specify a frequency variable, then PROC SHAPLEY assumes that the frequency of all observations is 1. The procedure expects nonnegative integer frequency values. If the frequency of an observation is not an integer, the frequency value is truncated to the largest integer that is less than or equal to the specified value. The procedure treats negative and missing frequencies as 0.

The optional WEIGHT statement names a numeric variable that provides the weight of each observation in the reference data set. If you do not specify a weight variable, then PROC SHAPLEY assumes that the weight of all observations is 1. It expects nonnegative weight values. The procedure treats negative and missing weights as 0.

PROC SHAPLEY uses both the frequency and weight variables in the reference data set to calculate the expectations that are used to approximate the Shapley value. It does not use the frequency or weight variable from the query data, even if the variables exist. The query data are always treated as if they have a frequency and weight of 1.

Results

When you specify the HYPERSHAP *method-type* option in the METHOD statement, PROC SHAPLEY creates the following ODS table.

ShapleyValues Table

The ShapleyValues table contains the estimated Shapley values for the query data. This table also contains an intercept value, which is the estimated average predicted value of the model that is being explained.

When you specify the KERNELSHAP *method-type* option in the METHOD statement, PROC SHAPLEY creates the following ODS tables.

ExplainerFidelity Table

The ExplainerFidelity table contains information about how accurate the regression model is in comparison to the machine learning model. The table displays the machine learning model and regression model predictions on the query observation, as well as the overall root mean square error for the regression model.

ExplainerInfo Table

The ExplainerInfo table contains information about the settings that were used in the procedure. This table helps you verify that the procedure used your intended settings.

ParameterEstimates Table

The ParameterEstimates table contains the parameter estimates from the regression model, along with information about the values of the input variables for the query. These estimates are the Shapley value approximations.

ODS Table Names

PROC SHAPLEY assigns a name to each table that it creates. You can use these names to refer to the table when you use the Output Delivery System (ODS) to select tables and create output data tables. Table 31.2 shows these table names and descriptions.

Table 31.2 ODS Tables Produced by PROC SHAPLEY

Table Name	Description	Statement	*method-type* Option
ExplainerFidelity	Indicator of the accuracy of the weighted linear regression model	METHOD	KERNELSHAP
ExplainerInfo	Information about what settings were used	METHOD	KERNELSHAP
ParameterEstimates	Parameter estimates for the weighted linear regression. These are the Shapley value approximations.	METHOD	KERNELSHAP
ShapleyValues	Shapley value approximations per variable	METHOD	HYPERSHAP

Examples: SHAPLEY Procedure

Example 31.1: Compute Shapley Values by Using the Kernel SHAP Method

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

This example shows how to use the SHAPLEY procedure to explain a prediction that is made by a forest model by using the Kernel SHAP method.

The following DATA step creates the input data table mylib.dmagecr:

```
data mylib.dmagecr;
   set sampsio.dmagecr;
   keep age amount coapp duration foreign job good_bad;
run;
```

The following DATA step creates the query data table mylib.query in your SAS library:

```
data mylib.query;
   set sampsio.dmagecr;
   keep age amount coapp duration foreign job good_bad;
   if _N_ = 20;
run;
```

The following statements run the FOREST procedure to build a forest model to predict whether the credit rating of each individual in the dmagecr data table is good or bad. The procedure also outputs an analytic store named forest_astore.

```
proc forest
   data=mylib.dmagecr
   seed=1234;
   input age amount duration / level=interval;
   input coapp foreign job /level=nominal;
   target good_bad;
   savestate rstore=mylib.forest_astore;
run;
```

The following statements run PROC SHAPLEY and produce ODS tables of the results:

```
proc shapley
   data=mylib.query
   referenceData=mylib.dmagecr;
   input age amount duration / level=interval;
   input coapp foreign job /level=nominal;
   predictedTarget P_good_badbad;
   method KERNELSHAP(seed=1234);
   astoreModel rstore=mylib.forest_astore;
run;
```

The DATA= option specifies the query data table. The REFERENCEDATA= option specifies the reference data table to use for Shapley analysis. The two INPUT statements specify that the age, amount, coapp, duration, foreign, and job variables be used as inputs. The PREDICTEDTARGET statement specifies that the P_good_badbad variable be used as the predicted target. The METHOD statement specifies that the Kernel SHAP method be used to approximate Shapley values. The SEED= option specifies the seed to use for pseudorandom number generation. The ASTOREMODEL statement with the RSTORE= option specifies that the analytic store that is saved in the data table mylib.forest_astore be used as the model for analysis.

The "Explainer Information" table, "Parameter Estimates" table, and "Explainer Fidelity" table that these statements produce are shown in Output 31.1.1, Output 31.1.2, and Output 31.1.3, respectively.

Output 31.1.1 Explainer Information

The SHAPLEY Procedure

Explainer Information	
Data Generation Method	Reference Distribution
Distance Method	SHAP Kernel
Explainer Type	Regression
Binary Encoding	All
Standardize Parameter Estimates	None
Include Missing as a Level	No
Number of Samples	3000
Bin Width	0.1
Seed	1234

Output 31.1.2 Parameter Estimates

Parameter Estimates								
		Nominal Variables						
Variable	Binary Variable	coapp	foreign	job	Estimate	Query Level	Lower Bound	Upper Bound
Intercept		.	.	.	0.1257126945	.	.	.
age	_bCode_AGE_	.	.	.	0.045381354	31	29.862453143	32.137546857
amount	_bCode_AMOUNT_	.	.	.	-0.026776358	3430	3147.7263124	3712.2736876
duration	_bCode_DURATION_	.	.	.	0.0051611944	24	22.794118555	25.205881445
coapp	_bCode_COAPP_	1	.	.	0.0248311738	1	.	.
foreign	_bCode_FOREIGN_	.	1	.	0.1876355464	1	.	.
job	_bCode_JOB_	.	.	3	-0.020327907	1	.	.

Output 31.1.3 Explainer Fidelity

Explainer Fidelity		
Model Prediction	Explainer Prediction	Explainer RMSE
0.3416687942	0.3416176981	0.1583813819

Example 31.2: Compute Shapley Values (Kernel SHAP Method) Using Prescored Data

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

This example shows how to use the SHAPLEY procedure to explain a prediction that is made by a forest model by using the Kernel SHAP method with prescored data.

NOTE: PROC SHAPLEY does not require a scoring function and does not generate new data when you use prescored data in this way. This can reduce computation time but can also result in less accurate Shapley value estimations. You should consider supplying the scoring function and allow the procedure to generate new data if you have access to the analytic store or the scoring code of the machine learning model.

The following DATA step creates the input data table mylib.dmagecr; adds the ID variable id; and keeps only the variables age, amount, coapp, duration, foreign, job, good_bad, and id:

```
data mylib.dmagecr;
   set sampsio.dmagecr;
   id = _N_;
   keep age amount coapp duration foreign job good_bad id;
run;
```

The following statements run the FOREST procedure to build a forest model to predict whether the credit rating of each individual in the dmagecr data table is good or bad. The procedure also outputs a scored data table named dmagecr_scored while copying all the input and ID variables.

```
proc forest
   data=mylib.dmagecr
   seed=1234;
   input age amount duration / level=interval;
   input coapp foreign job /level=nominal;
   target good_bad;
   output out=mylib.dmagecr_scored copyVars=(_ALL_);
run;
```

The following DATA step creates the prescored query data table mylib.query_scored in your SAS library:

```
data mylib.query_scored;
   set mylib.dmagecr_scored;
   if id = 20;
run;
```

The following statements run PROC SHAPLEY with the prescored data tables and produce ODS tables of the results:

```
proc shapley
   data=mylib.query_scored
   referenceData=mylib.dmagecr_scored;
   input age amount duration / level=interval;
   input coapp foreign job /level=nominal;
   predictedTarget P_good_badbad;
   method KERNELSHAP(seed=1234 useRawData);
run;
```

The DATA= option specifies the query data table. The REFERENCEDATA= option specifies the prescored data table as the reference data table to use for Shapley analysis. The two INPUT statements specify that the age, amount, coapp, duration, foreign, and job variables be used as inputs. The PREDICTEDTARGET statement specifies that the P_good_badbad variable be used as the predicted target. The METHOD statement specifies that the Kernel SHAP method be used to approximate Shapley values. The SEED= option specifies the seed to use for pseudorandom number generation. The USERAWDATA option specifies that PROC SHAPLEY should use the reference data table to approximate the Shapley values without generating new data.

NOTE: When you omit the ASTOREMODEL and CODE statements and specify the USERAWDATA option, PROC SHAPLEY requires both the query data table and the reference data table to be prescored.

The “Explainer Information” table, “Parameter Estimates” table, and “Explainer Fidelity” table that these statements produce are shown in Output 31.2.1, Output 31.2.2, and Output 31.2.3, respectively.

Output 31.2.1 Explainer Information

The SHAPLEY Procedure

Explainer Information	
Data Generation Method	None
Distance Method	SHAP Kernel
Explainer Type	Regression
Binary Encoding	All
Standardize Parameter Estimates	None
Include Missing as a Level	No
Bin Width	0.1
Seed	1234

Output 31.2.2 Parameter Estimates

Parameter Estimates								
		Nominal Variables						
Variable	**Binary Variable**	**coapp**	**foreign**	**job**	**Estimate**	**Query Level**	**Lower Bound**	**Upper Bound**
Intercept		.	.	.	0.1230277182	.	.	.
age	_bCode_AGE_	.	.	.	0.0207106122	31	29.862453143	32.137546857
amount	_bCode_AMOUNT_	.	.	.	-0.015444818	3430	3147.7263124	3712.2736876
duration	_bCode_DURATION_	.	.	.	0.0072105325	24	22.794118555	25.205881445
coapp	_bCode_COAPP_	1	.	.	-0.015812736	1	.	.
foreign	_bCode_FOREIGN_	.	1	.	0.1994427495	1	.	.
job	_bCode_JOB_	.	.	3	-0.009984922	1	.	.

Output 31.2.3 Explainer Fidelity

Explainer Fidelity		
Model Prediction	**Explainer Prediction**	**Explainer RMSE**
0.3276871873	0.3091491362	0.15974728

References

Lundberg, S. M., and Lee, S.-I. (2017). "A Unified Approach to Interpreting Model Predictions." In *Advances in Neural Information Processing Systems 30 (NIPS 2017)*, 4765–4774. La Jolla, CA: Neural Information Processing Systems Foundation.

Molnar, C. (2019). *Interpretable Machine Learning: A Guide for Making Black Box Models Explainable*. Self-published, https://christophm.github.io/interpretable-ml-book/.

Chapter 32
SPARSEML Procedure

Contents

Overview: SPARSEML Procedure **825**
PROC SPARSEML Features 826
Using SAS Viya Workbench 826
Getting Started: SPARSEML Procedure **826**
Syntax: SPARSEML Procedure **828**
PROC SPARSEML Statement 828
INPUT Statement 830
OUTPUT Statement 830
SAVESTATE Statement 831
Details: SPARSEML Procedure **831**
Displayed Output 833
ODS Table Names 833
Examples: SPARSEML Procedure **834**
Example 32.1: Sparse Data Set Example 834
Example 32.2: Junk Mail Classification 836
References **839**

Overview: SPARSEML Procedure

The SPARSEML procedure supports the use of sparse data as input data. It implements the support vector machines (SVM) algorithm in SAS Viya Workbench for binary classification. By using the coordinate descent optimization method, the SPARSEML procedure can train large-scale data sets that can be both wide and deep.

Like other predictive modeling tools in SAS Viya Workbench, the SPARSEML procedure uses input data to train a model and generates an analytic store that can be deployed through the ASTORE procedure.

PROC SPARSEML Features

The SPARSEML procedure has the following features:

- supports sparse data format
- supports classification of a binary target
- supports the coordinate descent method
- supports high dimensions of variables

Using SAS Viya Workbench

SAS Viya Workbench has greatly simplified the SAS programming experience by offering a syntax style nearly identical to that of SAS 9 and earlier releases. However, there are minor exceptions. In the past, SAS Viya procedures required the use of a CAS engine libref. This engine served multiple purposes, but it was primarily used to upload data to the CAS server and manage the data. Its additional purposes included support of new data types and improved performance for large data sets. No CAS server is needed for SAS Viya Workbench, because all computations are done in place for you in a single location.

In the following program, the LIBNAME statement creates a mylib libref that the examples in this chapter use to refer to your current or desired work directory:

```
libname mylib ".";
```

For more information, see *SAS Viya Workbench: Getting Started*.

Getting Started: SPARSEML Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

The following DATA step creates the data table smldata. This data set contains one variable named vars and 15 observations; vars is a character variable. Each observation contains the data information, which includes a target and a sequence of combined column indexes and column values. The target value is either 1 or –1. The column index starts at 1.

```
data smldata;
    length vars $ 20.;
    input vars $ 1-20;
    datalines;
1 1:-1 2:3
1
1 1:1 2:1
```

```
1 1:2 2:2
1 1:3 2:3
1 1:4 2:4
1 1:5 2:5
-1 2:2
-1 1:1 2:3
-1 1:2 2:4
-1 1:3 2:5
1 1:0 2:-5
1 1:5
-1 1:0 2:5
-1 1:2 2:8
;
run;
```

You can load the smldata data set into your SAS library in the following DATA step:

```
data mylib.smldata;
    set smldata;
run;
```

The following statements use PROC SPARSEML to run the sparse machine learning algorithm on the mylib.smldata data table:

```
proc sparseml data= mylib.smldata;
    input vars;
run;
```

The INPUT statement defines the input variable vars, which contains both target and sparse input values in sparse string format.

PROC SPARSEML generates several ODS tables, some of which are shown in Figure 32.1 through Figure 32.3.

The "Data Information" table in Figure 32.1 shows that the number of observations is 15, the number of features is 2, and the number of sparse elements is 26.

Figure 32.1 Sparse Data Information

The SPARSEML Procedure

Data Information	
Number of Rows	15
Number of Features	2
Number of Sparse Elements	26

The "Misclassification Matrix" table in Figure 32.2 shows that among the total of fifteen observations, nine observations are classified as 1, and six observations are classified as –1. The number of correctly predicted 1 observations is eight, and the number of correctly predicted –1 observations is six. Thus the accuracy is 93.33%, as indicated in the "Fit Statistics" table in Figure 32.3.

Figure 32.2 Misclassification Matrix

Misclassification Matrix			
	Training Prediction		
Observed	1	-1	Total
1	8	1	9
-1	0	6	6
Total	8	7	15

Figure 32.3 Fit Statistics

Fit Statistics	
Statistic	Training
Accuracy	0.9333
Error	0.0667
Sensitivity	0.8889
Specificity	1.0000

A relatively good model means that misclassification is low and both sensitivity and specificity are high.

Syntax: SPARSEML Procedure

The following statements are available in the SPARSEML procedure:

```
PROC SPARSEML <options> ;
    INPUT variable ;
    OUTPUT OUT=libref.data-table <option> ;
    SAVESTATE RSTORE=libref.data-table ;
```

The PROC SPARSEML statement and the INPUT statement are required.

The following sections describe the PROC SPARSEML statement and then describe the other statements in alphabetical order.

PROC SPARSEML Statement

```
PROC SPARSEML <options> ;
```

The PROC SPARSEML statement invokes the procedure.

You can specify the following *options*:

C=*number*
specifies the penalty value, where *number* must be a real number greater than 0.

By default, C=10.

DATA=*libref.data-table*
names the input data table for PROC SPARSEML to use. The default is the most recently created data table. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to the data. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 826.

data-table specifies the name of the input data table.

MAXITERS=*number*
specifies the maximum number of iterations before the process stops, where *number* is a positive integer.

By default, MAXITERS=100. In some cases, you can obtain a good model in fewer than 10 iterations.

NTHREADS=*number-of-threads*
specifies the number of threads to use in the computation. The default value is the number of CPUs available on the machine.

PRINTLEVEL=*level-value*
specifies a level value from which the ODS information tables are printed. You can specify the following *level-values*:

AGGRESSIVE generates all information tables.

BASIC generates basic information tables.

MODERATE generates standard information tables.

NONE suppresses information tables.

By default, PRINTLEVEL=MODERATE.

REGL1=*number*
specifies the L1 norm penalization weight. If *number* is greater than 0, then the L1 norm penalization is performed.

By default, REGL1=0. If you specify a positive *number*, the penalty value C is ignored.

NOTE: The *number* must be greater than or equal to 0. You can specify either the REGL1= or REGL2= option, but not both.

REGL2=*number*
specifies the L2 norm penalization weight. If *number* is greater than 0, then the L2 norm penalization is performed.

By default, REGL2=0. If you specify a positive *number*, the penalty value C is ignored.

NOTE: The *number* must be greater than or equal to 0. You can specify either the REGL2= or REGL1= option, but not both.

SEED=*value*
specifies a nonnegative integer as the random number seed.

By default, SEED=1.

TOLERANCE=*number*
specifies the minimal absolute tolerance at which the iteration stops. The *number* must be greater than 0.

By default, TOLERANCE=1.0E–6. In some cases, you can obtain a good model by using a *number* greater than or equal to 0.01.

INPUT Statement

INPUT *variable* **;**

The INPUT statement specifies the name of the input variable to use in training. The input variable is a character string that contains a target and a sequence of space-separated sparse variable values. The format of the input variable is *target* column_1:value_1 column_2:value_2 ... column_n:value_n.

The *target* is either 1 or –1. The value column_i is a positive integer that is the column index starting at 1, where $i = 1, 2, \ldots, n$. The value value_i is a double number that is the corresponding column value.

NOTE: For more information, see Example 32.1.

OUTPUT Statement

OUTPUT OUT=*libref.data-table* < *option* > **;**

The OUTPUT statement creates an output data table that contains the predicted values of the input data table.

You must specify the following option:

OUT=*libref.data-table*
names the output data table for PROC SPARSEML to use. You must specify this option before any other options. *libref.data-table* is a two-level name, where

libref	refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to where the data table is to be stored. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 826.
data-table	specifies the name of the output data table.

You can also specify the following *option*:

COPYVAR=*variable*

COPYVARS=(*variables***)**

lists one or more variables from the input data table that are transferred to the output data table.

SAVESTATE Statement

SAVESTATE RSTORE=*libref.data-table* **;**

The SAVESTATE statement creates an analytic store for the model and saves it as a binary object in a data table. You can use the analytic store in the ASTORE procedure to score new data. For more information, see Chapter 4, "ASTORE Procedure."

You must specify the following option:

RSTORE=*libref.data-table*

specifies a data table in which to save the analytic store for the model. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 826.

Details: SPARSEML Procedure

PROC SPARSEML uses a linear kernel to compute the support vector machine (SVM) learning classifiers for the binary pattern recognition problem. SVM is a powerful tool for finding robust classifiers; see Vapnik (1995); Burges (1998); Hsieh et al. (2008). Given a data set $X = \{x_1, x_2, \ldots, x_n\}$ and the corresponding labels $Y = \{y_1, \ldots, y_n\}$, the traditional SVM fitting model can be formulated as

$$\begin{array}{lrcll} \underset{w,b,\xi}{\text{minimize}} & \frac{1}{2}\|w\|^2 + C_{L_1}\sum_{i=1}^{n}\xi_i & & & \\ \text{subject to} & y_i(w^T x_i + b) & \geq & 1-\xi_i & \forall i \\ & \xi_i & \geq & 0 & \forall i \end{array}$$

where (w, b) determine the classifier and ξ_i, for each data point i, measures the error based on hinge loss that penalizes the misclassified points according to their distance from the corresponding classification margin.

The preceding model is called the L1-SVM model because the errors are penalized using the L1 norm. Similarly, the L2-SVM model can be formulated as

$$\begin{array}{lrcll} \underset{w,b,\xi}{\text{minimize}} & \frac{1}{2}\|w\|^2 + C_{L_2}\sum_{i=1}^{n}\xi_i^2 & & & \\ \text{subject to} & y_i(w^T x_i + b) & \geq & 1-\xi_i & \forall i \\ & \xi_i & \geq & 0 & \forall i \end{array}$$

You can select these models by setting the parameters C_{L_1} (using the REGL1= option) and C_{L_2} (using the REGL2 option), respectively. If you specify the REGL1= option, then the L1-SVM model is used. If you specify the REGL2= option, then the L2-SVM model is used. If neither option is specified, then the L1-SVM model with a default penalty value (10) is assumed.

For performance purposes, the related dual problem would be solved. The following reformulates the L1-SVM model to produce the dual problem,

$$\begin{array}{ll} \underset{\alpha}{\text{minimize}} & \frac{1}{2}\sum_{i=1}^{n}\sum_{j=1}^{n}\alpha_i\alpha_j y_i y_j (x_i^T x_j) - \sum_{i=1}^{n}\alpha_i \\ \text{subject to} & \sum_{i=1}^{n}\alpha_i y_i = 0 \\ & 0 \le \alpha_i \le C \end{array}$$

where the classifier would be recovered by

$$w = \sum_i \alpha_i y_i x_i$$

The algorithm that is used to solve the dual problem is referred to as the coordinate descent algorithm. It works on updating each coordinate by finding the δ that minimizes $f(\alpha + \delta e_i)$, where e_i is the ith column of the identity matrix. This is equivalent to solving a one-dimensional problem,

$$d := \arg \min_{0 \le \alpha + d e_i \le C} f(\alpha + d e_i) = \frac{1}{2}\|x_i\|^2 d^2 + (\sum_j \alpha_j (y_i y_j x_i^T x_j) - 1)d + \text{Const}$$

which is a quadratic that can easily be minimized. Defining $G := (\sum_j \alpha_j (y_i y_j x_i^T x_j) - 1)$, which is referred to as the gradient, would show whether a certain coordinate is suitable to be updated. Especially given the box constraint, the projected gradient (PG) would help determine suitable coordinates by setting

$$\text{PG} = \begin{cases} G_i & \text{if } 0 < \alpha_i < C \\ \min(0, G_i) & \text{if } \alpha_i = 0 \\ \max(0, G_i) & \text{if } \alpha_i = C \end{cases}$$

thus updating each coordinate at a time. The randomized coordinate descent algorithm is suitable for large-scale sparse SVM models.

Displayed Output

The following sections describe the output that PROC SPARSEML produces. The output is organized into various tables, which are described in the order of their appearance.

Number of Observations

The "Number of Observations" table contains the number of observations that are read and the number of observations that are used.

Data Information

The "Data Information" table contains the number of rows, the number of features, and the number of sparse elements.

Model Information

The "Model Information" table contains the initial training settings, such as task type, optimization technique, and so on.

Misclassification Matrix

The "Misclassification Matrix" table contains the target information, both observed and predicted. The columns include the observed target, predicted event, predicted nonevent, and total numbers of events or nonevents for the training data.

Fit Statistics

The "Fit Statistics" table contains the model accuracy information, which includes accuracy, error, sensitivity, and specificity. The statistics are calculated from the "Misclassification Matrix" table.

Run Time Summary

The "Run Time Summary" table contains the time that each task uses.

ODS Table Names

Each table that the SPARSEML procedure creates has a name associated with it. You must use this name to refer to the table when you use ODS statements. The name of each table and a short description of the contents are listed in Table 32.1.

Table 32.1 ODS Tables Produced by PROC SPARSEML

Table Name	Description	PRINTLEVEL= Value
DataInfo	Information for the training data	BASIC

Table 32.1 *continued*

Table Name	Description	Option
FitStats	Accuracy information about the training	BASIC
Misclassification	Misclassification matrix table	MODERATE
ModelInfo	Basic model information for the training	BASIC
NObs	Observation information about the input data	BASIC
TaskTime	Time used by each task	AGGRESSIVE

Examples: SPARSEML Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

Example 32.1: Sparse Data Set Example

This example uses the SPARSEML procedure to train a model by using the same data set, smldata, as in the section "Getting Started: SPARSEML Procedure" on page 826. Here, you can add a column id to the smldata data set to serve as the ID variable, as shown in the following statements:

```
data smldata;
    id = _N_;
    set smldata;
run;
```

You can load the smldata data set into your SAS library in the following DATA step:

```
data mylib.smldata;
    set smldata;
run;
```

The following statements use PROC SPARSEML to run the sparse machine learning algorithm on the mylib.smldata data table and generate the analytic store mylib.mystate:

```
proc sparseml data= mylib.smldata
    TOLERANCE=0.01
    C = 5.5
    MAXITERS=10
    ;
    input vars;
    savestate rstore=mylib.mystate;
    output out=mylib.trainingout copyvars=(id);
run;
```

The PROC SPARSEML statement has options such as TOLERANCE=, C=, and MAXITERS=. The INPUT statement defines the input variable vars, which includes both target and sparse input values. The

SAVESTATE statement specifies the remote data table mylib.mystate, where the analytic store is saved. The OUTPUT statement shows that the training output table is mylib.trainingout, which includes the variable id.

PROC SPARSEML generates several ODS tables, some of which are shown in Output 32.1.1 through Output 32.1.3.

The "Data Information" table in Output 32.1.1 shows that the number of observations is 15, the number of features is 2, and the number of sparse elements is 26.

Output 32.1.1 Sparse Data Information

The SPARSEML Procedure

Data Information	
Number of Rows	15
Number of Features	2
Number of Sparse Elements	26

The "Misclassification Matrix" table in Output 32.1.2 shows that among the total of fifteen observations, nine observations are classified as 1, and six observations are classified as –1. The number of correctly predicted 1 observations is four, and the number of correctly predicted –1 observations is six. Thus the accuracy is 66.67%, as indicated in the "Fit Statistics" table in Output 32.1.3.

Output 32.1.2 Misclassification Matrix

Misclassification Matrix			
	Training Prediction		
Observed	1	-1	Total
1	4	5	9
-1	0	6	6
Total	4	11	15

Output 32.1.3 Fit Statistics

Fit Statistics	
Statistic	Training
Accuracy	0.6667
Error	0.3333
Sensitivity	0.4444
Specificity	1.0000

The generated analytic store mylib.mystate can be used by PROC ASTORE to score data. For example, you can score the mylib.smldata data table as follows:

```
proc astore;
    score data=mylib.smldata
       out=mylib.smlscore
       rstore=mylib.mystate
       copyvars=(id)
       ;
```

```
    run;
quit;
```

The score output data are saved in the mylib.smlscore data table.

Example 32.2: Junk Mail Classification

This example shows how to use the SPARSEML procedure to create a model and use the model to predict whether an email is a junk email or a regular email in the Sashelp.JunkMail data set.

The Sashelp.JunkMail data set comes from a study that classifies whether an email is junk email (coded as 1) or not (coded as 0). The data were collected by Hewlett-Packard Labs and donated by George Forman. The data set Sashelp.JunkMail contains 4,601 observations, with 2 binary variables and 57 continuous explanatory variables. The response variable, Class, is a binary indicator of whether an email is considered spam or not. The partitioning variable, Test, is a binary indicator that is used to divide the data into training and testing sets. The 57 explanatory variables are continuous variables that represent frequencies of some common words and characters and lengths of uninterrupted sequences of capital letters in emails. For more information about the Sashelp.JunkMail data set, see the partition data example in the documentation of the LOGSELECT procedure in *SAS Visual Statistics: Procedures for SAS Viya Workbench*.

The following DATA step converts the Sashelp.JunkMail data to the sparse data format so that you can apply it to PROC SPARSEML. The converted table JunkMail contains three variables: Test, Class, and sparseString.

```
data JunkMail;
    length sparseString $600;
    length oneVariable  $60;
    set Sashelp.JunkMail;
    array vars{*} _NUMERIC_;
    if Class=0 then do;
        sparseString = put(-1, 2.);
    end; else do;
        sparseString = put(+1, 2.);
    end;
    do i=2 to 58;
        if vars{i} ne 0 then do;
            oneVariable = strip(put(i-1, 5.))||':'||strip(put(vars{i}, 10.3));
            sparseString = strip(sparseString)||' '||oneVariable;
        end;
    end;
    keep Test Class sparseString;
run;
```

The following DATA step loads the training data mylib.mailtrain into the SAS library. The training data contain about two-thirds of the data, which have a Test value of 0.

```
data mylib.mailtrain;
    id = _N_;
    set JunkMail;
    where Test=0;
    drop Test;
run;
```

The following DATA step loads the testing data mylib.mailtest into the SAS library. The testing data contain about one-third of the data, which have a Test value of 1.

```
data mylib.mailtest;
    id = _N_;
    set JunkMail;
    where Test=1;
    drop Test;
run;
```

The following statements perform the model training. In the options of the PROC SPARSEML statement, the penalty value C= is 0.1, and the value of the maximum iteration option MAXITERS= is 400. The SAVESTATE statement specifies that the trained model is saved in the analytic store mylib.mystate.

```
proc sparseml data=mylib.mailtrain
    C=0.1
    MAXITERS=400
;
    input sparseString;
    savestate rstore=mylib.mystate;
run;
```

The sparse data information, misclassification matrix, and fit statistics tables are shown in Output 32.2.1 through Output 32.2.3.

Output 32.2.1 Sparse Data Information

The SPARSEML Procedure

Data Information	
Number of Rows	3065
Number of Features	57
Number of Sparse Elements	39801

Output 32.2.2 Misclassification Matrix

Misclassification Matrix			
	Training Prediction		
Observed	1	-1	Total
1	1030	188	1218
-1	144	1703	1847
Total	1174	1891	3065

Output 32.2.3 Fit Statistics

Fit Statistics	
Statistic	**Training**
Accuracy	0.8917
Error	0.1083
Sensitivity	0.8456
Specificity	0.9220

To score testing data, you can use PROC ASTORE. The following statements score the testing data table mylib.mailtest and save the results in the data table mylib.scoreout:

```
proc astore;
    score data=mylib.mailtest out=mylib.scoreout
          rstore=mylib.mystate copyvars=(id Class);
run;
quit;
```

To calculate the misclassification for the testing data, the following DATA step creates the variable Score to show the predicted value:

```
data scoreout;
    set mylib.scoreout;
    Score=_I_;
    if Score=-1 then Score=0;
run;
```

You can compare the predicted value Score with the true value Class by using PROC FREQ. The results of the comparison are displayed in Output 32.2.4.

```
proc freq data=scoreout;
    table Class*Score;
run;
```

Output 32.2.4 Crosstabular Frequency Table

The FREQ Procedure

Frequency Percent Row Pct Col Pct	Table of Class by Score			
	Class(0 - Not Junk, 1 - Junk)	**Score**		
		0	**1**	**Total**
	0	859 55.92 91.29 89.76	82 5.34 8.71 14.16	941 61.26
	1	98 6.38 16.47 10.24	497 32.36 83.53 85.84	595 38.74
	Total	957 62.30	579 37.70	1536 100.00

The misclassification rate for the testing data is $(82 + 98)/1536 = 11.72\%$.

References

Burges, C. J. C. (1998). "A Tutorial on Support Vector Machines for Pattern Recognition." *Data Mining and Knowledge Discovery* 2:121–167.

Hsieh, C.-J., Chang, K.-W., Lin, C.-J., Keerthi, S. S., and Sundararajan, S. (2008). "A Dual Coordinate Descent Method for Large-Scale Linear SVM." In *Proceedings of the 25th International Conference on Machine Learning*, 408–415. New York: ACM.

Vapnik, V. N. (1995). *The Nature of Statistical Learning Theory*. New York: Springer-Verlag.

Chapter 33
STYLEGAN Procedure

Contents

Overview: STYLEGAN Procedure **842**
- PROC STYLEGAN Features 842
- Using SAS Viya Workbench 842

Getting Started: STYLEGAN Procedure **843**

Syntax: STYLEGAN Procedure **847**
- PROC STYLEGAN Statement 847
- OPTIMIZATION Statement 848
- OUTPUT Statement 849
- SAVESTATE Statement 849
- TRAIN Statement 850

Details: STYLEGAN Procedure **852**
- Optimization 856
- Invalid Images 857
- Displayed Output 857
  - Iteration History 857
  - Model Information 857
  - Number of Observations 857
  - Timing 857
  - Table Output 857
- ODS Table Names 858

Examples: STYLEGAN Procedure **858**
- Example 33.1: Generating Human Face Images 858
- Example 33.2: Generating Human Face Images — Continue Training 861

References **862**

Overview: STYLEGAN Procedure

The STYLEGAN Procedure trains a Style GAN model on images in SAS Viya Workbench. PROC STYLEGAN implements the Style GAN version 2 model by using the PyTorch library.

PROC STYLEGAN Features

The STYLEGAN procedure has the following main features:

- It performs generative modeling by using the state-of-the-art Style GAN 2 model, whose generator network can be used to generate realistic images.
- It supports multithreaded training on one graphics processing unit (GPU) device.

Using SAS Viya Workbench

SAS Viya Workbench has greatly simplified the SAS programming experience by offering a syntax style nearly identical to that of SAS 9 and earlier releases. However, there are minor exceptions. In the past, SAS Viya procedures required the use of a CAS engine libref. This engine served multiple purposes, but it was primarily used to upload data to the CAS server and manage the data. Its additional purposes included support of new data types and improved performance for large data sets.

No CAS server is needed for SAS Viya Workbench, because all computations are done in place for you in a single location. However, this SAS procedure needs to be able to use these new data and file types, so an advanced engine is required. For this purpose, SAS Viya Workbench supports the SASVIYA engine.

In the following program, the LIBNAME statement creates mylib and mydirlib librefs that the examples in this chapter use:

```
libname mylib sasviya;
libname mydirlib "/path/to/images";
```

Getting Started: STYLEGAN Procedure

NOTE: This example assumes that the SAS 9 engine libref mydirlib and the SASVIYA engine libref mylib have already been created. For more information, see the section "Using SAS Viya Workbench" on page 842 and Chapter 1, "Shared Concepts" (*SAS Viya Workbench: Computer Vision Procedures*).

This example shows how to use the STYLEGAN procedure to train a Style GAN model and generate new images by using a data set that contains 60,000 gray-scale images of fashion items at a resolution of 28×28 pixels. The data set includes 10 classes that represent different types of clothing and accessories: T-shirts, pants, pullovers, dresses, coats, sandals, shirts, sneakers, bags, and ankle boots. The Fashion MNIST data set is available at Deep Learning Example Data. Note that the shape of the input image for PROC STYLEGAN must be square and selected from {4, 8, 16, 32, 64, 128, 256, 512, 1024}. Figure 33.1 shows 100 images that are randomly selected from the Fashion MNIST data set.

Figure 33.1 100 Images Randomly Selected from the Fashion MNIST Data Set

First, you use the LOADIMAGES procedure (see Chapter 10, "LOADIMAGES Procedure" (*SAS Viya Workbench: Computer Vision Procedures*)) as follows to load the images from the specified path into the table mylib.fashionImages in your SAS library. This assumes that your SAS libref is named mylib, as in the section "Using SAS Viya Workbench" on page 842, but you can substitute any appropriately defined SAS libref. The specified path must be relative to the path that is associated with the mydirlib libref:

```
proc loadimages libref=mydirlib path='path-to-images-folder/';
   output out=mylib.fashionImages;
run;
```

Then, you use the PROCESSIMAGES procedure (see Chapter 12, "PROCESSIMAGES Procedure" (*SAS Viya Workbench: Computer Vision Procedures*)) as follows to resize the 28×28-pixel images that are loaded in the table mylib.fashionImages to 32×32-pixel images and to save the resized images in the mylib.resizedImages table:

```
proc processimages data=mylib.fashionImages;
     resize type=keep_aspect_ratio width=32;
     output out=mylib.resizedImages;
run;
```

Next, you use the following PROC STYLEGAN statements to train a Style GAN model and save the trained generator network in an analytic store:

```
proc stylegan    data=mylib.resizedImages imagesize=32
                 numsamples=100 useGPU seed=123;
optimization     numiters=5;
savestate        rstore=mylib.astore;
output           out=mylib.outtable;
run;
```

The DATA= option specifies the input images data set. The IMAGESIZE= option specifies the input image size, which is the image width of the square images in pixels. You specify the number of images that you want to generate in the NUMSAMPLES= option. The USEGPU option specifies that a GPU device is used to train the Style GAN model. The SEED= option specifies the random seed to use in the training process. You specify the number of training iterations to use for the GAN model in the NUMITERS= option in the OPTIMIZATION statement. One iteration processes one minibatch of your training data. Note that for the Fashion MNIST data, you can get a converged model by running 30,000 iterations. In this example, small option values are used for brevity. The SAVESTATE statement specifies the table to use for saving the trained model, and the OUTPUT statement specifies the table to use for outputting result images.

PROC STYLEGAN generates four ODS tables.

Figure 33.2 shows the iteration history, including the generator and discriminator loss values for each epoch of training.

Figure 33.2 Iteration History

The STYLEGAN Procedure

Iteration History				
Iteration Number	Generator Loss	Discriminator Loss	Path Length Regularization Loss	R1 Regularization Loss
1	3.738638	1.524220	1.755333	0.009592
2	2.746564	0.952844	1.755333	0.009592
3	13.112035	1.174145	1.755333	0.009592
4	1.670602	1.189783	1.755333	0.009592
5	13.850622	0.723998	0.849263	0.009592

Figure 33.3 shows the number of observations that are read and number of observations that are actually used.

Figure 33.3 Number of Observations

Number of Observations	
Number of Observations Read	60000
Number of Observations Used	60000

Figure 33.4 shows the values of the parameters that the Style GAN model uses.

Figure 33.4 Model Information

Model Information	
Exponential Decay Rate for the First-Moment Estimates	0
Exponential Decay Rate for the Second-Moment Estimates	0.99
Width of Square Images in Pixels	32
Use Large Network	True
Learning Rate for the Optimizer	0.002
Number of Images in One Minibatch	32
Number of Iterations	5
Seed for Random Initialization	123
Discriminator Regularization Frequency	16
Generator Regularization Frequency	4
Weight for Path Length Regularization	2
Weight for R1 Regularization	10

Figure 33.5 illustrates the timing of the various steps that PROC STYLEGAN performs.

Figure 33.5 Timing

Timing	
Task	**Time in Seconds**
Reserving GPU Device	2.043964
Loading Training Data	4.929908
Setting Up Model	2.118650
Training Model	15.908062
Generating Images	0.331942
Saving Generator to Astore	1.513873

In this example, PROC STYLEGAN generates the score output table mylib.outtable in the active library. This table contains 100 generated images from the trained model. Figure 33.6 shows 100 images that were generated by using PROC STYLEGAN.

Figure 33.6 Images Generated Using PROC STYLEGAN

The following statements run the SAVEIMAGES procedure (see Chapter 16, "SAVEIMAGES Procedure" (*SAS Viya Workbench: Computer Vision Procedures*)) to write the generated images from the table mylib.outtable in bitmap format onto a disk:

```
proc saveimages libref=mydirlib path="path-to-write-images"
    data=mylib.outtable encoding=BMP;
run;
```

You can generate additional images by using the trained Style GAN model that is saved in the mylib.astore table and PROC ASTORE (see Chapter 4, "ASTORE Procedure.")

The following statements use PROC ASTORE to generate 10 images by using the trained Style GAN model that is stored in mylib.astore and then save the results in the mylib.astoreout table:

```
proc astore;
setoption GENOBS 10;
setoption SEED 42;
score rstore=mylib.astore
      out=mylib.astoreout;
run;
```

The GENOBS option in the SETOPTION statement specifies the number of images to generate. To generate different images, change the SEED option value in the SETOPTION statement.

NOTE: For Style GAN analytic stores that are trained on the 2026.02 release or later releases, you can generate a specific number of images without input data by specifying the GENOBS option when you call PROC ASTORE. However, for Style GAN analytic stores that are trained on releases earlier than 2026.02, you must provide an input data set that contains the number of rows that corresponds to the number of images that you want to generate. See the corresponding document version for details.

Syntax: STYLEGAN Procedure

The following statements are available in the STYLEGAN procedure:

PROC STYLEGAN IMAGESIZE=*number* < *options* > **;**
OPTIMIZATION < *options* > **;**
OUTPUT OUT=*libref.data-table* **;**
SAVESTATE RSTORE=*libref.data-table* **;**
TRAIN < *options* > **;**

The PROC STYLEGAN, OUTPUT, and SAVESTATE statements are required.

The rest of this section provides detailed syntax information about each of the preceding statements, beginning with the PROC STYLEGAN statement. The remaining statements are described in alphabetical order.

PROC STYLEGAN Statement

PROC STYLEGAN IMAGESIZE=*number* < *options* > **;**

The PROC STYLEGAN statement invokes the procedure. Table 33.1 summarizes the options in this statement.

Table 33.1 PROC STYLEGAN Statement Options

Option	Description
DATA=	Specifies the name of the input table
DETERMINISTIC	Specifies deterministic cuDNN implementations
IMAGESIZE=	Specifies the image size in the training data. This option is required.
NUMSAMPLES=	Specifies the number of observations to generate
SCORESEED=	Specifies the random number seed to use for GAN scoring
SEED=	Specifies the random number seed to use for GAN training
USEGPU	Specifies that a GPU device is to be used
USEGPU(DEVICE=)	Specifies the ID of the GPU device that is to be used

You must specify the following option:

IMAGESIZE=*number*
specifies the image size (width of square images in pixels) in the training data. The *number* must be one of the values 4, 8, 16, 32, 64, 128, 256, 512, 1024.

You can also specify the following *options*:

DATA=*libref.data-table*
names the input data table for PROC STYLEGAN to use. The default is the most recently created data table. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to the data. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 842.

data-table specifies the name of the input data table.

DETERMINISTIC
tells the NVIDIA CUDA Deep Neural Network library (cuDNN) that you want only the deterministic implementations. **NOTE:** Specifying this option can significantly slow down the training process.

NUMSAMPLES=*number*
specifies the number of sample images to generate.

By default, NUMSAMPLES=1.

SCORESEED=*number*
specifies the seed to use for the random number generator for scoring. If you do not specify a seed or if you specify a value less than or equal to 0, the seed is generated by reading the time of day from the computer's clock.

SEED=*number*
specifies the seed to use for the random number generator for training. If you do not specify a seed or if you specify a value less than or equal to 0, the seed is generated by reading the time of day from the computer's clock.

USEGPU(*suboptions***)**
specifies that a GPU device is to be used. You can specify the following *suboption*:

DEVICE=*id-number*
specifies the ID number of the GPU device that is to be used.

By default, DEVICE=0.

OPTIMIZATION Statement

OPTIMIZATION < *options* > **;**

The OPTIMIZATION statement specifies the optimizer parameters to use in training the style GAN model. You can specify the following *options*:

ADAM (BETA1=*number* **BETA2=***number***)**
specifies the optimization method to use in training the GAN model. Currently only the Adam method is supported. You must specify the keyword ADAM when specifying the following suboptions:

BETA1=*number*
specifies the exponential decay rate for the first-moment estimates.

By default, BETA1=0.0.

BETA2=*number*
specifies the exponential decay rate for the second-moment estimates.

By default, BETA2=0.99.

LEARNINGRATE=*number*
specifies the learning rate for the optimizer.

By default, LEARNINGRATE=0.002.

MINIBATCHSIZE=*number*
specifies the number of images in one minibatch. The *number* must be a positive integer.

By default, MINIBATCHSIZE=32.

NUMITERS=*number*
specifies the number of minibatches to be used for training the GAN model. The *number* must be a positive integer.

By default, NUMITERS=10.

OUTPUT Statement

OUTPUT OUT=*libref.data-table* **;**

The OUTPUT statement creates an output data table that contains the results of PROC STYLEGAN.

OUT=*libref.data-table*
specifies the output table in which to store the generated images from the trained model. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 842.

SAVESTATE Statement

SAVESTATE RSTORE=*libref.data-table* **;**

The SAVESTATE statement creates an analytic store for the model and saves it as a binary object in a data table. You can use the analytic store in the ASTORE procedure to score new data. For more information, see Chapter 4, "ASTORE Procedure."

You must specify the following option:

RSTORE=*libref.data-table*
specifies a data table in which to save the analytic store for the model. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 842.

TRAIN Statement

TRAIN < *options* > **;**

The TRAIN statement specifies the training parameters to use for training the style GAN model.

Table 33.2 summarizes the options in this statement.

Table 33.2 TRAIN Statement Options

Option	Description
CHECKPOINTFREQ=	Specifies the frequency at which to save a checkpoint during training
CHECKPOINTIN=	Specifies the checkpoint table to be loaded to restore your model
CHECKPOINTOUT=	Specifies the table in which to save the current model as a checkpoint
NTRUNCATIONLATENTS=	Specifies the number of latents to use in calculating the truncation mean
PRINTFREQ=	Specifies the frequency at which to print losses to the log
REGDFREQ=	Specifies the frequency at which to regularize the discriminator
REGGFREQ=	Specifies the frequency at which to regularize the generator
REGPATHWEIGHT=	Specifies the weight for regularizing the path length loss
REGR1WEIGHT=	Specifies the truncation ratio for image generation
USESMALLNETWORK	Specifies that small networks are to be used

You can specify the following *options*:

CHECKPOINTFREQ=*integer*
specifies the frequency at which to save a checkpoint during the training.

By default, CHECKPOINTFREQ=5000.

CHECKPOINTIN=*libref.data-table*
specifies the checkpoint table to be loaded to restore your model. If you omit this option, the model is trained from scratch. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 842.

CHECKPOINTOUT=*libref.output-table*
specifies the table in which to save the current model as a checkpoint. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 842.

NTRUNCATIONLATENTS=*integer*
specifies the number of latents to use in calculating the truncation mean.

By default, NTRUNCATIONLATENTS=4096.

PRINTFREQ=*integer*
specifies the frequency at which to print losses to the log and the iteration history table.

By default, PRINTFREQ=1.

REGDFREQ=*integer*
specifies the frequency at which to regularize the discriminator.

By default, REGDFREQ=16.

REGGFREQ=*integer*
specifies the frequency at which to regularize the generator.

By default, REGGFREQ=4.

REGPATHWEIGHT=*number*
specifies the weight for regularizing the path length loss.

By default, REGPATHWEIGHT=2.0.

REGR1WEIGHT=*number*
specifies the weight for regularizing the R1 loss.

By default, REGR1WEIGHT=10.0.

TRUNCATION=*number*
specifies the truncation ratio for image generation.

By default, TRUNCATION=1.0.

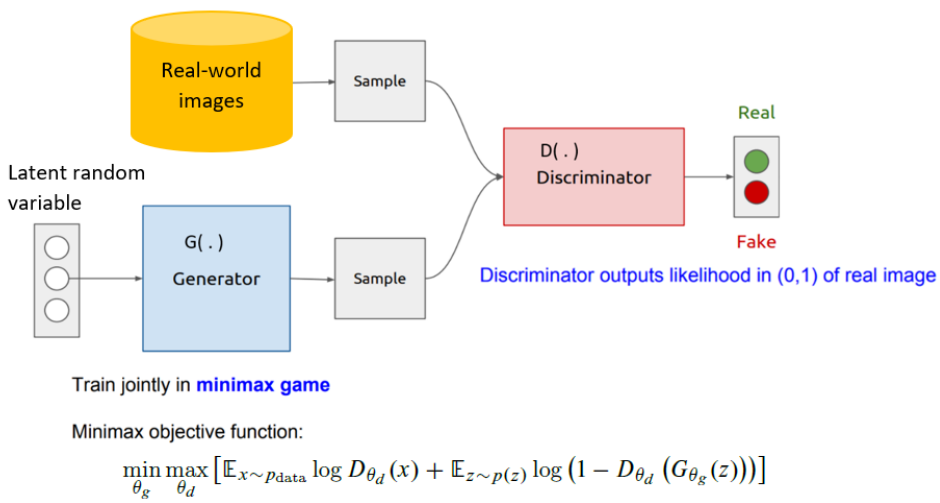
USESMALLNETWORK
specifies that small networks are to be used.

Details: STYLEGAN Procedure

The Style GAN 2 model is a state-of-the-art GAN model that solves the problems of the original Style GAN model. What is a generative adversarial network (GAN) model? A GAN model attempts to synthesize new data that have the same statistics as the training data, so that it becomes difficult to distinguish the real data from the fake data. In the architecture of GAN models, there are two neural networks: the generator, which generates data from noise (latent variables); and the discriminator, which distinguishes between data produced by the generator and real data.

A GAN model is characterized by learning the generator and the discriminator alternately, which it does by using the objective function shown in Figure 33.7.

Figure 33.7 GAN Model



Here, the generator produces data that have the same dimensions as the real data by using the latent z as input, and the discriminator outputs a value between 0 and 1 by using the sigmoid function of the final layer. In this way, the generator and the discriminator learn through repetition to work together, and eventually the GAN model can reach a Nash equilibrium when the following conditions are met:

- The discriminator can at best make a 50/50 guess about whether a sample is real.
- The generator can produce fake samples that are indistinguishable from the real data in the training data set.

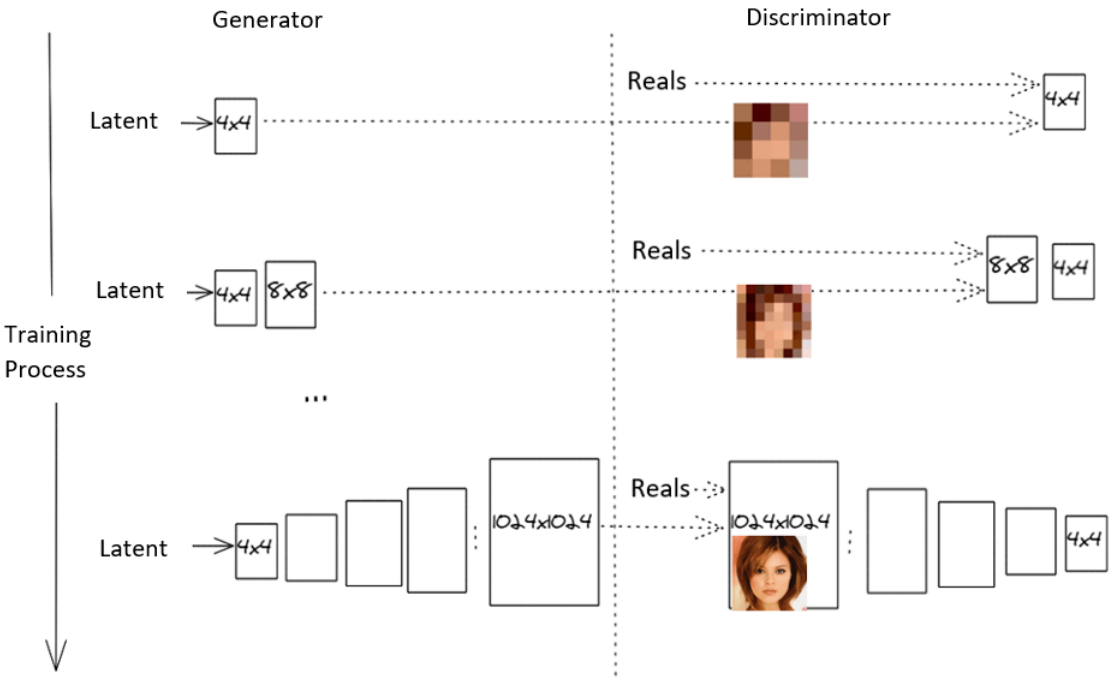
GANs have been extensively researched. One of the best-known models is Style GAN (Karras, Laine, and Aila 2019), which was proposed at the end of 2018. Figure 33.8 demonstrates that high-resolution images that you can generate using Style GAN can be mistaken for real ones. In this case, fake images of celebrities have been generated by training the Style GAN model on a data set of real celebrities' faces.

Figure 33.8 Images Generated by the Style GAN Model

Style GAN has two features: it generates high-resolution images by using progressive growing, and it generates these images by incorporating image styles into each layer, through the use of the adaptive instance normalization (AdaIN) module.

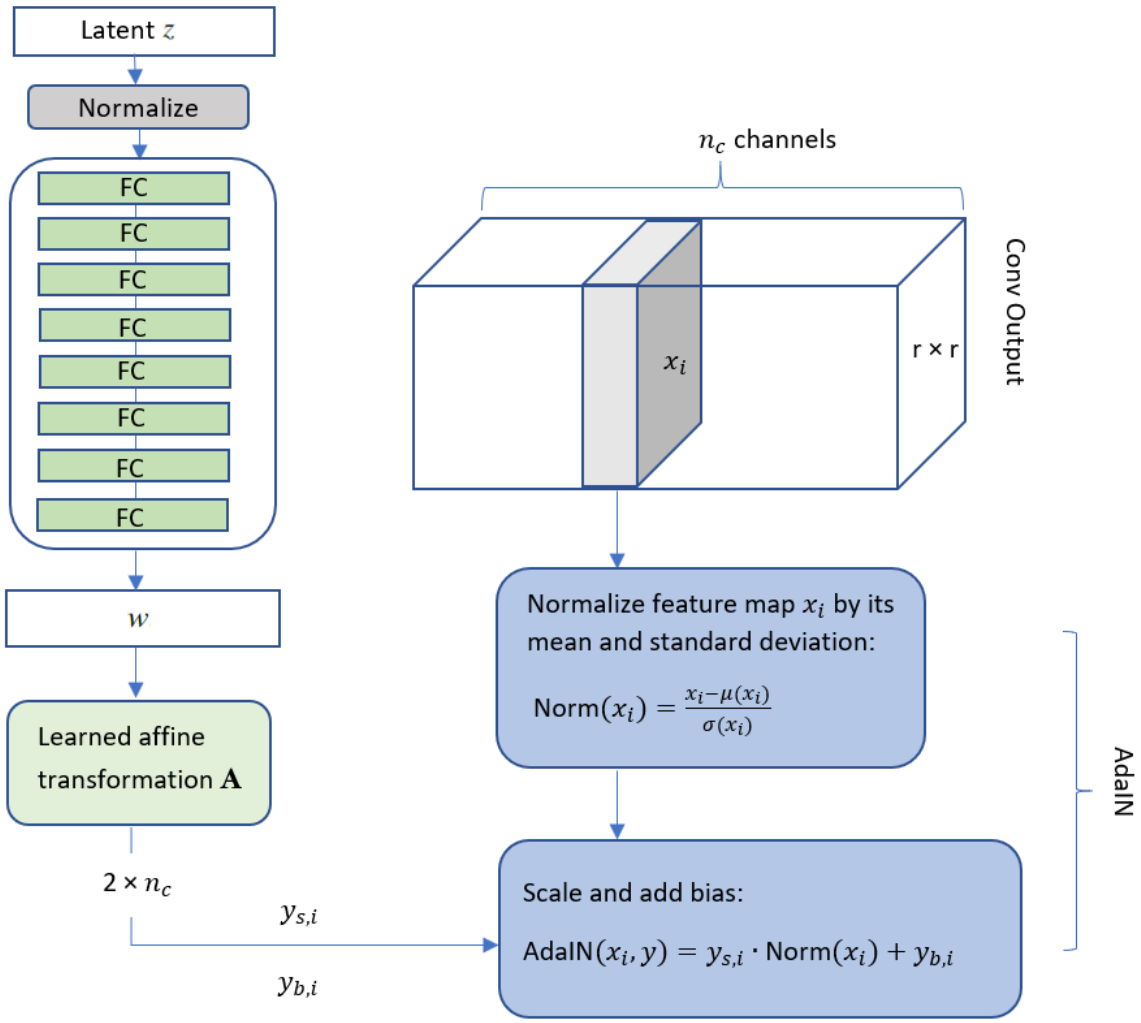
Progressive growing is a method of generating high-resolution images that was proposed in progressively growing GAN (PG-GAN) (Karras et al. 2018). It tries to generate high-resolution images by starting with low-resolution images and gradually expanding to high resolution by systematically adding new blocks of convolutional layers to the generators and discriminators. Both the generator and discriminator networks expand simultaneously. Figure 33.9 shows the training process that uses progressive growing.

Figure 33.9 Training Process Using Progressive Growing



Style GAN mainly improves on the existing architecture of the generator (G) network of PG-GAN to achieve better results and keeps the discriminator (D) network and loss functions untouched. The G network uses the AdaIN module in each resolution layer to transfer the encoded information w, which is created by a mapping network, to the generated image. The details of the AdaIN module are shown in Figure 33.10.

Figure 33.10 Details of AdaIN Module



Style GAN can create high-quality images, but after the results were closely examined by human testers (Karras et al. 2020), some issues were found. Here are the two main Style GAN issues:

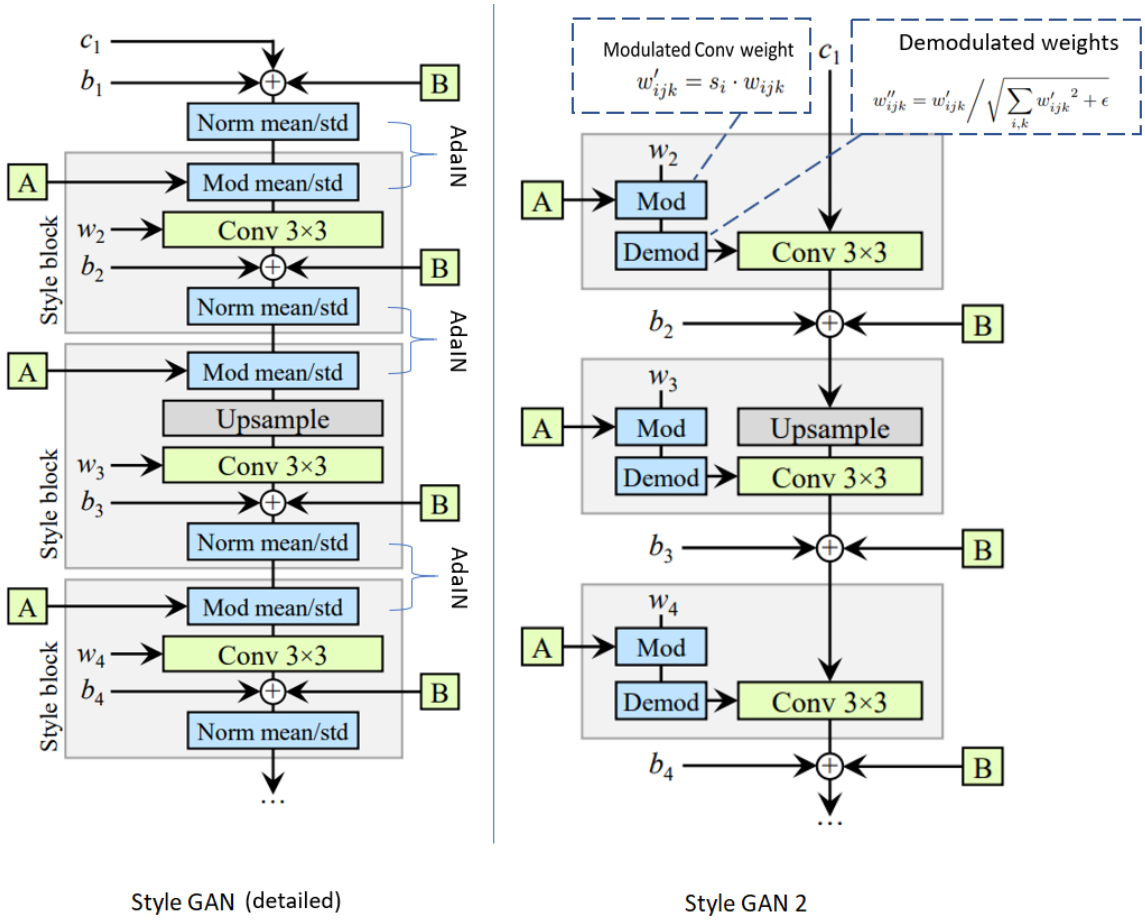
- *Water droplet–like artifacts.* This issue occurs because of the normalization layer (AdaIN). Images that include objects with sharp edges or sections with high contrast result in feature maps that have a small spike-type distribution. Even if the value of the spike is small in the original image feature map, the spike value is increased by normalization in the generator network and causes a large influence on the generated image (Karras et al. 2020). Karras et al. confirmed this by removing the normalization part and analyzing the results.

- *Phase artifacts.* The progressively grown generator appears to have a strong location preference for details. When the direction of the face changes, features like teeth and eyes should move with it in the same direction. However, sometimes the features remain stuck in place for several movements despite the change in the face's direction (Karras et al. 2020).

The Style GAN 2 model is an improved version of the Style GAN model. It makes several changes to both the model architecture and the training methods to solve the problems of the Style GAN model. Karras et al. redesigned the normalization that the generator network uses to remove the water droplet–like artifacts. Moreover, they introduced a regularization term in the generator loss to promote smooth mapping of the features and reduce the phase artifacts.

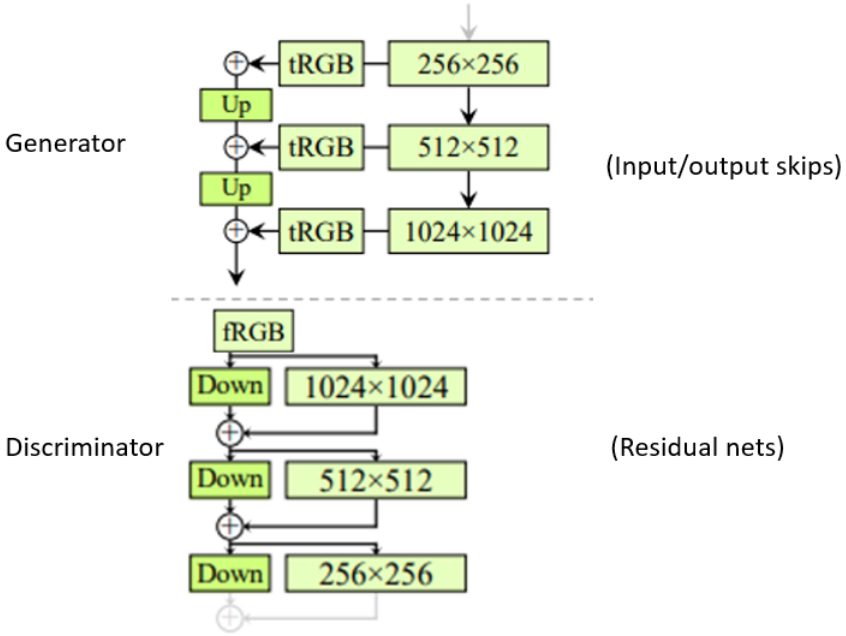
Figure 33.11 shows details of the redesigned generator normalization.

Figure 33.11 Redesigned Normalization in the Generator of Style GAN 2



Style GAN 2 implements a new network design that is based on skip connections and residual connections. This network is like the MSG-GAN (Karnewar and Wang 2020), which also uses multiple skip connections. The STYLEGAN procedure implements the architectures shown in Figure 33.12.

Figure 33.12 Style GAN 2 Architectures



By default, the architectures that PROC STYLEGAN implements correspond to the config-f architectures that are discussed in the Style GAN 2 paper (Karras et al. 2020). If you specify the USESMALLNETWORK option, then the architectures in PROC STYLEGAN correspond to the config-e architectures in the Style GAN 2 paper.

The loss function in Style GAN 2 consists of the nonsaturating loss with R1 regularization for the discriminator and the path length regularization for the generator. You can adjust the weights for the regularization terms by using the REGR1WEIGHT= option and the REGPATHWEIGHT= option in the TRAIN statement. You can use the REGDFREQ= option in the same statement to control how frequently R1 regularization is applied to the discriminator, and you can use the REGGFREQ= option to control how frequently path length regularization is applied to the generator.

For more information about the Style GAN 2 model, see Karras et al. (2020).

Optimization

PROC STYLEGAN supports the Adam optimizer. For more information about Adam optimization, see Kingma and Ba (2015).

Invalid Images

The STYLEGAN procedure checks for invalid input images. Valid images are defined as three-channel square color images whose width is specified by the IMAGESIZE= option in the PROC STYLEGAN statement. If the input image table contains any invalid images (rows), the procedure simply ignores them.

Displayed Output

The STYLEGAN procedure produces various tables. The following sections describe the output tables in alphabetical order.

Iteration History

The “Iteration History” table provides the iteration history of the optimization.

Model Information

The “Model Information” table provides information about the parameters that are used in the Style GAN 2 model.

Number of Observations

The “Number of Observations” table displays the information about the number of observations in the input data. The Number of Observations Read column displays the count of images that are read from the input data table, and the Number of Observations Used column displays the count of data that are actually used in the Style GAN 2 model training. The count in the Number of Observations Used column can be smaller if there are images (rows) in the input data table that have an invalid size or channel.

Timing

The “Timing” table displays the time spent in different tasks by PROC STYLEGAN.

Table Output

PROC STYLEGAN produces the following three output data tables.

Score Output

The score output table is specified in the OUT= option. This table has the generated images from the trained generator. The number of generated images is specified in the NUMSAMPLES= option.

Analytic Store Table

The analytic store table is specified in the RSTORE= option. This data table contains the saved generator network that is trained by PROC STYLEGAN. You can use this analytic store table to generate new images.

Checkpoint Out Table

If you specify the CHECKPOINTOUT= option, a table is generated to save the trained model as a checkpoint. You can load this table later by using the CHECKPOINTIN= option to continue training from where you previously stopped. This table has one row and one column. The column name is _checkpoint_. This column name cannot be changed if you want to reload the checkpoint by using the CHECKPOINTIN= option.

ODS Table Names

The STYLEGAN procedure assigns a name to each table that it creates. You can use this name to refer to the table when you use the Output Delivery System (ODS) to select tables and create output data tables. Table 33.3 lists the table names.

Table 33.3 ODS Tables Produced by PROC STYLEGAN

Table Name	Description	Statement	Option
IterHistory	Iteration history	PROC STYLEGAN	Default output
ModelInfo	Model information	PROC STYLEGAN	Default output
NObs	Number of observations	PROC STYLEGAN	Default output
Timing	Timing	PROC STYLEGAN	Default output

Examples: STYLEGAN Procedure

Example 33.1: Generating Human Face Images

NOTE: This example assumes that the SAS 9 engine libref mydirlib and the SASVIYA engine libref mylib have already been created. For more information, see the section "Using SAS Viya Workbench" on page 842 and Chapter 1, "Shared Concepts" (*SAS Viya Workbench: Computer Vision Procedures*).

This example shows how to use the STYLEGAN procedure to train a style GAN model and generate new images by using a data set that contains 70,000 color images of human faces at a resolution of 128 × 128 pixels. Note that the shape of the input image that you use in PROC STYLEGAN must be square.

First, you use the LOADIMAGES procedure (see Chapter 10, "LOADIMAGES Procedure" (*SAS Viya Workbench: Computer Vision Procedures*)) as follows to load the images from the specified path into the table mylib.images in your SAS library. This assumes that your SAS libref is named mylib, as in the section "Using SAS Viya Workbench" on page 842, but you can substitute any appropriately defined SAS libref. The specified path must be relative to the path that is associated with the mydirlib libref:

```
proc loadimages libref=mydirlib path='path-to-images-folder/';
    output out=mylib.images;
run;
```

Then, you use the following PROC STYLEGAN statements to train a style GAN model and save the trained generator network in an analytic store:

```
proc stylegan        data=mylib.images imagesize=128
                     numsamples=4 useGPU seed=123;
     optimization    ADAM(beta1=0.9 beta2=0.99) numiters=5
                     miniBatchSize=4 learningrate=0.001;
     train           useSmallNetwork truncation=0.95;
     savestate       rstore=mylib.astore;
     output          out=mylib.outtable;
run;
```

The DATA= option specifies the input images data set. The IMAGESIZE= option specifies the input image size, which is the image width of the square images in pixels. You specify the number of images that you want to generate in the NUMSAMPLES= option. The USEGPU option specifies that a GPU device is used to train the Style GAN model. The SEED= option specifies the random seed to use in the training process. You specify the optimization parameters in the OPTIMIZATION statement. The keyword ADAM specifies that the Adam optimizer is to be used in the training process. The BETA1= and BETA2= suboptions specify the exponential decay rate for the first-moment and second-moment estimates, respectively. You specify the number of training iterations to use for the GAN model in the NUMITERS= option. One iteration processes one minibatch of your training data. Note that for the human faces data, you can get a converged model by running 300,000 iterations with a minibatch size of 32. In this example, small option values are used for brevity. The TRAIN statement specifies the training parameters for the GAN model. The USESMALLNETWORK option indicates that small neural network models are to be used for the generator and discriminator. You specify the truncation ratio for image generation in the TRUNCATION= option. The SAVESTATE statement specifies the table to use for saving the trained model, and the OUTPUT statement specifies the table to use for outputting result images.

PROC STYLEGAN generates four ODS tables, which are shown in Output 33.1.1 through Output 33.1.4.

Output 33.1.1 shows the iteration history, including the generator and discriminator loss values for each epoch of training.

Output 33.1.1 Iteration History

The STYLEGAN Procedure

Iteration History				
Iteration Number	Generator Loss	Discriminator Loss	Path Length Regularization Loss	R1 Regularization Loss
1	0.806909	1.261505	1.744924	0.000185
2	0.999718	2.500470	1.744924	0.000185
3	3.764462	1.135563	1.744924	0.000185
4	4.969986	0.924546	1.744924	0.000185
5	4.048201	0.948312	1.375675	0.000185

Output 33.1.2 shows the number of observations that are read and number of observations that are actually used.

Output 33.1.2 Number of Observations

Number of Observations	
Number of Observations Read	70000
Number of Observations Used	70000

Output 33.1.3 shows the values of the parameters that the Style GAN model uses.

Output 33.1.3 Model Information

Model Information	
Exponential Decay Rate for the First-Moment Estimates	0.9
Exponential Decay Rate for the Second-Moment Estimates	0.99
Width of Square Images in Pixels	128
Use Large Network	False
Learning Rate for the Optimizer	0.001
Number of Images in One Minibatch	4
Number of Iterations	5
Seed for Random Initialization	123
Discriminator Regularization Frequency	16
Generator Regularization Frequency	4
Weight for Path Length Regularization	2
Weight for R1 Regularization	10

Output 33.1.4 illustrates the timing of the various steps that PROC STYLEGAN performs.

Output 33.1.4 Timing

Timing	
Task	**Time in Seconds**
Reserving GPU Device	0.000983
Loading Training Data	13.314346
Setting Up Model	0.790476
Training Model	8.708030
Generating Images	0.058983
Saving Generator to Astore	1.653507

In this example, PROC STYLEGAN generates the score output table mylib.outtable in the active library. This table contains four generated images from the trained model.

The following statements run the SAVEIMAGES procedure (see Chapter 16, "SAVEIMAGES Procedure" (*SAS Viya Workbench: Computer Vision Procedures*)) to write the generated images from the table mylib.outtable in bitmap format onto a disk:

```
proc saveimages libref=mydirlib path="path-to-write-images"
    data=mylib.outtable encoding=BMP;
run;
```

Example 33.2: Generating Human Face Images — Continue Training

NOTE: This example assumes that the SAS 9 engine libref mydirlib and the SASVIYA engine libref mylib have already been created. For more information, see the section "Using SAS Viya Workbench" on page 842 and Chapter 1, "Shared Concepts" (*SAS Viya Workbench: Computer Vision Procedures*).

This example is a follow-up to Example 33.1 and shows how to continue training a Style GAN model by using the CHECKPOINTIN= and CHECKPOINTOUT= options in the TRAIN statement. It uses the same data set as Example 33.1.

First, as in Example 33.1, you use PROC LOADIMAGES (see Chapter 10, "LOADIMAGES Procedure" (*SAS Viya Workbench: Computer Vision Procedures*)) as follows to load the images into the table mylib.images in your SAS library. This assumes that your SAS libref is named mylib, as in the section "Using SAS Viya Workbench" on page 842, but you can substitute any appropriately defined SAS libref. The specified path must be relative to the path that is associated with the mydirlib libref:

```
proc loadimages libref=mydirlib path='path-to-images-folder/';
    output out=mylib.images;
run;
```

Then, you use PROC STYLEGAN as follows to train a Style GAN model, save training checkpoints in a table, and save the trained generator network in an analytic store:

```
proc stylegan        data=mylib.images imagesize=128
                     numsamples=4 useGPU seed=123 scoreseed=321;
     optimization    numiters=2
                     miniBatchSize=8 learningrate=0.002;
     train           useSmallNetwork truncation=0.95
                     regDFreq=2 regGFreq=1 regPathWeight=3.3 regR1Weight=11.57
                     printFreq=1 checkpointFreq=1 checkpointOut=mylib.checkpointout;
     savestate       rstore=mylib.astore;
     output          out=mylib.outtable;
run;
```

In addition to the statements and options that are used in Example 33.1, the current example uses the following options. The SCORESEED= option specifies the random seed to use for scoring. The REGDFREQ= option specifies the regularization frequency of the discriminator model to be every two steps; The REGGFREQ= option regularizes the generator model at every step. The REGPATHWEIGHT= option specifies the weight for regularizing the path length loss. The REGR1WEIGHT= option specifies the truncation ratio for image generation. The PRINTFREQ= option prints loss values at every step. The CHECKPOINTFREQ= option saves checkpoints at every step of the training. The CHECKPOINTOUT= saves the training checkpoints to the mylib.checkpointout data table.

The following statements use PROC STYLEGAN to continue training a Style GAN model by using the last checkpoint that is saved in the mylib.checkpointout data table:

```
proc stylegan        data=mylib.images imagesize=128
                     numsamples=6 useGPU seed=42 scoreseed=50;
     optimization    numiters=4
                     miniBatchSize=4 learningrate=0.001;
     train           useSmallNetwork truncation=0.95
                     regDFreq=2 regGFreq=2 regPathWeight=2.5 regR1Weight=10.2
                     checkpointIn=mylib.checkpointout printFreq=2
```

```
                    checkpointFreq=2 checkpointOut=mylib.checkpointout2;
     savestate      rstore=mylib.astore2;
     output         out=mylib.outtable2;
run;
```

The CHECKPOINTIN= option tells PROC STYLEGAN to continue training from the last saved checkpoint available in the mylib.checkpointout data table. Note that the values of options such as SEED=, NUMITERS=, LEARNINGRATE=, and many others can differ from their values in the previous PROC STYLEGAN run that generated the mylib.checkpointout data table. The CHECKPOINTOUT= option saves the checkpoints from the recent training in the mylib.checkpointout2 data table. You can continue training the Style GAN model by using the checkpoints that are saved in the mylib.checkpointout2 data table in a similar fashion. PROC STYLEGAN generates results and ODS tables that are omitted here for brevity. See Example 33.1 for information about results and ODS tables.

References

Karnewar, A., and Wang, O. (2020). "MSG-GAN: Multi-scale Gradients for Generative Adversarial Networks." In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. New York: IEEE. https://openaccess.thecvf.com/content_CVPR_2020/papers/Karnewar_MSG-GAN_Multi-Scale_Gradients_for_Generative_Adversarial_Networks_CVPR_2020_paper.pdf.

Karras, T., Aila, T., Laine, S., and Lehtinen, J. (2018). "Progressive Growing of GANs for Improved Quality, Stability, and Variation." In *Proceedings of the Sixth International Conference on Learning Representations*. La Jolla, CA: ICLR. https://openreview.net/pdf?id=Hk99zCeAb.

Karras, T., Laine, S., and Aila, T. (2019). "A Style-Based Generator Architecture for Generative Adversarial Networks." In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. New York: IEEE. https://openaccess.thecvf.com/content_CVPR_2019/papers/Karras_A_Style-Based_Generator_Architecture_for_Generative_Adversarial_Networks_CVPR_2019_paper.pdf.

Karras, T., Laine, S., Aittala, M., Hellsten, J., Lehtinen, J., and Aila, T. (2020). "Analyzing and Improving the Image Quality of StyleGAN." In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR)*. New York: IEEE. https://openaccess.thecvf.com/content_CVPR_2020/papers/Karras_Analyzing_and_Improving_the_Image_Quality_of_StyleGAN_CVPR_2020_paper.pdf.

Kingma, D. P., and Ba, J. (2015). "Adam: A Method for Stochastic Optimization." In *Proceedings of the 3rd International Conference for Learning Representations*. Baixas, France: ISCA. https://arxiv.org/abs/1412.6980.

Chapter 34
SVDD Procedure

Contents

Overview: SVDD Procedure . . . **864**
- PROC SVDD Features . . . 864
- Using SAS Viya Workbench . . . 865

Getting Started: SVDD Procedure . . . **865**

Syntax: SVDD Procedure . . . **870**
- PROC SVDD Statement . . . 870
- CODE Statement . . . 871
- DISPLAY Statement . . . 872
- DISPLAYOUT Statement . . . 873
- ID Statement . . . 874
- INPUT Statement . . . 874
- KERNEL Statement . . . 874
- SAVESTATE Statement . . . 875
- SOLVER Statement . . . 876
- WEIGHT Statement . . . 878

Details: SVDD Procedure . . . **878**
- Mathematical Formulation . . . 878
- Radial Basis Function (RBF) Bandwidth Parameter . . . 881
- Fraction Outlier Parameter . . . 884
- SVDD Formulation with a Weight Variable . . . 885
- Comparison with One-Class Support Vector Machines . . . 886
- SVDD Solvers . . . 887
- Scoring Multiple SVDD Models with the ASTORE Procedure . . . 888
- Displayed Output . . . 889
- ODS Table Names . . . 891
- Output Data Tables . . . 891

Examples: SVDD Procedure . . . **892**
- Example 34.1: Analysis of Aircraft Engine Degradation . . . 892
- Example 34.2: Automatic RBF Kernel Bandwidth Parameter Value Selection . . . 896
- Example 34.3: Stochastic Subset Solver . . . 907
- Example 34.4: Analysis of Tennessee Eastman Process Data Using Multiple Bandwidth Values and Consensus-Based Classification . . . 914
- Example 34.5: Bandwidth Calculation Using the Peak Criterion Method . . . 920

References . . . **927**

Overview: SVDD Procedure

The SVDD procedure implements the support vector data description (SVDD) algorithm (Tax and Duin 2004). SVDD is a one-class classification technique that is useful in applications where data that belong to one class are abundant but data about any other class are scarce or missing. Fraud detection, equipment health monitoring, and process control are some examples of application areas where the majority of the data belong to one class. You can use SVDD to model such one-class data and subsequently use the model to perform outlier detection.

In its simplest form, an SVDD model is obtained by building a minimum-radius hypersphere around the one-class training data. The hypersphere provides a compact spherical description of the training data. This training data description can be used to determine whether a new observation is similar to the training data observations. The distance from any new observation to the hypersphere center is computed and compared with the hypersphere radius. If the distance is more than the radius, the observation is designated as an outlier. Using kernel functions in SVDD formulation provides a more flexible description of training data. Such description is nonspherical and conforms to the geometry of the data. PROC SVDD implements only the flexible data description.

SVDD is useful for obtaining a geometric description of data and in most applications also for detecting outliers. SVDD is used in domains where most of the data are in one class. Applications of SVDD include the following:

- multivariate process control (Sun and Tsung 2003; Sukchotrat, Kim, and Tsung 2009; Kakde, Peredriy, and Chaudhuri 2017)
- equipment prognostics and health management (Benkedjouh et al. 2012; Tax, Ypma, and Duin 1999a, b)
- cybersecurity and intrusion detection (Kang, Jeong, and Kong 2012)
- fraud identification (Jeong, An, and Nam 2016; Juszczak et al. 2008; Jeong, An, and Nam 2016)
- hyperspectral image analysis (Banerjee, Burlina, and Diehl 2006; Banerjee, Burlina, and Meth 2007; Sakla et al. 2011; Liao et al. 2018)
- analysis of health data from wearable devices (Shin, Lee, and Park 2011; Yang et al. 2010)

PROC SVDD Features

The following list provides important features of the SVDD procedure:

- multithreaded SVDD training
- support for both interval and nominal variables
- a WEIGHT statement that enables you to assign weights to observations
- automatic value selection of the radial basis function (RBF) kernel bandwidth parameter

- choice of solvers:
  - stochastic subset solver that enables you to quickly obtain an approximate solution
  - active-set solver that enables you to obtain a more accurate solution

Using SAS Viya Workbench

SAS Viya Workbench has greatly simplified the SAS programming experience by offering a syntax style nearly identical to that of SAS 9 and earlier releases. However, there are minor exceptions. In the past, SAS Viya procedures required the use of a CAS engine libref. This engine served multiple purposes, but it was primarily used to upload data to the CAS server and manage the data. Its additional purposes included support of new data types and improved performance for large data sets. No CAS server is needed for SAS Viya Workbench, because all computations are done in place for you in a single location.

In the following program, the LIBNAME statement creates a mylib libref that the examples in this chapter use to refer to your current or desired work directory:

```
libname mylib ".";
```

For more information, see *SAS Viya Workbench: Getting Started*.

Getting Started: SVDD Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

This example demonstrates how you can use the SVDD procedure to obtain a description of two-variable data and perform outlier detection. The training data in this example are refrigerant analysis data (Heckert and Filliben 2000). The data table contains 360 observations and 31 variables. The variables represent various process measurements that are taken during the refrigerant analysis. This example uses variables y5 (inlet water temperature 1) and y4 (pressure drop 6) to perform an SVDD training. Variables y4 and y5 are selected to highlight the ability of the SVDD algorithm to obtain an accurate description of the training data.

The following DATA step creates the refrigerant (which is stored in the Work library by default):

```
data refrigerant;
   infile datalines delimiter=',';
   input y4 y5 ;
   label y4="Pressure Drop 6" y5="Inlet Water Temperature 1";
   datalines;
2.0562,25.0904
2.04147,25.1676
2.04922,25.6362

   ... more lines ...
```

The following code standardizes variables y4 and y5:

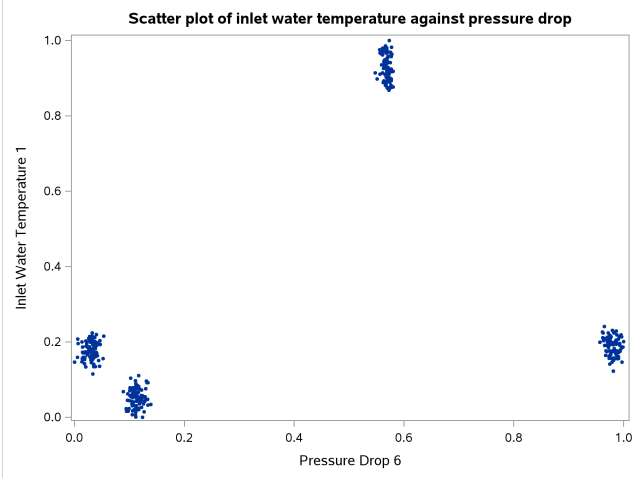
```
proc sql noprint;
   select min(y4), min(y5), max(y4), max(y5)
   into :miny4, :miny5, :maxy4, :maxy5
   from  Refrigerant;
quit;


data refrigerant;
   set refrigerant;
   y4=(y4-&miny4)/%sysevalf(&maxy4-&miny4);
   y5=(y5-&miny5)/%sysevalf(&maxy5-&miny5);
run;
```

The following statements plot inlet water temperature 1 against pressure drop 6:

```
proc sgplot data=Refrigerant;
   title "Refrigerant Analysis";
   title1 "Scatter plot of inlet water temperature against pressure drop";
   scatter x=y4 y=y5/markerattrs=(size=3 symbol=circlefilled);
run;
```

Output 34.1 shows the results. The scatter plot indicates four distinct clusters.

Figure 34.1 Scatter Plot of Inlet Water Temperature against Pressure Drop



You can load the work.refrigerant data set into your SAS session by specifying your libref in the first statement in the following DATA step:

```
data mylib.refrigerant;
   set refrigerant;
run;
```

The following statements execute the SVDD algorithm on the mylib.refrigerant data table and produce Figure 34.2 through Figure 34.5.

```
proc svdd data=mylib.Refrigerant;
   id y4 y5;
   input y4 y5/level=interval;
   kernel rbf / bw=0.021;
   savestate rstore=mylib.state;
run;
```

The INPUT statement defines the input variables y4 and y5 as interval variables. The KERNEL statement specifies the kernel function as a radial basis function (RBF) and specifies a value of 0.021 for the bandwidth parameter. The SAVESTATE statement creates an analytic store for the model and saves it as a binary object in the mylib.state data table. You can use the analytic store later in the ASTORE procedure to score new data. The "Model Information" table in Figure 34.2 summarizes the key options and the input data variables.

Figure 34.2 Refrigerant Analysis Model Information

Model Information	
Optimization Method	Active Set
Kernel Type	RBF
RBF Kernel Bandwidth	0.021
Bandwidth Selection Method	User specified
Bandwidth Relative Scale	0.1663083933
Expected Outlier Fraction	1E-6
Optimization Tolerance	0.0001
Number of Interval Variables	2
Number of Nominal Variables	0

The "Training Results" table in Figure 34.3 shows the number of support vectors and the R^2 threshold value.

Figure 34.3 Refrigerant Analysis Training Results

Training Results	
Number of Support Vectors	39
Number of Support Vectors on Boundary	39
Number of Dropped Observations	0
Threshold R Square Value	0.94341
Constant (C_r) Value	0.05659
Run Time (seconds)	0.00684

Figure 34.4 displays the number of observations in the training data set.

Figure 34.4 Refrigerant Analysis Training Observations

Number of Observations Read	360
Number of Observations Used	360

The "Optimization Summary" table in Figure 34.5 shows whether the solution is optimal, the number of iterations that were required, and the objective function value.

Figure 34.5 Refrigerant Analysis Optimization Summary

Optimization Summary	
Number of Iterations	1
Objective Value	0.0565942537
Infeasibility	1.023959E-12
Optimization Status	Optimal
Degenerate	No

To evaluate the quality of the training results, scoring is performed on a 200 × 200 data grid. The following DATA step creates a data set named scoreds to contain the scoring observations:

```
data scoreds;
   do i=0 to 1 by 0.005;
      do  j=0 to 1 by 0.005;
         output;
      end;
   end;
rename i=y5 j=y4;
run;
```

You can load the scoreds data set into your SAS session by specifying your SAS libref in the first statement in the following DATA step:

```
data mylib.scoreds;
   set scoreds;
run;
```

The following code scores the mylib.scoreds data by using the score analytic store that was created during training:

```
proc astore;
   score data=mylib.scoreds
   out=mylib.score_results
   rstore=mylib.state;
quit;
```

The following code copies the score_results into the Work library:

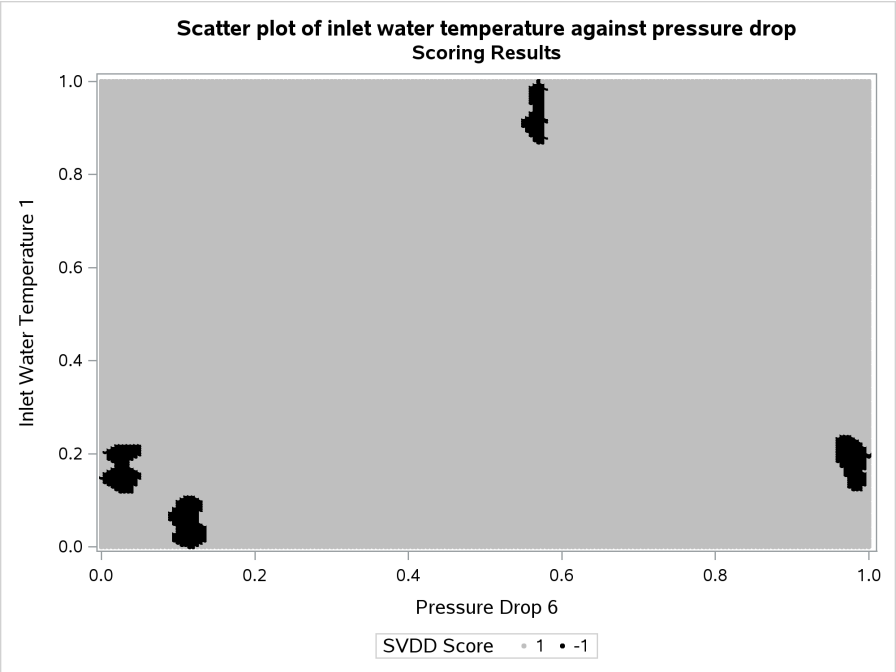
```
data work.score_results;
   set mylib.score_results;
run;
```

The following code plots the scoring results:

```
proc sgplot data=score_results(where=(_svddscore_ in (-1,1)));
styleattrs datacontrastcolors=(ligr black) datasymbols=(circlefilled);
title "Refrigerant Analysis";
title1 "Scatter plot of inlet water temperature against pressure drop";
title2 "Scoring Results";
scatter x=y4 y=y5/group=_svddscore_ markerattrs=(size=3);
run;
```

Figure 34.6 shows the scatter plot of the scoring data set. The gray area indicates the observations that are identified as outliers, and the black area indicates observations that are identified as inliers. Comparison between Figure 34.1 and the training data in Figure 34.6 indicates that SVDD has correctly identified observations outside the training data as outliers. In this example, the SVDD procedure described four disjoint clusters. Any observation not part of these four clusters was identified as an outlier during scoring. This example illustrates that SVDD procedure can be used to describe data with multiple disjoint clusters, without the need to specify the correct number of clusters in advance. The scoring results in Figure 34.6 indicate that the description obtained using SVDD takes essential geometric features of the training data into account. The scoring correctly identified any observations with pressure drop 6 and inlet water temperature 1 values different from the training data values correctly identified as outliers.

Figure 34.6 Scoring Results Scatter Plot



Syntax: SVDD Procedure

The following statements are available in the SVDD procedure:

```
PROC SVDD <options> ;
    CODE FILE=filename ;
    DISPLAY <table-list> </options> ;
    DISPLAYOUT table-spec-list </options> ;
    ID variables ;
    INPUT variables / <LEVEL=INTERVAL | NOMINAL> ;
    KERNEL  kernel-type / kernel-parameter ;
    SAVESTATE RSTORE=libref.data-table ;
    SOLVER solver-type / <options> ;
    WEIGHT variable ;
```

The PROC SVDD statement, the KERNEL statement, and at least one INPUT statement are required.

The following sections describe the PROC SVDD statement and then describe the other statements in alphabetical order.

PROC SVDD Statement

```
PROC SVDD <options> ;
```

The PROC SVDD statement invokes the procedure. You can specify the following *options*:

DATA=*libref.data-table*
names the input data table for PROC SVDD to use. The default is the most recently created data table. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to the data. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 865.

data-table specifies the name of the input data table.

NBINS=*number*
specifies the number of bins to display, where *number* must be an integer greater than 2. By default, NBINS=20.

NOPRINT
suppresses the generation of ODS outputs. If you specify this option, no ODS tables are generated.

NTHREADS=*number-of-threads*
specifies the number of threads to be used in the computation. The default value is the number of CPUs available in the machine.

OUTLIER_FRACTION=*number*

FRACTION=*number*

FRAC=*number*

specifies the expected fraction of the training data that consists of outliers, where *number* is a real number between 0 and 1.

By default, OUTLIER_FRACTION=1E–6.

SCORE

scores the observations in the data table that is specified in the DATA= option and generates a frequency table of _SVDDDISTANCE_ values.

STD

STANDARDIZE

standardizes the variables that are specified in the INPUT statement with LEVEL=INTERVAL by subtracting their mean and dividing by their standard deviation.

UCC

USECONSTANTCOL

uses variables that have constant values for SVDD scoring. (By default, the variables that have constant values are ignored.) This option is useful in scenarios where certain variables have constant value in the training data, but variation in the values of those variables is expected in the scoring data. This option ensures that variables with constant values are used for scoring.

OUTSV=*libref.data-table*

SV=*libref.data-table*

specifies the output data to contain the training data observations that are identified as support vectors, the corresponding value of the Lagrange multiplier (column _SVDDALPHA_), and an indicator (column _SVDDOUTLIER_) that flags the position of the support vectors relative to the data boundary. Support vectors whose _SVDDOUTLIER_ value is 0 fall on the data boundary, and support vectors whose _SVDDOUTLIER_ value is 1 fall outside the data boundary. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 865.

CODE Statement

CODE FILE=*filename* **;**

The CODE statement generates SAS DATA step code that mimics the computations that are done by the OUTPUT statement.

You must specify the following option:

FILE=*filename*
specifies the name of the file to write the SAS score code to.

The CODE statement is optional.

NOTE: You can use the CODE statement when you specify multiple bandwidth values in the KERNEL statement. You cannot use the code file for scoring, if the scoring data include a nominal variable of the VARCHAR data type.

DISPLAY Statement

DISPLAY < *table-list* > < / *options* > **;**

The DISPLAY statement enables you to specify a list of display tables to display or exclude. This statement is similar to the ODS SELECT, ODS EXCLUDE, and ODS TRACE statements. However, the DISPLAY statement can improve performance when a large number of tables could be generated (such as in BY-group processing). The DISPLAY statement produces only the requested tables, but ODS statements generate all the tables and then keep the requested tables.

If you use both DISPLAY and ODS statements together, the DISPLAY statement takes precedence over the ODS statements. Note that the ODS EXCLUDE statement processes tables after they have been filtered by the DISPLAY statement. In some cases, it might appear that the ODS EXCLUDE statement is taking precedence because it can further filter the tables. For more information about ODS, see *SAS Output Delivery System: Procedures Guide*.

You can specify the *table-list* as a list of table names, paths, partial pathnames, and regular expressions.

The table names that you can specify are listed in the section "ODS Table Names" on page 891. A path is a table name that is prefixed with dot-separated grouping information. For example, a SelectionSummary table that a procedure produces during a selection routine might have the path *Bygroup1.Summary.SelectionSummary*. A partial pathname does not include all groups; for example, *SelectionSummary* and *Summary.SelectionSummary* are partial pathnames for *Bygroup1.Summary.SelectionSummary*.

When you specify a table name or partial pathname, all display tables whose paths end in the specified name are selected for display or exclusion. For example, both *SelectionSummary* and *Summary.SelectionSummary* select *Bygroup1.Summary.SelectionSummary*.

A regular expression is enclosed in forward slashes (/). For example, specifying "/tions/" selects all pathnames that contain the substring "tions"; in particular, the *Bygroup1.Summary.SelectionSummary* table is selected. Specifying "!/tions/" selects all pathnames that do not contain the substring "tions"; in particular, the *Bygroup1.Summary.SelectionSummary* table is not selected.

You can specify the following *options* after a slash (/):

CASESENSITIVE
performs a case-sensitive comparison of table names in the *table-list* to display table names when tables are subsetted for display. To preserve case, you must enclose table names in the *table-list* in quotation marks.

EXCLUDE
displays all display tables except those that you specify in the *table-list*.

EXCLUDEALL
suppresses display of all tables. This option takes precedence over the other options.

TRACE
displays the display table names, labels, and paths.

DISPLAYOUT Statement

DISPLAYOUT *table-spec-list* < / *options* > **;**

The DISPLAYOUT statement enables you to create output tables from your displayed output. This statement is similar to the ODS OUTPUT statement. For more information about ODS, see *SAS Output Delivery System: Procedures Guide*.

The *table-spec-list* specifies a list of output tables to create. Each entry in the list has either a *key*=*value* format or a *key* format:

key=*value* specifies *key* as the ODS table name, path, or partial pathname, and specifies *value* as the output table name.

key specifies *key* as the ODS table name and also as the output table name.

The ODS table names that you can specify are listed in the section "ODS Table Names" on page 891. You cannot specify the ODS table named OutputCasTables in the *table-spec-list*.

Table names and partial pathnames are discussed under the DISPLAY statement. The DISPLAYOUT statement does not support regular expressions.

You can specify the following *options* after a slash (/):

INCLUDEALL
creates output tables for all display tables. The name of the created output table is the same as the name of the corresponding display table. If you specify this option, the *table-spec-list* specification is ignored.

NOREPLACE
does not replace any existing output table of the same name.

ID Statement

ID *variables* **;**

The ID statement lists one or more variables to be copied from the input data table to the output data tables that are specified in the OUTSV= option in the PROC SVDD statement and the RSTORE= option in the SAVESTATE statement.

INPUT Statement

INPUT *variables* / < **LEVEL=INTERVAL | NOMINAL** > **;**

The INPUT statement specifies the names of *variables* to be used in training. Only interval, binary, and nominal variables are accepted. If you want to use different options for different variables, you can specify multiple INPUT statements.

You can specify the following option after a slash (/):

LEVEL=INTERVAL | NOMINAL
specifies whether the specified input *variables* are continuous or categorical. You can specify the following values:

INTERVAL specifies that the input *variables* are continuous.

NOMINAL specifies that the input *variables* are categorical.

By default, LEVEL=INTERVAL for numeric variables and LEVEL=NOMINAL for categorical variables. Binary variables are considered to be categorical variables.

KERNEL Statement

KERNEL *kernel-type* / *kernel-parameter* **;**

The KERNEL statement specifies the type of kernel and any associated parameters to be used during training. Only the radial basis kernel function is supported.

You must specify the following *kernel-type* and *kernel-parameters*:

RBF / **BW=***s*

RBF / **BW=***(number list)* | *(n* **TO** *m* **BY** *p)*

RBF / **BW=MEAN (DELTA=***number***)**

RBF / **BW=MEAN2**

RBF / **BW=TRACE (NREP=***number***)**
uses a radial basis function (also known as the Gaussian kernel function) during training. The kernel is defined as

$$K(x, y) = \exp \frac{-\|x - y\|^2}{2s^2}$$

where x and y are vectors and s is the bandwidth parameter. The bandwidth parameter is determined in one of the following ways:

- You can specify the bandwidth parameter in the form BW=*s*, where *s* must be a positive nonzero real number. The processing time usually increases as the value of *s* decreases. At very small values of *s*, the processing time can be very long.
- You can specify multiple bandwidth parameter values in the form BW=(*number list*) or BW=(*n* TO *m* BY *p*). The maximum number of bandwidth parameters that you can specify or iterate over is 64. You can use multiple bandwidth parameters to train multiple SVDD models in a single run.
- You can instruct PROC SVDD to select an appropriate value by specifying BW=MEAN(DELTA=*number*) to request that the bandwidth parameter be computed using the mean criterion. The DELTA=*number* suboption specifies the tolerance value for the mean criterion. By default, DELTA=1E–6. For more information about the mean criterion, see the section "Mean Criterion" on page 883.
- You can instruct PROC SVDD to select an appropriate value by specifying BW=MEAN2 to request that the bandwidth parameter be computed using the modified mean criterion. For more information about the modified mean criterion, see the section "Modified Mean Criterion" on page 883.
- You can instruct PROC SVDD to select an appropriate value by specifying BW=TRACE(NREP=*number*) to request that the bandwidth parameter be computed using the trace criterion. The NREP=*number* suboption specifies the number of representative points. By default, NREP=5. For more information about the trace criterion, see the section "Trace Criterion" on page 884.

SAVESTATE Statement

SAVESTATE RSTORE=*libref.data-table* **;**

The SAVESTATE statement creates an analytic store for the model and saves it as a binary object in a data table. You can use the analytic store in the ASTORE procedure to score new data. For more information, see Chapter 4, "ASTORE Procedure."

You must specify the following option:

RSTORE=*libref.data-table*

specifies the name of the file in which to save the analytic store. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 865.

SOLVER Statement

SOLVER *solver-type* < /*options* > **;**

The SOLVER statement specifies the type of optimization solver for SVDD training. You must specify one of the following *solver-types*; the default type is ACTSET.

ACTSET
uses the active-set solver for SVDD training. For more information about this solver, see the section "Active-Set Solver" on page 887.

STOCHS
uses a stochastic subset solver for SVDD training. For more information about this solver, see the section "Stochastic Subset Solver" on page 887.

FASTINC
uses a fast incremental solver for SVDD training. For more information about this solver, see the section "Fast Incremental Solver" on page 888.

You can specify the following *option* after a slash (/) for either *solver-type*:

MAXTIME=*t*
specifies an upper limit of *t* seconds for solving the SVDD optimization problem. If you omit this option, the solver does not stop based on the amount of time elapsed. The value of *t* can be any positive number; the default value is 1,800 seconds, or 30 minutes. This option puts a time budget only on the solvers, while the actual running time may be longer because of various kinds of overheads.

You can specify the following options after a slash (/) only when the *solver-type* is ACTSET or STOCHS:

MAXITER=*number*
specifies the maximum number of iterations of the stochastic solver, where *number* must be an integer greater than 0. To obtain a solution more quickly, specify a lower value for *number*. By default, MAXITER=25 for the active-set solver and MAXITER=200 for the stochastic subset solver.

STOL=*number*
specifies the tolerance value for the solver, where *number* must be a real number greater than 0. To obtain a solution more quickly, specify a higher value for *number*. By default, STOL=1E–4.

You can specify the following options after a slash (/) only when the *solver-type* is FASTINC or STOCHS:

MAXSV=*number*
specifies the maximum number of support vectors, where *number* must be an integer greater than 0. To obtain a solution more quickly, specify a lower value for *number*. By default, MAXSV=3500.

You can specify the following options after a slash (/) only when the *solver-type* is FASTINC:

OTHRESH=*number*

specifies the threshold for identifying outliers. The fast incremental solver considers an observation z to be an outlier if

$$\max_i(K(x_i, z)) < \textit{number}$$

where $x_i, i = 1 \ldots n$ is the set of support vectors and $K(x_i, z) = \exp \dfrac{-\|x_i - z\|^2}{2s^2}$

By default, OTHRESH=1E–20. The OTHRESH option is used only for the FASTINC solver and it is ignored for the ACTSET and the STOCHS solvers.

You can specify the following options after a slash (/) only when the *solver-type* is STOCHS:

CTOL=*number*

specifies the tolerance value that is used for detecting convergence of the center, where *number* must be a real number greater than 0. To obtain a solution more quickly, specify a higher value for *number*. By default, CTOL=1E–2.

NMATCH=*number*

specifies a convergence criterion for the stochastic solver, where *number* must be an integer greater than 1. If the radius and center values converge for the specified *number* of consecutive iterations, then convergence is declared. By default, NMATCH=5. To obtain a solution more quickly, specify a lower value for *number*.

NSAMP=*number*

specifies the number of observations that are sampled in each iteration of the stochastic solver, where *number* must be an integer greater than 0. To obtain a solution more quickly, specify a lower value for *number*. By default, NSAMP=10.

RTOL=*number*

specifies the tolerance value that is used to detect convergence of threshold value, where *number* must be a real number greater than 0. To obtain a solution more quickly, specify a higher value for *number*. By default, RTOL=1E–2.

SEED=*number*

specifies a seed value that is used for selecting a random sample, where *number* must be an integer. The default value of SEED is based on the system clock. If you want reproducible results, specify an integer for *number*.

WEIGHT Statement

WEIGHT *variable* **;**

The WEIGHT statement names a *variable* in the training data that specifies the weight to be assigned to each observation. The values of *variable* must be nonnegative. Observations that have higher weight have a better chance of being designated as support vectors.

Details: SVDD Procedure

PROC SVDD uses the radial basis kernel function to describe one-class training data and perform outlier detection. For more information about the theory and use of SVDD, see Tax and Duin (2004).

Mathematical Formulation

This section describes the mathematical formulation of SVDD.

Normal Data Description

The most elemental form of SVDD is a normal data description. The SVDD model for normal data description builds a minimum-radius hypersphere around the data. PROC SVDD does not support this description because it is very simplistic and often leads to higher rates of misclassification. PROC SVDD supports the flexible data description (as described in the section "Flexible Data Description" on page 879). SVDD formulation can be expressed in either of the following forms:

Primal Form

The objective function in primal form is

$$\min R^2 + C \sum_{i=1}^{n} \xi_i$$

subject to

$$\|x_i - a\|^2 \leq R^2 + \xi_i \text{ for all } i = 1, \ldots, n$$
$$\xi_i \geq 0 \text{ for all } i = 1, \ldots, n$$

where $x_i \in \mathbb{R}^m$ for i=1, ..., n represents the training data, R is the radius and represents the decision variable, ξ_i is the slack for each variable, a is the center, $C = \frac{1}{nf}$ is the penalty constant that controls the trade-off between the volume and the errors, and f is the expected outlier fraction.

Dual Form

The dual formulation is obtained from primal formulation by using the method of Lagrange multipliers. The objective function is

$$\max \sum_{i=1}^{n} \alpha_i (x_i \cdot x_i) - \sum_{i,j} \alpha_i \alpha_j (x_i \cdot x_j)$$

subject to

$$\sum_{i=1}^{n} \alpha_i = 1$$
$$0 \leq \alpha_i \leq C \text{ for all } i = 1, \ldots, n$$

where $\alpha_i \in \mathbb{R}$ are the Lagrange multipliers and $C = \frac{1}{nf}$ is the penalty constant.

Duality Information The solution to the dual formulation of SVDD provides values of Lagrange multipliers α_i for all training data observations $i = 1, \ldots, n$. Based on the value of α_i, the position of an observation in the context of the data description can be determined as follows:

Inside position:

$$\|x_i - a\| < R \implies \alpha_i = 0$$

Boundary position:

$$\|x_i - a\| = R \implies 0 < \alpha_i < C$$

Outside position:

$$\|x_i - a\| > R \implies \alpha_i = C$$

The center of the data description a can be obtained by using the values of α_i for all n observations in the training data as follows:

Center position:

$$\sum_{i=1}^{n} \alpha_i x_i = a$$

The circular data boundary can include a significant amount of space in which training observations are very sparsely distributed. Scoring with this model can increase the probability of false positives. Hence, instead of a circular shape, a compact bounded outline around the data is often desired. Such an outline should approximate the shape of the one-class training data. Such a data boundary is possible with the use of kernel functions.

Flexible Data Description

The support vector data description is made flexible by replacing the inner product $(x_i \cdot x_j)$ present in the objective function of ("Dual Form" on page 878) with a suitable kernel function $K(x_i, x_j)$. The radial basis kernel function is defined as

$$K(x, y) = \exp \frac{-\|x - y\|^2}{2s^2}$$

where x and y are any two observations from the training data and s is the bandwidth parameter. Using a kernel function, the objective function is

$$\max \sum_{i=1}^{n} \alpha_i K(x_i, x_i) - \sum_{i,j} \alpha_i \alpha_j K(x_i, x_j)$$

subject to

$$\sum_{i=1}^{n} \alpha_i = 1$$
$$0 \le \alpha_i \le C \text{ for all } i = 1, \ldots, n$$

The results for the center, inside, boundary, and outside positions, which are described in the section "Duality Information" on page 879, hold when the kernel function is used in the mathematical formulation.

The threshold R^2 is calculated as

$$R^2 = K(x_k, x_k) - 2\sum_{i} \alpha_i K(x_i, x_k) + \sum_{i,j} \alpha_i \alpha_j K(x_i, x_j)$$

using any $x_k \in SV_{<C}$, where $SV_{<C}$ is the set of support vectors for which $\alpha_k < C$.

Scoring

For each observation z in the scoring data set, the distance $\mathrm{dist}^2(z)$ is calculated as follows:

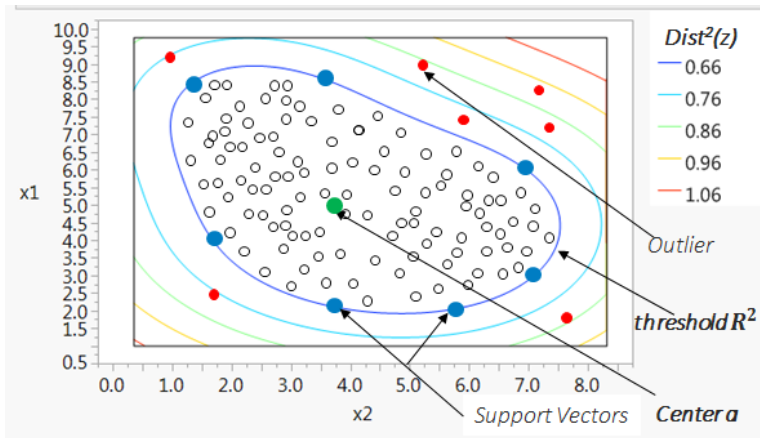
$$\mathrm{dist}^2(z) = K(z, z) - 2\sum_{i} \alpha_i K(x_i, z) + \sum_{i,j} \alpha_i \alpha_j K(x_i, x_j)$$

Observations in the scoring data set for which $\mathrm{dist}^2(z) > R^2$ are designated as outliers.

Visual Explanation of SVDD

Figure 34.7 illustrates key SVDD statistics. The unfilled markers indicate the bivariate training data set. The filled green marker indicates the center a. The filled blue markers are support vectors, which define the data boundary with a threshold R^2 value of 0.66. The plot shows contour lines for distance values from 0.76 to 1.06. The red markers indicate the outlier observations from the scoring data; their distance values are greater than the threshold R^2 value. The plot illustrates that as points move away from the training data, their distance value increases. It can be noted that the contours corresponding to different distance values follow the natural geometry of the training data.

Figure 34.7 Key SVDD Statistics

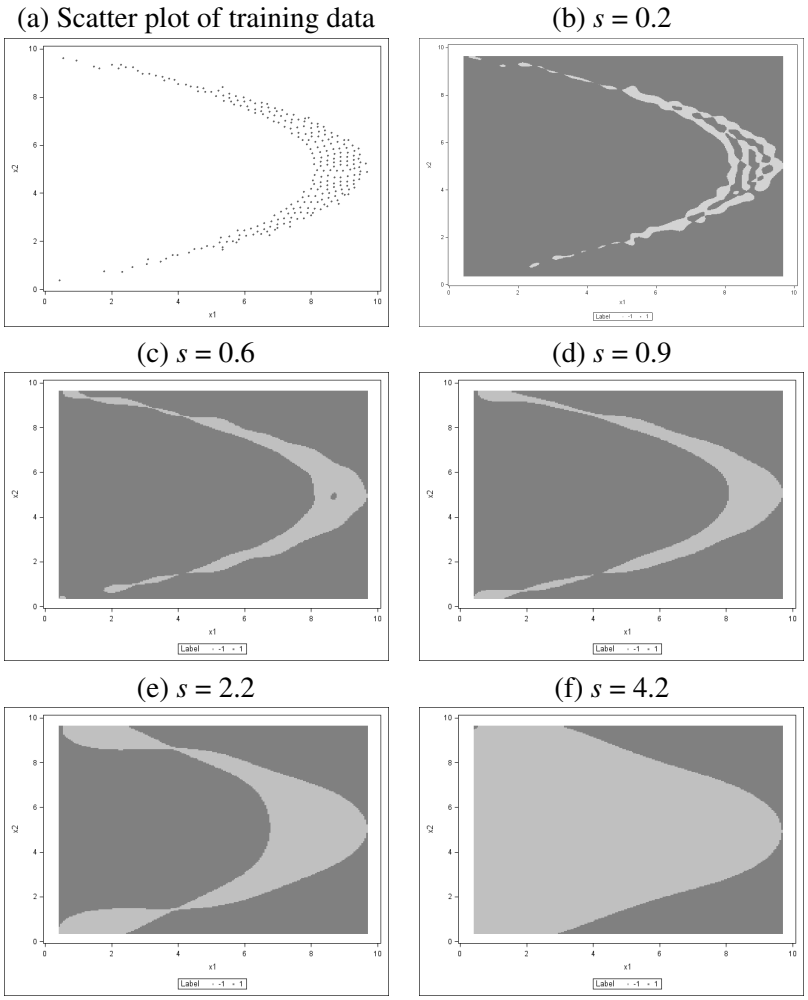


Radial Basis Function (RBF) Bandwidth Parameter

The flexible data description is preferred when the data boundary needs to closely follow the shape of the data. The tightness of the boundary is a function of the number of support vectors. In the case of an RBF kernel, it is observed that if the value of the outlier fraction f is kept constant, the number of support vectors identified by the SVDD algorithm is a function of the RBF bandwidth parameter s. At a very low value of s, the number of support vectors is very high and approaches the number of observations. As the value of s increases, the number of support vectors decreases. It is also observed that at lower values of s, the data boundary is extremely wiggly. As s is increased, the data boundary becomes less wiggly, and it starts to follow the shape of the data. At higher values of s, the data boundary starts becoming spherical. The processing time increases as the value of s decreases. At very small values of s, the processing time can be very long.

Figure 34.8 illustrates the effect of s on the data boundary.

Figure 34.8 Effect of RBF Bandwidth Parameter on Data Boundary



Subfigure (a) shows the training data. The training data were trained using different values of s. To confirm the shape of the data boundary, a 200×200 data grid was scored using training results. The scoring results are shown in subfigures (b) through (f). For subfigures (b) through (f), light gray indicates inside points and dark gray indicates outside points. All five results are obtained using $f = 0.0001$. At the lower value of $s = 0.2$, the data boundary is extremely wiggly. As the value of s is increased, the data boundary starts becoming less wiggly and conforms to the geometry of the training data. The higher value of $s = 4.2$ provides a very spherical data boundary. The data boundary that is obtained using $s = 0.9$ provides the best description of the data.

The selection of an appropriate value of s is tricky, and you must often experiment with several values until you obtain a good data boundary. If labeled data that belong to multiple classes are available, you can use cross validation to select an appropriate value of s. If training data observations belong to a single class, you cannot use cross validation. In such a scenario, you can use the distance to the farthest neighbor (DFN) method (Xiao et al. 2014), the peak criterion method (Kakde et al. 2017; Peredriy, Kakde, and Chaudhuri 2017), the mean or median criterion method (Chaudhuri et al. 2017), the modified mean criterion method (Liao et al. 2018) or the trace criterion method (Chaudhuri et al. 2021) to select an appropriate value of s. The SVDD procedure supports the mean criterion and modified mean criterion methods of selecting an appropriate value of s. You can use the mean criterion method by specifying BW=MEAN, the modified mean

criterion method by specifying BW=MEAN2 and the trace criterion method by specifying BW=TRACE as described in the KERNEL statement section.

Mean Criterion

The SVDD results depend on the training data through the kernel matrix. For training data that consist of n observations, the kernel matrix is an $n \times n$ matrix whose ij element is

$$\exp(-\frac{\|x_i - x_j\|^2}{2s^2})$$

where x_i and x_j are any two observations in the training data and s is the bandwidth parameter. When the kernel matrix is close to the identity matrix, SVDD overfits the data. The mean criterion chooses an s value such that the kernel matrix is sufficiently different from the identity matrix. The mean criterion provides a closed-form expression for computing the bandwidth value as

$$\sqrt{\frac{\bar{D}^2}{\ln \frac{n-1}{\delta^2}}}$$

where

$$\bar{D}^2 = \frac{\sum_{i<j} ||x_i - x_j||^2}{\binom{n}{2}}$$

For more information about the mean criterion, see Chaudhuri et al. (2017). The unsupervised bandwidth tuning is an extremely difficult problem, so it is quite possible that there is a class of data sets for which the mean criterion does not produce good results.

Modified Mean Criterion

The modified mean criterion method is an empirical method of bandwidth selection based on the peak method (Kakde et al. 2017) and the mean criterion method (Chaudhuri et al. 2017). This criterion provides the following closed-form expression for determining the value of parameter δ that is used in the mean criteria,

$$\delta = -0.14818008\phi^4 + 0.284623624\phi^3 - 0.252853808\phi^2 + 0.159059498\phi - 0.001381145$$

where $\phi = \frac{1}{\ln(n-1)}$ and n is the number of observations in the training data set.

The modified mean criterion method uses this value of delta and computes the bandwidth value as

$$\sqrt{\frac{\bar{D}^2}{\ln \frac{n-1}{\delta^2}}}$$

where

$$\bar{D}^2 = \frac{\sum_{i<j} ||x_i - x_j||^2}{\binom{n}{2}}$$

For more information about the modified mean criterion method, see Liao et al. (2018).

Trace Criterion

The trace criterion is an empirical method of bandwidth selection that is based on the low-rank representation of the kernel matrix. The rank of the low-rank kernel matrix can be specified by using the NREP= option, as described in the KERNEL statement. The default value of NREP is 5. If the number of clusters in the training data are known, you can get a better bandwidth value by specifying the number of clusters as the value of the NREP= option. This method has been seen to work well with high-dimensional data. For more information about the trace criterion method, see (Chaudhuri et al. 2021).

Bandwidth Relative Scale

The bandwidth relative scale value is the ratio of the user-specified RBF bandwidth parameter value to the bandwidth parameter value, computed using the mean criteria, with DELTA=1E–6. The bandwidth relative scale value is reported in the ModelInfo ODS table. If your analysis is taking a long time to complete or your model is overfitting the data, you can use the bandwidth relative scale value to determine whether resubmitting the analysis with a different bandwidth value is warranted. A very small value of the bandwidth relative scale (less than one) indicates that the user-specified bandwidth value is too low for the training data. In such a scenario, you can improve the performance or quality of your training results by resubmitting the analysis with a higher bandwidth value.

There are multiple factors (such as, but not limited to, solver type, training data, and solver tolerances) that can affect the processing time and quality of your results. Hence, there might be situations in which adjusting the bandwidth parameter value alone might not improve the performance or quality of your results.

Fraction Outlier Parameter

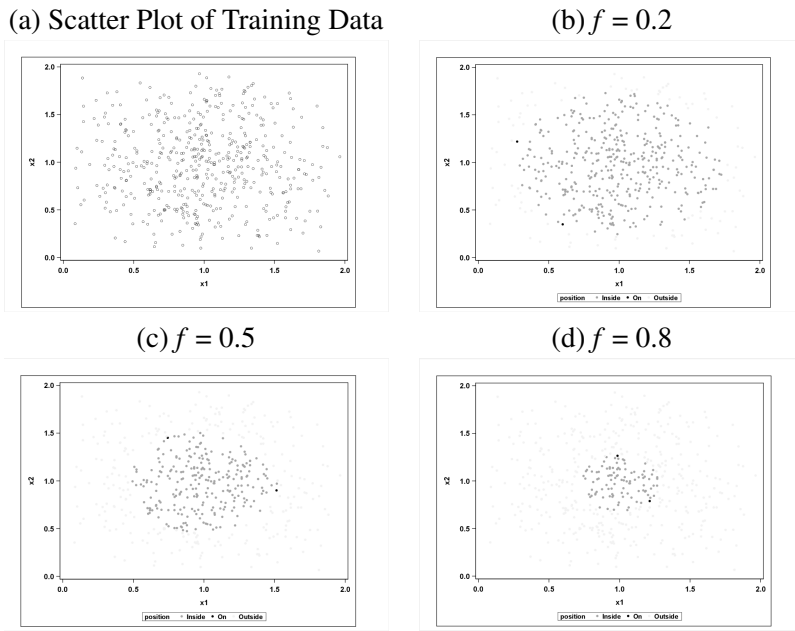
As outlined in the section "Mathematical Formulation" on page 878, the penalty constant C controls the trade-off between the volume and accuracy. The constant C is defined as

$$C = \frac{1}{nf}$$

where n is the number of observations in the training data set and f is the fraction of outliers.

Keeping the value of the RBF bandwidth parameter s constant, higher values of f produce a tighter boundary around the training data and thereby lower the enclosed volume. As the value of f is decreased, the enclosed volume tends to increase. Figure 34.9 illustrates the effect of f on the data boundary.

Figure 34.9 Effect of f on Data Boundary



For subfigures (b) through (d), light gray indicates outside points, dark gray indicates inside points, and black indicates support vectors. All three results are obtained using $s = 2$. As the value of f is increased from 0.2 to 0.8, the data boundary becomes tighter and the enclosed area is reduced. A data description that is obtained using an incorrect value of f can be misleading. If the specified value of f is greater than the true value, it can lead to a higher misclassification rate because observations are incorrectly classified as outliers. The corresponding threshold R^2 value is also smaller. On the other hand, a specified value of f that is less than its true value can also lead to a higher misclassification rate because outliers are classified as inliers.

It is recommended that you use domain expertise and select SVDD training data such that the data truly reflect the single class of interest. The corresponding f can then be set to a very small value.

SVDD Formulation with a Weight Variable

You can use the WEIGHT statement to specify observation weights. The weight can be observation frequency. The following expression shows the dual formulation of SVDD with a kernel function and a weight variable:

$$\max \sum_{i=1}^{n} \alpha_i K(x_i, x_i) - \sum_{i,j} \alpha_i \alpha_j K(x_i, x_j)$$

subject to

$$\sum_{i=1}^{n} \alpha_i = 1$$

$$0 \le \alpha_i \le \frac{w_i}{(\sum_{i=1}^{n} w_i) f}, \text{ for all } i = 1, \ldots, n$$

where

$$K(x, y) = \exp \frac{-\|x - y\|^2}{2s^2}$$

where x and y are any two observations from the training data and s is the bandwidth parameter; $\alpha_i \in \mathbb{R}$ are the Lagrange multipliers, for all $i = 1, \ldots, n$; and w_i are the weight values, for all $i = 1, \ldots, n$.

Using a weight variable in SVDD formulation results in observations with higher weights having higher upper bounds on the value of the Lagrange multipliers. Such observations are more likely to end up with a positive nonzero value of their Lagrange multipliers and hence are designated as support vectors.

The WEIGHT statement has no impact on the results when the value of the fraction outlier parameter f is less than $\frac{\min w_i}{\sum_{i=1}^{n} w_i}$ for all $i = 1, \ldots, n$.

Comparison with One-Class Support Vector Machines

The one-class support vector machines (OCSVM) is a one-class classification technique similar to the SVDD. Instead of obtaining a bounding hypersphere around the training data, the OCSVM algorithm finds the maximal margin hyperplane that best separates the training data from the origin (Schölkopf et al. 1999). Similar to SVDD, the OCSVM algorithm uses kernel functions to map training data into a higher-dimensional feature space. The dual formulation of OCSVM can be expressed as

$$\min \ \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j K(x_i, x_j)$$

subject to

$$\sum_{i=1}^{n} \alpha_i = 1$$

$$0 \le \alpha_i \le \frac{1}{\nu n} \text{ for all } i = 1, \ldots, n$$

where $\alpha_i \in \mathbb{R}$ are the Lagrange multipliers, ν is a parameter that controls the trade-off between maximizing the distance of the hyperplane from the origin and the number of data points contained by the hyperplane, n is the number of points in the training data, and $K(x_i, x_j)$ is the kernel function.

For the radial basis kernel function (RBF)

$$K(x, y) = \exp \frac{-\|x - y\|^2}{2s^2}$$

where x and y are any two observations from the training data and s is the bandwidth parameter.

With the RBF kernel function, the first term of the SVDD objective function of the flexible data description, as outlined in "Flexible Data Description" on page 879 evaluates to 1. Hence the OCSVM formulation is equivalent to the SVDD formulation when the RBF kernel function is used.

SVDD Solvers

You can perform SVDD training using either the active-set solver or the stochastic subset solver as described in the SOLVER Statement.

Active-Set Solver

The active set is defined as the set of indices that satisfy bound constraints as equality. The inactive set is a set of indices that satisfy bound constraints as strict inequality. For SVDD dual formulation, the active set refers to the set indices of observations for which $\alpha = 0$ or $\alpha = C$, and the inactive set refers to set indices of observations for which $0 < \alpha < C$. The active-set method seeks to form the solution by determining which indices of bound constraints belong to the active set and by using the inactive set (whose size is relatively small) to solve the optimization problem. Even though there is a great number of ways to partition bound constraint indices into active and inactive sets, the number of combinations tried in practice is usually small. A consequence is that the run time for the active-set method for a particular problem is hard to predict. The computational cost of the active-set method is also controlled by the number of support vectors, and that number is highly dependent on both the problem and the choice of the kernel bandwidth value. As a rule, the run time increases as the kernel bandwidth value decreases and the corresponding number of support vectors increases.

Stochastic Subset Solver

The stochastic subset solver for SVDD training provides an efficient way to train your SVDD model and is recommended for training large data sets. The stochastic solver performs the following steps on training data with a sample size *n*, Gaussian bandwidth parameter *s*, fraction outlier *f*, and convergence criteria parameters that are specified in the SOLVER statement:

1. Initialize a master set of support vectors SV^* by computing the SVDD of a random sample S_0. Set $i = 1$.
2. Select a random sample S_i. Compute its SVDD. Obtain a set of support vectors SV_i. When the WEIGHT statement is specified, the probability of an observation becoming part of the sample is proportional to the observation's weight value.
3. Compute the SVDD of $(SV^* \cup SV_i)$. Designate corresponding support vectors as SV^*. Set $i = i + 1$.
4. Repeat steps 2 and 3 until convergence of the threshold value R^2 and center a.

When convergence is obtained, the resulting master set of support vectors SV^* is set as an input to the active-set solver and a solution is obtained. For more information about the stochastic subset solver, see Chaudhuri et al. (2018).

Fast Incremental Solver

The fast incremental solver uses incremental updates to the set of support vectors for performing SVDD training. This algorithm builds on an observation that all support vectors on the boundary have the same distance to the center of sphere in a higher-dimensional feature space as mapped by the Gaussian kernel function. This method uses rank-1 updates and a fast bookkeeping operation to update the solution. If a new point does not substantially alter the solution, then it is discarded. For more information about the fast incremental solver, see (Jiang et al. 2019).

Scoring Multiple SVDD Models with the ASTORE Procedure

The KERNEL statement section outlines how you can specify multiple BW= option values in that statement. You can use the SAVESTATE statement to generate a single analytical store that contains scoring models corresponding to each bandwidth value. If the BW= option contains *n* bandwidth values, then the analytical store will contain *n* scoring models. The scoring model for each bandwidth value is stored inside the analytic store with a unique value of the modelID variable. When the BW= option contains *n* bandwidth values, the value of the modelID variable ranges from 1 to *n*.

In the ASTORE procedure, you can use a subset of scoring models by specifying the SETOPTION statement with the MODELID option. The following examples show how to use the SETOPTION statement in PROC ASTORE.

The following statements execute the SVDD algorithm on the mylib.refrigerant data table by using three bandwidth values, 0.05, 0.1, and 0.15:

```
proc svdd data=mylib.Refrigerant score nbins=10;
id y4 y5;
input y4 y5/level=interval;
kernel rbf / bw=(0.05 0.1 0.15);
savestate rstore=mylib.state;
run;
```

The analytic store mylib.state contains three scoring models, with modelID 1, 2, and 3 corresponding to the bandwidth values 0.05, 0.1, and 0.15, respectively.

The following statements execute the ASTORE procedure without the SETOPTION statement. This causes the scoring data to be scored using all three models, corresponding with the bandwidth values 0.05, 0.1, and 0.15.

```
proc astore;
   score data=mylib.scoreds
   out=mylib.score_results
   rstore=mylib.state;
quit;
```

The following statements execute the ASTORE procedure with the SETOPTION statement. This causes the scoring data to be scored using the model for which MODELID=2, corresponding to the option BW=0.1. You must specify the SETOPTION statement before the SCORE statement.

```
proc astore;
   setoption modelid 2;
   score data=mylib.scoreds
   out=mylib.score_results
   rstore=mylib.state;
quit;
```

You can instruct the ASTORE procedure to keep bandwith values that correspond to each model in the OUT data table by setting the BWCOL option to 1. You can remove the bandwidth value from the OUT data table by setting the BWCOL option to 0. The default value of the BWCOL option is 0.

The following statements execute the ASTORE procedure with the SETOPTION statement. The BWCOL option is set to 1, which causes the mylib.score_result to include bandwidth columns for each model. You must specify the SETOPTION statement before the SCORE statement.

```
proc astore;
   setoption bwcol 1;
   score data=mylib.scoreds
   out=mylib.score_results
   rstore=mylib.state;
quit;
```

Displayed Output

PROC SVDD displays various tables for analysis. The following sections describe the output in the order of its appearance when you specify the related options. The "Number of Observations," "Nominal Variables," "Status Information," and OutputCasTables tables are produced once per procedure run. The "Model Information," "Training Results," "Optimization Summary," and "Frequency Distribution for SVDD Distance" tables are produced for each SVDD model that is trained. When you specify multiple bandwidth parameter values in the form BW=(*number list*) or BW=(*n* TO *m* BY *p*) in the KERNEL statement, multiple SVDD models are trained, and each model produces its own set of model-specific tables.

Number of Observations

The "Number of Observations" table displays the number of observations that are read from the input data table and the number that are used. This table is referred to as the Nobs table in Table 34.1.

Nominal Variables

The "Nominal Variables" table shows the number of levels for each nominal variable. This table is referred to as the DescStatsNom table in Table 34.1.

Model Information

The "Model Information" table displays the optimization method, kernel type, RBF kernel bandwidth, bandwidth selection method, optimization tolerance, number of interval variables, and number of nominal variables. If all the input variables are interval, the table also displays the bandwidth relative scale. If you specify the STD option, the table also displays the standardization flag. If the *solver-type* in the SOLVER statement is ACTSET, the table also displays the expected outlier fraction. If the *solver-type* in the SOLVER

statement is STOCHS, the table also displays the stochastic method, sample size, and seed value. If the *solver-type* in the SOLVER statement is FASTINC, the table also displays the outlier threshold. This table is referred to as the ModelInfo table in Table 34.1.

Training Results

The "Training Results" table displays the number of support vectors; number of support vectors on the boundary; number of outliers when you specify the SCORE option in the PROC SVDD statement; number of dropped observations; threshold R-square value that is used in scoring (see the section "Scoring" on page 880); constant (C_r) value, which is the final term in the distance calculation formula (see the section "Scoring" on page 880); total run time (in seconds); and bandwidth calculation time (in seconds) when the bandwidth is calculated (when the BW= option value in the KERNEL statement is not a fixed value or a number list). This table is referred to as the TrainingResults table in Table 34.1.

Optimization Summary

The "Optimization Summary" table displays the number of iterations; objective value; infeasibility; optimization status; and degenerate flag, which indicates how the threshold R-square value is determined. The degenerate flag is set to "No" when the inactive set is nonempty and the threshold R-square value is the distance that is calculated from the inactive set. The degenerate flag is set to "Yes" when the inactive set is empty and the threshold R-square value is the minimum distance that is calculated from the active set. This table is referred to as the OptSummary table in Table 34.1.

Frequency Distribution for SVDD Distance

The "Frequency Distribution for SVDD Distance" table summarizes the empirical distribution of the SVDD distance values for the training data. The distances are grouped in bins, where the "Bin Midpoint" represents the center of each distance interval. For each bin, the table reports the number of observations ("Count"), the percentage of total observations in the bin ("Percent"), and the cumulative percentage up to that bin ("Cumulative"). This table is referred to as the Histogram table in Table 34.1.

Status Information

The "Status Information" table displays the SVDD model instances and their status. This table is referred to as the StatusTable table in Table 34.1.

OutputCasTables Table

The OutputCasTables table is a special table that has information about each table that is created. The information for each table consists of the table name, the library in which the table resides, and the number of columns and rows in the table. Because this table is not a typical ODS table that contains analytical results, you cannot include it in the *table-spec-list* in the DISPLAYOUT statement.

ODS Table Names

Each table that the SVDD procedure creates has a name associated with it. You must use this name to refer to the table when you use ODS statements. The name of each table and a short description of the contents are listed in Table 34.1.

Table 34.1 ODS Tables Produced by PROC SVDD

Table Name	Description	Statement	Option
DescStatsNom	Number of levels of nominal variables in the input data	PROC SVDD	Default
Histogram	Frequency table of _SVDDDISTANCE_ values of training data	PROC SVDD	SCORE
ModelInfo	Basic model information for the training	PROC SVDD	Default
Nobs	Observation information about the input data	PROC SVDD	Default
OptSummary	Statistics related to optimization solver	PROC SVDD	Default
OutputCasTables	See the section "OutputCasTables Table" on page 890	PROC SVDD	Default
StatusTable	Status of each training model	PROC SVDD	Default
TrainingResults	Displays the training results	PROC SVDD	Default

Output Data Tables

PROC SVDD creates one data table to contain the training data observations that are identified as support vectors, the corresponding value of the Lagrange multiplier, and an indicator that flags the position of the support vectors relative to the data boundary. You specify the names of this data table in the OUTSV= option in the PROC SVDD statement.

For information about the output data table, see the section "OutputCasTables Table" on page 890.

Examples: SVDD Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

Example 34.1: Analysis of Aircraft Engine Degradation

Sun and Tsung (2003) introduced an SVDD-based K-chart, which is a nonparametric multivariate control chart that is used for statistical process control and can also be used for monitoring equipment health and operating data. A K-chart is implemented in two phases: In phase I, observations from normal operations of the process are collected and are used to train a SVDD model and obtain the threshold R^2 value. This model of normal operations is then operationalized in phase II for process or equipment monitoring. For each new observation z, its distance value $\mathrm{dist}^2(z)$ is computed and compared to the threshold R^2 value. Observations for which $\mathrm{dist}^2(z) > R^2$ generally indicate something abnormal in the process.

This example uses a K-chart to illustrate the analysis of aircraft engine degradation and uses data from Saxena et al. (2008); Saxena and Goebel (2008). The data that are used in this example consist of the flight history of 15 engines. For each flight, three variables are related to the engine's operating conditions and 21 variables are for sensor measurements. Because each engine degrades at a different rate, the number of flights until the end of life is different for each engine.

The following code creates a phase I (normal operations) data set (the engine data set in the Work library) by selecting the first 25% of observations of a random sample of five engines. The analysis assumes that the first 25% of observations represent normal operations of the aircraft engine.

```
data engine;
   input Engine cycle x1-x24;
   datalines;

   ... more lines ...
```

The following statements creates a subset of work.engine by selecting the first 25% of observations that belong to five randomly selected engines:

```
data eid(keep=engine);
   set engine;
   by engine;
   if first.engine then output;
run;

data reid;
   retain nleft 5;
   set eid nobs=n;
   r = ranuni(1994);
   if r <= nleft/n then do;
      nleft = nleft - 1;
```

```
      output;
   end;
   n = n - 1;
run;

 %macro a;

data _null_;
   set reid;
   call symput('e'||left(_n_), engine);
run;

 %do i=1 %to 5;
    %if &i=1 %then %do;
       %let st=&&e&i;
   %end;
   %else %do;
       %let st=&st &&e&i;
   %end;
 %end;

data eid_cy;
   set engine;
   by engine ;
   if engine in (&st) and last.engine=1 then output;
run;

data eid_cy;
   set eid_cy;
   ds1=floor(0.25*cycle);
run;

/* The following DATA steps create the training data set */
data train;
   set engine;
   if engine in (&st) then output;
run;

data train;
   merge train(in=a) eid_cy(in=b);
   by engine;
   if a=b;
run;

data train;
   set train (where=(cycle le ds1));
run;

/* The following DATA step creates the scoring data set. The scoring data set
contains flight data for all engines, which were not part of the training data set */
data score;
   set engine;
   if engine not in (&st)  then output;
run;
```

```
%mend a;
%a
```

You can load the work.train data set into your SAS session by specifying your SAS libref in the first statement in the following DATA step. These statements assume that your SAS libref is named mylib, as in the section "Using SAS Viya Workbench" on page 865, but you can substitute any appropriately defined SAS libref.

```
data mylib.train;
   set train;
run;
```

You can load the work.score data set into your SAS session by specifying your SAS libref in the first statement in the following DATA step:

```
data mylib.score;
   set score;
run;
```

The following statements execute the SVDD algorithm on the mylib.train data table and produce Figure 34.1.1 through Figure 34.1.2.

```
proc svdd data=mylib.train;
   id engine cycle;
   input x1-x24/level=interval;
   kernel rbf / bw=94;
   savestate rstore=mylib.state;
run;
```

The INPUT statement defines 24 input variables (x1–x24) as interval variables. The KERNEL statement specifies the radial basis function as the kernel function and specifies a value of 94 for the bandwidth parameter. The SAVESTATE statement creates an analytic store for the model and saves it as a binary object in the mylib.state data table. You can use the analytic store later in the ASTORE procedure to score new data. The "Model Information" table in Output 34.1.1 summarizes the key options and input data variables.

Output 34.1.1 Aircraft Engine Degradation Analysis Model Information

Model Information	
Optimization Method	Active Set
Kernel Type	RBF
RBF Kernel Bandwidth	94
Bandwidth Selection Method	User specified
Bandwidth Relative Scale	0.797583926
Expected Outlier Fraction	1E-6
Optimization Tolerance	0.0001
Number of Interval Variables	24
Number of Nominal Variables	0

The "Training Results" table in Output 34.1.2 shows the number of support vectors and the threshold R^2 value.

Output 34.1.2 Aircraft Engine Degradation Analysis Training Results

Training Results	
Number of Support Vectors	13
Number of Support Vectors on Boundary	13
Number of Dropped Observations	0
Threshold R Square Value	0.80366
Constant (C_r) Value	0.19634
Run Time (seconds)	0.00411

Output 34.1.3 displays the number of observations in the training data set.

Output 34.1.3 Aircraft Engine Degradation Analysis Training Observations

Number of Observations Read	225
Number of Observations Used	225

The "Optimization Summary" table in Output 34.1.4 shows whether the solution is optimal, the number of iterations that were required and the objective function value.

Output 34.1.4 Aircraft Engine Degradation Analysis Optimization Summary

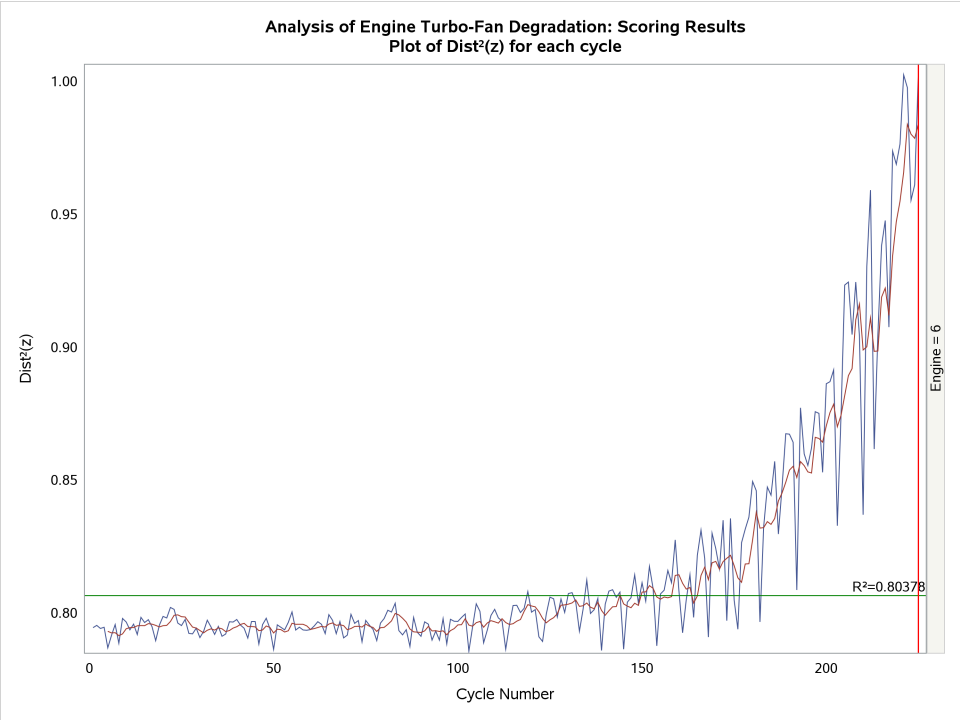
Optimization Summary	
Number of Iterations	1
Objective Value	0.1963442117
Infeasibility	6.019851E-12
Optimization Status	Optimal
Degenerate	No

The following PROC ASTORE code scores the mylib.score data, which contain the entire flight history for 10 engines. In the context of K-charts, this scoring constitutes the phase II analysis, where new observations are scored to infer any departure from the normal operations.

```
proc astore;
   score data=mylib.score
   out=mylib.test_out
   rstore=mylib.state;
quit;
```

Output 34.1.5 shows the scoring results for Engine 6. The plot indicates the $\mathrm{dist}^2(z)$ value that is plotted on the Y axis against the flight or cycle number on the X axis. The blue line indicates the $\mathrm{dist}^2(z)$ value, and the red line indicates a moving average that is computed using five observations. The horizontal reference line indicates the threshold R^2 value of 0.80378 that was obtained from training, which also acts as an upper control limit (UCL) for the $\mathrm{dist}^2(z)$ value. The vertical reference line indicates the last cycle (flight) of the first engine. Output 34.1.5 shows that the $\mathrm{dist}^2(z)$ value serves as a good proxy for engine degradation.

Output 34.1.5 Scoring Results Scatter Plot



.

Example 34.2: Automatic RBF Kernel Bandwidth Parameter Value Selection

This example demonstrates how you can use automatic kernel bandwidth selection for the radial basis function (RBF) in the SVDD procedure. In this example, the use of the mean criterion is illustrated first, followed by the use of the modified mean criterion and the trace criterion. The training data in this example are refrigerant analysis data (Heckert and Filliben 2000). These data are also used in the section "Getting Started: SVDD Procedure" on page 865, where the value of the RBF kernel bandwidth parameter is predetermined by the specification of the BW=0.021 option in the KERNEL statement. In the current example, the SVDD procedure computes the RBF kernel bandwidth parameter value by using the automatic bandwidth selection methods, alleviating the need to explicitly specify a value. The data contain 360 observations and 31 variables. The variables represent various process measurements that are taken during the refrigerant analysis. This example uses the variables y5 (inlet water temperature 1) and y4 (pressure drop 6) to perform an SVDD training. The variables y4 and y5 are selected to highlight the ability of the SVDD algorithm to obtain an accurate description of the training data.

The following DATA step creates the Refrigerant data set (which is stored in the Work library by default):

```
data refrigerant;
   infile datalines delimiter=',';
   input y4 y5 ;
   label y4="Pressure Drop 6" y5="Inlet Water Temperature 1";
   datalines;
2.0562,25.0904
```

```
2.04147,25.1676
2.04922,25.6362

   ... more lines ...
```

The following code standardizes the variables y4 and y5:

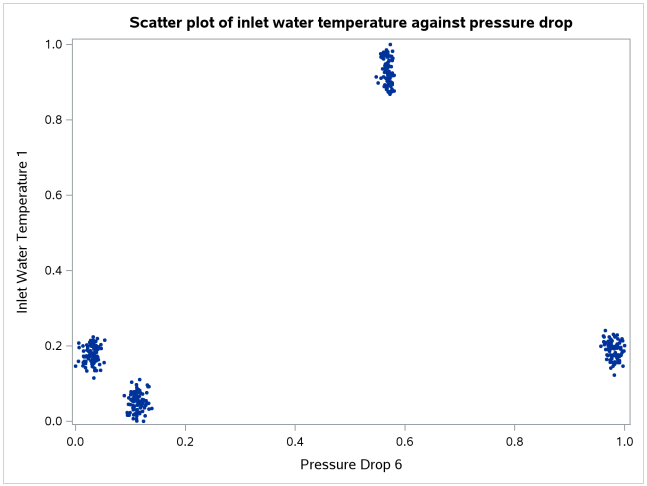
```
proc sql noprint;
   select min(y4), min(y5), max(y4), max(y5)
   into :miny4, :miny5, :maxy4, :maxy5
   from  Refrigerant;
quit;


   data Refrigerant;
      set Refrigerant;
      y4=(y4-&miny4)/%sysevalf(&maxy4-&miny4);
      y5=(y5-&miny5)/%sysevalf(&maxy5-&miny5);
   run;
```

The following PROC SGPLOT statements plot inlet water temperature 1 against pressure drop 6:

```
proc sgplot data=Refrigerant;
   title "Refrigerant Analysis";
   title1 "Scatter plot of inlet water temperature against pressure drop";
   scatter x=y4 y=y5/markerattrs=(size=3 symbol=circlefilled);
run;
```

Output 34.2.1 shows the results. The scatter plot indicates four distinct clusters.

Output 34.2.1 Scatter Plot of Inlet Water Temperature against Pressure Drop



You can load the Work.Refrigerant data set into your SAS session by specifying your SAS libref in the first statement in the following DATA step:

```
data mylib.Refrigerant;
      set Refrigerant;
run;
```

These statements assume that your SAS libref is named mylib, as in the section "Using SAS Viya Workbench" on page 865, but you can substitute any appropriately defined SAS libref.

The following statements execute the SVDD algorithm on the mylib.Refrigerant data table and produce Output 34.2.2 through Output 34.2.6.

```
proc svdd data=mylib.Refrigerant score nbins=10;
   id y4 y5;
   input y4 y5/level=interval;
   kernel rbf / bw=mean;
   savestate rstore=mylib.state;
run;
```

The SCORE option in the PROC SVDD statement requests that the training data set be scored. The NBINS=10 option creates a histogram data set with 10 bins. The INPUT statement defines the input variables y4 and y5 as interval variables. The KERNEL statement specifies the kernel function as a radial basis function (RBF), and the BW=MEAN option instructs the SVDD procedure to select value of the RBF bandwidth parameter by using the mean criterion. The SAVESTATE statement creates an analytic store for the model and saves it as a binary object in the mylib.state data table. You can use the analytic store later in the ASTORE procedure to score new data. The "Model Information" table in Output 34.2.2 summarizes the key options and the input data variables, and it provides the value of the RBF kernel bandwidth parameter, which is computed to be 0.1262 by using the mean criterion.

Output 34.2.2 Refrigerant Analysis Model Information

Model Information	
Optimization Method	Active Set
Kernel Type	RBF
RBF Kernel Bandwidth	0.1262714382
Bandwidth Selection Method	Mean (delta = 1e-06)
Bandwidth Relative Scale	1
Expected Outlier Fraction	1E-6
Optimization Tolerance	0.0001
Number of Interval Variables	2
Number of Nominal Variables	0

The "Training Results" table in Output 34.2.3 shows the number of support vectors and the R^2 threshold value.

Output 34.2.3 Refrigerant Analysis Training Results

Training Results	
Number of Support Vectors	7
Number of Support Vectors on Boundary	7
Number of Outliers	0
Number of Dropped Observations	0
Threshold R Square Value	0.76304
Constant (C_r) Value	0.23696
Run Time (seconds)	0.00384
Bandwidth Calculation Time (seconds)	0.00000310

Output 34.2.4 displays the number of observations in the training data set.

Output 34.2.4 Refrigerant Analysis Training Observations

Number of Observations Read	360
Number of Observations Used	360

The "Optimization Summary" table in Output 34.2.5 shows whether the solution is optimal, the number of iterations that were required, and the objective function value.

Output 34.2.5 Refrigerant Analysis Optimization Summary

Optimization Summary	
Number of Iterations	1
Objective Value	0.2369581441
Infeasibility	1.897082E-11
Optimization Status	Optimal
Degenerate	No

The "Histogram" table in Output 34.2.6 shows the frequency table of _SVDDDISTANCE_ values of observations in the mylib.Refrigerant training data set.

Output 34.2.6 Refrigerant Analysis Histogram Table

Frequency Distribution for SVDD Distance			
Bin Midpoint	Count	Percent (%)	Cumulative (%)
0.716108	28	7.78	7.78
0.721048	45	12.50	20.28
0.725989	73	20.28	40.56
0.730929	62	17.22	57.78
0.735869	48	13.33	71.11
0.740810	37	10.28	81.39
0.745750	30	8.33	89.72
0.750691	12	3.33	93.06
0.755631	15	4.17	97.22
0.760572	10	2.78	100.00

To evaluate the quality of the training results, scoring is performed on a 200 × 200 data grid. The following DATA step creates a data set named scoreds to contain the scoring observations:

```
data scoreds;
   do i=0 to 1 by 0.005;
      do  j=0 to 1 by 0.005;
         output;
      end;
   end;
   rename i=y5 j=y4;
run;
```

You can load the scoreds data set into your SAS session by specifying your SAS libref in the first statement in the following DATA step:

```
data mylib.scoreds;
   set scoreds;
run;
```

The following PROC ASTORE code scores the mylib.scoreds data table by using the score analytic store that was created during training:

```
proc astore;
   score data=mylib.scoreds
   out=mylib.score_results
   rstore=mylib.state;
quit;
```

The following code copies the score_results data table into the Work library:

```
data work.score_results;
   set mylib.score_results;
run;
```

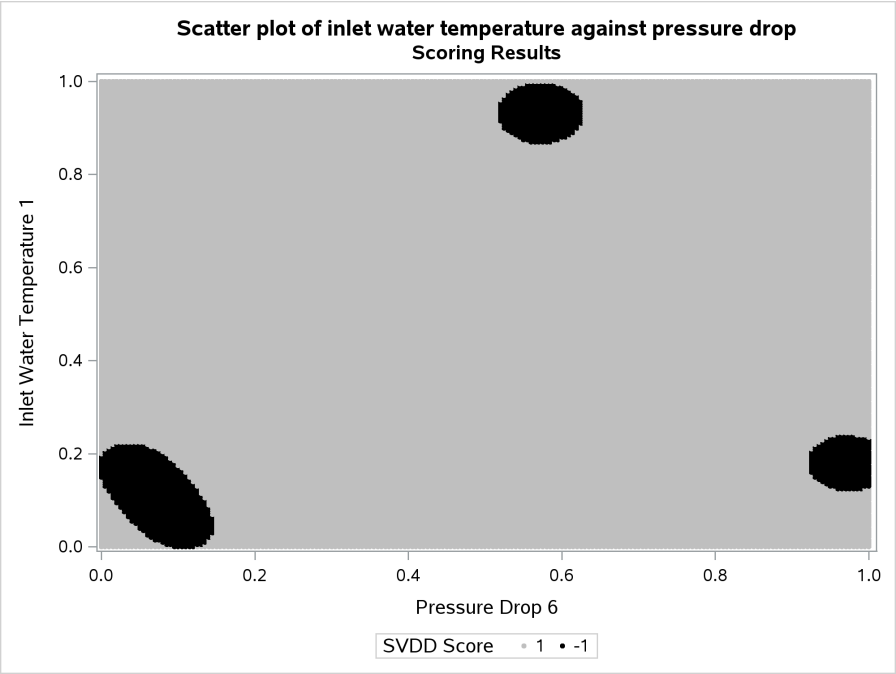
The following PROC SGPLOT code plots the scoring results:

```
proc sgplot data=score_results(where=(_svddscore_ in (-1,1)));
   styleattrs datacontrastcolors=(ligr black) datasymbols=(circlefilled);
   title "Refrigerant Analysis: Mean Criterion";
   title1 "Scatter plot of inlet water temperature against pressure drop";
   title2 "Scoring Results";
   scatter x=y4 y=y5/group=_svddscore_ markerattrs=(size=3);
run;
```

Output 34.2.7 shows the scatter plot of the scoring data. The gray area indicates the observations that are identified as outliers, and the black area indicates observations that are identified as inliers. In this example, the mean criterion was used to compute the RBF kernel bandwidth parameter value. Comparison between Output 34.2.7 and the training data in Output 34.2.1 indicates that PROC SVDD identified three out of four clusters in the original data. The training data observations in the two clusters located in the lower left corner were combined into one cluster. In addition, the description of clusters that is obtained in this example is more circular. This is expected because the RBF bandwidth parameter value of 0.1262, which is computed using the mean criterion, is higher than the value 0.021, which is specified in the section "Getting Started: SVDD Procedure" on page 865. Despite these shortcomings, the mean criterion provides a reasonable approximation

of the training data. The scatter plot of the scoring data in Output 34.2.7 indicates that PROC SVDD was able to correctly identify a majority of the outliers.

Output 34.2.7 Scoring Results Scatter Plot: Bandwidth Computation Using the Mean Criterion



The use of the modified mean criterion for bandwidth selection is illustrated next.

The following statements execute the SVDD algorithm on the mylib.Refrigerant data table and produce the results shown in Output 34.2.8 through Output 34.2.12. The SVDD algorithm uses the bandwidth value that is computed using the modified mean criterion.

```
proc svdd data=mylib.Refrigerant score nbins=10;
id y4 y5;
input y4 y5/level=interval;
kernel rbf / bw=mean2;
savestate rstore=mylib.state;
run;
```

The SCORE option in the PROC SVDD statement requests that the training data table be scored. The NBINS=10 option creates a histogram data table that contains 10 bins. The INPUT statement defines the input variables y4 and y5 as interval variables. The KERNEL statement specifies the kernel function as a radial basis function (RBF), and the BW=MEAN2 option instructs the SVDD procedure to select the value of the RBF bandwidth parameter by using the modified mean criterion. The SAVESTATE statement creates an analytic store for the model and saves it as a binary object in the data table mylib.state. You can use the analytic store later in the ASTORE procedure to score new data. The "Model Information" table in Output 34.2.8 summarizes the key options and the input data variables, and it provides the value of the RBF kernel bandwidth parameter, which is computed using the modified mean criterion to be 0.197.

Output 34.2.8 Refrigerant Analysis Model Information

Model Information	
Optimization Method	Active Set
Kernel Type	RBF
RBF Kernel Bandwidth	0.1971705477
Bandwidth Selection Method	Modified Mean
Bandwidth Relative Scale	1.5614817616
Expected Outlier Fraction	1E-6
Optimization Tolerance	0.0001
Number of Interval Variables	2
Number of Nominal Variables	0

The "Training Results" table in Output 34.2.9 shows the number of support vectors and the R^2 threshold value.

Output 34.2.9 Refrigerant Analysis Training Results

Training Results	
Number of Support Vectors	7
Number of Support Vectors on Boundary	7
Number of Outliers	0
Number of Dropped Observations	0
Threshold R Square Value	0.71933
Constant (C_r) Value	0.28067
Run Time (seconds)	0.00398
Bandwidth Calculation Time (seconds)	0.00000191

Output 34.2.10 displays the number of observations in the training data table.

Output 34.2.10 Refrigerant Analysis Training Observations

Number of Observations Read	360
Number of Observations Used	360

The "Optimization Summary" table in Output 34.2.11 shows whether the solution is optimal, the number of iterations that were required, and the objective function value.

Output 34.2.11 Refrigerant Analysis Optimization Summary

Optimization Summary	
Number of Iterations	1
Objective Value	0.280665654
Infeasibility	5.399015E-13
Optimization Status	Optimal
Degenerate	No

The "Histogram" table in Output 34.2.12 shows the frequency table of _SVDDDISTANCE_ values of observations in the mylib.Refrigerant training data table.

Output 34.2.12 Refrigerant Analysis Histogram Table

Frequency Distribution for SVDD Distance			
Bin Midpoint	Count	Percent (%)	Cumulative (%)
0.656727	10	2.78	2.78
0.663317	22	6.11	8.89
0.669908	36	10.00	18.89
0.676498	24	6.67	25.56
0.683088	28	7.78	33.33
0.689678	43	11.94	45.28
0.696269	97	26.94	72.22
0.702859	59	16.39	88.61
0.709449	27	7.50	96.11
0.716039	14	3.89	100.00

The following PROC ASTORE code scores the mylib.scoreds data table by using the score analytic store that was created during training. The mylib.scoreds data table that is used in the following code is the same scoring data used earlier to evaluate the quality of results that were obtained using the SVDD model that was trained with the mean criterion. Using the same data enables you to objectively compare the bandwidth values that are obtained using the mean criterion and the modified mean criterion.

```
proc astore;
score data=mylib.scoreds
out=mylib.score_results
rstore=mylib.state;
quit;
```

The following code copies the score_results into the Work library:

```
data work.score_results;
set mylib.score_results;
run;
```

The following PROC SGPLOT code plots the scoring results:

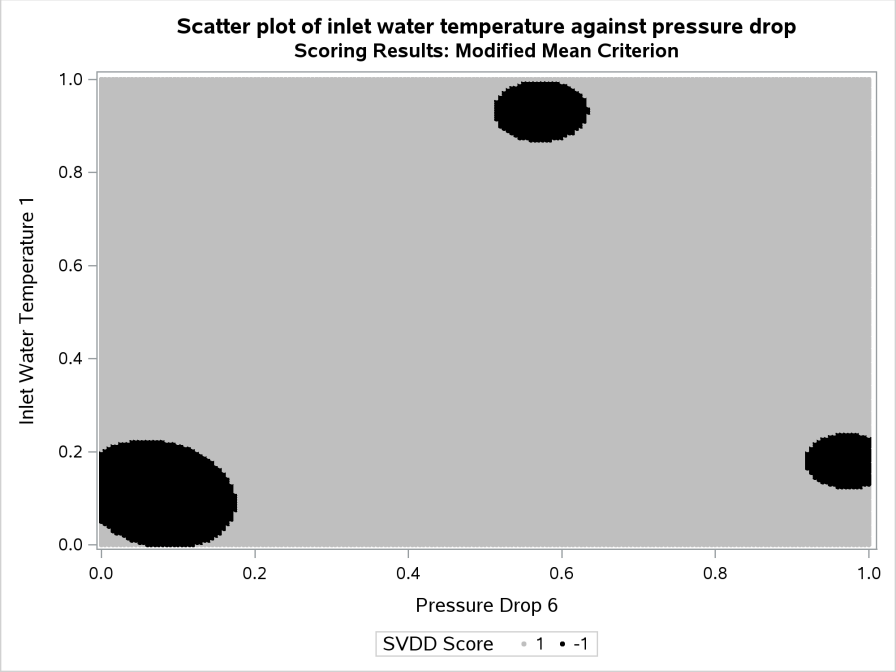
```
proc sgplot data=score_results(where=(_svddscore_ in (-1,1)));
styleattrs datacontrastcolors=(ligr black) datasymbols=(circlefilled);
title "Refrigerant Analysis";
title1 "Scatter plot of inlet water temperature against pressure drop";
title2 "Scoring Results: Modified Mean Criterion";
scatter x=y4 y=y5/group=_svddscore_ markerattrs=(size=3);
run;
```

Output 34.2.13 shows the scatter plot of the scoring data table. The gray area indicates the observations that are identified as outliers, and the black area indicates observations that are identified as inliers. In this example, the modified mean criterion was used to compute the RBF kernel bandwidth parameter value. Comparison between Output 34.2.13 and the training data in Output 34.2.1 indicates that PROC SVDD identified three out of four clusters in the original data. The training data observations in the two clusters that are located in the lower left corner were combined into one cluster. In addition, the description of clusters that is obtained in this example is more circular. This is expected because the RBF bandwidth parameter value of 0.197, which is computed using the modified mean criterion, is higher than the value 0.021, which is

specified in the section "Getting Started: SVDD Procedure" on page 865. Comparing the scoring results in Output 34.2.13 and Output 34.2.7 indicates that the description that is obtained using the modified mean criterion is more circular than the description that is obtained using the mean criterion with a default value of DELTA. Despite these shortcomings, the modified mean criterion provides a reasonable approximation of the training data. The scatter plot of the scoring data in Output 34.2.7 indicates that PROC SVDD was able to correctly identify a majority of the outliers.

Output 34.2.13 shows trace results.

Output 34.2.13 Scoring Results Scatter Plot: Bandwidth Computation Using the Modified Mean Criterion



The use of the trace criterion for bandwidth selection is illustrated next.

The following statements execute the SVDD algorithm on the mylib.Refrigerant data table and produce the results shown in Output 34.2.14 through Output 34.2.18. The SVDD algorithm uses the bandwidth value that is computed using the trace criterion.

```
proc svdd data=mylib.Refrigerant score nbins=10;
id y4 y5;
input y4 y5/level=interval;
kernel rbf / bw=trace;
savestate rstore=mylib.state;
run;
```

The SCORE option in the PROC SVDD statement requests that the training data table be scored. The NBINS=10 option creates a histogram data table that contains 10 bins. The INPUT statement defines the input variables y4 and y5 as interval variables. The KERNEL statement specifies the kernel function as a radial basis function (RBF), and the BW=TRACE option instructs the SVDD procedure to select the value of the RBF bandwidth parameter by using the trace criterion. The SAVESTATE statement creates an analytic store for the model and saves it as a binary object in the data table mylib.state. You can use the analytic store later in the ASTORE procedure to score new data. The "Model Information" table in

Output 34.2.14 summarizes the key options and the input data variables, and it provides the value of the RBF kernel bandwidth parameter, which is computed using the trace criterion to be 0.0132.

Output 34.2.14 Refrigerant Analysis Model Information

Model Information	
Optimization Method	Active Set
Kernel Type	RBF
RBF Kernel Bandwidth	0.0107502582
Bandwidth Selection Method	Trace (nrep = 5)
Bandwidth Relative Scale	0.0851361033
Expected Outlier Fraction	1E-6
Optimization Tolerance	0.0001
Number of Interval Variables	2
Number of Nominal Variables	0

The "Training Results" table in Output 34.2.15 shows the number of support vectors and the R^2 threshold value.

Output 34.2.15 Refrigerant Analysis Training Results

Training Results	
Number of Support Vectors	100
Number of Support Vectors on Boundary	100
Number of Outliers	0
Number of Dropped Observations	0
Threshold R Square Value	0.97464
Constant (C_r) Value	0.02546
Run Time (seconds)	0.01238
Bandwidth Calculation Time (seconds)	0.01661

Output 34.2.16 displays the number of observations in the training data table.

Output 34.2.16 Refrigerant Analysis Training Observations

Number of Observations Read	360
Number of Observations Used	360

The "Optimization Summary" table in Output 34.2.17 shows whether the solution is optimal, the number of iterations that were required, and the objective function value.

Output 34.2.17 Refrigerant Analysis Optimization Summary

Optimization Summary	
Number of Iterations	1
Objective Value	0.0254636933
Infeasibility	0.0000660755
Optimization Status	Optimal
Degenerate	No

The "Histogram" table in Output 34.2.18 shows the frequency table of _SVDDDISTANCE_ values of observations in the mylib.Refrigerant training data table.

Output 34.2.18 Refrigerant Analysis Histogram Table

Frequency Distribution for SVDD Distance			
Bin Midpoint	Count	Percent (%)	Cumulative (%)
0.968225	6	1.67	1.67
0.968900	6	1.67	3.33
0.969575	8	2.22	5.56
0.970249	12	3.33	8.89
0.970924	11	3.06	11.94
0.971599	24	6.67	18.61
0.972274	33	9.17	27.78
0.972949	32	8.89	36.67
0.973624	53	14.72	51.39
0.974299	175	48.61	100.00

The following PROC ASTORE code scores the mylib.scoreds data table by using the score analytic store that was created during training. The mylib.scoreds data table that is used in the following code is the same scoring data used earlier to evaluate the quality of results that were obtained using the SVDD model that was trained with the mean criterion. Using the same data enables you to objectively compare the bandwidth values that are obtained using the mean criterion, the modified mean criterion, and the trace criterion.

```
proc astore;
score data=mylib.scoreds
out=mylib.score_results
rstore=mylib.state;
quit;
```

The following code copies the score_results into the Work library:

```
data work.score_results;
set mylib.score_results;
run;
```

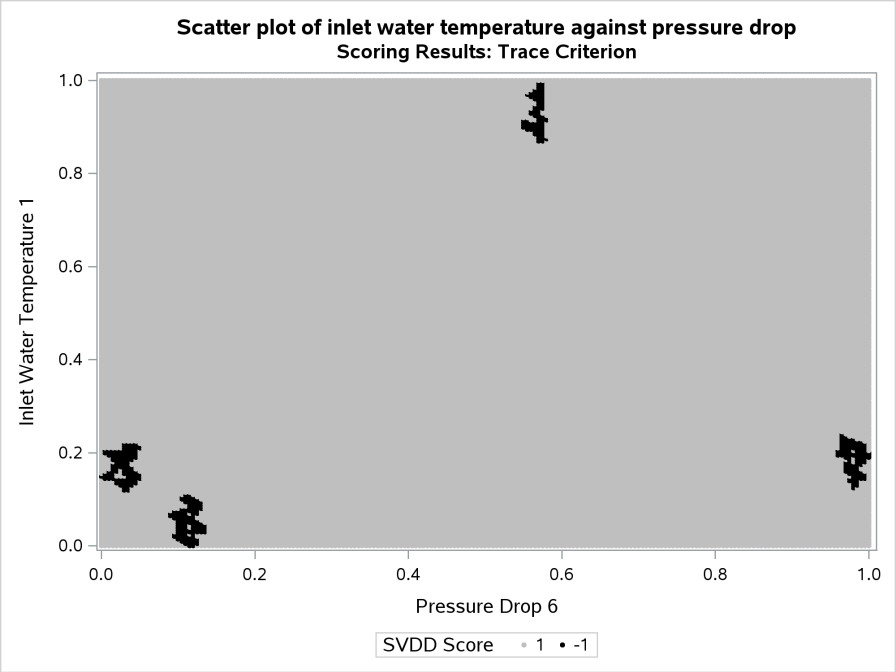
The following PROC SGPLOT code plots the scoring results:

```
proc sgplot data=score_results(where=(_svddscore_ in (-1,1)));
styleattrs datacontrastcolors=(ligr black) datasymbols=(circlefilled);
title "Refrigerant Analysis";
title1 "Scatter plot of inlet water temperature against pressure drop";
title2 "Scoring Results: Trace Criterion";
scatter x=y4 y=y5/group=_svddscore_ markerattrs=(size=3);
run;
```

Output 34.2.13 shows the scatter plot of the scoring data table. The gray area indicates the observations that are identified as outliers, and the black area indicates observations that are identified as inliers. In this example, the trace criterion was used to compute the RBF kernel bandwidth parameter value. Comparison between Output 34.2.19 and the training data in Output 34.2.1 indicates that PROC SVDD identified all four clusters in the original data. The RBF bandwidth parameter value of 0.013, which is computed using the trace

criterion, is close to the value 0.021, which is specified in the section "Getting Started: SVDD Procedure" on page 865. Comparing the scoring results in Output 34.2.13 and Output 34.2.7 with the scoring results in Output 34.2.19 indicates that the description that is obtained using the trace criterion is more accurate than the descriptions that were obtained using the modified mean criterion and mean criterion with a default value of DELTA. Output 34.2.19 shows trace results.

Output 34.2.19 Scoring Results Scatter Plot: Bandwidth Computation Using the Trace Criterion



Example 34.3: Stochastic Subset Solver

The stochastic subset solver for SVDD training provides an efficient way to train an SVDD model and is recommended for training large data sets. For more information, see the section "Stochastic Subset Solver" on page 887. This example compares the active-set solver with the stochastic subset solver in terms of processing time and quality of results.

The following DATA step creates the two_donut data set, which contains 4,000,000 observations and is stored in the Work library by default:

```
%let r_min=2;
%let r_max=5;

   data two_donut;
      call streaminit(24215);
      two_pi = 2 * constant("pi");
      a = &r_min * &r_min;
      b = (&r_max  + &r_min) * ( &r_max - &r_min) ;
      do _n_ = 1 to 2000000;
         u1 = rand("uniform");
```

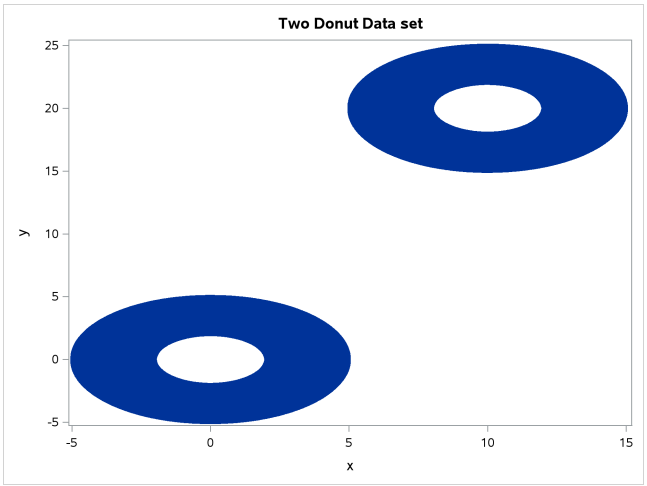
```
            r = sqrt( a + u1 * b );
            u2 = rand("uniform");
            t = two_pi * u2;
            x = r * cos(t); y = r * sin(t);
            output;
            x = x + 10; y = y + 20;
            output;
         end;
         keep x y;
      run;
```

The following statements plot the two_donut data set:

```
ods graphics/maxobs=48000000;
proc sgplot data=two_donut;
   title "Two Donut Data set";
   scatter x=x y=y/markerattrs=(size=3 symbol=circlefilled);
run;
```

Output 34.3.1 shows the results.

Output 34.3.1 Scatter Plot of Two-Donut Data



You can load the Work.two_donut data set into your SAS session by specifying your SAS libref in the first statement in the following DATA step:

```
data mylib.two_donut;
  set two_donut;
run;
```

These statements assume that your SAS libref is named mylib, as in the section "Using SAS Viya Workbench" on page 865, but you can substitute any appropriately defined SAS libref. The following statements execute the SVDD algorithm on the mylib.two_donut data table by using the active-set solver and produce Output 34.3.2 through Output 34.3.5:

```
proc svdd data=mylib.two_donut std;
   id x y;
   input x y/level=interval;
   kernel rbf/bw=MEAN ;
   solver actset/stol=1e-6;
   savestate rstore=mylib.actset;
```

```
run;
```

The STD option in the PROC SVDD statement standardizes the mylib.two_donut data set. The INPUT statement defines the input variables x and y as interval variables. The KERNEL statement specifies the kernel function as a radial basis function (RBF), and the BW=MEAN option requests that the SVDD procedure select the value of the RBF bandwidth parameter by using the mean criterion. The SOLVER statement specifies ACTSET (active-set solver) as the SVDD solver. The SAVESTATE statement creates an analytic store for the model and saves it as a binary object in the mylib.actset data table. You can use the analytic store later in the ASTORE procedure to score new data.

The "Model Information" table in Output 34.3.2 summarizes the key options and the input data variables, and it shows that the value of the RBF kernel bandwidth parameter (which is computed by using the mean criterion) is 0.3055.

Output 34.3.2 Two Donut Model Information: Active-Set Solver

Model Information	
Optimization Method	Active Set
Kernel Type	RBF
RBF Kernel Bandwidth	0.305591793
Bandwidth Selection Method	Mean (delta = 1e-06)
Bandwidth Relative Scale	1
Expected Outlier Fraction	1E-6
Optimization Tolerance	1E-6
Standardization	True
Number of Interval Variables	2
Number of Nominal Variables	0

The "Training Results" table in Output 34.3.3 shows the number of support vectors and the R^2 threshold value.

Output 34.3.3 Two Donut Analysis Training Results: Active-Set Solver

Training Results	
Number of Support Vectors	51
Number of Support Vectors on Boundary	51
Number of Dropped Observations	0
Threshold R Square Value	0.90931
Constant (C_r) Value	0.09069
Run Time (seconds)	11.82589
Bandwidth Calculation Time (seconds)	0.00353

Output 34.3.4 displays the number of observations in the training data set.

Output 34.3.4 Two Donut Training Observations: Active-Set Solver

Number of Observations Read	4000000
Number of Observations Used	4000000

The "Optimization Summary" table in Output 34.3.5 shows whether the solution is optimal, the number of iterations that were required, and the objective function value.

Output 34.3.5 Two Donut Optimization Summary: Active-Set Solver

Optimization Summary	
Number of Iterations	1
Objective Value	0.0906858129
Infeasibility	9.411096E-7
Optimization Status	Optimal
Degenerate	No

The following statements execute the SVDD algorithm on the mylib.two_donut data table by using the STOCHS solver and produce Output 34.3.6 through Output 34.3.9.

```
proc svdd data=mylib.two_donut std;
   id x y;
   input x y/level=interval;
   kernel rbf/bw=MEAN;
   solver stochs/nsamp=10 nmatch=10 rtol=1e-6 ctol=1e-6 stol=1e-6 seed=10;
   savestate rstore=mylib.stochs;
run;
```

The STD option in PROC SVDD statement standardizes the mylib.two_donut data set. The INPUT statement defines the input variables x and y as interval variables. The KERNEL statement specifies the kernel function as a radial basis function (RBF), and the BW=MEAN option requests that the SVDD procedure select the value of the RBF bandwidth parameter by using the mean criterion. The SOLVER statement specifies STOCHS (stochastic subset solver) as the SVDD solver. The SAVESTATE statement creates an analytic store for the model and saves it as a binary object in the mylib.stochs data table. You can use the analytic store later in the ASTORE procedure to score new data.

The "Model Information" table in Output 34.3.6 summarizes the key options and the input data variables, and it shows that the value of the RBF kernel bandwidth parameter (which is computed by using the mean criterion) is 0.3055.

Output 34.3.6 Two Donut Model Information: Stochastic Subset Solver

Model Information	
Optimization Method	Stochastic Subset
Kernel Type	RBF
RBF Kernel Bandwidth	0.305591793
Bandwidth Selection Method	Mean (delta = 1e-06)
Bandwidth Relative Scale	1
Sample Size	10
Seed	10
Optimization Tolerance	1E-6
Standardization	True
Number of Interval Variables	2
Number of Nominal Variables	0

The "Training Results" table in Output 34.3.7 shows the number of support vectors and the R^2 threshold

value.

Output 34.3.7 Two Donut Analysis Training Results: Stochastic Subset Solver

Training Results	
Number of Support Vectors	39
Number of Support Vectors on Boundary	39
Number of Dropped Observations	0
Threshold R Square Value	0.90879
Constant (C_r) Value	0.09121
Run Time (seconds)	0.33995
Bandwidth Calculation Time (seconds)	0.00355

Output 34.3.8 displays the number of observations in the training data set.

Output 34.3.8 Two Donut Training Observations: Stochastic Subset Solver

Number of Observations Read	4000000
Number of Observations Used	4000000

The "Optimization Summary" table in Output 34.3.9 shows whether the solution is optimal, the number of iterations that were required, and the objective function value.

Output 34.3.9 Two Donut Optimization Summary: Stochastic Subset Solver

Optimization Summary	
Number of Iterations	200
Objective Value	0.0912108656
Infeasibility	1.584388E-11
Optimization Status	Maximum number of iterations reached.
Degenerate	No

The run time, the threshold R^2 value, and the number of support vectors for the active-set (ACTSET) solver and the stochastic subset (STOCHS) solver are provided in "Model Information" table in Output 34.3.2 and Output 34.3.6. Comparison of the run time indicates that in this example, the stochastic subset solver performance is orders of magnitude faster than the active-set solver. The threshold R^2 values for the two solvers are close. The number of support vectors produced by the stochastic subset solver are less than the number produced by the active-set solver.

To compare the quality of the training results obtained using the ACTSET and STOCHS solvers, scoring is performed on a 200×200 data grid. The following DATA step creates a data set named scoreds to contain the scoring observations. Note that the scale of the observations in the scoreds scoring data set matches the scale of the data in the original, nonstandardized two_donut data set. The analytic stores mylib.actset and mylib.stochs contain the code necessary to standardize the scoring data set on the same scale as the training data set.

```
proc sql noprint;
   select min(x), min(y), max(x), max(y)
   into :minx, :miny, :maxx, :maxy
   from  two_donut;
```

```
quit;

   data scoreds;
      do i=&minx to &maxx by %sysevalf((&maxx-&minx)/200);
         do  j=&miny to &maxy by %sysevalf((&maxy-&miny)/200);
            output;
         end;
      end;
      rename i=x j=y;
   run;
```

You can load the scoreds data set into your SAS session by specifying your SAS libref in the first statement in the following DATA step:

```
   data mylib.scoreds;
      set scoreds;
   run;
```

The following code scores the mylib.scoreds using the score analytic store that was created using the ACTSET solver during training.

```
proc astore;
   score data=mylib.scoreds
   out=mylib.scoreact
   rstore=mylib.actset;
run;
```

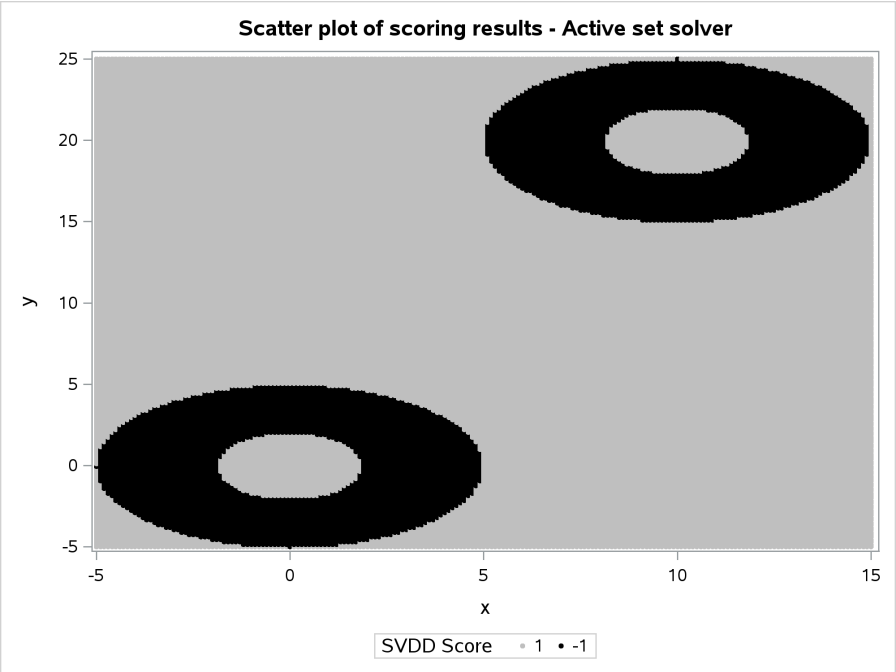
The following code copies the mylib.scoreact data table into the Work library:

```
   data work.scoreact;
      set mylib.scoreact;
   run;
```

The following code plots the scoring results, as shown in Output 34.3.10:

```
proc sgplot data=scoreact(where=(_svddscore_ in (-1,1)));
   styleattrs datacontrastcolors=(ligr black) datasymbols=(circlefilled);
   title "Two Donut Data set";
   title1 "Scatter plot of scoring results - Active set solver";
   scatter x=x y=y/group=_svddscore_ markerattrs=(size=3);
run;
```

Output 34.3.10 Scoring Results Scatter Plot: Active-Set Solver



The following code scores the mylib.scoreds data table by using the score analytic store that was created using the STOCHS solver during training:

```
proc astore;
   score data=mylib.scoreds
   out=mylib.scorestochs
   rstore=mylib.stochs;
run;
```

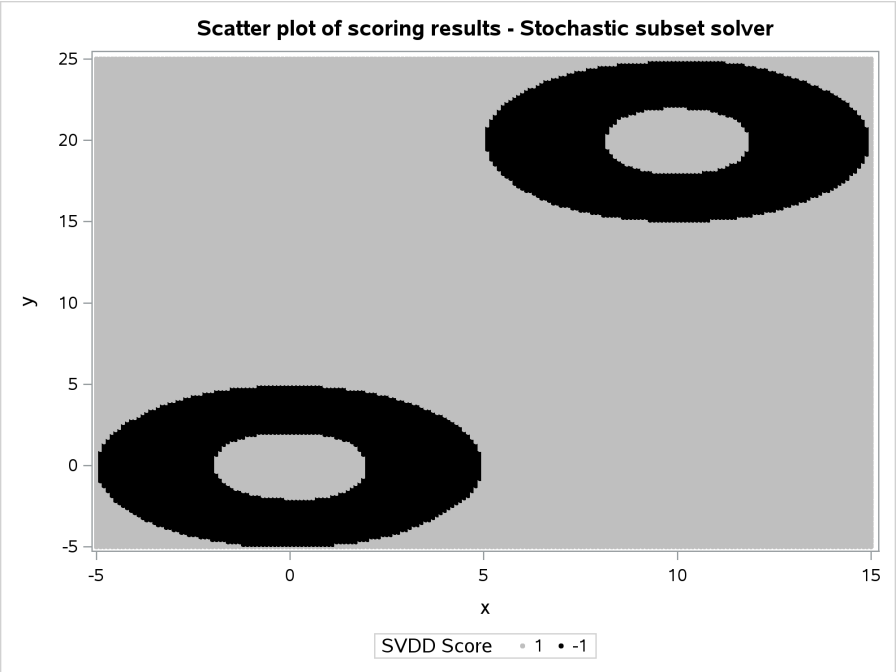
The following code copies the scorestochs data table into the Work library:

```
   data Work.scorestochs;
     set mylib.scorestochs;
   run;
```

The following code plots the scoring results:

```
proc sgplot data=scorestochs(where=(_svddscore_ in (-1,1)));
   styleattrs datacontrastcolors=(ligr black) datasymbols=(circlefilled);
   title "Two Donut Data set";
   title1 "Scatter plot of scoring results - Stochastic subset solver";
   scatter x=x y=y/group=_svddscore_ markerattrs=(size=3);
run;
```

Output 34.3.11 Scoring Results Scatter Plot: Stochastic Subset Solver



Output 34.3.10 and Output 34.3.11 show the scatter plots of the scoring data set that were obtained using the active-set (ACTSET) and the stochastic subset (STOCHS) solvers, respectively. The gray area indicates the observations that are identified as outliers, and the black area indicates observations that are identified as inliers. Comparison between Output 34.3.10 and Output 34.3.11 indicates that quality of the SVDD training that was obtained using active-set (ACTSET) and the stochastic subset (STOCHS) solvers is nearly identical.

In this example, the stochastic subset solver provides an order of magnitude faster performance compared to the active-set solver. The run-time advantage of the stochastic subset solver depends on the number of variables in the training data set. When a training data set has a large number of variables, the active-set solver might be faster than the stochastic subset solver.

Example 34.4: Analysis of Tennessee Eastman Process Data Using Multiple Bandwidth Values and Consensus-Based Classification

The Tennessee Eastman (TE) process is a realistic model of a chemical process that is widely used in the academic community to study multivariate process control problems. The TE process is based on an actual industrial process, with some modifications to protect the proprietary nature of the process (Downs and Vogel 1993). MATLAB simulation code that was provided by Ricker (2002) was used to generate the TE process data. Data were generated for the normal operations of the process and 20 different faulty conditions. Each observation consists of 41 variables, x1–x41; 22 variables are measured continuously (every 6 seconds on average), and 19 variables are sampled at a specified interval of either 0.1 or 0.25 hours. The variable x42 indicates the operating mode: 0 represents the normal conditions, and 1 to 20 represent the condition when the corresponding fault was present. Table 34.2 shows the total number of observations for normal operation and for each fault condition.

This example analyzes the TE process data by using multiple bandwidth values and demonstrates consensus-based classification of the scoring results.

Table 34.2 The Tennessee Eastman Data

Operation Mode	# Obs	Operation Mode	# Obs
Normal	900	Fault 11	372
Fault 1	294	Fault 12	427
Fault 2	1,007	Fault 13	465
Fault 3	653	Fault 14	1,729
Fault 4	807	Fault 15	1,729
Fault 5	603	Fault 16	1,124
Fault 6	280	Fault 17	1,953
Fault 7	217	Fault 18	1,741
Fault 8	759	Fault 19	1,729
Fault 9	1,465	Fault 20	1,473
Fault 10	1,362		

The following DATA step creates the TE data set (which is stored in the Work library by default):

```
data te;
input x1-x42;
datalines;

... more lines ...
```

The following code creates the training and scoring data sets. The ID variable is created to uniquely identify an observation. The training data set consists of 800 observations that belong to normal operations. The scoring data set contains 100 observations that belong to normal operations and 20,189 observations when the fault conditions 1 to 20 were present.

```
data TE;
    set TE;
    id=_n_;
run;

data train score;
set TE;
    if id le 800 then output train;
    else if id gt 800 then output score;
run;
```

You can load the Work.train and Work.score data sets into your SAS session by specifying your SAS libref in the first statement in the following DATA step:

```
data mylib.train;
set train;
run;
```

```
data mylib.score;
set score;
run;
```

These statements assume that your SAS libref is named mylib, as in the section "Using SAS Viya Workbench" on page 865, but you can substitute any appropriately defined SAS libref.

The following statements execute the SVDD algorithm on the mylib.train data table and produce the results in Output 34.4.1 through Output 34.4.4:

```
proc svdd data=mylib.train score nbins=10;
id id x42;
input x1-x41/level=interval;
kernel rbf / bw=(5 to 20 by 1);
savestate rstore=mylib.scorecat;
run;
```

The SCORE option in the PROC SVDD statement requests that the training data table be scored. The NBINS=10 option creates a histogram data table that contains 10 bins. The INPUT statement defines the input variables x1 through x41 as interval variables. The KERNEL statement specifies the kernel function as a radial basis function (RBF), and the BW=(5 to 20 by 1) option instructs the SVDD procedure to train 16 SVDD models that correspond to bandwidth values from 5 to 20. The SAVESTATE statement creates an analytic store for the model and saves it as a binary object in the mylib.scorecat data table. You can use the analytic store later in the ASTORE procedure to score new data.

Output 34.4.1 displays the number of observations in the training data table.

Output 34.4.1 TE Process Analysis Training Observations

Number of Observations Read	800
Number of Observations Used	800

Because the SVDD procedure is executed with 16 bandwidth values, the "Model Information," "Training Results," "Optimization Summary," and "Histogram" tables are generated for each bandwidth value. For the sake of brevity, only tables that correspond to two bandwidth values are provided here.

The "Model Information" table in Output 34.4.2 summarizes the key options and the input data variables for the SVDD model that is trained using the option BW=5.

Output 34.4.2 TE Process Analysis Model Information (MODELID=1, BW=5)

Model Information	
Optimization Method	Active Set
Kernel Type	RBF
RBF Kernel Bandwidth	5
Bandwidth Selection Method	User specified
Bandwidth Relative Scale	0.7242664168
Expected Outlier Fraction	1E-6
Optimization Tolerance	0.0001
Number of Interval Variables	41
Number of Nominal Variables	0

The "Training Results" table in Output 34.4.3 shows the number of support vectors and the R^2 threshold value for the SVDD model that is trained using the option BW=5.

Output 34.4.3 TE Process Analysis Training Results (MODELID=1, BW=5)

Training Results	
Number of Support Vectors	327
Number of Support Vectors on Boundary	327
Number of Outliers	7
Number of Dropped Observations	0
Threshold R Square Value	0.98803
Constant (C_r) Value	0.01208
Run Time (seconds)	0.05276

The "Optimization Summary" table in Output 34.4.4 shows whether the solution is optimal, the number of iterations that were required, and the objective function value for the SVDD model that is trained using the option BW=5.

Output 34.4.4 TE Process Analysis Optimization Summary (MODELID=1, BW=5)

Optimization Summary	
Number of Iterations	1
Objective Value	0.0120808289
Infeasibility	0.0000885828
Optimization Status	Optimal
Degenerate	No

The "Histogram" table in Output 34.4.5 shows whether the solution is optimal, the number of iterations that were required, and the objective function value for the SVDD model that is trained using the option BW=5.

Output 34.4.5 TE Process Analysis Histogram (MODELID=1, BW=5)

Frequency Distribution for SVDD Distance			
Bin Midpoint	**Count**	**Percent (%)**	**Cumulative (%)**
0.983452	4	0.50	0.50
0.983946	8	1.00	1.50
0.984440	10	1.25	2.75
0.984934	39	4.88	7.63
0.985428	43	5.38	13.00
0.985922	63	7.88	20.88
0.986417	70	8.75	29.63
0.986911	87	10.88	40.50
0.987405	98	12.25	52.75
0.987899	378	47.25	100.00

The "Model Information" table in Output 34.4.6 summarizes the key options and the input data variables for the SVDD model that is trained using the option BW=20.

Output 34.4.6 TE Process Analysis Model Information (MODELID=16, BW=20)

Model Information	
Optimization Method	Active Set
Kernel Type	RBF
RBF Kernel Bandwidth	20
Bandwidth Selection Method	User specified
Bandwidth Relative Scale	2.8970656673
Expected Outlier Fraction	1E-6
Optimization Tolerance	0.0001
Number of Interval Variables	41
Number of Nominal Variables	0

The "Training Results" table in Output 34.4.7 shows the number of support vectors and the R^2 threshold value for the SVDD model that is trained using the option BW=20.

Output 34.4.7 TE Process Analysis Training Results (MODELID=16, BW=20)

Training Results	
Number of Support Vectors	36
Number of Support Vectors on Boundary	36
Number of Outliers	1
Number of Dropped Observations	0
Threshold R Square Value	0.88994
Constant (C_r) Value	0.11006
Run Time (seconds)	0.00932

The "Optimization Summary" table in Output 34.4.8 shows whether the solution is optimal, the number of iterations that were required, and the objective function value for the SVDD model that is trained using the option BW=5.

Output 34.4.8 TE Process Analysis Optimization Summary (MODELID=16, BW=20)

Optimization Summary	
Number of Iterations	1
Objective Value	0.1100559877
Infeasibility	3.472149E-6
Optimization Status	Optimal
Degenerate	No

The "Histogram" table in Output 34.4.9 shows whether the solution is optimal, the number of iterations that were required, and the objective function value for the SVDD model that is trained using the option BW=5.

Output 34.4.9 TE Process Analysis Histogram (MODELID=16, BW=20)

Frequency Distribution for SVDD Distance			
Bin Midpoint	Count	Percent (%)	Cumulative (%)
0.869560	2	0.25	0.25
0.871706	2	0.25	0.50
0.873853	7	0.88	1.38
0.875999	11	1.38	2.75
0.878145	27	3.38	6.13
0.880292	18	2.25	8.38
0.882438	40	5.00	13.38
0.884585	153	19.13	32.50
0.886731	305	38.13	70.63
0.888878	235	29.38	100.00

The following PROC ASTORE code scores the mylib.score data table by using the score analytic store that was created during training:

```
proc astore;
score data=mylib.score
out=mylib.score_results
rstore=mylib.scorecat;
quit;
```

The following code copies the score_results into the Work library:

```
data work.score_results;
set mylib.score_results;
run;
```

The score_results table has the following columns:

- Variables ID and X42, which are specified using the COPYVARS option
- Variables BW_i, where i = 1 to 16. These indicate the bandwidth values that the SVDD model uses.
- Variables _SVDDDISTANCE_i, where i = 1 to 16. These indicate the value of _SVDDDISTANCE_ that are obtained by scoring against the *i*th model.
- Variables _SVDDSCORE_i, where i = 1 to 16. These indicate the value of _SVDDSCORE_ that is obtained by scoring against the *i*th model.

The SVDDScore values in the score_results data table, which correspond to different models, can be used to develop a consensus-based strategy to determine the degree to which observations in the scoring data table are outliers or inliers. The proportion of models that designate an observation as an outlier or inlier can be used to determine this degree.

The following DATA step reads the score_results data table and computes the proportion of models that classify an observation as an outlier or an inlier. This DATA step creates two data sets: score_results_normal, which contains observations that belong to normal operating mode, and score_results_fault, which contains observations that belonging to faulty operating mode.

```
data score_results_normal score_results_fault;
set score_results;
label p_inside="Proportion Inside" p_outside="Proportion Outside";
array SVDDSCORE{16} _SVDDSCORE_1  - _SVDDSCORE_16 ;
inside=0;
outside=0;
do j=1 to 16;
   if SVDDSCORE[j]=-1 then inside=inside+1;
   if SVDDSCORE[j]=1  then outside=outside+1;
end;
p_inside=inside/16;
p_outside=outside/16;
if x42=0 then output score_results_normal;
else output score_results_fault;
run;
```

The following PROC SORT code sorts the score_results_normal data in ascending order of the p_inside column. Observations whose p_inside values are higher can be treated as more likely to be inliers than observations whose p_inside values are lower.

```
proc sort data=score_results_normal;
by descending p_inside;
run;
```

The following PROC SORT code sorts the score_results_fault data in ascending order of the p_outside column. Observations whose p_outside values are higher can be treated as more likely to be outliers than observations whose p_outside values are lower.

```
proc sort data=score_results_fault;
by descending p_outside;
run;
```

This example illustrates how to use multiple bandwidth values and a consensus-based methodology to classify observations in the scoring data table. In such analysis, the bandwidth values can be selected in the neighborhood of the bandwidth value that is obtained using either the mean criterion or the modified mean criterion.

Example 34.5: Bandwidth Calculation Using the Peak Criterion Method

This example illustrates the peak criterion method (Kakde et al. 2017) for bandwidth selection. The peak criterion method suggests the values of Gaussian bandwidth s, where the second derivative of the value of the optimal dual objective function (as defined in the section "Flexible Data Description" on page 879) with respect to s first reaches 0. The bandwidth value that is obtained using the peak criterion method provides a data boundary that closely follows the data shape. For certain data sets, the method provides a range of values where this criterion is met. Any value of s within this range provides a good data boundary.

The peak criterion method involves computing the optimal dual objective function value over a range of bandwidth values. These computations can be performed by specifying multiple bandwidth parameter values in the form of BW=*n* TO *m* BY *p* in the KERNEL statement in order to train multiple SVDD models in a single run.

The training data set in this example is star-shaped. This example first computes the bandwidth value by using the peak criterion method. The bandwidth value thus obtained is subsequently used to train an SVDD model. To evaluate the quality of the training results, scoring is performed on a 200 × 200 data grid.

The following DATA step creates the star (which is stored in the Work library by default):

```
data star;
infile datalines delimiter=',';
input x1 x2 ;
datalines;
0.07197,6.48334
0.28467,6.35686
0.45698,6.35741
0.57731,6.13851

   ... more lines ...
```

The following statements plot variable x1 against variable x2:

```
proc sgplot data=star;
title "Star-shaped Training Data";
scatter x=x1 y=x2/markerattrs=(size=3 symbol=circlefilled);
run;
```

Output 34.5.1 shows the results.

Output 34.5.1 Star-Shaped Training Data

You can load the work.star data set into your SAS session by specifying your SAS libref in the following DATA step:

```
data mylib.star;
set star;
run;
```

These statements assume that your SAS libref is named mylib, as in the section "Using SAS Viya Workbench" on page 865, but you can substitute any appropriately defined SAS libref.

The following statements execute the SVDD algorithm on the mylib.star data table. The optSummary and modelInfo ODS data sets are stored the in optimization_summary and model_info data sets, respectively.

```
ods output optSummary=optimization_summary modelinfo=model_info;
proc svdd data=mylib.star;
id x1 x2;
input x1-x2/level=interval;
kernel rbf / bw=(0.1 to 5 by 0.1);
solver actset/stol=1e-10;
savestate rstore=mylib.scorecat;
run;
```

The INPUT statement defines the input variables x1 and x2 as interval variables. The KERNEL statement specifies the kernel function as a radial basis function (RBF), and the BW=(0.1 to 5 by 0.1) option instructs the SVDD procedure to train 50 SVDD models that correspond to bandwidth values from 0.1 to 5. The SAVESTATE statement creates an analytic store for the model and saves it as a binary object in the mylib.scorecat data table.

Output 34.5.2 displays the number of observations in the training data table.

Output 34.5.2 Peak Criterion Method: Star-Shaped Data

Number of Observations Read	582
Number of Observations Used	582

The SVDD procedure is executed with 50 bandwidth values, and it generates the "Model Information," "Training Results," "Optimization Summary," and "Histogram" tables for each bandwidth value. For the sake of brevity, only tables that correspond to one bandwidth value are provided here.

The "Model Information" table in Output 34.5.3 summarizes the key options and the input data variables for the SVDD model that is trained using the option BW=0.1.

Output 34.5.3 Peak Criterion Method: Star-Shaped Data (MODELID=1, BW=0.1)

Model Information	
Optimization Method	Active Set
Kernel Type	RBF
RBF Kernel Bandwidth	0.1
Bandwidth Selection Method	User specified
Bandwidth Relative Scale	0.1643794954
Expected Outlier Fraction	1E-6
Optimization Tolerance	1E-10
Number of Interval Variables	2
Number of Nominal Variables	0

The "Training Results" table in Output 34.5.4 shows the number of support vectors and the R^2 threshold value for the SVDD model that is trained using the option BW=0.1.

Output 34.5.4 Peak Criterion Method: Star-Shaped Data (MODELID=1, BW=0.1)

Training Results	
Number of Support Vectors	582
Number of Support Vectors on Boundary	582
Number of Dropped Observations	0
Threshold R Square Value	0.99758
Constant (C_r) Value	0.00242
Run Time (seconds)	0.14306

The "Optimization Summary" table in Output 34.5.5 shows whether the solution is optimal, the number of iterations that were required, and the objective function value for the SVDD model that is trained using the option BW=0.1.

Output 34.5.5 Peak Criterion Method: Star-Shaped Data (MODELID=1, BW=0.1)

Optimization Summary	
Number of Iterations	1
Objective Value	0.002415936
Infeasibility	7.771561E-16
Optimization Status	Optimal
Degenerate	No

The following code uses the optimization_summary and model_info data sets to obtain the optimal objective function and bandwidth values, respectively, for each model. Subsequently, these two data sets are merged to create one data set named peak_input.

```
data model_info(keep=model nvalue rename=(nvalue=bw));
set model_info (where=(rowid="BW"));
run;
data optimization_summary(keep=model value rename=(value=obj));
set optimization_summary (where=(rowid="OBJ"));
run;
proc sort data=model_info; by model; run;
proc sort data=optimization_summary; by model; run;
data peak_input;
merge model_info(in=a) optimization_summary(in=b);
by model;
if a=b;
run;
```

The following code computes the first and second derivative of the optimal objective function value with respect to the bandwidth value s. The sby macro variable specifies the bandwidth step size of 0.1, as used earlier in the SVDD procedure.

```
%let sby=0.1;
data peak_input;
set peak_input;
lag_obj=lag(obj);
lag2_obj=lag2(obj);
run;
```

```
data peak_input;
set peak_input;
if _n_ > 1 then d_obj = (obj - lag_obj)/(&sby);
if _n_ > 2 then d2_obj = (obj - 2 * lag_obj + lag2_obj) / ( &sby * &sby ) ;
run;
```

To decide whether the value of the second derivative is 0, a penalized B-spline is fitted to the second derivative by using the TRANSREG procedure in SAS/STAT software. If the 95% confidence interval of the fitted value of second derivative contains 0, the second derivative value is considered to be approximately 0.

```
proc transreg data=peak_input;
model identity(d2_obj) = pbspline(bw / nknots=20 evenly sbc);
output out=peak_res predicted cli clm;
run;
```

The following code determines the bandwidth value where the second derivative of the optimal objective function value with respect to *s* first reaches 0:

```
   data peak_res;
   set peak_res(where=(d2_obj ne .));
   if ((CIUd2_obj ge 0) and (CILd2_obj le 0)) then zero_flg=1;
   else zero_flg=0;
   label d2_obj = "Second Derivative";
   run;


proc sql noprint;
select min(bw) into :peak_bw
from peak_res
where zero_flg=1;
quit;

   data peak_res;
   set peak_res;
   if round(bw,0.0001)= &peak_bw then do;
   ss=bw;
   label="Bandwidth="||trim(left(bw));
   end;
   run;
```

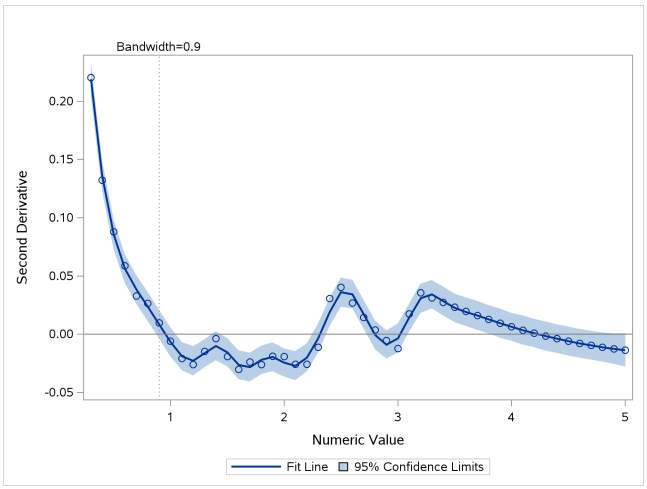
The following code plots the second derivative of the optimal objective function value against bandwidth value *s*:

```
proc template;
define statgraph bandplot;
begingraph;
entrytitle " ";
layout overlay;
bandplot x=bw limitupper=CIUd2_obj
limitlower=CILd2_obj /
name="band1" modelname="fit"
legendlabel="95% Confidence Limits";
scatterplot x=bw y=d2_obj / primary=true;
seriesplot x=bw y=Pd2_obj / name="fit"
legendlabel="Fit Line" lineattrs=(thickness=2);;
```

```
referenceline x=ss / curvelabel=label
lineattrs=(color=gray pattern=dot thickness=1);
referenceline y=0 /lineattrs=(color=gray pattern=solid );
discretelegend  "fit" "band1" ;
endlayout;
endgraph;
end;
run;

proc sgrender data=peak_res(where=(d2_obj ne .)) template=bandplot;
run;
```

Output 34.5.6 shows the results, which indicate that the value of second derivative of the optimal objective function value with respect to s first reaches 0 at $s = 0.9$. This represents the bandwidth value that is obtained using the peak criterion method.

Output 34.5.6 Peak Criterion: Plot of the 2nd Derivative



The following statements execute the SVDD algorithm on the mylib.star data table by using the bandwidth value (0.9) that was obtained using the peak criterion method.

```
proc svdd data=mylib.star;
id x1 x2;
input x1 x2/level=interval;
kernel rbf / bw=0.9;
savestate rstore=mylib.state;
run;
```

To evaluate the quality of the training results, scoring is performed on a 200×200 data grid. The following DATA step creates a data set named scoreds to contain the scoring observations:

```
data scoreds;
do x1=0 to 10 by 0.05;
   do  x2=0 to 10 by 0.05;
      output;
   end;
end;
run;
```

You can load the scoreds data set into your SAS session by specifying your SAS libref in the following DATA step:

```
data mylib.scoreds;
set scoreds;
run;
```

The following code scores the mylib.scoreds data by using the score analytic store that was created during training:

```
proc astore;
score data=mylib.scoreds
out=mylib.score_results
rstore=mylib.state;
quit;
```

The following code copies the score_results into the Work library:

```
data work.score_results;
set mylib.score_results;
run;
```

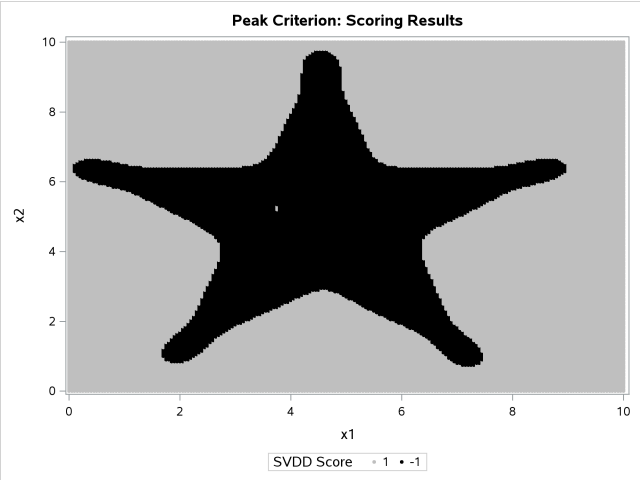
The following code plots the scoring results:

```
proc sgplot data=score_results(where=(_svddscore_ in (-1,1)));
styleattrs datacontrastcolors=(ligr black) datasymbols=(circlefilled);
title "Peak Criterion: Scoring Results";
scatter x=x1 y=x2/group=_svddscore_ markerattrs=(size=3);
run;
```

Output 34.5.7 shows the scatter plot of the scoring data set. The gray area indicates the observations that are identified as outliers, and the black area indicates observations that are identified as inliers. Comparison between Output 34.5.7 and the training data in Output 34.5.1 indicates that the bandwidth that was obtained using the peak criterion method provides an accurate description of the training data.

As outlined in this example, determining the bandwidth that is suggested by the peak criterion method requires that the SVDD solution be computed multiple times for the training data for a list of bandwidth values that lie on a grid. The parameters of the grid such as the starting value, ending value, and increment can affect the quality of results. Also, it is necessary to set the solver tolerance, STOL, to a very low value (such as the value 1E–10 that is used in this example).

Output 34.5.7 Peak Criterion: Scoring Results



References

Banerjee, A., Burlina, P., and Diehl, C. (2006). "A Support Vector Method for Anomaly Detection in Hyperspectral Imagery." *IEEE Transactions on Geoscience and Remote Sensing* 44:2282–2291.

Banerjee, A., Burlina, P., and Meth, R. (2007). "Fast Hyperspectral Anomaly Detection via SVDD." In *Proceedings of the 2007 IEEE International Conference on Image Processing*, vol. 4, 101–104. Piscataway, NJ: IEEE.

Benkedjouh, T., Medjaher, K., Zerhouni, N., and Rechak, S. (2012). "Fault Prognostic of Bearings by Using Support Vector Data Description." In *IEEE Conference on Prognostics and Health Management (PHM)*, 1–7. Piscataway, NJ: Institute of Electrical and Electronics Engineers.

Chaudhuri, A., Kakde, D., Jahja, M., Xiao, W., Jiang, H., Kong, S., and Peredriy, S. (2018). "Sampling Method for Fast Training of Support Vector Data Description." In *Proceedings of the 2018 Annual Reliability and Maintainability Symposium (RAMS)*. Piscataway, NJ: Institute of Electrical and Electronics Engineers.

Chaudhuri, A., Kakde, D., Sadek, C., Gonzalez, L., and Kong, S. (2017). "The Mean and Median Criterion for Automatic Kernel Bandwidth Selection for Support Vector Data Description." In *2017 IEEE 17th International Conference on Data Mining Workshops (ICDMW)*. Piscataway, NJ: Institute of Electrical and Electronics Engineers.

Chaudhuri, A., Sadek, C., Kakde, D., Wang, H., Hu, W., Jiang, H., Kong, S., et al. (2021). "The Trace Kernel Bandwidth Criterion for Support Vector Data Description." *Pattern Recognition* 111.

Downs, J. J., and Vogel, E. F. (1993). "A Plant-Wide Industrial Process Control Problem." *Computers and Chemical Engineering* 17:245–255.

Heckert, N. A., and Filliben, J. J. (2000). "Dataplot." Accessed January 17, 2017. http://www.itl.nist.gov/div898/software/dataplot/.

Jeong, M.-K., An, S.-H., and Nam, K. (2016). "SVDD-Based Financial Fraud Detection Method through Respective Learnings of Normal/Abnormal Behaviors." *International Journal of Security and Its Applications* 10:429–438.

Jiang, H., Wang, H., Hu, W., Kakde, D., and Chaudhuri, A. (2019). "Fast Incremental SVDD Learning Algorithm with the Gaussian Kernel." In *Thirty-Third AAAI Conference on Artificial Intelligence*. Palo Alto, CA: AAAI Press.

Juszczak, P., Adams, N. M., Hand, D. J., Whitrow, C., and Weston, D. J. (2008). "Off-the-Peg and Bespoke Classifiers for Fraud Detection." *Computational Statistics and Data Analysis* 52:4521–4532.

Kakde, D., Chaudhuri, A., Kong, S., Jahja, M., Jiang, H., and Silva, J. (2017). "Peak Criterion for Choosing Gaussian Kernel Bandwidth in Support Vector Data Description." In *2017 IEEE International Conference on Prognostics and Health Management (ICPHM) (PHM2017)*. Piscataway, NJ: Institute of Electrical and Electronics Engineers.

Kakde, D., Peredriy, S., and Chaudhuri, A. (2017). "A Non-parametric Control Chart for High Frequency Multivariate Data." In *Proceedings of the 2017 Annual Reliability and Maintainability Symposium (RAMS)*. Piscataway, NJ: Institute of Electrical and Electronics Engineers.

Kang, I., Jeong, M. K., and Kong, D. (2012). "A Differentiated One-Class Classification Method with Applications to Intrusion Detection." *Expert Systems with Applications* 39:3899–3905.

Liao, Y., Kakde, D., Chaudhuri, A., Jiang, H., Sadek, C., and Kong, S. (2018). "A New Bandwidth Selection Criterion for Using SVDD to Analyze Hyperspectral Data." In *Proceedings of the 2018 Defense and Security Symposium*. Bellingham, WA: SPIE.

Peredriy, S., Kakde, D., and Chaudhuri, A. (2017). "Kernel Bandwidth Selection for SVDD: The Sampling Peak Criterion Method for Large Data." In *Proceedings of the 2017 Big Data Conference*, 3540–3549. Piscataway, NJ: Institute of Electrical and Electronics Engineers.

Ricker, N. L. (2002). "Tennessee Eastman Challenge Archive, MATLAB 7.x Code." University of Washington, Seattle, Department of Chemical Engineering. Accessed March 21, 2016. http://depts.washington.edu/control/LARRY/TE/download.html.

Sakla, W., Chan, A., Ji, J., and Sakla, A. (2011). "An SVDD-Based Algorithm for Target Detection in Hyperspectral Imagery." *IEEE Geoscience and Remote Sensing Letters* 8:384–388.

Saxena, A., and Goebel, K. (2008). "Turbofan Engine Degradation Simulation Data Set." Accessed January 17, 2017. NASA Ames Prognostics Data Repository. NASA Ames Research Center, Moffett Field, CA. https://www.nasa.gov/intelligent-systems-division/discovery-and-systems-health/pcoe/pcoe-data-set-repository/.

Saxena, A., Goebel, K., Simon, D., and Eklund, N. (2008). "Damage Propagation Modeling for Aircraft Engine Run-to-Failure Simulation." In *Proceedings of the International Conference on Prognostics and Health Management, 2008*, 1–9. Piscataway, NJ: IEEE.

Schölkopf, B., Williamson, R. C., Smola, A. J., Shawe-Taylor, J., and Platt, J. C. (1999). "Support Vector Method for Novelty Detection." In *Proceedings of the Twelfth International Conference on Neural Information Processing*, 582–588. Cambridge, MA: MIT Press.

Shin, J. H., Lee, B., and Park, K. S. (2011). "Detection of Abnormal Living Patterns for Elderly Living Alone Using Support Vector Data Description." *IEEE Transactions on Information Technology in Biomedicine* 15:438–448.

Sukchotrat, T., Kim, S. B., and Tsung, F. (2009). "One-Class Classification-Based Control Charts for Multivariate Process Monitoring." *IIE Transactions* 42:107–120.

Sun, R., and Tsung, F. (2003). "A Kernel-Distance-Based Multivariate Control Chart Using Support Vector Methods." *International Journal of Production Research* 41:2975–2989.

Tax, D. M. J., and Duin, R. P. W. (2004). "Support Vector Data Description." *Machine Learning* 54:45–66.

Tax, D. M. J., Ypma, A., and Duin, R. P. W. (1999a). "Pump Failure Detection Using Support Vector Data Descriptions." In *Advances in Intelligent Data Analysis: Third International Symposium, IDA-99*, edited by D. J. Hand, J. N. Kok, and M. R. Berthold, 415–426. Berlin: Springer-Verlag.

Tax, D. M. J., Ypma, A., and Duin, R. P. W. (1999b). "Support Vector Data Description Applied to Machine Vibration Analysis." In *Proceedings of the Fifth Annual Conference of the Advanced School for Computing and Imaging*, 398–405. Delft, Netherlands: ASCI.

Xiao, Y., Wang, H., Zhang, L., and Xu, W. (2014). "Two Methods of Selecting Gaussian Kernel Parameters for One-Class SVM and Their Application to Fault Detection." *Knowledge-Based Systems* 59:75–84.

Yang, J., Wang, S., Chen, N., Chen, X., and Shi, P. (2010). "Wearable Accelerometer Based Extendable Activity Recognition System." In *Proceedings of the 2010 IEEE International Conference on Robotics and Automation (ICRA)*, 3641–3647. Piscataway, NJ: IEEE.

Chapter 35
SVMACHINE Procedure

Contents

Overview: SVMACHINE Procedure **932**
PROC SVMACHINE Features 932
Using SAS Viya Workbench 933
Getting Started: SVMACHINE Procedure **933**
Syntax: SVMACHINE Procedure **935**
PROC SVMACHINE Statement 935
AUTOTUNE Statement 938
CODE Statement 945
CROSSVALIDATION Statement 945
FREQ Statement 945
ID Statement 946
INPUT Statement 946
KERNEL Statement 946
MITIGATEBIAS Statement 948
OUTPUT Statement 950
PARTITION Statement 951
SAVESTATE Statement 951
TARGET Statement 952
WEIGHT Statement 952
Details: SVMACHINE Procedure **953**
Support Vector Classification 953
Support Vector Regression 954
Interior Point Optimization Methods 954
Coordinate Descent Optimization Method 955
Active-Set Optimization Method 956
Reduced SVM Optimization Method 957
Scoring Process 957
k-fold Cross Validation 958
Bias Mitigation 958
Displayed Output 960
ODS Table Names 962
ODS Graphics 964
Examples: SVMACHINE Procedure **965**
Example 35.1: Home Equity Loan Case 965
Example 35.2: Support Vector Regression 968
Example 35.3: Coordinate Descent Method 969

Example 35.4: Active-Set Method 971
Example 35.5: Reduced SVM Method 972
Example 35.6: Mitigating Bias of a Support Vector Classification 974
Example 35.7: Tuning the Hyperparameters of a Support Vector Machine 978
References **980**

Overview: SVMACHINE Procedure

The SVMACHINE procedure implements support vector machine (SVM) algorithms in SAS Viya Workbench. The SVM algorithms include support vector classification (SVC) and support vector regression (SVR). As a popular data mining model, SVM algorithms compute support vector machine learning classifiers for the binary pattern recognition problem and regression problem. They have been broadly used in fields such as image classification, handwriting recognition, financial decision, text mining, and so on.

Like other predictive modeling tools, PROC SVMACHINE uses input data to train a model and provides information about the model. The procedure executes an SVM algorithm (by applying the interior point optimization method, the coordinate descent optimization method, or the active-set optimization method during training) and can generate a SAS analytic store for scoring future data. PROC SVMACHINE uses both linear and nonlinear kernels to do computations in SAS Viya Workbench.

PROC SVMACHINE Features

The SVMACHINE procedure has the following features:

- supports large-scale training data
- supports both continuous and categorical inputs
- supports classification of a binary target
- supports regression of an interval target
- supports the interior point optimization method
- supports the coordinate descent optimization method
- supports the active-set optimization method
- supports the reduced support vector machine (RSVM) optimization method
- supports the analytic store model
- supports the weighted support vector machine algorithm
- can train the model while minimizing the bias metric that you specify

Using SAS Viya Workbench

SAS Viya Workbench has greatly simplified the SAS programming experience by offering a syntax style nearly identical to that of SAS 9 and earlier releases. However, there are minor exceptions. In the past, SAS Viya procedures required the use of a CAS engine libref. This engine served multiple purposes, but it was primarily used to upload data to the CAS server and manage the data. Its additional purposes included support of new data types and improved performance for large data sets. No CAS server is needed for SAS Viya Workbench, because all computations are done in place for you in a single location.

In the following program, the LIBNAME statement creates a mylib libref that the examples in this chapter use to refer to your current or desired work directory:

```
libname mylib ".";
```

For more information, see *SAS Viya Workbench: Getting Started*.

Getting Started: SVMACHINE Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

This example trains the model by using German credit benchmark data, which are available in the sampsio.dmagecr data set. This data set contains 1,000 observations, each of which contains an applicant's information, including the applicant's credit rating (GOOD or BAD). The binary target is named GOOD_BAD. Other input variables are Checking, Duration, History, and so on.

The contents of the sampsio.dmagecr data set are described at http://support.sas.com/documentation/cdl/en/emgs/59885/HTML/default/a001026918.htm.

You can load the sampsio.dmagecr data set into your SAS library in the following DATA step:

```
data mylib.dmagecr;
   set sampsio.dmagecr;
run;
```

These statements assume that your SAS libref is named mylib, as in the section "Using SAS Viya Workbench" on page 933, but you can substitute any appropriately defined SAS libref.

The following statements execute the SVM algorithm on the mylib.dmagecr data table and produce the results shown in Figure 35.1 through Figure 35.3.

```
proc svmachine data=mylib.dmagecr;
   input checking history purpose savings employed marital coapp
              property other job housing telephon foreign/level=nominal;
   input duration amount installp resident existcr depends age/level=interval;
   target good_bad;
run;
```

The first INPUT statement defines the input variables Checking, History, Purpose, Savings, Employed, Marital, Coapp, Property, Other, Job, Housing, Telephon, and Foreign as categorical variables. The second

INPUT statement defines the input variables Duration, Amount, Installp, Resident, Existcr, Depends, and Age as continuous variables. The TARGET statement defines good_bad to be the target variable (the variable that is predicted). The "Training Results" table in Figure 35.1 shows that the inner product of weights is 11.6121718, the bias is –2.1296773, and the number of support vectors is 531, where 481 of those vectors are on the margin. The table also shows that the maximum decision function value (Maximum F) is 2.57131793 and the minimum decision function value (Minimum F) is –4.6513481.

Figure 35.1 German Credit Data Training Results

The SVMACHINE Procedure

Training Results	
Inner Product of Weights	11.6121718
Bias	-2.1296773
Total Slack (Constraint Violations)	492.87883
Norm of Longest Vector	4.17809329
Number of Support Vectors	531
Number of Support Vectors on Margin	481
Maximum F	2.57131793
Minimum F	-4.6513481
Number of Effects	20
Columns in Data Matrix	61

The "Misclassification Matrix" table in Figure 35.2 shows that among the total of 1,000 observations, 700 observations are classified as good and 300 observations are classified as bad. The number of correctly predicted good observations is 626, and the number of correctly predicted bad observations is 158. Thus the accuracy is 78.4%, as indicated in the "Fit Statistics" table in Figure 35.3.

Figure 35.2 German Credit Misclassification Matrix

Misclassification Matrix			
	Training Prediction		
Observed	good	bad	Total
good	626	74	700
bad	142	158	300
Total	768	232	1000

Figure 35.3 German Credit Accuracy

Fit Statistics	
Statistic	Training
Accuracy	0.7840
Error	0.2160
Sensitivity	0.8943
Specificity	0.5267

A relatively good model means that misclassification is low while both sensitivity and specificity are high. In PROC SVMACHINE, you can adjust training parameters and use different kernels to obtain a better model.

Syntax: SVMACHINE Procedure

The following statements are available in the SVMACHINE procedure:

```
PROC SVMACHINE <options> ;
    AUTOTUNE <options> ;
    CODE FILE=filename ;
    CROSSVALIDATION <options> ;
    FREQ variable ;
    ID variables ;
    INPUT variables / <LEVEL=INTERVAL | NOMINAL> ;
    KERNEL kernel-type / <kernel-parameter> ;
    MITIGATEBIAS mitigate-options ;
    OUTPUT OUT=libref.data-table <option> ;
    PARTITION partition-option ;
    SAVESTATE RSTORE=libref.data-table ;
    TARGET variable </options> ;
    WEIGHT variable ;
```

The PROC SVMACHINE statement, the TARGET statement, and at least one INPUT statement are required.

The following sections describe the PROC SVMACHINE statement and then describe the other statements in alphabetical order.

PROC SVMACHINE Statement

```
PROC SVMACHINE <options> ;
```

The PROC SVMACHINE statement invokes the procedure.

You can specify the following *options*:

C=*number*

specifies the penalty value, where *number* is a real number greater than 0.

By default, C=1.0.

NOTE: For the CD method, if neither the REGL1= nor REGL2= option is specified, then REGL1=C.

DATA=*libref.data-table*

names the input data table for PROC SVMACHINE to use. The default is the most recently created data table. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to the data. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 933.

data-table specifies the name of the input data table.

EARLYSTOP

generates a model that is based on the validation accuracy. The iterations stop when the validation accuracy stops improving, and the model from the previous iteration is selected. The iteration accuracy from both the training data and the validation data is displayed in the "Iteration Report" table.

NOTE: The EARLYSTOP option is ignored and a warning message is issued if the PARTITION statement is omitted. This option is ignored if you specify METHOD=CD, METHOD=ACTIVESET, or METHOD=RSVM, or perform SVR training.

EPSILON=*number*

specifies the insensitive loss value, where *number* is a nonnegative real number.

By default, EPSILON=0.01.

NOTE: This option is for SVR training.

ITERATIONREPORT

generates a table named "Iteration Report," which displays the training accuracy for each iteration. If you also specify the PARTITION statement and the validation data and testing data exist, then the "Iteration Report" table also displays the validation accuracy and testing accuracy, respectively, for each iteration. This option is ignored if you specify the AUTOTUNE statement; specify METHOD=CD, METHOD=ACTIVESET, or METHOD=RSVM; or perform SVR training.

NOTE: Generating the accuracy report for each iteration is computationally expensive and requires a significant amount of time.

MAXITER=*number*

specifies the maximum number of iterations before the process stops, where *number* is a positive integer.

By default, MAXITER=25. If you specify METHOD=CD, then by default MAXITER=100. In some cases, you can obtain a good model in fewer iterations.

MAXSV=*number*

specifies the maximum number of support vectors, where *number* is a positive integer.

By default, MAXSV=3500.

NOTE: This option applies only when METHOD=ACTIVESET or METHOD=RSVM. The actual number of support vectors can be a little larger than the specified *number* because of the nature of the active-set method.

METHOD=ACTIVESET | CD | IPOINT | RSVM

specifies the optimization method to use for training the data. You can specify the following methods:

ACTIVESET	uses the active-set method.
CD	uses the coordinate descent method.
IPOINT	uses the interior point method.
RSVM	uses the reduced SVM method.

By default, METHOD=IPOINT. If you specify METHOD=CD, you can also specify additional options, such as REGL1=, REGL2=, and SEED=. If you specify METHOD=RSVM, you can also specify additional options, such as SAMPLETYPE=, SAMPLERATE= or SAMPLESIZE=, and SEED=.

NOTE: When METHOD=ACTIVESET, METHOD=CD, or METHOD=RSVM, support vector regression is not supported.

NOTE: When METHOD=ACTIVESET, only a relatively small training data set is supported.

NOTE: When METHOD=RSVM, only the RBF kernel is supported.

NOPRINT
suppresses the generation of ODS outputs. If you specify this option, no ODS tables are generated.

NOSCALE
uses the original data during training.

NTHREADS=*number-of-threads*
specifies the number of threads to use in the computation. The default value is the number of CPUs available in the machine.

PRINTTARGET
generates the "Predicted Probability Variables" table, which displays the target variable and the predicted probability variables, and the "Predicted Target Variable" table, which displays the predicted target variable.

By default, these two tables are not generated. For SVR training, only the "Predicted Target Variable" table is generated.

REGL1=*number*
specifies the L1 penalty value when METHOD=CD, where *number* is a real number greater than 0.

By default, REGL1=1.0.

NOTE: If you specify a valid REGL2= option value, then the REGL1= option is ignored.

REGL2=*number*
specifies the L2 penalty value when METHOD=CD, where *number* is a real number greater than 0.

NOTE: If you specify a valid REGL2= option value, then the REGL1= option is ignored.

SAMPLERATE=*number*
specifies the sample rate when METHOD=RSVM, where *number* is a float number between 0 and 1.

By default, SAMPLERATE=0.1.

NOTE: If you specify both the SAMPLERATE= and SAMPLESIZE= options, then the SAMPLERATE= option is ignored.

SAMPLESIZE=*number*
specifies the sample size to use when METHOD=RSVM, where *number* is a positive integer less or equal to the number of observations.

NOTE: If you specify both the SAMPLESIZE= and SAMPLERATE= options, then the SAMPLERATE= option is ignored.

SAMPLETYPE=SIMPLE | STRATIFY

specifies the sampling strategy to use when METHOD=RSVM. You can specify the following sampling strategies:

SIMPLE uses the simple random sampling strategy.

STRATIFY uses the stratified random sampling strategy that is based on the target values.

By default, SAMPLETYPE=SIMPLE.

SCALE

scales the input variables to between 0 and 1 during training.

By default, all interval input variables are scaled before the training.

SEED=*number*

specifies the random number seed when METHOD=CD or METHOD=RSVM, where *number* is a nonnegative integer.

By default, SEED=1. If you specify a value of 0 or a negative integer, then the seed is generated by reading the time of day from the computer's clock.

TOLERANCE=*number*

specifies the minimal absolute tolerance at which the iteration stops. The tolerance *number* must be greater than or equal to 1.0E–12.

By default, TOLERANCE=1.0E–6. If METHOD=ACTIVESET or METHOD=RSVM, then by default TOLERANCE=0.0001. In some cases, you can obtain a good model by specifying a *number* greater than or equal to 0.01.

USEMISS

uses missing values for input variables. Missingness is treated as a special level for a categorical variable, and missing values for a continuous variable are imputed to the mean before training.

By default, all observations that have missing values are dropped during the training process.

AUTOTUNE Statement

AUTOTUNE < *options* > **;**

The AUTOTUNE statement searches for the best combination of values of the C, DEGREE, KERNEL_METHOD, RBFPARAMETER, REGL2, SIGMOIDPARAMETER1, and SIGMOIDPARAMETER2 hyperparameters. You cannot specify both the AUTOTUNE statement and the CROSSVALIDATION statement in the same procedure run.

Table 35.1 summarizes the *options* that you can specify in the AUTOTUNE statement. For more information about all options except the TUNINGPARAMETERS= option, see the option's description in the section "AUTOTUNE Statement" on page 6 in Chapter 2, "Shared Concepts." The TUNINGPARAMETERS= option is described following table Table 35.1.

NOTE: Processing the AUTOTUNE statement is computationally expensive and requires a significant amount of time.

Table 35.1 AUTOTUNE Statement Options

Option	Description
APPENDLOOKUP	Specifies that the table specified in the HISTORYTABLE= option contain the rows from the table specified in the LOOKUPTABLE= option
CONSTRAINTS=	Specifies the constraints on the output metrics
EVALHISTORY=	Specifies how to report the evaluation history of the tuner
FOLDCOLUMN=	Specifies the column in the data table in which the cross validation fold value is indicated
FRACTION=	Specifies the fraction of observations to use for validation
HISTORYTABLE=	Specifies the table that contains the evaluation history
KFOLD=	Specifies the number of folds for *k*-fold cross validation
LIVEUPDATE	Specifies that the table specified in the HISTORYTABLE= option be updated at every evaluation
LOOKUPTABLE=	Specifies the table to use for evaluation lookup
MAXBAYES=	Specifies the maximum number of points in the kriging model
MAXEVALS=	Specifies the maximum number of evaluations
MAXITER=	Specifies the maximum number of iterations when SEARCHMETHOD=GA or SEARCHMETHOD=BAYESIAN
MAXTIME=	Specifies the maximum time for all iterations
MAXTRAINTIME=	Specifies the maximum time for a model training
NCONVITER=	Specifies the number of convergence iterations
NOGRIDSHUFFLE	Requests that the grid points not be shuffled
NOLOCALSEARCH	Disables local search optimization
NPARALLEL=	Specifies the number of parallel sessions
NSUBSESSIONWORKERS=	Specifies the number of workers in parallel sessions
OBJECTIVE=	Specifies the objective function
POPSIZE=	Specifies the population size when SEARCHMETHOD=GA or SEARCHMETHOD=BAYESIAN
SAMPLESIZE=	Specifies the sample size when SEARCHMETHOD=LHS or SEARCHMETHOD=RANDOM
SEARCHMETHOD=	Specifies the search method that the optimizer uses
SECONDOBJECTIVE=	Specifies the second objective to use for tuning
SELECTINITPOINT	Specifies that the tuner select the best evaluation from the lookup table
TARGETEVENT=	Specifies the target event for ROC-based calculations
TRAINFRACTION=	Specifies the fraction of observations to use for training
TUNINGPARAMETERS=	Specifies the custom tuning parameters
USEPARAMETERS=	Specifies how to handle the TUNINGPARAMETERS= option

TUNINGPARAMETERS=(*suboption* **| . . . | <** *suboption* **>)**

TUNEPARMS=(*suboption* **| . . . | <** *suboption* **>)**

specifies which options to tune and which ranges to tune over. If USEPARAMETERS=STANDARD, this option is ignored.

You can specify one or more of the following *suboptions*:

C (LB=*number* UB=*number* VALUES=*value-list* INIT=*number* EXCLUDE)
specifies the penalty values to be used when tuning the SVM model, where *number* or any value in *value-list* is a real number greater than 0. For more information, see the C= option in the PROC SVMACHINE statement.

You can specify the following additional suboptions:

LB=*number*
specifies the minimum penalty value to consider when tuning the SVM model. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=1E-10.

UB=*number*
specifies the maximum penalty value to consider when tuning the SVM model. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=100.

VALUES=*value-list*
specifies a list of penalty values to consider when tuning the SVM model, where *value-list* is a space-separated list of numbers greater than 0. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial penalty value for the tuner to use.

By default, INIT=1.

EXCLUDE
excludes the penalty suboption from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

DEGREE (LB=*number* UB=*number* VALUES=*value-list* INIT=*number* EXCLUDE)
specifies the degree that is used in a polynomial kernel when tuning the SVM model, where *number* and values in the *value-list* can be 2 or 3. This suboption applies only to the polynomial SVM kernel. Therefore its value is shown as missing in the output tables when the value of the KERNEL_METHOD suboption is not POLYNOMIAL_IPOINT or POLYNOMIAL_ACTIVESET. If the number of model features is greater than 32, the DEGREE suboption is not tuned. Instead its value is set to 2 when the polynomial kernel is specified by the tuner. The number of model features is calculated as the number of interval inputs plus the sum of all levels of all the nominal inputs.

You can specify the following additional suboptions:

LB=*number*
specifies a lower bound of the degree to consider when tuning the SVM model. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=2.

UB=*number*
specifies an upper bound of the degree to consider when tuning the SVM model. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=3.

VALUES=*value-list*
specifies a list of values to consider for the degree in the kernel, where *value-list* is a space-separated list of numbers. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial degree of the kernel in the SVM model.

By default, INIT=2.

EXCLUDE
excludes the degree from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

KERNEL_METHOD (VALUES=*value-list* **INIT=***value* **EXCLUDE)**
specifies the combined values of the KERNEL and METHOD suboptions that are used when tuning the SVM model. The tuner uses the KERNEL_METHOD suboption instead of two separate suboptions, because the KERNEL and METHOD suboptions can be combined in only a few very specific combinations, and setting their values independently during the tuning would result in a large number of invalid combinations. For targets of nominal type, the value of the KERNEL_METHOD suboption is one of the following seven strings: LINEAR_IPOINT, POLYNOMIAL_IPOINT, LINEAR_CD, LINEAR_ACTIVESET, POLYNOMIAL_ACTIVESET, RBF_ACTIVESET, or SIGMOID_ACTIVESET. For targets of interval type, the value of the KERNEL_METHOD suboption is one of the following two strings: LINEAR_IPOINT or POLYNOMIAL_IPOINT. If the number of model features exceeds 100, the polynomial kernel is excluded from tuning. In that case the following values of the KERNEL_METHOD suboption are not used: POLYNOMIAL_IPOINT and POLYNOMIAL_ACTIVESET.

You can specify the following additional suboptions:

VALUES=*value-list*
specifies a list of values to consider for the KERNEL_METHOD suboption, where *value-list* is a space-separated list of strings.

INIT=*value*
specifies the initial value of the KERNEL_METHOD suboption.

By default, INIT=LINEAR_IPOINT.

EXCLUDE
excludes the KERNEL_METHOD suboption from the tuning process. If you specify this suboption, any specified VALUES= or INIT= suboptions are ignored.

RBFPARAMETER (LB=*number* UB=*number* VALUES=*value-list* INIT=*number* EXCLUDE)
specifies the RBFPARAMETER (K_PAR) suboption values to be used when tuning the SVM model, where *number* or any value in the *value-list* is a real number greater than 0. This suboption applies only to the RBF kernel. Therefore its value is shown as missing in the output tables when the value of the KERNEL_METHOD suboption is not RBF_ACTIVESET. For more information about the RBFPARAMETER (K_PAR) suboption, see the RBF option in the KERNEL statement.

You can specify the following additional suboptions:

LB=*number*
specifies the minimum RBFPARAMETER suboption value to consider when tuning the SVM model. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=0.1.

UB=*number*
specifies the maximum RBFPARAMETER suboption value to consider when tuning the SVM model. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=100.

VALUES=*value-list*
specifies a list of RBFPARAMETER suboption values to consider when tuning the SVM model, where *value-list* is a space-separated list of numbers greater than 0. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial RBFPARAMETER suboption value for the tuner to use.

By default, INIT=0.1.

EXCLUDE
excludes the RBFPARAMETER suboption from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

REGL2 (LB=*number* UB=*number* VALUES=*value-list* INIT=*number* EXCLUDE)
specifies the REGL2 suboption values to be used when tuning the SVM model, where *number* or any value in the *value-list* is a real number greater than 0. For more information about this suboption, see the REGL2 section.

You can specify the following additional suboptions:

LB=*number*
specifies the minimum REGL2 suboption value to consider when tuning the SVM model. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=0.1.

UB=*number*
specifies the maximum REGL2 suboption value to consider when tuning the SVM model. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=100.

VALUES=*value-list*
specifies a list of REGL2 suboption values to consider when tuning the SVM model, where *value-list* is a space-separated list of numbers greater than 0. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial REGL2 suboption value for the tuner to use.

By default, INIT=0.1.

EXCLUDE
excludes the REGL2 suboption from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

SIGMOIDPARAMETER1 (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**

specifies the SIGMOIDPARAMETER1 (K_PAR1) suboption values to be used when tuning the SVM model, where *number* or any value in the *value-list* is a real number greater than 0. This suboption applies only to the sigmoid kernel. Therefore its value is shown as missing in the output tables when the value of the KERNEL_METHOD suboption is not SIGMOID_ACTIVESET. For more information about the SIGMOIDPARAMETER1 suboption, see the SIGMOID option in the KERNEL statement.

You can specify the following additional suboptions:

LB=*number*
specifies the minimum SIGMOIDPARAMETER1 suboption value to consider when tuning the SVM model. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=0.1.

UB=*number*
specifies the maximum SIGMOIDPARAMETER1 suboption value to consider when tuning the SVM model. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=10.

VALUES=*value-list*
specifies a list of SIGMOIDPARAMETER1 suboption values to consider when tuning the SVM model, where *value-list* is a space-separated list of numbers greater than 0. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial SIGMOIDPARAMETER1 suboption value for the tuner to use.

By default, INIT=0.1.

EXCLUDE
excludes the SIGMOIDPARAMETER1 suboption from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

SIGMOIDPARAMETER2 (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**

specifies the SIGMOIDPARAMETER2 (K_PAR2) suboption values to be used when tuning the SVM model, where *number* or any value in the *value-list* is a real number. This suboption applies only to the sigmoid kernel. Therefore its value is shown as missing in the output tables when the value of the KERNEL_METHOD suboption is not SIGMOID_ACTIVESET. For more information about the SIGMOIDPARAMETER2 suboption, see the SIGMOID option in the KERNEL statement.

You can specify the following additional suboptions:

LB=*number*
specifies the minimum SIGMOIDPARAMETER2 suboption value to consider when tuning the SVM model. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=–10.

UB=*number*
specifies the maximum SIGMOIDPARAMETER2 suboption value to consider when tuning the SVM model. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=–0.1.

VALUES=*value-list*
specifies a list of SIGMOIDPARAMETER2 suboption values to consider when you are tuning the SVM model, where *value-list* is a space-separated list of real numbers. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial SIGMOIDPARAMETER2 suboption value for the tuner to use.

By default, INIT=–0.1.

EXCLUDE
excludes the SIGMOIDPARAMETER2 suboption from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, or INIT= suboptions are ignored.

CODE Statement

CODE FILE=*filename* **;**

The CODE statement generates SAS DATA step code that mimics the computations that the OUTPUT statement performs.

You must specify the following option:

FILE=*filename*
specifies the name of the file to write the SAS score code to.

The CODE statement is optional. When METHOD=ACTIVESET or METHOD=RSVM, the CODE statement is ignored.

CROSSVALIDATION Statement

CROSSVALIDATION < *options* > **;**

The CROSSVALIDATION statement performs *k*-fold cross validation to find the average estimated validation error. You cannot specify this statement if you specify the PARTITION statement.

You can specify the following *options*:

KFOLD=*number*
specifies the number of partition folds in the cross validation process, where *number* is between 2 and 20, inclusive. By default, KFOLD=5.

NOPARALLEL
specifies that *k*-fold cross validation not be run in parallel. By default, the process runs in parallel.

FREQ Statement

FREQ *variable* **;**

The *variable* in the FREQ statement identifies a numeric variable in the input data table that contains the frequency of occurrence of each observation. PROC SVMACHINE treats each observation as if it appears f times, where f is the value of the FREQ *variable* for the observation. If f is not an integer, it is truncated to an integer. If f is less than 1 or missing, the observation is not used in the analysis. When you omit the FREQ statement, each observation is assigned a frequency of 1.

NOTE: If you specify the CD method, then the FREQ statement is ignored.

ID Statement

ID *variables* **;**

The ID statement lists one or more variables that are to be copied from the input data table to the output data tables that are specified in the OUT= option in the OUTPUT statement and the RSTORE= option in the SAVESTATE statement.

INPUT Statement

INPUT *variables* / < **LEVEL=INTERVAL | NOMINAL** > **;**

The INPUT statement specifies the names of *variables* to be used in training. Only interval, binary, and nominal variables are accepted. If you want to use different options for different variables, you can specify multiple INPUT statements.

You can specify the following *option* after a slash (/):

LEVEL=INTERVAL | NOMINAL
specifies whether the specified input *variables* are continuous or categorical. You can specify the following values:

INTERVAL specifies that the input *variables* are continuous.

NOMINAL specifies that the input *variables* are categorical.

By default, LEVEL=INTERVAL for numeric variables and LEVEL=NOMINAL for categorical variables. Binary variables are considered to be categorical variables.

KERNEL Statement

KERNEL *kernel-type* / < *kernel-parameter* > **;**

The KERNEL statement specifies the type of kernel and any associated parameters to be used during training.

You can specify the following *kernel-types*:

LINEAR
uses a linear kernel during training. No *kernel-parameter* is needed. The kernel is defined as

$$k(x_1, x_2) = < x_1, x_2 >$$

where x_1 and x_2 are two vectors and $<, >$ is the inner product.

POLYNOMIAL < / DEGREE=*number* **>**

uses a polynomial kernel during training. Specify the polynomial degree as the *kernel-parameter*, in the form DEGREE=*number*, where *number* is 2 or 3 when METHOD=IPOINT. For example, specify KERNEL POLYNOMIAL / DEGREE=2. The kernel is defined as

$$k(x_1, x_2) = (< x_1, x_2 > +1)^p$$

where p is the degree of the polynomial. When METHOD=ACTIVESET, the *number* can be any positive integer.

By default, DEGREE=2.

NOTE: The NOSCALE option in the PROC SVMACHINE statement is not supported when the degree is greater than 3.

RBF < / K_PAR=*number* **>**

uses a radial basis function (RBF) kernel during training. Specify K_PAR as the *kernel-parameter*. For example, specify KERNEL RBF / K_PAR=1. The kernel is defined as

$$k(x_1, x_2) = \exp\left(-\frac{||x_1 - x_2||^2}{2\sigma^2}\right)$$

where σ is a positive number, which is K_PAR. The value of K_PAR must be greater than or equal to 0.0001.

By default, K_PAR = $\sqrt{\mathrm{nftr}}$, where nftr is the number of features. The aliases of K_PAR are KERNELPARM and RBFPARAMETER.

SIGMOID < / K_PAR1=*number* **K_PAR2=***number* **>**

uses a sigmoid kernel during training. Specify two *kernel-parameters*, K_PAR1 and K_PAR2. For example, specify KERNEL SIGMOID / K_PAR1=1 K_PAR2=–1. The kernel is defined as

$$k(x_1, x_2) = \tanh(p_1 < x_1, x_2 > +p_2)$$

where p_1 is a positive number (K_PAR1) and p_2 is a real number (K_PAR2).

By default, K_PAR1=1 and K_PAR2=–1. The aliases of K_PAR1 are KERNELPARM1 and SIGMOIDPARAMETER1. The aliases of K_PAR2 are KERNELPARM2 and SIGMOIDPARAMETER2.

By default, the *kernel-type* is LINEAR.

NOTE: For the CD method, the KERNEL statement is ignored.

NOTE: When METHOD=RSVM, only the RBF kernel is supported.

MITIGATEBIAS Statement

MITIGATEBIAS SENSITIVEVAR=*variable* **PREDICTEDVARS=(***variable-names***)** **PREDICTEDEVENTS=***"event-list"* < *options* > **;**

The MITIGATEBIAS statement enables PROC SVMACHINE to iteratively train a model while minimizing the bias metric that you specify. To use this statement, you must also specify the SAVESTATE statement and a categorical response (target) variable. You cannot also specify the AUTOTUNE statement or the CROSSVALIDATION statement.

You must specify the following options:

PREDICTEDEVENTS="*event-list***"**

PEVENTS="*event-list***"**

specifies the events that correspond to each variable in the PREDICTEDVARS= option. You must specify the PREDICTEDEVENTS= and PREDICTEDVARS= options with the same number of items in the same order. Items in the *event-list* must be separated by the delimiter that you specify in the DELIMETER= option.

PREDICTEDVARS=(*variable-names***)**

PVARS=(*variable-names***)**

specifies the names of the variables that contain the posterior probability for each level in model prediction that corresponds to the response (target) variable. You must specify the PREDICTEDVARS= and PREDICTEDEVENTS= options with the same number of items in the same order. Duplicate variables are not allowed.

SENSITIVEVAR=*variable*

specifies the sensitive variable to use in order to reduce the value of the bias measurement that you specify in the BIASMETRIC= option.

You can also specify the following *options*:

BIASMETRIC=*bias-metric*

specifies the type of bias measurement. You can specify the following *bias-metrics*:

DEMOGRAPHICPARITY specifies demographic parity as the bias measurement.

EQUALIZEDODDS specifies equalized odds as the bias measurement.

EQUALOPPORTUNITY specifies equal opportunity as the bias measurement.

PREDICTIVEPARITY specifies predictive parity as the bias measurement.

By default, BIASMETRIC=DEMOGRAPHICPARITY.

BOUND=*number*

specifies the bound value for the exponentiated gradient reduction algorithm.

By default, BOUND=100.

DELIMITER="*character***"**
DLM="*character***"**

specifies the delimiter to be used to separate events that you specify in the PREDICTEDEVENTS= option. A delimiter is used when event names contain embedded spaces (such as "Fuel Oil") or special characters (such as ";" or "*"). You must use quotation marks around *character*. You can specify only one delimiter; combinations of delimiters are not supported. The valid values of the DELIMITER= option are " " (space), ";" (semicolon), "*" (asterisk), "." (period), and "," (comma).

By default, DELIMITER=" " (space).

If you specify a delimiter other than " " (space), PROC SVMACHINE treats any spaces (including empty spaces) as valid characters. You are responsible for mapping events correctly in order to obtain correct results.

LEARNINGRATE=*number*

specifies the step size to use in updating the exponentiated gradient reduction algorithm.

By default, LEARNINGRATE=0.01.

LOGLEVEL=0 | 1 | 2

specifies the level of log information to print.

0 prints only warnings and errors.

1 prints some notes.

2 prints notes for each iteration.

By default, LOGLEVEL=1.

MAXITER=*number*

specifies the maximum number of iterations to run the exponentiated gradient reduction algorithm.

By default, MAXITER=10.

SEED=*number*

specifies the seed for the pseudorandom number generator. By default, the random number stream is based on the time of day from the computer's clock. A 0 or negative seed value also results in random number streams that are based on the time of day from the computer's clock. If you want a reproducible random number sequence between runs, specify a value greater than 0.

By default, SEED=0.

TARGETEVENT="*event***"**

specifies the formatted value of the response (target) variable that represents the event of interest.

By default, *event* is the first level that you specify in the PREDICTEDEVENTS= option.

TOLERANCE=*number*

specifies the parity constraint violation tolerance. If you set *number* to 0, PROC SVMACHINE trains for the maximum number of iterations, which you specify in the MAXITER= option.

By default, TOLERANCE=0.005.

TUNEBOUND
specifies that the bound value must be tuned.

OUTPUT Statement

OUTPUT OUT=*mylib.data-table* < *option* > **;**

The OUTPUT statement creates an output data table that contains the predicted values of the input data table.

You must specify the following option:

OUT=*libref.data-table*
names the output data table for PROC SVMACHINE to use. You must specify this option before any other options. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to where the data table is to be stored. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 933.

data-table specifies the name of the output data table.

You can also specify the following *option*:

COPYVAR=*variable*
COPYVARS=(*variables***)**
lists one or more variables from the input data table that are transferred to the output data table.

ROLE< =*variable* **>**
specifies a numeric *variable* that indicates the role that each observation plays in fitting the model. The default name is _ROLE_. Table 35.2 shows how this *variable* is interpreted for each observation.

Table 35.2 Role Interpretation

Value	Observation Role
0	Not used
1	Training
2	Validation
3	Testing

If you do not partition the input data by specifying a PARTITION statement, then the role variable value is 1 for observations that are used in fitting the model and 0 for observations that have at least one missing or invalid value for the target or input variables.

PARTITION Statement

PARTITION *partition-option* **;**

The PARTITION statement specifies how observations in the input data set are logically partitioned into disjoint subsets for model training, validation, and testing. For more information, see the section "Using Validation and Test Data" on page 23 in Chapter 2, "Shared Concepts." Either you can designate a variable in the input data table and a set of formatted values of that variable to determine the role of each observation, or you can specify proportions to use for randomly assigning observations to each role.

You must specify exactly one of the following *partition-options*:

FRACTION(< TEST=*fraction* **> < VALIDATE=***fraction* **> < SEED=***number* **>)**
randomly assigns specified proportions of the observations in the input data table to the roles. You specify the proportions for testing and validation by using the TEST= and VALIDATE= suboptions. If you specify both the TEST= and VALIDATE= suboptions, then the sum of the specified fractions must be less than 1 and the remaining fraction is the proportion of the observations assigned to the training role. The SEED= option specifies an integer that is used to start the pseudorandom number generator for random partitioning of data for training, testing, and validation. If you do not specify SEED=*number* or if *number* is less than or equal to 0, the seed is generated by reading the time of day from the computer's clock.

ROLE=*variable* **(< TEST='***value***' > < TRAIN='***value***' > < VALIDATE='***value***' >)**
ROLEVAR=*variable* **(< TEST='***value***' > < TRAIN='***value***' > < VALIDATE='***value***' >)**
names the *variable* in the input data table whose values are used to assign roles to each observation. This *variable* cannot also appear as an analysis variable in other statements or options. The TEST=, TRAIN=, and VALIDATE= suboptions specify the formatted values of this variable that are used to assign observation roles. If you do not specify the TRAIN= suboption, then all observations whose role is not determined by the TEST= or VALIDATE= suboption are assigned to the training role.

SAVESTATE Statement

SAVESTATE RSTORE=*mylib.data-table* **;**

The SAVESTATE statement creates an analytic store for the model and saves it as a binary object in a data table. You can use the analytic store in the ASTORE procedure to score new data. For more information, see Chapter 4, "ASTORE Procedure."

You must specify the following option:

RSTORE=*mylib.data-table*
specifies a data table in which to save the analytic store for the model. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 933.

TARGET Statement

TARGET *variable* < /*options* > **;**

The TARGET statement names the target *variable* whose values PROC SVMACHINE predicts. The target *variable* must be binary or interval, and it must be different from the variables in the INPUT statement.

You can specify following *options* after a slash (/):

LEVEL=INTERVAL | NOMINAL
specifies whether the specified target is continuous or categorical. You can specify the following values:

INTERVAL specifies that the target is continuous and performs SVR training.

NOMINAL specifies that the target is categorical and performs SVC training.

By default, LEVEL=NOMINAL.

For SVC training, you can also specify either of the following options to define how to levelize the target values. By default, the target values are levelized in descending order.

ASC
ASCENDING
levelizes the target values in ascending order.

DESC
DESCENDING
levelizes the target values in descending order.

NOTE: If you specify LEVEL=INTERVAL when METHOD=ACTIVESET or METHOD=CD, then PROC SVMACHINE issues an error message and stops.

WEIGHT Statement

WEIGHT *variable* **;**

The *variable* in the WEIGHT statement is used as a weight to perform a weighted analysis of the data. Observations that have nonpositive or missing weights are not included in the analysis. The values of the WEIGHT variable can be nonintegral and are not truncated. When you omit the WEIGHT statement, all observations that are used in the analysis are assigned a weight of 1.

NOTE: If you specify the CD method, then the WEIGHT statement is ignored.

Details: SVMACHINE Procedure

PROC SVMACHINE uses linear or nonlinear kernels to compute support vector machine (SVM) learning classifiers for the binary pattern recognition problem and the regression problem. The binary pattern recognition problem is called support vector classification (SVC), and the regression problem is called support vector regression (SVR). For more information about the theory and use of SVM learning, see Vapnik (1995); Burges (1998); Cristianini and Shawe-Taylor (2000); Smola and Schölkopf (2004).

Support Vector Classification

In the linear kernel case, support vector classification (SVC) computes the parameters w and β that define the decision function,

$$f(x) = w^T x + \beta$$

by solving the following quadratic optimization problem,

$$\begin{array}{lll} \underset{w,\beta,z}{\text{minimize}} & \frac{1}{2} w^T w + C e^T z & \\ \text{subject to} & \mathbf{D}\mathbf{X} w + \beta d & \geq \quad e - z \\ & z & \geq \quad 0 \end{array}$$

where $\mathbf{X}$ denotes the $m \times n$ matrix whose rows correspond to the observations, $\mathbf{D}$ denotes a diagonal matrix whose diagonals are 1 or –1, e is the vector of all ones, z denotes the slack variables that relax the classification constraints, and C denotes the penalty term. The corresponding dual optimization problem is

$$\begin{array}{ll} \underset{\alpha}{\text{minimize}} & \frac{1}{2} \alpha^T \mathbf{Q} \alpha - e^T \alpha \\ \text{subject to} & d^T \alpha = 0 \\ & 0 \leq \alpha \leq C \end{array}$$

where $\mathbf{Q} = \mathbf{D}\mathbf{X}\mathbf{X}^T\mathbf{D}$ and α is the Lagrange multiplier.

If you specify the FREQ or WEIGHT statement, the objective function, $\frac{1}{2} w^T w + C e^T z$, in the quadratic optimization problem becomes $\frac{1}{2} w^T w + C F^T z$, where the all-ones vector e is replaced by the frequency or weight vector F. If you specify both the FREQ and WEIGHT statements, F is the product of frequency and weight.

For a more general discussion about duality in the context of quadratic programming, see the chapter "The OPTQP Procedure" in *SAS/OR User's Guide: Mathematical Programming*. At the solution, w is related to α by the equation $w = \mathbf{X}^T \mathbf{D} \alpha$. Observations that correspond to nonzero entries in α are called support vectors. Observations that correspond to entries in α that are active at their upper bound are called support vectors on the margin.

In the nonlinear case, the dual optimization problem satisfies $Q_{ij} = d_i d_j k(x_i, x_j)$, where $k(x, y)$ denotes the selected kernel function, which is defined in the section "KERNEL Statement" on page 946. The corresponding decision function is defined in terms of β, α, and the support vectors as follows:

$$f(x) = \beta + \sum_{\alpha_i > 0} \alpha_i d_i k(x, x_i)$$

When nonlinear kernels are used, the dimension of the corresponding primal problem can be prohibitively large, or even infinite. Polynomial kernels are a special case in that the primal problem definition can be formed explicitly; in this case, the matrix **X** corresponds to explicitly projected observations.

Support Vector Regression

For support vector regression (SVR), the primal problem is defined as

$$\begin{array}{ll} \underset{w,\beta,z,z^*}{\text{minimize}} & \frac{1}{2}w^T w + Ce^T(z+z^*) \\ \text{subject to} & \begin{array}{rcl} d - \mathbf{X}w - \beta e & \leq & \varepsilon e + z \\ \mathbf{X}w + \beta e - d & \leq & \varepsilon e + z^* \\ z, z^* & \geq & 0 \end{array} \end{array}$$

where **X** denotes the $m \times n$ matrix as defined in SVC and z and z^* are slack variables.

The corresponding dual optimization problem becomes

$$\begin{array}{ll} \underset{\alpha,\alpha^*}{\text{minimize}} & \frac{1}{2}(\alpha-\alpha^*)^T\mathbf{X}\mathbf{X}^T(\alpha-\alpha^*) + \varepsilon e^T(\alpha+\alpha^*) - d^T(\alpha-\alpha^*) \\ \text{subject to} & e^T(\alpha-\alpha^*) = 0 \\ & 0 \leq \alpha, \alpha^* \leq Ce \end{array}$$

where α and α^* are the Lagrange multipliers.

With the same model function $f(x) = w^T x + \beta$, the parameter w in the primal problem can be calculated as

$$w = \sum_{i=1}^{m}(\alpha_i - \alpha_i^*)x_i = \mathbf{X}^T(\alpha-\alpha^*)$$

As with SVC, if you specify the FREQ or WEIGHT statement, the objective function, $\frac{1}{2}w^T w + Ce^T(z+z^*)$, in the primal problem becomes $\frac{1}{2}w^T w + CF^T(z+z^*)$, where the all-ones vector e is replaced by the frequency or weight vector F.

Interior Point Optimization Methods

A popular approach for accurately obtaining the global solution to an SVM optimization problem is interior point (IPOINT) methods. Interior point methods are attractive in that the required number of iterations is relatively small and does not grow dramatically with problem size. The cost per iteration for interior point methods can be quite high for large-scale problems unless there is an underlying structure that can be exploited. For example, interior point methods can be extremely efficient for problems where the number of variables is smaller than the number of constraints. When both the number of variables and the number of constraints are large, interior point methods become intractable for dense problems.

PROC SVMACHINE applies a primal-dual interior point method to linear and polynomial kernels of degree 2 and 3. In the polynomial kernel case, **X** is obtained by explicitly projecting each observation into the design matrix. The resulting number of columns is given by the binomial coefficient

$$\binom{n+p}{p} = \frac{(n+p)!}{p!n!}$$

where n denotes the number of columns in the levelized design matrix and p denotes the polynomial degree. Thus, a degree of 2 is not recommended when the number of columns in the design matrix is much greater than 100. Similarly, a degree of 3 is not recommended when the number of columns in the design matrix is much greater than 32.

Primal-dual interior point methods perturb the optimality conditions in order to create a system of nonlinear equations that satisfy the requirements of Newton's method. This perturbed system of nonlinear equations has the property that an interior solution (with respect to the inequality constraints) always exists; safeguards are then wrapped around Newton's method to ensure that the interior (and hence feasible) approximate solution is obtained. Interior point methods have the additional property that as the perturbation term goes to 0, the approximate solution converges to the true solution.

The dominant cost of each iteration comes from the need to solve a system of linear equations of size $(m + n) \times (m + n)$. Because PROC SVMACHINE assumes that m might be very large, it uses block reduction strategies similar to those described in Gertz and Griffin (2005, 2010) to reduce the size of this system to a matrix of size $n \times n$, where n denotes the number of columns in the design (or projected kernel) matrix. Then it performs dense matrix factorization on the resulting system. For problems where $m \gg n$, the dominant computational cost occurs during the block-row reduction step. To reduce the solution time, this operation is both distributed and threaded.

Coordinate Descent Optimization Method

In addition to supporting the interior point method, PROC SVMACHINE also supports the coordinate descent (CD) method. The CD method does not support nonlinear kernels; it supports only linear kernels.

The CD method approaches the solution by solving the dual optimization problem through two layers of iterations: inner-layer iteration and outer-layer iteration. Inner-layer iteration solves variable-based small quadratic optimization problems. Outer-layer iteration synchronizes the final weight of each inner-layer iteration. Each outer-layer iteration contains m inner-layer iterations, where m is the number of observations. The cost per outer-layer iteration is $O(mn)$, which is much lower than the cost of iterations for the interior point method. The overall cost of iterations for the CD method is much lower than the cost of iterations for the interior point method, and it requires more iterations to achieve the same solution.

Because inner-layer iteration depends on the number of observations, the CD method can support very wide data sets. For high-cardinality data sets, the CD method outperforms the interior point method in both memory usage and CPU time. The CD method can handle these problems even if both the number of variables and the number of observations are large.

In addition to supporting the C penalty (the L1-SVM model), the CD method also supports the L2 penalty (the L2-SVM model). You can specify the penalty parameter regL1 to train an L1-SVM model and specify the penalty parameter regL2 to train an L2-SVM model.

To obtain a better model quickly, the CD method also adopts the data shuffling technique by training the subproblem in each inner-layer iteration randomly. You can specify the random number seed by using the SEED= option.

For more information about the CD method, see Hsieh et al. (2008).

NOTE: In the current release, the CD method does not support SVR training or the FREQ and WEIGHT statements.

Active-Set Optimization Method

In order to support nonlinear kernels such as the radial basis function (RBF), PROC SVMACHINE also supports the active-set method. This method supports linear, polynomial, RBF, and sigmoid kernels; for more information, see the section "KERNEL Statement" on page 946.

The active-set optimization method applies to the dual optimization problem. The active-set method seeks to form the solution by determining which inequality constraints are active at the solution, and they quickly converge as soon as the true active set is known. For the SVM case, active constraints correspond to the bound constraints where $\alpha_i \in \{0, C\}$. If the active set is known, you can determine the optimal values of the remaining entries in α by solving a single system of linear equations whose size is on the order of the number of inactive constraints. Because there are 2^m possible ways to partition the bound constraints into active and inactive sets, it is possible to create worst-case-scenario problems in which a prohibitive number of combinations are tried before a solution is reached. However, in practice, the number of combinations that are tried is usually much smaller. A consequence of this observation is that the run time of the active-set method for a particular problem is much harder to predict.

The computational costs of the active-set method are controlled by the number of support vectors, and this number is highly dependent on both the problem and the choice of kernel. As a rule, the run time increases with the number of inactive support vectors. A significant computational cost of the active-set method comes from the need to update the residual of the optimality error. Therefore, PROC SVMACHINE is multithreaded to reduce the solution time.

For the same problem, you might observe that the number of support vectors that the interior point method finds is much larger than the number of support vectors that the active-set method finds. This can occur whenever the dual solution, α, is not unique; it results when the interior point method tries to follow a positive trajectory of approximate minimizers to the solution. This implies that the interior point method initially treats every vector as a support vector and uncovers non-support vectors in the limit, when it is pushed sufficiently close to the boundary. In contrast, active-set methods add support vectors to the approximate solution incrementally and have a better chance of finding a sparser solution.

The model training of the active-set method is supported on a single-machine node. When data are distributed, the training is performed at the first worker node where the data are aggregated from other nodes before the computation. Unlike the interior point and coordinate descent methods, the active-set method supports only problems that involve relatively small data sets. For example, when the number of observations is greater than 100,000 and the number of features is greater than 1,000, the memory requirement can dramatically increase and the training time is much longer.

NOTE: Currently, the active-set method does not support SVR training.

Reduced SVM Optimization Method

Training a nonlinear support vector machine (SVM) at scale remains a computationally demanding optimization problem. Although the active-set algorithm performs effectively on small to medium-sized data sets, its scalability to larger data sets is limited. In particular, when you use a radial basis function (RBF) kernel, both memory consumption and training time can increase substantially. As the number of support vectors grows, model training can take hours or even days, and in extreme cases it can fail to converge within practical resource limits. These challenges reflect the inherent computational complexity of nonlinear SVM optimization.

To improve training efficiency for large-scale problems, PROC SVMACHINE implements the reduced support vector machine (RSVM) method. RSVM preselects a subset of the training observations as candidate support vectors and solves a reduced quadratic programming problem that is defined on this subset. This strategy reduces the optimization complexity from $O(m^2)$ to $O(ml)$, where m denotes the number of training observations and l represents the number of preselected candidates. By keeping l relatively small, PROC SVMACHINE decreases both training time and memory usage while maintaining acceptable model performance.

The RSVM candidate selection process is governed by the SAMPLERATE= and SAMPLESIZE= options in the PROC SVMACHINE statement, which determine the size of the candidate subset. The SAMPLETYPE= option enables you to specify two sampling strategies: SIMPLE and STRATIFY. The SIMPLE strategy performs uniform random sampling, selecting candidates up to the proportion of the full training observations that you specify in the SAMPLERATE= option or up to the limit that you specify in the SAMPLESIZE= option. The STRATIFY strategy also performs random sampling, but it preserves the class distribution of the target variable within the selected subset.

A trade-off of the RSVM approach is that predictive accuracy can be lower than the accuracy that is obtained by the active-set method, because the optimization is restricted to a subset of candidate support vectors. However, for large-scale data sets, the substantial reductions in computational time and memory requirements often make RSVM the more practical and computationally feasible alternative.

For more information about the RSVM method, see Omheni and Griffin (2025).

NOTE: Currently, the RSVM method does not support SVR training. For this method, only the RBF kernel is supported.

Scoring Process

The scoring process for the interior point method and coordinate descent method is straightforward. As long as the training weight parameters and bias are known, the scoring process is just a linear combination. The event or nonevent is decided by the decision function. The decision function variable _P_ is defined in both the training output table and the scored data table. If the value of _P_ is less than or equal to 0, then the prediction is an event; otherwise, it is a nonevent. For SVR training, the predicted variable is P_target, where target is the target variable.

The analytic store is generated when you specify the SAVESTATE statement. In this case, you can score the data by using the ASTORE procedure; see Chapter 4, "ASTORE Procedure." For the interior point and coordinate descent methods, the score code is provided if you specify the CODE statement. In this case, you can score the data by using a traditional SAS DATA step.

k-fold Cross Validation

The CROSSVALIDATION statement performs *k*-fold cross validation to assess the accuracy of a model. During cross validation, all data are divided into *k* subsets (folds), where *k* is the value of the KFOLD= option. For each fold, a new model is trained on the (*k*–1) folds and then validated using the selected (holdout) fold. The assessment metrics are then averaged across all the holdout folds. The CROSSVALIDATION statement creates a table that has *k*+1 rows. The first *k* rows contain the assessment metrics for each holdout fold, and the last row contains the average across all the holdout folds. Which metrics the table includes depends on whether the target is nominal or interval. For nominal targets, the table includes the values of square error, mean consequential error, and multiclass log loss. For interval targets, the table includes the values of square error, absolute error, and square logarithmic error.

Bias Mitigation

With the widespread use of AI and machine learning models in decision-making processes, there is concern about the social impacts of these models. Bias can be embedded in both the training data and the trained model, either of which can lead to unfair or biased predictions for certain groups or populations. To reduce bias in the data and models, researchers have developed three main categories of bias mitigation methods:

- Preprocessing methods, which transform the data prior to model training. These methods try to eliminate the correlations between the input features and the sensitive feature while retaining as much information from the input data as possible.
- In-process methods, which consider fairness constraints during the model training process. These methods actively adjust the model parameters during training to create a model that produces fair predictions and classifications.
- Postprocessing methods, which do not directly alter the input data or the model training process. These methods adjust the predictive outputs from the learned models to compensate for the bias in the model.

You can reduce bias in the trained model by specifying the MITIGATEBIAS statement. This statement uses the exponentiated gradient reduction (EGR) algorithm, which is an in-process method of mitigating bias.

Exponentiated Gradient Reduction (EGR) Algorithm

The exponentiated gradient reduction (EGR) algorithm was introduced at ICML 2018 (Agarwal et al. 2018). As an in-process algorithm, it considers fairness constraints during the model training process. The EGR algorithm has three main advantages. First, it is model-agnostic, which means that you can apply it to a wide variety of machine learning models. The EGR algorithm works like a wrapper that is built around existing machine learning models. The wrapper iteratively reweights and possibly relabels the data, and then it trains a new classifier model on the basis of reweighted data in each iteration. By doing so, it reduces the complex optimization problem of training a machine learning model with fairness constraints to a series of standard model training problems without the fairness constraints. Second, the EGR algorithm supports various fairness measurements, such as demographic parity, equal opportunity, and equalized odds. Finally, the algorithm has strong theoretical support in terms of its optimization trade-off between fairness and accuracy. In theory, the EGR algorithm returns solutions that satisfy all the fairness constraints and also achieve the optimal error rate of the best fair solution.

Training a fair machine learning model is in general a nonlinear programming problem. In nonlinear optimization tasks, it is not uncommon to get suboptimal solutions because the constraints are infeasible or because the algorithm is stuck in a local minimum. As with many other machine learning algorithms, the effectiveness of the EGR algorithm depends on many factors, such as the quality of the input data, the selection of fairness constraints, the quality of the machine learning model that it wraps around, and the values of the hyperparameters.

PROC SVMACHINE supports bias mitigation for multilevel nominal target variables and multilevel nominal sensitive variables. Note that when you include a large number of levels in the sensitive variable, the EGR algorithm searches more complex optimization scenarios; this can result in suboptimal solutions. Therefore, the solutions from this algorithm might not always be desirable when the number of levels in the sensitive variable is high. You should review and validate your sensitive variable, training models, and fairness constraints before applying the EGR algorithm.

Fairness Constraints

PROC SVMACHINE supports the following fairness measurements: demographic parity, predictive parity, equal opportunity, and equalized odds. The goal of the EGR algorithm is to bring the fairness metrics of groups as close together as possible, as well as to convert fair machine learning to a sequence of standard cost-sensitive machine learning problems whose solutions yield a randomized classifier with the lowest error rate, subject to the desired constraints. As a result, the EGR algorithm iteratively trains the machine learning models by using different weights to reduce the fairness disparities without significantly increasing the loss of the final machine learning model. For example, when there are two protected groups in the sensitive variable, the fairness constraints that the EGR algorithm requires for demographic parity are as follows:

$$\mu_1^t - \mu_0^t \leq c$$
$$\mu_1^t - \mu_0^t \geq -c$$
$$\mu_2^t - \mu_0^t \leq c$$
$$\mu_2^t - \mu_0^t \geq -c$$

where μ_1^t is the demographic parity for the first protected group during iteration t; μ_2^t is the demographic parity for the second protected group during iteration t; μ_0^t is the demographic parity for the entire population during iteration t; and c is the predefined violation tolerance threshold.

Exponentiated Gradient Reduction Algorithm Workflow

Two key hyperparameters in the EGR algorithm are specified by the BOUND= option (denoted by B) and the LEARNINGRATE= option (denoted by lr); both are available in the MITIGATEBIAS statement. To derive the fair machine learning algorithm, the fairness constraints (see the section "Fairness Constraints" on page 959) are added to the loss function of the cost-sensitive machine learning model by introducing the Lagrange multipliers $\lambda_{j,t}$, where j is the index in the constraint set. The multipliers $\lambda_{j,t}$ are updated on the basis of the degree of constraint violations $\gamma_{j,t}$, B, and lr. Then $\lambda_{j,t}$ is used to derive the new weights for each observation in the input data set for the next round of training. Note that the values of B and lr play important roles in the performance of the EGR algorithm, based on experiments. The next section discusses how to come up with appropriate values for B and lr.

Bound Value Recommendation

Because the EGR algorithm is designed to strike a balance between model accuracy and fairness constraint violations, it does not perform well unless you specify appropriate values for the BOUND= and LEARNINGRATE= options in the MITIGATEBIAS statement. The BOUND= option is important for the sample

reweighting and relabeling process. It controls how heavily the EGR algorithm penalizes fairness constraint violations, and therefore it determines the trade-off between the fairness constraints and the model accuracy. When the bound is too small, there is not enough momentum to drive the model close enough to the saddle points in the solution sets. In this case, the bias metric values of the different subgroups do not approach each other.

The LEARNINGRATE= option controls how quickly the algorithm reduces fairness constraint violations. If the learning rate is too small, the algorithm converges very slowly, resulting in a long training process; if the learning rate is too large, the algorithm becomes unstable and might not converge. In general, the learning rate and the bound value are coupled, and changing one of these values changes the optimal value of the other.

The MITIGATEBIAS statement offers a built-in bound value recommendation function to automatically search for a good candidate bound value when you request it. To eliminate the coupling effect of the bound and the learning rate, PROC SVMACHINE fixes the initial learning rate at 0.01 during the tuning process. The procedure runs the bound value recommendation function if you specify the TUNEBOUND option. This function finds the best bound value through multiple iterations by analyzing the convergence patterns of the fairness metrics. PROC SVMACHINE first runs the EGR algorithm with some initial bound value for a few iterations, while monitoring the convergence pattern for signs of stagnation or overshooting to determine whether the bound value is too small or too large. It then adjusts the bound value until it finds an appropriate value or reaches the maximum number of iterations that are allowed. After the recommendation process, PROC SVMACHINE applies the recommended bound value to the EGR algorithm to train a final model. Note that because of the nonlinear nature of the optimization process, the bound value recommendation is also subject to the problem of local minima. This recommendation should be used as a general guideline for setting the bound value; you might find a better solution by further fine-tuning.

Displayed Output

The following sections describe the output that PROC SVMACHINE produces by default. The output is organized into various tables, which are discussed in the order of their appearance.

Model Information

The “Model Information” table contains the initial training settings, such as task type, optimization technique, and kernel function type. If the kernel function type is polynomial, then the kernel degree is also displayed.

Number of Observations

The “Number of Observations” table contains the number of observations and the number of observations used. If the FREQ statement is specified, the sum of the frequency values is reported. If the WEIGHT statement is specified, the sum of the weight values is reported. If both the FREQ statement and WEIGHT statement are specified, the sum of the weight values is the sum of the product of frequency and weight.

NOTE: For the CD method, both the FREQ and WEIGHT statements are ignored.

Training Results

The "Training Results" table contains the model information. The information includes but is not limited to the inner product of weights, bias, maximum and minimum F (the maximum and minimum values, respectively, of the decision function), and the number of effects.

Optimization Iteration History

The "Optimization Iteration History" table contains the number of iterations, the complementarity, and the feasibility. The complementarity is controlled by the TOLERANCE= option (which specifies the minimal absolute tolerance at which an iteration stops) and the MAXITER= option (which controls the number of iterations).

NOTE: For the CD method, the active-set method, the reduced SVM method, and SVR training, the "Optimization Iteration History" table is not generated.

Misclassification Matrix

The "Misclassification Matrix" table contains the target information, both observed and predicted. The columns include the observed target, predicted event, predicted nonevent, and total numbers of events or nonevents for the training data.

NOTE: For SVR training, the "Misclassification Matrix" table is not generated.

Fit Statistics

The "Fit Statistics" table contains the model accuracy information, which includes accuracy, error, sensitivity, and specificity. The statistics are calculated from the "Misclassification Matrix" table.

NOTE: For SVR training, the "Fit Statistics" table is not generated.

Iteration Report

The "Iteration Report" table contains the model accuracy value for each iteration step. If the PARTITION statement is specified, then the specified validation or testing data accuracy is also reported.

NOTE: For the CD method, the active-set method, the reduced SVM method, and SVR training, the "Iteration Report" table is not generated.

Predicted Probability Variables

The "Predicted Probability Variables" table contains the target variable and the predicted probability variables. The names of the predicted probability variables begin with P_.

NOTE: For SVR training, the "Predicted Probability Variables" table is not generated.

Predicted Target Variable

The “Predicted Target Variable” table contains the predicted target variable. For SVC training, the variable name begins with I_. For SVR training, the variable name begins with P_.

Constraint History

The “Constraint History” table lists the values of the various constraints at each iteration of the EGR algorithm. This table is produced by the MITIGATEBIAS statement.

Group Metrics History

The “Group Metrics History” table displays the metrics for accuracy and fairness for each subpopulation over each iteration of the EGR algorithm. This table is produced by the MITIGATEBIAS statement.

Iteration History

The “Iteration History” table displays the values of accuracy and bias metric at each iteration of the EGR algorithm. This table is produced by the MITIGATEBIAS statement.

Mitigation Information

The “Mitigation Information” table displays information about the options that you use for bias mitigation. This table is produced by the MITIGATEBIAS statement.

Timing

The “Timing” table displays the breakdown of how long bias mitigation spent on the various tasks of training models. This table is produced by the MITIGATEBIAS statement.

Weight History

The “Weight History” table displays the weights that are assigned by the EGR algorithm for each level of the response (target) variable and for each subpopulation. This table is produced by the MITIGATEBIAS statement.

ODS Table Names

Each table that the SVMACHINE procedure creates has a name associated with it. You must use this name to refer to the table when you use ODS statements. The name of each table and a short description of the contents are listed in Table 35.3.

Table 35.3 ODS Tables Produced by PROC SVMACHINE

Table Name	Description	Statement	Option
BestConfiguration	Hyperparameters and objective function values for the best configuration	AUTOTUNE	Default
CrossValidateMLFitStat	Per-fold and average assessment metrics of *k*-fold cross validation	CROSSVALIDATION	Default
ConstraintHistory	Values of the various constraints at each iteration of the EGR algorithm	MITIGATEBIAS	Default
EvaluationHistory	Values of the hyperparameters and the objective function for all configurations	AUTOTUNE	Default
FitStatistics	Accuracy information about the training	PROC SVMACHINE	Default
GroupMetricsHistory	Metrics for accuracy and fairness for each subpopulation over each iteration of the EGR algorithm	MITIGATEBIAS	Default
IterationHistory	Values of the model accuracy and bias metric over each iteration of the EGR algorithm	MITIGATEBIAS	Default
IterationReport	Accuracy report for each iteration	PROC SVMACHINE	ITERATIONREPORT
IterHistory	Optimization iteration history	PROC SVMACHINE	Default
Misclassification	Misclassification matrix table	PROC SVMACHINE	Default
MitigationInformation	Information about the options that are used for bias mitigation	MITIGATEBIAS	Default
ModelInfo	Basic model information for the training	PROC SVMACHINE	Default
NObs	Observation information about the input data	PROC SVMACHINE	Default
PredIntoName	Predicted target variable	PROC SVMACHINE	PRINTTARGET

Table 35.3 *continued*

Table Name	Description	Statement	Option
PredProbName	Predicted probability variables	PROC SVMACHINE	PRINTTARGET
Timing	Breakdown of how long bias mitigation spent on the various tasks of training models	MITIGATEBIAS	Default
TrainingResult	Displays the training results	PROC SVMACHINE	Default
TunerInfo	Setup values that are used by the tuner	AUTOTUNE	Default
TunerResults	Values of the hyperparameters, the objective function for the default configuration (iteration 0), and up to 10 best configurations found	AUTOTUNE	Default
TunerSummary	Statistics about the tuning process	AUTOTUNE	Default
TunerTiming	Total time spent on different tasks during tuning	AUTOTUNE	Default
WeightHistory	Weights that are assigned by the EGR algorithm for each level of the response (target) variable and for each subpopulation	MITIGATEBIAS	Default

ODS Graphics

You can refer to every graph that is produced through ODS Graphics by name. The name of the graph that PROC SVMACHINE generates is listed in Table 35.4.

Table 35.4 Graph Produced by PROC SVMACHINE

ODS Graph Name	Plot Description	Statement	Option
IterationHistoryPlot	Iteration history of the model accuracy and bias metric	MITIGATEBIAS	Default

Examples: SVMACHINE Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

Example 35.1: Home Equity Loan Case

This example shows how you can use PROC SVMACHINE to create scoring code that can be used to score future home equity loan applications. The data set Hmeq, which is in the Sampsio library that SAS provides, contains observations for 5,960 mortgage applicants. A variable named Bad indicates whether the customer has paid on the loan or has defaulted on it. Table 35.5 describes the variables in Hmeq.

Table 35.5 Variables in the Home Equity (Hmeq) Data Set

Variable	Role	Level	Description
Bad	Response	Binary	1 = customer defaulted on the loan or is seriously delinquent 0 = customer is current on loan payments
CLAge	Predictor	Interval	Age of oldest credit line in months
CLNo	Predictor	Interval	Number of credit lines
DebtInc	Predictor	Interval	Debt-to-income ratio
Delinq	Predictor	Interval	Number of delinquent credit lines
Derog	Predictor	Interval	Number of major derogatory reports
Job	Predictor	Nominal	Occupational category
Loan	Predictor	Interval	Requested loan amount
MortDue	Predictor	Interval	Amount due on existing mortgage
nInq	Predictor	Interval	Number of recent credit inquiries
Reason	Predictor	Binary	`'DebtCon'` = debt consolidation `'HomeImp'` = home improvement
Value	Predictor	Interval	Value of current property
YoJ	Predictor	Interval	Years at present job

You can load the Hmeq data set into your SAS library in the following DATA step:

```
data mylib.hmeq;
   set sampsio.hmeq;
run;
```

The following statements execute the SVM algorithm on the mylib.hmeq data table:

```
filename codefile temp;
proc svmachine data=mylib.hmeq;
   input reason job derog delinq ninq / level=nominal;
   input loan mortdue value yoj clage clno debtinc / level=interval;
   target bad / desc;
   code file=codefile;
```

```
run;
```

The first INPUT statement defines the input variables Reason, Job, Derog, Delinq, and Ninq as categorical variables. The second INPUT statement defines the input variables Loan, MortDue, Value, YoJ, CLAge, CLNo, and DebtInc as continuous variables. The TARGET statement defines Bad (which is a binary variable) as the target variable and specifies the order of the target variable as descending. The CODE statement generates DATA step scoring code and stores it in the filename *codefile*. The scoring code can be used to score other home equity loan applications.

PROC SVMACHINE generates several ODS tables, some of which are shown in Output 35.1.1 through Output 35.1.5.

The "Model Information" table in Output 35.1.1 shows that the kernel function is linear, and the penalty parameter value is 1 (both of which are default values).

Output 35.1.1 Model Information

The SVMACHINE Procedure

Model Information	
Task Type	C_CLAS
Optimization Technique	Interior Point
Scale	YES
Kernel Function	Linear
Penalty Method	C
Penalty Parameter	1
Maximum Iterations	25
Tolerance	1e-06

The observations table in Output 35.1.2 shows that the total number of observations is 5,960 and the number of observations used in the training is 3,364.

Output 35.1.2 Number of Observations

Number of Observations Read	5960
Number of Observations Used	3364

The "Training Results" table in Output 35.1.3 shows the inner product of weights, bias, total slack, and so on.

Output 35.1.3 Training Results

Training Results	
Inner Product of Weights	19.8000318
Bias	-1.5372926
Total Slack (Constraint Violations)	532.92348
Norm of Longest Vector	2.72195233
Number of Support Vectors	3361
Number of Support Vectors on Margin	267
Maximum F	1.0000874
Minimum F	-2.9999943
Number of Effects	12
Columns in Data Matrix	49

The "Misclassification Matrix" table in Output 35.1.4 displays the original observations and predicted values. Here the true positive is 43, the false negative is 257, the true negative is 3,055, and the false positive is 9.

Output 35.1.4 Misclassification Matrix

Misclassification Matrix			
	Training Prediction		
Observed	**1**	**0**	**Total**
1	43	257	300
0	9	3055	3064
Total	52	3312	3364

The "Fit Statistics" table in Output 35.1.5 shows information about the accuracy, error, sensitivity, and specificity.

Output 35.1.5 Fit Statistics

Fit Statistics	
Statistic	**Training**
Accuracy	0.9209
Error	0.0791
Sensitivity	0.1433
Specificity	0.9971

In addition to these ODS tables, PROC SVMACHINE also generates the "Optimization Iteration History" table. The CODE statement in this example generates SAS code and stores at in the filename *codefile*. Advanced SAS users can use the SAS code to easily score their data.

Example 35.2: Support Vector Regression

In this example, support vector regression training is performed. For the sake of simplicity, the same data set, mylib.dmagecr, is used. Instead of using the binary target good_bad, amount is chosen as the interval target. The insensitive loss parameter epsilon is set to 5.0, and the LEVEL=INTERVAL option in the TARGET statement requests support vector regression training.

The following statements execute SVR algorithm on the mylib.dmagecr data table and produce Output 35.2.1 and Output 35.2.2:

```
proc svmachine data=mylib.dmagecr epsilon=5.0 noscale;
   input checking history purpose savings employed marital coapp
         property other job housing telephon foreign/level=nominal;
   input duration installp resident existcr depends age/level=interval;
   target amount / level=INTERVAL;
run;
```

The "Model Information" table is displayed in Output 35.2.1.

Output 35.2.1 Model Information

The SVMACHINE Procedure

Model Information	
Task Type	ESP_REG
Optimization Technique	Interior Point
Scale	NO
Kernel Function	Linear
Penalty Method	C
Penalty Parameter	1
Insensitive Loss Parameter	5
Maximum Iterations	25
Tolerance	1e-06

The "Training Results" table is displayed in Output 35.2.2.

Output 35.2.2 Training Result

Training Results	
Inner Product of Weights	128231.898
Bias	912.616613
Total Slack (Constraint Violations)	1298997.98
Norm of Longest Vector	87.2467764
Mean Squared Error	4483863.04
Number of Effects	19
Columns in Data Matrix	60

Example 35.3: Coordinate Descent Method

In this example, you can apply the same German credit benchmark data that are specified in the section "Getting Started: SVMACHINE Procedure" on page 933. Instead of using the default interior point method, you can apply the coordinate descent method and save the model to an analytic store. After that, you can load the testing data set sampsio.dmagescr into your SAS library and apply the model to the testing data by using the ASTORE procedure.

You can load the sampsio.dmagecr data set into your SAS library as described in the section "Getting Started: SVMACHINE Procedure" on page 933.

The following statements execute the SVM coordinate descent algorithm on the mylib.dmagecr data table and save the training model as mylib.dmageModel. The generated results are listed in Output 35.3.1 through Output 35.3.3.

```
proc svmachine data=mylib.dmagecr
         method=CD regL1=1.0;
   input checking history purpose savings employed marital coapp
         property other job housing telephon foreign/level=nominal;
   input duration amount installp resident existcr depends age/level=interval;
   target good_bad;
   savestate rstore=mylib.dmageModel;
run;
```

The procedure option METHOD=CD defines the training algorithm as coordinate descent. The option REGL1=1.0 specifies that the L1 penalty value is used. The SAVESTATE statement shows that the trained model is saved as mylib.dmageModel.

The "Model Information" table is displayed in Output 35.3.1. From this table, you can see that the optimization technique is coordinate descent and the penalty method is "regL1," as well as other parameter settings.

Output 35.3.1 German Credit Model Information

The SVMACHINE Procedure

Model Information	
Task Type	C_CLAS
Optimization Technique	Coordinate Descent
Scale	YES
Kernel Function	Linear
Penalty Method	regL1
Penalty Parameter	1
Maximum Iterations	100
Tolerance	1e-06
Seed	1

The "Misclassification Matrix" table in Output 35.3.2 shows that among the total of 1,000 observations, 700 observations are classified as good and 300 observations are classified as bad. The number of correctly predicted good observations is 633, and the number of correctly predicted bad observations is 154. Thus the accuracy is 78.70%, as indicated in the "Fit Statistics" table in Output 35.3.3.

Output 35.3.2 German Credit Misclassification Matrix

Misclassification Matrix			
	Training Prediction		
Observed	good	bad	Total
good	633	67	700
bad	146	154	300
Total	779	221	1000

Output 35.3.3 German Credit Accuracy

Fit Statistics	
Statistic	Training
Accuracy	0.7870
Error	0.2130
Sensitivity	0.9043
Specificity	0.5133

Now you can load the data set sampsio.dmagescr into the SAS library and apply the model mylib.dmageModel by using the ASTORE procedure to generate the score results:

```
data mylib.dmagescr;
   set sampsio.dmagescr;
run;

proc astore;
    score data=mylib.dmagescr out=mylib.dmagescore
            rstore=mylib.dmageModel copyvar=(CUSTID);
run;
quit;

proc print data=mylib.dmagescore(obs=10);
run;
```

Output 35.3.4 displays 10 observations from the data table mylib.dmagescore.

Output 35.3.4 German Credit Score Results

Obs	_P_	P_good_badgood	P_good_badbad	I_good_bad	_WARN_	CUSTID
1	-0.13668	0.51525	0.48475	good		649
2	-0.95590	0.60667	0.39333	good		325
3	-0.75900	0.58470	0.41530	good		288
4	-1.93547	0.71598	0.28402	good		858
5	-2.27788	0.75419	0.24581	good		151
6	0.16156	0.47229	0.52771	bad		348
7	-2.19034	0.74442	0.25558	good		212
8	-1.18724	0.63248	0.36752	good		990
9	-1.66171	0.68543	0.31457	good		303
10	-2.73913	0.80566	0.19434	good		166

Example 35.4: Active-Set Method

In this example, you can apply the same home equity loan data as in "Example 35.1: Home Equity Loan Case" on page 965. But instead of using the default interior point method, you can apply the active-set method and save the model to the analytic store mylib.hmeqModel.

The following statements run the SVM active-set algorithm on the mylib.hmeq data table and save the training model as mylib.hmeqModel. The generated results are listed in Output 35.4.1 through Output 35.4.3.

```
proc svmachine data=mylib.hmeq method=ACTIVESET noscale;
   input reason job derog delinq ninq / level=nominal;
   input loan mortdue value yoj clage clno debtinc / level=interval;
   target bad;
   kernel RBF / k_par=1.0;
   savestate rstore=mylib.hmeqModel;
run;
```

The METHOD=ACTIVESET option defines the training algorithm as active set. The SAVESTATE statement shows that the trained model is saved as mylib.hmeqModel.

The "Model Information" table is displayed in Output 35.4.1. From this table, you can see that the optimization technique is active set, the kernel function is RBF, and the kernel parameter value is 1.

Output 35.4.1 HMEQ Model Information

The SVMACHINE Procedure

Model Information	
Task Type	C_CLAS
Optimization Technique	Active Set
Scale	NO
Kernel Function	RBF
RBF Parameter	1
Penalty Method	C
Penalty Parameter	1
Maximum Iterations	25
Tolerance	0.0001

The "Misclassification Matrix" table in Output 35.4.2 shows that among the total of 3,364 training observations, 300 observations are classified as 1 and 3,064 observations are classified as 0. The number of correctly predicted observations in class 1 is 300, and the number of correctly predicted observations in class 0 is 3,064. Thus the accuracy is 100.00%, as indicated in the "Fit Statistics" table in Output 35.4.3.

Output 35.4.2 HMEQ Misclassification Matrix

Misclassification Matrix			
	Training Prediction		
Observed	**1**	**0**	**Total**
1	300	0	300
0	0	3064	3064
Total	300	3064	3364

Output 35.4.3 HMEQ Accuracy

Fit Statistics	
Statistic	**Training**
Accuracy	1.0000
Error	0.0000
Sensitivity	1.0000
Specificity	1.0000

Now you can load the data set into your SAS library, and you can apply the model mylib.hmeqModel to generate the score results by using the ASTORE procedure as specified in "Example 35.3: Coordinate Descent Method" on page 969.

Example 35.5: Reduced SVM Method

In this example, you can apply the SVMACHINE procedure to the same home equity loan data as in "Example 35.1: Home Equity Loan Case" on page 965. But instead of using the default interior point method, you can apply the reduced support vector machine (RSVM) method and save the model to an analytic store.

The following statements run the RSVM algorithm on the mylib.hmeq data table and save the training model to the analytic store mylib.rsvmModel. The results are shown in Output 35.5.1 through Output 35.5.3.

```
proc svmachine data=mylib.hmeq method=RSVM noscale
        sampleType=STRATIFY sampleRate=0.4 seed=1234;
   input reason job derog delinq ninq / level=nominal;
   input loan mortdue value yoj clage clno debtinc / level=interval;
   target bad;
   kernel RBF / k_par=1.0;
   savestate rstore=mylib.rsvmModel;
run;
```

The METHOD=RSVM option defines the training algorithm as reduced SVM. The SAMPLETYPE= option value is STRATIFY, which applies stratified sampling to the target variable bad. The sampling rate is 0.4, which is 40% of valid observations of the training data. The SAVESTATE statement saves the trained model to the analytic store mylib.rsvmModel.

The "Model Information" table is displayed in Output 35.5.1. From this table, you can see that the optimization technique is reduced SVM, the kernel function is RBF, and the kernel parameter value is 1.

Output 35.5.1 HMEQ Model Information

The SVMACHINE Procedure

Model Information	
Task Type	C_CLAS
Optimization Technique	Reduced SVM
Scale	NO
Kernel Function	RBF
RBF Parameter	1
Penalty Method	C
Penalty Parameter	1
Tolerance	0.0001
Seed	1234
Sample Type	STRATIFY
Sample Rate	0.4

The "Misclassification Matrix" table in Output 35.5.2 shows that among the total of 3,364 training observations, 300 observations are in class 1 and 3,064 observations are in class 0. The number of correctly predicted observations in class 1 is 120, and the number of correctly predicted observations in class 0 is 3,064. Thus the accuracy is 94.65%, as indicated in the "Fit Statistics" table in Output 35.5.3.

Output 35.5.2 HMEQ Misclassification Matrix

Misclassification Matrix			
	Training Prediction		
Observed	**1**	**0**	**Total**
1	120	180	300
0	0	3064	3064
Total	120	3244	3364

Output 35.5.3 HMEQ Accuracy

Fit Statistics	
Statistic	**Training**
Accuracy	0.9465
Error	0.0535
Sensitivity	0.4000
Specificity	1.0000

Now you can load the data set into your SAS library, and you can apply the model mylib.rsvmModel to generate the score results by using the ASTORE procedure as specified in "Example 35.3: Coordinate Descent Method" on page 969.

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

Example 35.6: Mitigating Bias of a Support Vector Classification

This example uses the SVMACHINE procedure to create a support vector classification model while minimizing the equal opportunity bias for the Reason variable. The response variable for the support vector classification model is Bad, a classification variable that has two values (0 for payment of loan and 1 for default). The other variables are predictor variables for the model.

The following statements load the data table mylib.hmeq into your SAS library. For this example, the statements assume that your SAS library is named mylib, but you can substitute any appropriately defined SAS library.

```
data mylib.hmeq;
   set sampsio.hmeq;
run;
```

The following statements use PROC SVMACHINE to create a support vector classification model without minimizing the equal opportunity bias:

```
proc svmachine data=mylib.hmeq c=0.1;
   input clage clno debtinc loan mortdue value yoj delinq derog job ninq;
   input delinq derog job ninq / level=nominal;
   target bad;
   output out=mylib.scored copyvars=(_ALL_);
   savestate rstore=mylib.model;
run;
```

The C= option specifies 0.1 as the penalty for training. The two INPUT statements specify clage, clno, debtinc, loan, mortdue, value, yoj, delinq, derog, job, and ninq as the input variables. The TARGET statement specifies bad as the target variable. The OUTPUT statement generates the data table mylib.scored, which contains the predicted mortgage default from the model for each observation and contains all variables from the input data table as copy variables. The SAVESTATE statement creates an analytic store for the model and saves it as a binary object in the data table mylib.model.

Output 35.6.1 displays the fit statistics for the classification model.

Output 35.6.1 Fit Statistics Table

The SVMACHINE Procedure

Fit Statistics	
Statistic	**Training**
Accuracy	0.9156
Error	0.0844
Sensitivity	0.0584
Specificity	1.0000

After you train a support vector classification model, you might want to view the equal opportunity bias of the trained model. You can do this by running the ASSESSBIAS procedure. For more information about PROC ASSESSBIAS, see Chapter 3, "ASSESSBIAS Procedure."

The following statements use PROC ASSESSBIAS to assess the bias measurements of the model:

```
proc assessbias data=mylib.scored;
   target bad / event="1" level=nominal;
   var P_Bad1;
   fitstat pvar=P_BAD0 / pevent="0";
   sensitiveVar reason;
run;
```

Output 35.6.2 displays the bias metric values of the model. PROC ASSESSBIAS also produces other output not shown here.

Output 35.6.2 Bias Metrics Table

The ASSESSBIAS Procedure

Bias Metrics					
Bias Statistic	**Bias Statistic Label**	**Bias Statistic Value**	**Base Level**	**Compare Level**	**Note**
DemographicParity	Demographic Parity (Statistical Parity)	0.0165		HomeImp	
PredictiveParity	Predictive Parity	0.1373		HomeImp	
EqualAccuracy	Equal Accuracy	0.0344	DebtCon	HomeImp	
EqualizedOdds	Equalized Odds	0.0242	DebtCon	HomeImp	The maximum TPR difference is greater than the maximum FPR difference.
EqualOpportunity	Equal Opportunity	0.0242	DebtCon	HomeImp	

In order to use the MITIGATEBIAS statement to reduce the equal opportunity bias, you must specify the levels of the response (target) variable and the corresponding predicted variables. You can use the DESCRIBE statement in the ASTORE procedure to obtain this information. The following statements use PROC ASTORE to describe the model:

```
proc astore;
   describe rstore=mylib.model;
run;
```

Output 35.6.3 lists the output variables created by the model. The predicted variables are P_BAD1 and P_BAD0. Their corresponding response (target) levels are 1 and 0, respectively.

Output 35.6.3 Output Variables Table

The ASTORE Procedure

Output Variables			
Name	**Length**	**Type**	**Label**
_P_	8	Num	Decision Function
P_BAD1	8	Num	Predicted: BAD=1
P_BAD0	8	Num	Predicted: BAD=0
I_BAD	12	Character	Into: BAD
WARN	4	Character	Warnings

The following statements use PROC SVMACHINE to create a support vector classification model, while minimizing the equal opportunity bias for reason by using the MITIGATEBIAS statement:

```
ods graphics on;
proc svmachine data=mylib.hmeq c=0.1;
   input clage clno debtinc loan mortdue value yoj delinq derog job ninq;
   input delinq derog job ninq / level=nominal;
   target bad;
   mitigatebias biasmetric=EQUALOPPORTUNITY bound=75 pevents="1 0"
      pvars=(P_BAD1 P_BAD0) seed=12345 sensitivevar=reason targetevent="1";
   output out=mylib.scored copyvars=(_ALL_);
   savestate rstore=mylib.model;
run;
```

The BIASMETRIC= option specifies that equal opportunity be used as the bias metric. The BOUND= option specifies 75 as the bound value for the EGR algorithm. The PEVENTS= option lists the levels of the response (target) variable bad. The predicted variables correspond to the levels that are listed in the PVARS= option. The PVARS= option lists the predicted variables that contain the posterior probability for each level in model prediction that corresponds to the response (target) variable Bad. The SEED= option specifies the seed for the pseudorandom number generator to be 12345. The SENSITIVEVAR= option names the variable reason as the sensitive variable for the purpose of reducing the value of the bias measurement. The TARGETEVENT= option specifies the value 1 as the event of interest.

Output 35.6.4 displays the information about the options that are used for bias mitigation. Output 35.6.5 displays the model accuracy and bias metric values over each iteration of the EGR algorithm. Output 35.6.6 shows the iteration history, which is a graphical representation of the iteration history table. The graph displays the bias metric and the misclassification rate for each iteration. PROC SVMACHINE selects the iteration that has the lowest bias metric. In this case, the 8th iteration is selected; the graph indicates this selection by using the legend and a vertical line. The MITIGATEBIAS statement also creates other tables, which are not shown here but are listed in the section "ODS Table Names" on page 962.

Output 35.6.4 Mitigation Information Table

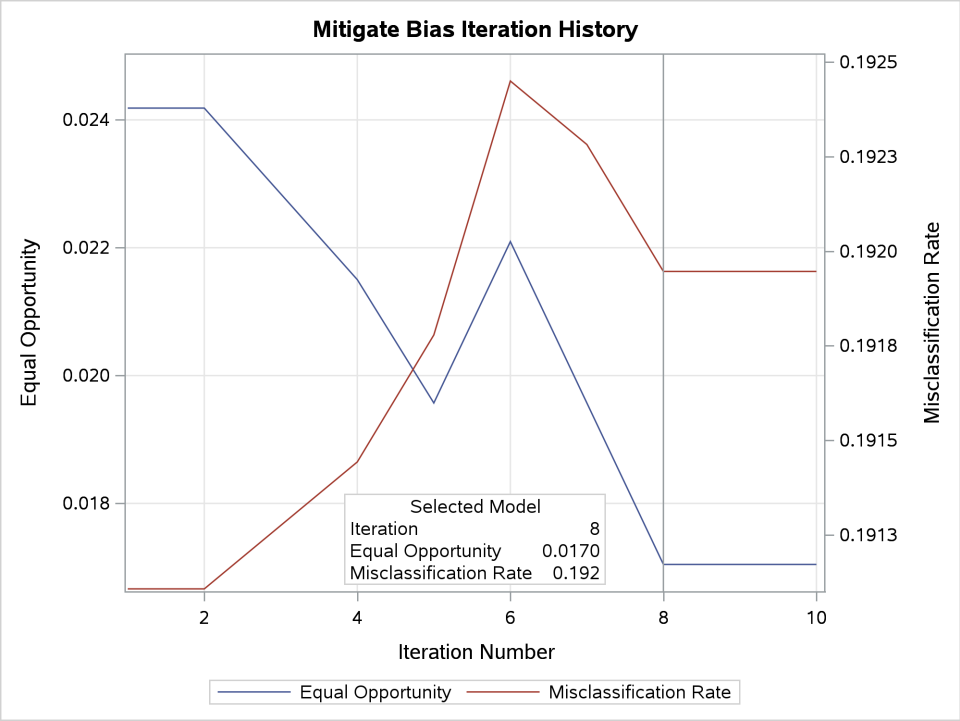
The SVMACHINE Procedure

Mitigation Information		
Description	**Character Value**	**Value**
Bias Metric	EQUALOPPORTUNITY	.
Sensitive Variable	REASON	.
Response	BAD	.
Bound Value	75	75
Learning Rate	0.01	0.01
Maximum Iteration	10	10
Tolerance	0.005	0.005
Event Level	1	.

Output 35.6.5 Iteration History Table

EGR Iteration History			
Iteration	**Equal Opportunity**	**Misclassification Rate**	**Multi-Class Log Loss**
1	0.024185	0.1911	0.7700
2	0.024185	0.1911	0.7641
3	0.022843	0.1913	0.7673
4	0.021500	0.1914	0.7757
5	0.019571	0.1918	0.7790
6	0.022096	0.1924	0.7692
7	0.019571	0.1923	0.7820
8	0.017045	0.1919	0.7867
9	0.017045	0.1919	0.7875
10	0.017045	0.1919	0.7904

Output 35.6.6 Iteration History Graph



NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

Example 35.7: Tuning the Hyperparameters of a Support Vector Machine

This example illustrates how you can use the AUTOTUNE statement in the SVMACHINE procedure to tune the hyperparameters of a support vector machine (SVM) model. It uses the home equity data set Hmeq, which is in the Sampsio library.

The Hmeq data set contains 5,960 observations with 13 variables. The response variable, Bad, is a binary indicator to predict which clients will default on their loan.

The following statements load the data table Hmeq into your SAS library. For this example, the statements assume that your SAS library is named mylib, but you can substitute any appropriately defined SAS library.

```
data mylib.hmeq;
   set sampsio.hmeq;
run;
```

The following statements use PROC SVMACHINE with the AUTOTUNE statement to automatically tune the hyperparameters of an SVM model:

```
proc svmachine data=mylib.hmeq;
   input clage clno debtinc loan mortdue value yoj delinq derog job ninq;
   input delinq derog job ninq / level=nominal;
   target bad;
   autotune
      /* Tuning Parameters
      You do not need to specify any tuning parameters for the default
      tuning process. If you want to make adjustments to the default
      tuning process, uncomment the following block of code and change
      any of the tuning parameters' attributes.

      tuningParameters=(
         C                  ( lb=1e-10 ub=100   init=1.0  )
         DEGREE             ( lb=2     ub=3     init=2    )
         KERNEL_METHOD      ( values=LINEAR_IPOINT POLYNOMIAL_IPOINT
            LINEAR_CD LINEAR_ACTIVESET POLYNOMIAL_ACTIVESET
            RBF_ACTIVESET SIGMOID_ACTIVESET init=LINEAR_IPOINT )
         RBFPARAMETER       ( lb=0.1   ub=100   init=0.1  )
         SIGMOIDPARAMETER1  ( lb=0.1   ub=10.0  init=0.1  )
         SIGMOIDPARAMETER2  ( lb=-10   ub=-0.1  init=-0.1 )
         REGL2              ( lb=0.1   ub=100   init=0.1  )
      )
      */
   ;
   /* Remove this line to see all results */
   ods select BestConfiguration EvaluationHistoryPlot IterationHistoryPlot;
run;
```

The preceding statements produce the table and the plots shown in Output 35.7.1 through Output 35.7.3. The table in Output 35.7.1 displays the evaluation number, the values of the tuning parameters, and the error metric value for the best SVM model that the tuner found. Note that the ODS SELECT statement limits the displayed results to a single table and two plots. You can remove this statement to display all tables and plots.

For the full list of ODS tables that PROC SVMACHINE produces, see Table 35.3.

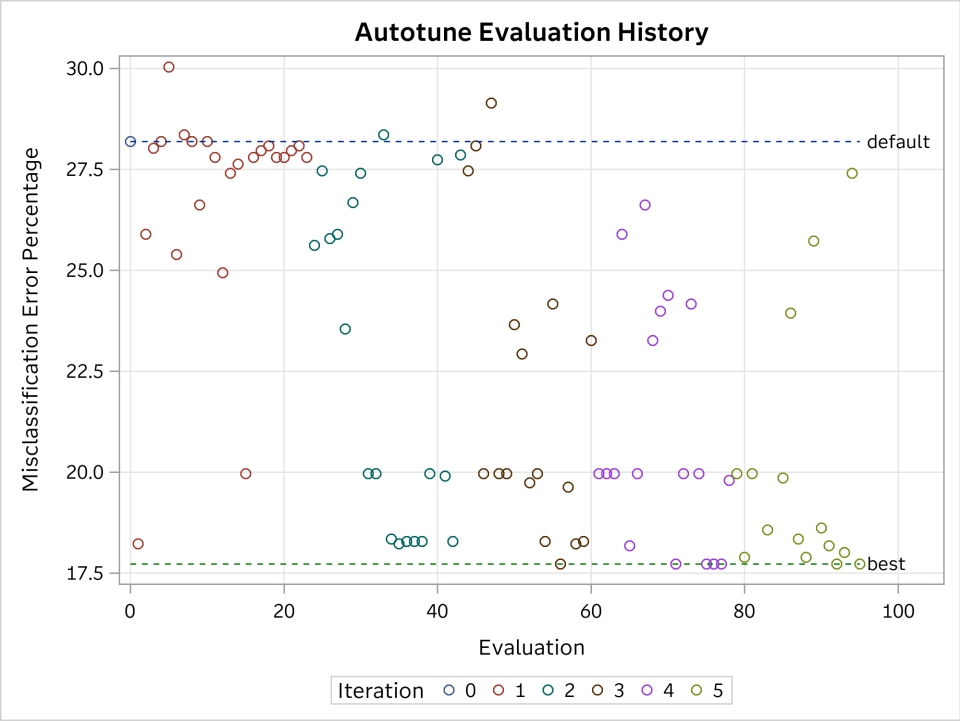
Output 35.7.1 Best Configuration Table

The SVMACHINE Procedure

Best Configuration	
Evaluation	56
Kernel/Method	SIGMOID_ACTIVESET
Penalty (C)	63.8888889
Sigmoid Parameter 1	1.2
Sigmoid Parameter 2	-5.6
Misclassification Error Percentage	17.73

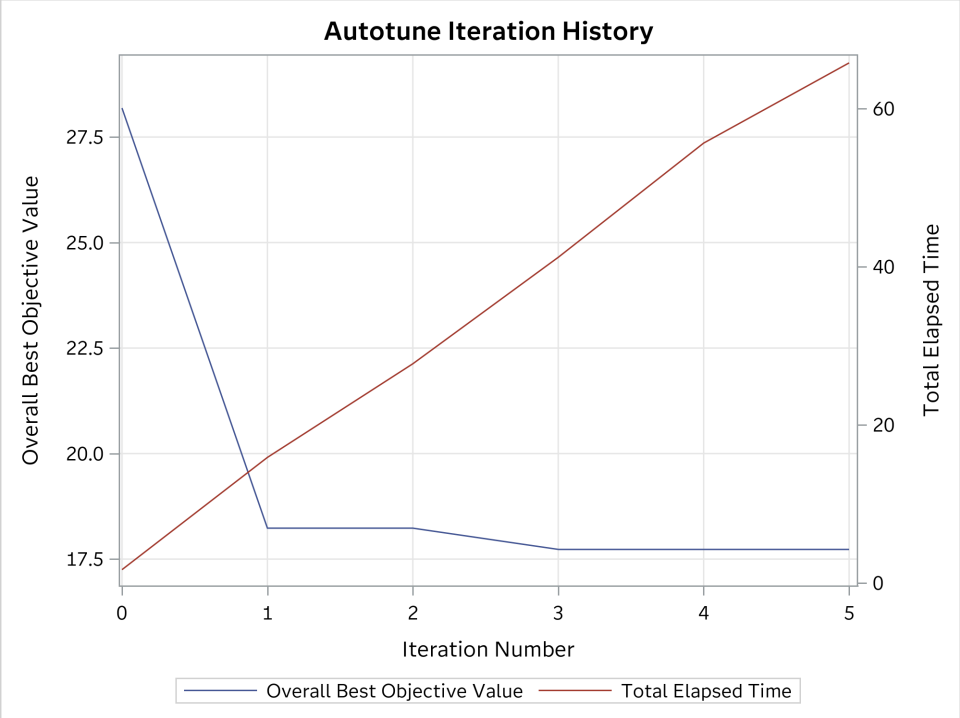
Output 35.7.2 displays a scatter plot of all configurations that the tuner tried. The objective values are shown on the Y axis, and the evaluation numbers are shown on the X axis.

Output 35.7.2 Evaluation History Plot



The plot in Output 35.7.3 displays how the best found objective value and the elapsed time changed with each iteration of the tuner.

Output 35.7.3 Iteration History Plot



References

Agarwal, A., Beygelzimer, A., Dudik, M., Langford, J., and Wallach, H. (2018). "A Reductions Approach to Fair Classification." In *Proceedings of the 35th International Conference on Machine Learning*, 60–69. PMLR.

Burges, C. J. C. (1998). "A Tutorial on Support Vector Machines for Pattern Recognition." *Data Mining and Knowledge Discovery* 2:121–167.

Cristianini, N., and Shawe-Taylor, J. (2000). *An Introduction to Support Vector Machines and Other Kernel-Based Learning Methods*. New York: Cambridge University Press.

Gertz, E. M., and Griffin, J. D. (2005). *Support Vector Machine Classifiers for Large Data Sets*. Technical Report ANL/MCS-TM-289, Mathematics and Computer Science Division, Argonne National Laboratory.

Gertz, E. M., and Griffin, J. D. (2010). "Using an Iterative Linear Solver in an Interior-Point Method for Generating Support Vector Machines." *Computational Optimization and Applications* 47:431–453.

Hsieh, C.-J., Chang, K.-W., Lin, C.-J., Keerthi, S. S., and Sundararajan, S. (2008). "A Dual Coordinate Descent Method for Large-Scale Linear SVM." In *Proceedings of the 25th International Conference on Machine Learning*, 408–415. New York: ACM.

Omheni, R., and Griffin, J. D. (2025). "Distributed nonlinear support vector machines." US Patent 12,353,501.

Smola, A. J., and Schölkopf, B. (2004). "A Tutorial on Support Vector Regression." *Statistics and Computing* 14:199–222. https://doi.org/10.1023/B:STCO.0000035301.49549.88.

Vapnik, V. N. (1995). *The Nature of Statistical Learning Theory*. New York: Springer-Verlag.

Chapter 36
TABULARGAN Procedure

Contents

Overview: TABULARGAN Procedure **984**
PROC TABULARGAN Features 984
Using SAS Viya Workbench 984
Getting Started: TABULARGAN Procedure **985**
Syntax: TABULARGAN Procedure **988**
PROC TABULARGAN Statement 988
AEOPTIMIZATION Statement 989
GANOPTIMIZATION Statement 990
GMM Statement 991
INPUT Statement 992
OUTPUT Statement 993
SAVESTATE Statement 993
TRAIN Statement 993
Details: TABULARGAN Procedure **995**
The CPCTGAN Model 995
Centroid Information 999
Displayed Output 999
Iteration History 999
Level Frequencies for Nominal Variables 999
Model Information 999
Number of Observations 999
Table Output 1000
ODS Table Names 1000
Examples: TABULARGAN Procedure **1000**
Example 36.1: Analyzing Home Mortgage Data 1001
Example 36.2: Analyzing Home Mortgage Data with a Centroids Table 1003
References **1006**

Overview: TABULARGAN Procedure

The TABULARGAN procedure trains a correlation-preserving conditional tabular generative adversarial network (CPCTGAN) model on tabular data in SAS Viya Workbench. The procedure implements the CPCTGAN model by using the PyTorch library.

PROC TABULARGAN Features

The TABULARGAN procedure has the following main features:

- It generates synthetic tabular data that preserve the pairwise correlations between variables from the original data.
- It supports multithreaded training on one graphics processing unit (GPU) device.

Using SAS Viya Workbench

SAS Viya Workbench has greatly simplified the SAS programming experience by offering a syntax style nearly identical to that of SAS 9 and earlier releases. However, there are minor exceptions. In the past, SAS Viya procedures required the use of a CAS engine libref. This engine served multiple purposes, but it was primarily used to upload data to the CAS server and manage the data. Its additional purposes included support of new data types and improved performance for large data sets. No CAS server is needed for SAS Viya Workbench, because all computations are done in place for you in a single location.

In the following program, the LIBNAME statement creates a mylib libref that the examples in this chapter use to refer to your current or desired work directory:

```
libname mylib ".";
```

For more information, see *SAS Viya Workbench: Getting Started*.

Getting Started: TABULARGAN Procedure

This example shows how to use the TABULARGAN procedure to analyze and augment the Fish data set in the Sashelp library. The Fish data set contains 159 observations and seven variables. Among these variables, Species is a nominal variable that contains the names of various fish species. The other six variables are interval variables that contain statistics of the fish: height, weight, width, and three measures of length. The following DATA step creates the input data table mylib.fish in your SAS library. These statements assume that your SAS libref is named mylib, as in the section "Using SAS Viya Workbench" on page 984, but you can substitute any appropriately defined SAS libref.

```
data mylib.fish;
   set sashelp.fish;
run;
```

The following PROC TABULARGAN statement trains a tabular GAN model and saves the trained model in an analytic store:

```
proc tabulargan          data=mylib.fish seed=123 numSamples=10 useGPU;
     input               weight length1 length2 length3/level=interval;
     input               species/level=nominal;
     aeoptimization      numEpochs=3;
     ganoptimization     numEpochs=5;
     train               miniBatchSize=50;
     savestate           rstore=mylib.astore;
     output              out=mylib.out;
run;
```

The interval and nominal input variables are determined using the INPUT statements. You specify the number of epochs for training the autoencoder models in the AEOPTIMIZATION statement. Similarly, you specify the number of epochs for training the generative adversarial network (GAN) models in the GANOPTIMIZATION statement. Note that the typical number of epochs for the autoencoder and GAN models should be greater than 100 for proper model training. The appropriate number of epochs, specified in the NUMEPOCHS= option, should be found for each use case. The SAVESTATE statement specifies the table to use for saving the trained model, and the OUTPUT statement specifies the table to use for output results.

Figure 36.1 shows the iteration history, including the autoencoder loss value and the generator and discriminator loss values for each epoch of training.

Figure 36.1 Iteration History

The TABULARGAN Procedure

Iteration History			
Epoch Number	Autoencoder Loss	Generator Loss	Discriminator Loss
1	0.345445	.	.
2	0.282496	.	.
3	0.233644	.	.
1	.	1.957608	0.061005
2	.	1.930210	-0.032211
3	.	1.951218	0.016106
4	.	1.956728	-0.006330
5	.	1.991888	0.003569

Figure 36.2 shows the number of observations that are read and the number of observations that are used. Because the mylib.fish data table contains missing values, the number of observations read does not equal the number of observations used in this example. When the input data table contains missing values, PROC TABULARGAN ignores the observations (rows) that have missing values in the input data table and does not use them in the analysis.

Figure 36.2 Number of Observations

Number of Observations	
Number of Observations Read	159
Number of Observations Used	158

Figure 36.3 shows the values of the parameters that are used in the tabular GAN model.

Figure 36.3 Model Information

Model Information	
Generator Embedding Dimension	128
Number of Observations in One Minibatch	50
Number of Observations Group Together in Applying the Discriminator	10
Weight for Regularizing the Discriminator	10
Exponential Decay Rate for the First-Moment Estimates for the Autoencoder's Optimizer	0.9
Exponential Decay Rate for the Second-Moment Estimates for the Autoencoder's Optimizer	0.999
Learning Rate for the Autoencoder's Optimizer	0.001
Number of Epochs for the Autoencoder's Training	3
Weight Decay for the Autoencoder's Optimizer	1E-8
Exponential Decay Rate for the First-Moment Estimates to Optimize GAN	0.5
Exponential Decay Rate for the Second-Moment Estimates to Optimize GAN	0.9
Learning Rate for the GAN Optimizer	0.00002
Number of Epochs for the GAN Training	5
Weight Decay for the Generator's Optimizer	0.0001
Weight Decay for the Discriminator's Optimizer	1E-6
Seed for Random Initialization	123
Whether to Use Log Frequency of Categorical Levels in the Conditional Sampling	True

Figure 36.4 shows the level names and frequencies of the nominal variable Species.

Figure 36.4 Level Frequencies of Nominal Variables

Level Frequencies for Nominal Variables		
Variable Name	Level	Frequency
Species	Bream	34
Species	Parkki	11
Species	Perch	56
Species	Pike	17
Species	Roach	20
Species	Smelt	14
Species	Whitefish	6

You can generate additional synthetic data by using the trained tabular GAN model that is saved in the mylib.astore table and PROC ASTORE (see Chapter 4, "ASTORE Procedure").

The following statements use PROC ASTORE to generate 100 synthetic data observations by using the trained tabular GAN model that is stored in mylib.astore and then save the results in the mylib.astoreout table:

```
proc astore;
setoption GENOBS 100;
setoption SEED 42;
score rstore=mylib.astore
      out=mylib.astoreout;
run;
```

The GENOBS option in the SETOPTION statement specifies the number of synthetic observations to generate. To generate different synthetic data, change the SEED option value in the SETOPTION statement.

NOTE: For tabular GAN analytic stores that are trained on the 2026.02 release or later releases, you can generate a specific number of observations without input data by specifying the GENOBS option when you call PROC ASTORE. However, for tabular GAN analytic stores that are trained on releases earlier than 2026.02, you must provide an input data set that contains the number of rows that corresponds to the number of observations that you want to generate. See the corresponding document version for details.

Syntax: TABULARGAN Procedure

The following statements are available in the TABULARGAN procedure:

PROC TABULARGAN < *options* > **;**
  AEOPTIMIZATION < **ADAM***(suboptions)* > < *options* > **;**
  GANOPTIMIZATION < **ADAM***(suboptions)* > < *options* > **;**
  GMM < *options* > **;**
  INPUT *variables* < / **LEVEL=NOMINAL** | **INTERVAL** > **;**
  OUTPUT OUT=*libref.data-table* **;**
  SAVESTATE RSTORE=*libref.data-table* **;**
  TRAIN < *options* > **;**

The PROC TABULARGAN, INPUT, OUTPUT, and SAVESTATE statements are required. The INPUT statement can appear multiple times.

The rest of this section provides detailed syntax information about each of the preceding statements, beginning with the PROC TABULARGAN statement. The remaining statements are described in alphabetical order.

PROC TABULARGAN Statement

PROC TABULARGAN < *options* > **;**

The PROC TABULARGAN statement invokes the procedure. Table 36.1 summarizes the options in this statement.

Table 36.1 PROC TABULARGAN Statement Options

Option	Description
DATA=	Specifies the name of the input table
DETERMINISTIC	Specifies deterministic CuDNN implementations
NUMSAMPLES=	Specifies the number of observations to generate
SCORESEED=	Specifies the random number seed to use for GAN scoring
SEED=	Specifies the random number seed to use for GAN training
USEGPU	Specifies that a GPU device is to be used
USEGPU(DEVICE=)	Specifies the ID of the GPU device that is to be used

You can specify the following *options*:

DATA=*libref.data-table*
names the input data table for PROC TABULARGAN to use. The default is the most recently created data table. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to the data. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 984.

data-table specifies the name of the input data table.

DETERMINISTIC
tells the NVIDIA CUDA Deep Neural Network library (CuDNN) that you want only the deterministic implementations. **NOTE:** Specifying this option can significantly slow down the training process.

NUMSAMPLES=*number*
specifies the number of sample observations to generate.

By default, NUMSAMPLES=1000.

SCORESEED=*number*
specifies the seed to use for the random number generator for scoring. If you do not specify a seed or if you specify a value less than or equal to 0, the seed is generated by reading the time of day from the computer's clock.

SEED=*number*
specifies the seed to use for the random number generator for training. If you do not specify a seed or if you specify a value less than or equal to 0, the seed is generated by reading the time of day from the computer's clock.

USEGPU(*suboptions***)**
specifies that a GPU device is to be used. You can specify the following *suboption*:

DEVICE=*id-number*
specifies the ID number of the GPU device that is to be used.

By default, DEVICE=0.

AEOPTIMIZATION Statement

AEOPTIMIZATION < **ADAM***(suboptions)* > < *options* > **;**

The AEOPTIMIZATION statement specifies the optimizer parameters to use in training the autoencoder model.

You can specify the following *options*:

ADAM (BETA1=*number* **BETA2=***number***)**
specifies the optimization method to use in training the autoencoder model. Currently only the Adam method is supported. You must specify the keyword ADAM when specifying the following suboptions:

BETA1=*number*
specifies the exponential decay rate for the first-moment estimates.

By default, BETA1=0.9.

BETA2=*number*
specifies the exponential decay rate for the second-moment estimates.

By default, BETA2=0.999.

LEARNINGRATE=*number*
specifies the learning rate for the autoencoder's optimizer.

By default, LEARNINGRATE=0.001.

NUMEPOCHS=*number*
specifies the number of epochs to use in training the autoencoder model. The *number* must be a positive integer.

By default, NUMEPOCHS=150.

WEIGHTDECAY=*number*
specifies the weight decay for the autoencoder's optimizer.

By default, WEIGHTDECAY=0.00000001.

GANOPTIMIZATION Statement

GANOPTIMIZATION < **ADAM***(suboptions)* > < *options* > **;**

The GANOPTIMIZATION statement specifies the optimizer parameters to use in training the GAN model. You can specify the following *options*:

ADAM (BETA1=*number* **BETA2=***number***)**
specifies the optimization method to use in training the GAN model. Currently only the Adam method is supported. You must specify the keyword ADAM when specifying the following suboptions:

BETA1=*number*
specifies the exponential decay rate for the first-moment estimates.

By default, BETA1=0.5.

BETA2=*number*
specifies the exponential decay rate for the second-moment estimates.

By default, BETA2=0.9.

LEARNINGRATE=*number*
specifies the learning rate for the optimizer.

By default, LEARNINGRATE=0.00002.

NUMEPOCHS=*number*
specifies the number of epochs to use in training the model GAN model. The *number* must be a positive integer.

By default, NUMEPOCHS=150.

WEIGHTDECAYD=*number*
specifies the weight decay for the discriminator's optimizer.

By default, WEIGHTDECAYD=0.0001.

WEIGHTDECAYG=*number*
specifies the weight decay for the generator's optimizer.

By default, WEIGHTDECAYG=0.000001.

GMM Statement

GMM < *options* > **;**

The GMM statement specifies the Gaussian mixture model (GMM) parameters to be used to calculate the centroid information for the interval variables.

Table 36.2 summarizes the options in this statement.

Table 36.2 GMM Statement Options

Option	Description
ALPHA=	Specifies the concentration parameter for the Dirichlet process
CENTROIDSTABLE=	Specifies the table that contains the centroid information
MAXCLUSTERS=	Specifies the GMM maximum number of clusters
SEED=	Specifies the random number seed to use for GMM training
VB	Specifies the inference parameters for the variational Bayesian (VB) model
VB(COVARIANCE=)	Specifies the GMM covariance matrix type
VB(MAXVBITER=)	Sets the maximum number of iterations for VB inference
VB(THRESHOLD=)	Specifies the GMM convergence threshold

You can specify the following *options*:

ALPHA=*number*
specifies the concentration parameter for the Dirichlet process.

By default, ALPHA=1.0.

CENTROIDSTABLE=*libref.data-table*
specifies the table that contains Gaussian mixture model (GMM) centroid information for continuous variables. Providing this table is optional. When you specify this option, PROC TABULARGAN does not calculate the centroid information; instead it uses the specified table and ignores the remaining options. However, if you omit this option, you can use the MAXCLUSTERS=, SEED=, and VB options to tune the GMM parameters.

libref.data-table is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the input data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 984.

MAXCLUSTERS=*number*
specifies the GMM maximum number of clusters.

By default, MAXCLUSTERS=100.

SEED=*number*
specifies a value to use for starting the pseudorandom number generator for initialization. If you do not specify a seed or if you specify a value less than or equal to 0, the seed is generated by reading the time of day from the computer's clock.

VB(*suboptions***)**
specifies the inference parameters for the variational Bayesian (VB) model. You must specify the keyword VB, followed by one or more of these *suboptions*:

COVARIANCE=DIAGONAL | FULL
specifies the GMM covariance matrix type.

DIAGONAL
specifies the covariance matrices of the Gaussian distributions as diagonal matrices.

FULL
specifies the covariance matrices of the Gaussian distributions as full matrices.

By default, COVARIANCE=DIAGONAL.

MAXVBITER=*number*
sets the maximum number of iterations for VB inference.

By default, MAXVBITER=100.

THRESHOLD=*number*
specifies the GMM convergence threshold.

By default, THRESHOLD=0.01.

INPUT Statement

INPUT *variables* < / **LEVEL=INTERVAL | NOMINAL** > **;**

The INPUT statement names input *variables* that share a common option. You can specify the INPUT statement multiple times.

You can specify the following option:

LEVEL=INTERVAL | NOMINAL
specifies the level of measurement of the variables. You can specify the following values:

INTERVAL specifies that the level of measurement of the variables is interval.

NOMINAL specifies that the level of measurement of the variables is nominal.

By default, LEVEL=INTERVAL for numeric variables and LEVEL=NOMINAL for categorical variables.

OUTPUT Statement

OUTPUT OUT=*libref.data-table* **;**

The OUTPUT statement creates an output data table that contains the results of PROC TABULARGAN.

OUT=*libref.data-table*
specifies the output table in which to store the generated tabular data from the trained model. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 984.

SAVESTATE Statement

SAVESTATE RSTORE=*libref.data-table* **;**

The SAVESTATE statement creates an analytic store for the model and saves it as a binary object in a data table. You can use the analytic store in the ASTORE procedure to score new data. For more information, see Chapter 4, "ASTORE Procedure."

You must specify the following option:

RSTORE=*libref.data-table*
specifies a data table in which to save the analytic store for the model. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 984.

TRAIN Statement

TRAIN < *options* > **;**

The TRAIN statement specifies the training parameters to use for training the tabular GAN model.

Table 36.3 summarizes the options in this statement.

Table 36.3 TRAIN Statement Options

Option	Description
CHECKPOINTFREQ=	Specifies the frequency at which to save a checkpoint during training
CHECKPOINTIN=	Specifies the checkpoint table to be loaded to restore your model
CHECKPOINTOUT=	Specifies the table in which to save the current model as a checkpoint
EMBEDDINGDIM=	Specifies the dimensions of the input noise to the generator
MINIBATCHSIZE=	Specifies the number of observations in one minibatch
PACKSIZE=	Specifies the number of samples to group together in applying the discriminator
PRINTFREQ=	Specifies the frequency at which to print losses to the log

Table 36.3 *continued*

Option	Description
REGR1WEIGHT=	Specifies the weight to use in regularizing the discriminator
USEORIGLEVELFREQ	Uses the log-frequency of categorical levels in the conditional sampling

You can specify the following *options*:

CHECKPOINTFREQ=*number*
specifies the frequency at which to save a checkpoint during training.

By default, CHECKPOINTFREQ=5000.

CHECKPOINTIN=*libref.data-table*
specifies the checkpoint table to be loaded to restore your model. If you omit this parameter, the model is trained from scratch. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 984.

CHECKPOINTOUT=*libref.data-table*
specifies the table in which to save the current model as a checkpoint. *libref.data-table* is a two-level name, where *libref* refers to the library, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the DATA= option and the section "Using SAS Viya Workbench" on page 984.

EMBEDDINGDIM=*number*
specifies the dimensions of the input noise to the generator.

By default, EMBEDDINGDIM=128.

MINIBATCHSIZE=*number*
specifies the number of observations in one minibatch. The *number* should be smaller than the total number of observations in the input data set.

By default, MINIBATCHSIZE=500.

PACKSIZE=*number*
specifies the number of samples to group together in applying the discriminator. The *number* must be a factor of the value that you specify in the MINIBATCHSIZE= option.

By default, PACKSIZE=10.

PRINTFREQ=*number*
specifies the frequency at which to print losses to the log and the iteration history table.

By default, PRINTFREQ=1.

REGR1WEIGHT=*number*
specifies the weight to use in regularizing the discriminator.

By default, REGR1WEIGHT=10.

USEORIGLEVELFREQ
uses the log-frequency of categorical levels in the conditional sampling.

Details: TABULARGAN Procedure

The success of generative adversarial networks (GANs) in generating realistic images is well known. However, you can also apply GANs in generating tabular data. Tabular data are one of the most common and important data types. Data such as clinical trial records, financial transaction data, and census data are all represented in tabular format. It is crucial to be able to generate synthetic data sets in which sensitive attributes and personal information are not disclosed. The correlation-preserving conditional tabular GAN (CPCTGAN) model is one of the generative models that you can use for this purpose.

The CPCTGAN Model

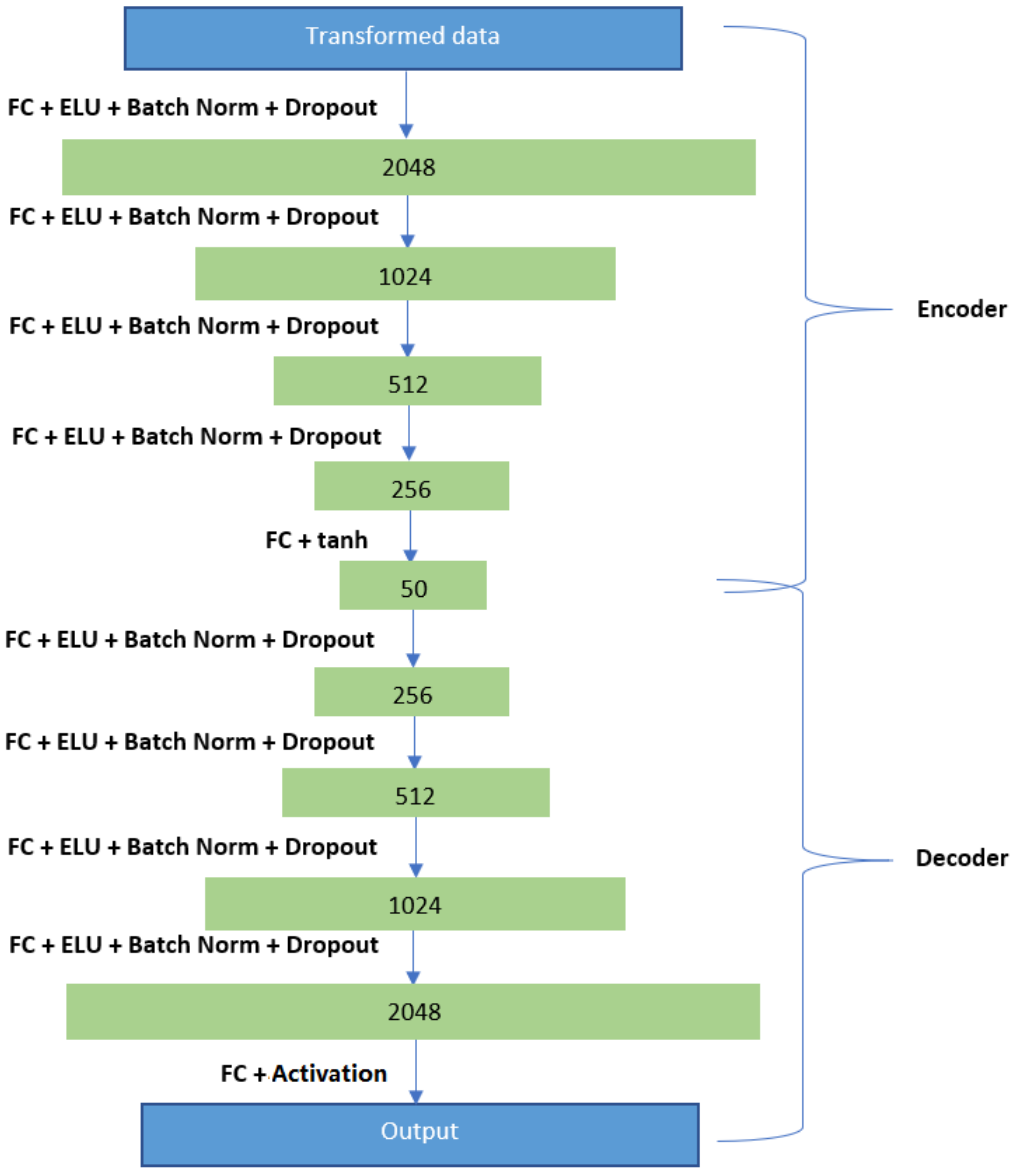
The CPCTGAN model combines the concepts of the conditional tabular GAN (CTGAN) (Xu et al. 2019) model and the correlation-capturing GAN (CorGAN) (Torfi and Fox 2020) model. From the CTGAN model, the CPCTGAN model inherits the data transformation, conditional generator, and training-by-sampling mechanisms. From the CorGAN model, it adopts the mechanism to pretrain an autoencoder model and then appends the pretrained decoder to the generator in the GAN model training.

The CPCTGAN model's training and synthetic data generation involve the following steps:

1. It transforms training data by discretizing the interval variables and using one-hot encoding for nominal variables. The discretization of interval variables is achieved by using the centroids table that contains centroid information for each interval variable. For the detailed transformation formulas, see Torfi and Fox (2020).

2. It trains an autoencoder model on the transformed data by using the reconstruction loss. The mean square error is used as the loss criterion. Figure 36.5 shows the detailed architecture of the autoencoder.

Figure 36.5 Detailed Architecture of the Autoencoder



Each green box in the figure represents a hidden layer, and the number in the box represents the output dimension of the corresponding layer. "FC" means that the layer is fully connected. "Batch Norm" means that the layer uses batch normalization. "Dropout" means that the layer uses the dropout technique in which randomly selected neurons are ignored during training. "ELU" means that the layer uses the exponential linear unit function as the activation function. The abbreviation "tanh" indicates that the layer uses the hyperbolic tangent activation function. "Activation" indicates that the layer uses a specialized activation function (a mix of the tanh and softMax functions).

3. It appends the pretrained decoder from the last step to the generator in the CTGAN model to form the CPCTGAN model, and then it trains the CPCTGAN model on the transformed data by using the same mechanism that the CTGAN model training uses. Specifically, the conditional generator is

trained to generate synthetic rows, each of which is conditioned on a conditional vector *cond*, which represents one randomly chosen category in one of the discrete columns. The output of the conditional generator is then fed into the decoder that maps the output to the transformed data space. With the training-by-sampling mechanism, the conditional vector *cond* and the training data are sampled according to the frequency or log-frequency of each category, so the GAN model can evenly explore all possible discrete values. The Wasserstein loss (Gulrajani et al. 2017) is used in this step to train the GAN model. Figure 36.6 shows the architecture of the CPCTGAN model.

Figure 36.6 CPCTGAN Model Architecture

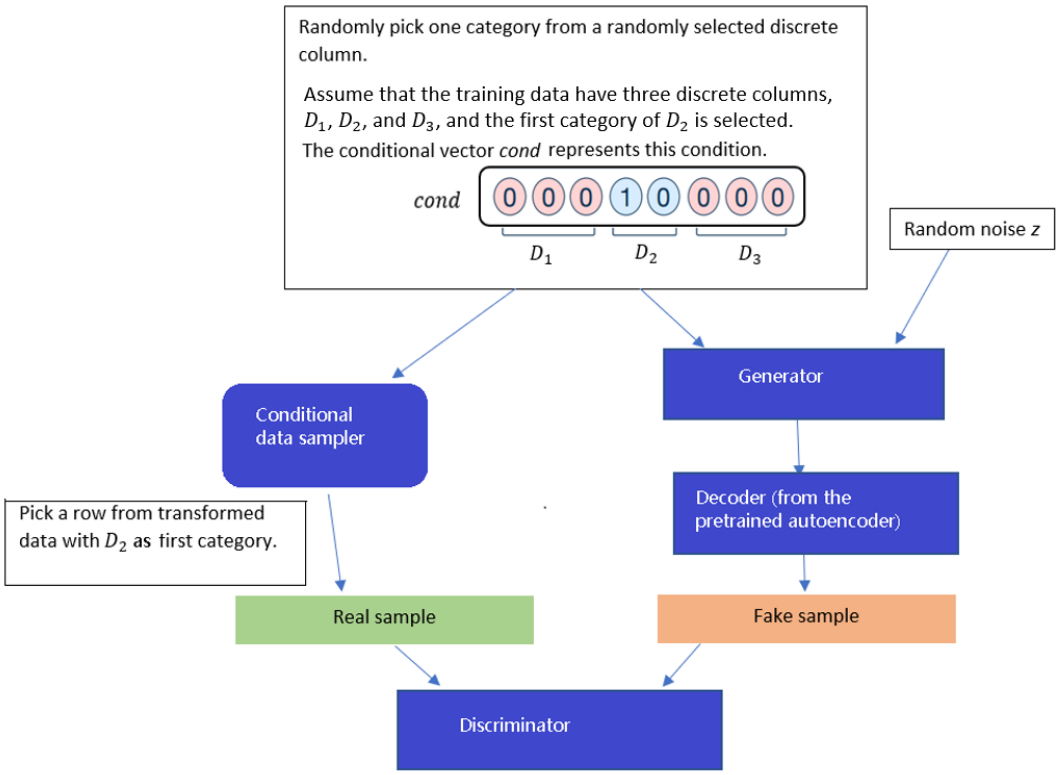


Figure 36.7 shows the detailed architecture of the generator in the CPCTGAN model, where ReLU means that the layer uses the rectified linear unit activation function.

Figure 36.7 Detailed Architecture of the Generator

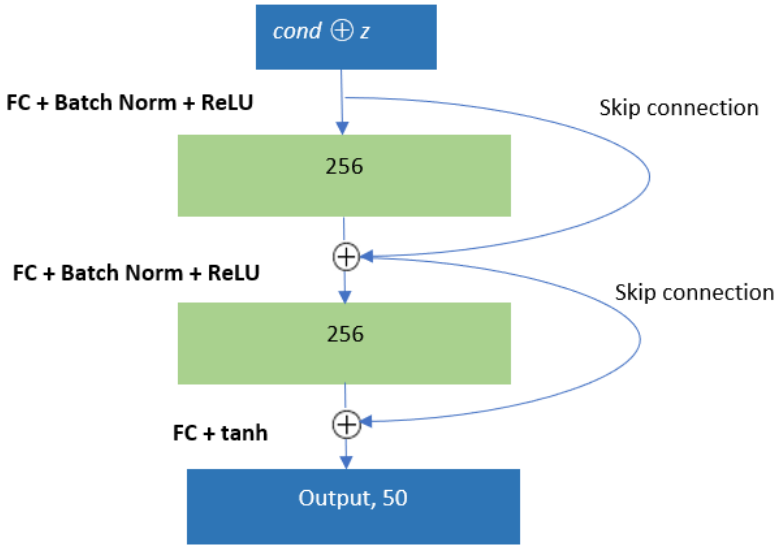
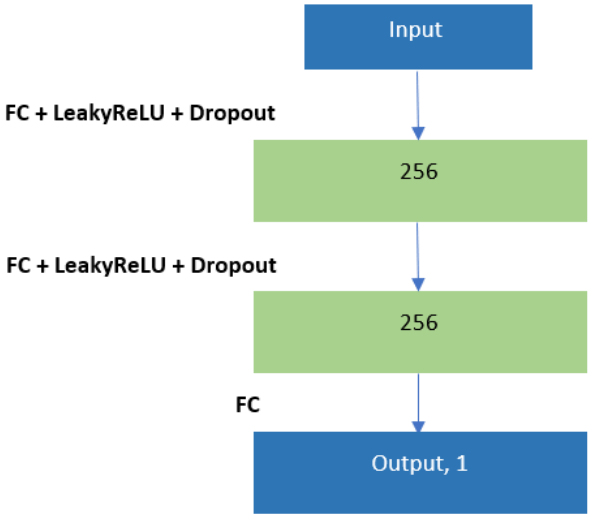


Figure 36.8 shows the detailed architecture of the discriminator in the CPCTGAN model, where LeakyReLU means that the layer uses the leaky rectified linear unit activation function.

Figure 36.8 Detailed Architecture of the Discriminator



4. When training is completed, the conditional generator, which is appended with the decoder, is used to generate new observations in the transformed data space. A reverse transformation then converts the generated samples from the transformed data space to the original data space.

Centroid Information

If you specify any interval variable for synthetic data generation, the TABULARGAN procedure requires information about the centroid of this variable. By default, the procedure calculates the centroid information by using the parameters that you specify in the GMM statement. The centroid information for the interval variables enables the procedure to model multimodal data efficiently and to generate more realistic data.

Alternatively, you can provide the centroid information to PROC TABULARGAN as a table by specifying the CENTROIDSTABLE option in the GMM statement. The centroids table should contain the following information, in this order: variable name, centroid ID, weight, mean, and standard deviation. You can use the GMM procedure (see Chapter 13, "GMM Procedure") to estimate the number of modes and to fit a Gaussian mixture model to each variable. You can find an example of how to generate the centroids table in Example 36.2.

Displayed Output

The TABULARGAN procedure produces various tables. The following sections describe the output tables in alphabetical order.

Iteration History

The "Iteration History" table provides the iteration history of the optimization.

Level Frequencies for Nominal Variables

The "Level Frequencies for Nominal Variables" table provides frequency counts of all levels of the nominal variables.

Model Information

The "Model Information" table provides information about the parameters that are used in the tabular GAN model.

Number of Observations

The "Number of Observations" table displays the information about the number of observations in the input data. The Number of Observations Read column displays the number of observations that are read from the input data table, and the Number of Observations Used column displays the number of observations that are actually used in the tabular GAN model training. The count in the Number of Observations Used column can be lower if there are observations in the input data table that have missing values.

Table Output

The TABULARGAN procedure produces the following output data tables.

Score Output

The score output table is specified in the OUTPUT statement. This table has the generated data from the trained generator. The number of generated rows is specified in the NUMSAMPLES option of the PROC TABULARGAN statement.

Analytic Store Table

The analytic store table is specified in the SAVESTATE statement. This data table contains the saved model that the TABULARGAN procedure trains. You can use this analytic store table to generate new data.

ODS Table Names

The TABULARGAN procedure assigns a name to each table that it creates. You can use this name to refer to the table when you use the Output Delivery System (ODS) to select tables and create output data tables. Table 36.4 lists the table names.

Table 36.4 ODS Tables Produced by PROC TABULARGAN

Table Name	Description	Statement	Option
IterHistory	Iteration history	PROC TABULARGAN	Default output
LevelFreq	Level frequencies for nominal variables	INPUT	Default output
ModelInfo	Model information	PROC TABULARGAN	Default output
NObs	Number of observations	PROC TABULARGAN	Default output

Examples: TABULARGAN Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

Example 36.1: Analyzing Home Mortgage Data

This example shows how to use the TABULARGAN procedure to analyze and augment the hmeq data table, which contains data about home mortgage applicants. The following DATA step creates the input data table mylib.hmeq in your SAS library. These statements assume that your SAS libref is named mylib, as in the section "Using SAS Viya Workbench" on page 984, but you can substitute any appropriately defined SAS libref.

```
data mylib.hmeq;
   set sampsio.hmeq (obs=600);
run;
```

The following statements run PROC TABULARGAN to train a tabular GAN model and save the trained model in an analytic store:

```
proc tabulargan          data=mylib.hmeq seed=123 numSamples=5 useGPU(device=0);
     input                value clage/level=interval;
     input                bad job/level=nominal;
     gmm                  alpha=1 maxClusters=10 seed=42 VB(maxVbIter=30);
     aeoptimization       ADAM LearningRate=0.0001 numEpochs=3;
     ganoptimization      ADAM(beta1=0.55 beta2=0.95)  numEpochs=5;
     train                embeddingDim=64 miniBatchSize=300 useOrigLevelFreq;
     savestate            rstore=mylib.astore;
     output               out=mylib.out;
run;
```

Note that the typical number of epochs for the autoencoder and GAN should be greater than 100 for proper training of the models. The appropriate number of epochs, specified in the NUMEPOCHS= option, should be found for each use case. Here small option values are used for brevity. PROC TABULARGAN generates four ODS tables, which are shown in Output 36.1.1 through Output 36.1.4.

Output 36.1.1 shows the iteration history, including the autoencoder loss value and the generator and discriminator loss values for each epoch of training.

Output 36.1.1 Iteration History

The TABULARGAN Procedure

Iteration History			
Epoch Number	Autoencoder Loss	Generator Loss	Discriminator Loss
1	0.313142	.	.
2	0.222675	.	.
3	0.147904	.	.
1	.	1.264446	0.002022
2	.	1.283346	0.029679
3	.	1.281197	0.002076
4	.	1.293178	-0.046795
5	.	1.343589	-0.010720

Output 36.1.2 shows the number of observations that are read and the number of observations that are used.

Because the mylib.hmeq data table contains missing values, the number of observations read does not equal the number of observations used in this example. When the input data table contains missing values, PROC TABULARGAN ignores the observations (rows) that have missing values in the input data table and does not use them in the analysis.

Output 36.1.2 Number of Observations

Number of Observations	
Number of Observations Read	600
Number of Observations Used	498

Output 36.1.3 shows the values of the parameters that are used in the tabular GAN model.

Output 36.1.3 Model Information

Model Information	
Generator Embedding Dimension	64
Number of Observations in One Minibatch	300
Number of Observations Group Together in Applying the Discriminator	10
Weight for Regularizing the Discriminator	10
Exponential Decay Rate for the First-Moment Estimates for the Autoencoder's Optimizer	0.9
Exponential Decay Rate for the Second-Moment Estimates for the Autoencoder's Optimizer	0.999
Learning Rate for the Autoencoder's Optimizer	0.0001
Number of Epochs for the Autoencoder's Training	3
Weight Decay for the Autoencoder's Optimizer	1E-8
Exponential Decay Rate for the First-Moment Estimates to Optimize GAN	0.55
Exponential Decay Rate for the Second-Moment Estimates to Optimize GAN	0.95
Learning Rate for the GAN Optimizer	0.00002
Number of Epochs for the GAN Training	5
Weight Decay for the Generator's Optimizer	0.0001
Weight Decay for the Discriminator's Optimizer	1E-6
Seed for Random Initialization	123
Whether to Use Log Frequency of Categorical Levels in the Conditional Sampling	True

Output 36.1.4 shows the level names and frequencies of the nominal variables bad and job.

Output 36.1.4 Level Frequencies of Nominal Variables

Level Frequencies for Nominal Variables		
Variable Name	Level	Frequency
BAD	0	300
BAD	1	198
JOB	Mgr	59
JOB	Office	90
JOB	Other	223
JOB	ProfExe	87
JOB	Sales	27
JOB	Self	12

The following PROC PRINT statement prints the generated synthetic samples shown in Output 36.1.5:

```
proc print   data = mylib.out;
run;
```

Output 36.1.5 Output

Obs	VALUE	CLAGE	BAD	JOB
1	271190.81	-10.907	1	ProfExe
2	18348.42	135.995	1	ProfExe
3	58962.11	289.372	0	Other
4	11862.95	18.502	1	Mgr
5	218497.81	77.910	1	Other

Example 36.2: Analyzing Home Mortgage Data with a Centroids Table

This example uses the same home mortgage data that Example 36.1 uses. It shows how to calculate the centroids table by using the GMM procedure (see Chapter 13, "GMM Procedure") and then how to use that table in the TABULARGAN procedure to analyze and augment the hmeq data table. The following DATA step creates the input data table mylib.hmeq in your SAS library. These statements assume that your SAS libref is named mylib, as in the section "Using SAS Viya Workbench" on page 984, but you can substitute any appropriately defined SAS libref.

```
data mylib.hmeq;
   set sampsio.hmeq (obs=600);
run;
```

The following macro runs PROC GMM over interval variables that are selected for analysis by PROC TABULARGAN to calculate the centroids table. In this example, the variables value and clage are selected for analysis.

```
%macro runloop;
/* Specify all interval input variables*/
%let names=value clage;
/* Loop over all variables that need centroids generation */
%do i=1 %to %sysfunc(countw(&names));
    %let name&i = %scan(&names, &i, %str( ));
    /* Call PROC GMM to cluster each variable */
    proc gmm
         data=mylib.hmeq
         seed=42
         maxClusters=10
         alpha=1
         inference=VB (maxVbIter=30 covariance=diagonal threshold=0.01);
         input &&name&i;
         ods select ClusterInfo;
         ods output ClusterInfo = work.outinfo&i;
    run;

    /* Save variable name, weights, mean,     */
```

```
    /* and standard deviation of each cluster */
    data  outinfo&i;
        varname = "&&name&i";
        set  outinfo&i(rename=(&&name&i.._Mean=Mean
                            &&name&i.._Variance=Var));
        /* Calculate standard deviation from variance*/
        std = sqrt(Var);
        drop Var;
    run;

    /* Construct centroids table from saved weights */
    %if &i=1 %then %do;
        data centroids;
            set outinfo&i;
        run;
    %end;
    %else %do;
        data centroids;
        set centroids outinfo&i;
        run;
    %end;
%end;
%mend;

/* Run the runloop macro to generate the centroids table */
%runloop;
```

The following DATA step uploads the centroidstable data table to your SAS library:

```
data mylib.centroidstable;
    set centroids;
run;
```

The following statements run PROC TABULARGAN to train a tabular GAN model and save the trained model in an analytic store:

```
proc tabulargan         data=mylib.hmeq seed=123 numSamples=5 useGPU(device=0);
     input               value clage/level=interval;
     input               bad job/level=nominal;
     gmm                 centroidsTable=mylib.centroidstable;
     aeoptimization      ADAM numEpochs=3;
     ganoptimization     ADAM numEpochs=3;
     train               miniBatchSize=300;
     savestate           rstore=mylib.astore;
     output              out=mylib.out;
run;
```

Note that the typical number of epochs for the autoencoder and GAN should be greater than 100 for proper training of the models. The appropriate number of epochs, specified in the NUMEPOCHS= option, should be found for each use case. Here small option values are used for brevity. PROC TABULARGAN generates four ODS tables, which are shown in Output 36.2.1 through Output 36.2.4.

Output 36.2.1 shows the iteration history, including the autoencoder loss value and the generator and discriminator loss values for each epoch of training.

Output 36.2.1 Iteration History

The TABULARGAN Procedure

Iteration History			
Epoch Number	Autoencoder Loss	Generator Loss	Discriminator Loss
1	0.309802	.	.
2	0.292259	.	.
3	0.192475	.	.
1	.	1.410194	0.025480
2	.	1.399884	-0.017609
3	.	1.415137	-0.019580

Output 36.2.2 shows the number of observations that are read and the number of observations that are used. Because the mylib.hmeq data table contains missing values, the number of observations read does not equal the number of observations used in this example. When the input data table contains missing values, PROC TABULARGAN ignores the observations (rows) that have missing values in the input data table and does not use them in the analysis.

Output 36.2.2 Number of Observations

Number of Observations	
Number of Observations Read	600
Number of Observations Used	498

Output 36.2.3 shows the values of the parameters that are used in the tabular GAN model.

Output 36.2.3 Model Information

Model Information	
Generator Embedding Dimension	128
Number of Observations in One Minibatch	300
Number of Observations Group Together in Applying the Discriminator	10
Weight for Regularizing the Discriminator	10
Exponential Decay Rate for the First-Moment Estimates for the Autoencoder's Optimizer	0.9
Exponential Decay Rate for the Second-Moment Estimates for the Autoencoder's Optimizer	0.999
Learning Rate for the Autoencoder's Optimizer	0.001
Number of Epochs for the Autoencoder's Training	3
Weight Decay for the Autoencoder's Optimizer	1E-8
Exponential Decay Rate for the First-Moment Estimates to Optimize GAN	0.5
Exponential Decay Rate for the Second-Moment Estimates to Optimize GAN	0.9
Learning Rate for the GAN Optimizer	0.00002
Number of Epochs for the GAN Training	3
Weight Decay for the Generator's Optimizer	0.0001
Weight Decay for the Discriminator's Optimizer	1E-6
Seed for Random Initialization	123
Whether to Use Log Frequency of Categorical Levels in the Conditional Sampling	True

Output 36.2.4 shows the level names and frequencies of the nominal variables bad and job.

Output 36.2.4 Level Frequencies of Nominal Variables

Level Frequencies for Nominal Variables		
Variable Name	Level	Frequency
BAD	0	300
BAD	1	198
JOB	Mgr	59
JOB	Office	90
JOB	Other	223
JOB	ProfExe	87
JOB	Sales	27
JOB	Self	12

The following PROC PRINT statement prints the generated synthetic samples shown in Output 36.2.5:

```
proc print   data = mylib.out;
run;
```

Output 36.2.5 Output

Obs	VALUE	CLAGE	BAD	JOB
1	175404.61	369.246	0	Mgr
2	45851.11	-86.237	0	Self
3	-34735.97	-40.260	1	Other
4	-19828.26	602.963	1	ProfExe
5	-32926.37	129.364	1	Other

References

Gulrajani, I., Ahmed, F., Arjovsky, M., Dumoulin, V., and Courville, A. (2017). "Improved Training of Wasserstein GANs." In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, 5769–5779. https://proceedings.neurips.cc/paper/2017/file/892c3b1c6dccd52936e27cbd0ff683d6-Paper.pdf.

Torfi, A., and Fox, E. A. (2020). "CorGAN: Correlation-Capturing Convolutional Generative Adversarial Networks for Generating Synthetic Healthcare Records." Accepted to be published in the 33rd International FLAIRS Conference, AI in Healthcare Informatics. https://arxiv.org/abs/2001.09346.

Xu, L., Skoularidou, M., Cuesta-Infante, A., and Veeramachaneni, K. (2019). "Modeling Tabular Data Using Conditional GAN." In *Neural Information Processing Systems (NeurIPS)*. https://papers.nips.cc/paper/2019/file/254ed7d2de3b23ab10936522dd547b78-Paper.pdf.

Chapter 37
TSNE Procedure

Contents

Overview: TSNE Procedure **1008**
PROC TSNE Features 1008
Using SAS Viya Workbench 1008
Getting Started: TSNE Procedure **1009**
Syntax: TSNE Procedure **1011**
PROC TSNE Statement 1011
AUTOTUNE Statement 1013
DISPLAY Statement 1016
DISPLAYOUT Statement 1017
INPUT Statement 1018
OUTPUT Statement 1018
WEIGHT Statement 1018
Details: TSNE Procedure **1019**
Displayed Output 1020
ODS Table Names 1020
Output Data Tables 1021
Examples: TSNE Procedure **1022**
Example 37.1: Tuning Hyperparameters of a *t*-SNE Model 1022
References **1024**

Overview: TSNE Procedure

The TSNE procedure implements the *t*-distributed stochastic neighbor embedding (*t*-SNE) dimension reduction method in SAS Viya Workbench. The *t*-SNE method is well suited for visualization of high-dimensional data, as well as for feature engineering and preprocessing for subsequent clustering and modeling. PROC TSNE computes a low-dimensional representation, also called an embedding, of high-dimensional data into two or three dimensions. Unlike other dimension reduction methods, such as principal component analysis, *t*-SNE is appropriate for nonlinear data and emphasizes existing groupings in the data. The method is named *t*-SNE because it models the pairwise distances in low dimensions according to Student's *t*-distribution. The *t*-distribution with one degree of freedom has heavier tails than the Gaussian distribution, which means that it assigns higher probability values to large distances. This allows *t*-SNE to relax pairwise distances for non-neighboring observations, whereas distances between closely neighboring observations are more exactly preserved. This behavior is desirable because it mitigates the crowding problem in high-dimensional data representation and makes existing groups in the data visually evident. You can use PROC TSNE to perform computation in parallel by making full use of multicore computers.

The TSNE procedure operates on an input data table. For each observation in the input data table, the procedure returns either two or three computed columns that contain the embedding coordinates. It computes the embedding by minimizing the Kullback-Leibler divergence between the joint probabilities in high dimensions and the joint probabilities in low dimensions, and it stores the embedding in the output data table that is specified by the OUTPUT statement. To speed up computations, PROC TSNE uses the Barnes-Hut approximation to the perplexity loss.

PROC TSNE Features

PROC TSNE enables you to use parallel execution for the *t*-distributed stochastic neighbor embedding (*t*-SNE) dimension reduction method.

The procedure has the following basic features:

- is multithreaded
- returns low-dimensional embeddings that are based on a parallel implementation of the *t*-SNE algorithm

Using SAS Viya Workbench

SAS Viya Workbench has greatly simplified the SAS programming experience by offering a syntax style nearly identical to that of SAS 9 and earlier releases. However, there are minor exceptions. In the past, SAS Viya procedures required the use of a CAS engine libref. This engine served multiple purposes, but it was primarily used to upload data to the CAS server and manage the data. Its additional purposes included support of new data types and improved performance for large data sets. No CAS server is needed for SAS Viya Workbench, because all computations are done in place for you in a single location.

In the following program, the LIBNAME statement creates a mylib libref that the examples in this chapter use to refer to your current or desired work directory:

```
libname mylib ".";
```

For more information, see *SAS Viya Workbench: Getting Started*.

Getting Started: TSNE Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

This example shows how to use the TSNE procedure to obtain an embedding from observations in a data table. The example uses the Iris data from Fisher (1936), which contain morphological measurements of 50 specimens from each of three different species of iris flowers: *Iris setosa*, *I. versicolor*, and *I. virginica*. Mezzich and Solomon (1980) discuss a variety of cluster analyses that use the Iris data. PROC TSNE returns a two-dimensional representation of each observation. The analysis uses four variables: SepalLength, SepalWidth, PetalLength, and PetalWidth. The remaining variables in the data table are not used.

You can load the Sashelp.Iris data set into your SAS library in the following DATA step. These statements assume that your libref is named mylib, but you can substitute any appropriately defined libref.

```
data mylib.iris;
    set sashelp.iris;
    id=_n_;
run;
```

The following statements run PROC TSNE and output the results to ODS tables:

```
proc tsne
   data            = mylib.iris
   nDimensions     = 2
   perplexity      = 5
   learningRate    = 100
   maxIters        = 500;
   input             SepalLength SepalWidth PetalLength PetalWidth;
   output out      = mylib.tsne_out copyvars=(id species);
run;
```

The NDIMENSIONS=2 option requests that the model return two embedding dimensions; the PERPLEXITY=5 option specifies the perplexity value; the LEARNINGRATE=100 option specifies the learning rate for the optimization; and the INPUT statement specifies that the SepalLength, SepalWidth, PetalLength, and PetalWidth variables be used as inputs. The OUTPUT statement requests that the embedding be written to the data table mylib.tsne_out, and the COPYVARS= option requests that the ID and Species variables be copied to the output.

The following statements download the data table mylib.tsne_out from the SAS library to the local SAS data set tsne_out:

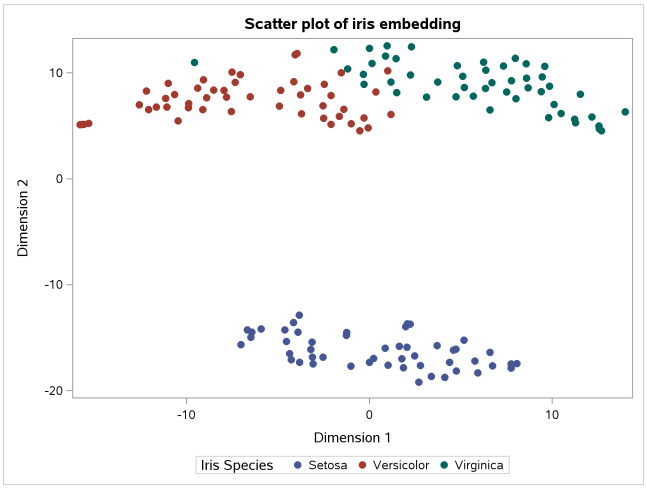
```
data tsne_out;
    set mylib.tsne_out;
run;
```

The following PROC SGPLOT statements plot embedding dimension _DIM_2_ against embedding dimension _DIM_1_:

```
proc sgplot data=tsne_out;
   title "Iris embedding";
   title1 "Scatter plot of iris embedding";
   scatter x=_DIM_1_ y=_DIM_2_ / group=species
                                 markerattrs=(symbol=CircleFilled);
run;
```

Output 37.1 shows the results. The colors in the scatter plot indicate three distinct clusters, which correspond to the three iris species.

Figure 37.1 Scatter Plot of Iris Data Embedding



Syntax: TSNE Procedure

The following statements are available in the TSNE procedure:

```
PROC TSNE <options> ;
    AUTOTUNE <options> ;
    DISPLAY <table-list> </ options> ;
    DISPLAYOUT table-spec-list </ options> ;
    INPUT variables ;
    OUTPUT OUT=libref.data-table <options> ;
    WEIGHT variable ;
```

The PROC TSNE statement and at least one INPUT statement are required. You can specify multiple INPUT statements.

The following sections describe the PROC TSNE statement and then describe the other statements in alphabetical order.

PROC TSNE Statement

```
PROC TSNE <options> ;
```

The PROC TSNE statement invokes the procedure. Table 37.1 summarizes the *options* available in the PROC TSNE statement.

Table 37.1 PROC TSNE Statement Options

Option	Description
Input Data Table Option	
DATA=	Specifies the input data table
***t*-SNE Options**	
LEARNINGRATE=	Specifies the learning rate for the optimization
MAXITERS=	Specifies the maximum number of iterations
METHOD=	Specifies the algorithm to use for gradient calculation
NDIMENSIONS=	Specifies the number of dimensions to return for the embedding
NOPRINT	Suppresses ODS output
NTHREADS=	Specifies the number of threads to use
PERPLEXITY=	Specifies the perplexity value
SEED=	Specifies the seed to be used for pseudorandom number generation

You can specify the following *options*:

DATA=*libref.data-table*
names the input data table for PROC TSNE to use. The default is the most recently created data table. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to the data. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 1008.

data-table specifies the name of the input data table.

LEARNINGRATE=*number*
specifies the learning rate for the gradient descent optimization, where *number* is a positive real number. The learning rate controls the amount by which the embeddings are updated at each iteration.

By default, LEARNINGRATE=1000. This value can be tuned with the AUTOTUNE statement.

MAXITERS=*number*
specifies the maximum number of iterations for the algorithm to perform, where *number* is an integer greater than or equal to 1.

By default, MAXITERS=300. This value can be tuned with the AUTOTUNE statement.

METHOD=BARNES_HUT | EXACT
specifies the algorithm to use for gradient calculation. You can specify the following values:

BARNES_HUT uses the Barnes-Hut approximation method.

EXACT uses the exact method.

By default, METHOD=EXACT.

NDIMENSIONS=*number*
specifies the number of dimensions to return, where *number* is either 2 or 3.

By default, NDIMENSIONS=2.

NOPRINT
suppresses ODS output.

NTHREADS=*number-of-threads*
specifies the number of threads to use for the computation, where *number-of-threads* is an integer. The default value is the maximum number of available threads.

PERPLEXITY=*number*
specifies the perplexity value to use for the computation, where *number* is a real number greater than zero. For more information, see the section "Details: TSNE Procedure" on page 1019.

By default, PERPLEXITY=30. This value can be tuned with the AUTOTUNE statement.

SEED=*random-seed*

specifies an integer that is used to start the pseudorandom number generator. This option enables you to reproduce the same sample output. If you do not specify a seed or you specify a value less than or equal to 0, the seed is generated from reading the time of day from the computer's clock.

By default, SEED=0.

AUTOTUNE Statement

AUTOTUNE < *options* > **;**

The AUTOTUNE statement searches for the best combination of values of the LEARNINGRATE=, MAXITERS=, and PERPLEXITY= options in the PROC TSNE statement.

Table 37.2 summarizes the *options* that you can specify in the AUTOTUNE statement. For more information about all options except the TUNINGPARAMETERS= option, see the option's description in the section "AUTOTUNE Statement" on page 6 in Chapter 2, "Shared Concepts." The TUNINGPARAMETERS= option is described following Table 37.2.

Table 37.2 AUTOTUNE Statement Options

Option	Description
APPENDLOOKUP	Specifies that the table specified in the HISTORYTABLE= option contain the rows from the table specified in the LOOKUPTABLE= option
EVALHISTORY=	Specifies how to report the evaluation history of the tuner
HISTORYTABLE=	Specifies the table that contains the evaluation history
LIVEUPDATE	Specifies that the table specified in the HISTORYTABLE= option be updated at every evaluation
LOOKUPTABLE=	Specifies the table to use for evaluation lookup
MAXBAYES=	Specifies the maximum number of points in the kriging model
MAXEVALS=	Specifies the maximum number of evaluations
MAXITER=	Specifies the maximum number of iterations when SEARCHMETHOD=GA or SEARCHMETHOD=BAYESIAN
MAXTIME=	Specifies the maximum time for all iterations
MAXTRAINTIME=	Specifies the maximum time for a model training
NCONVITER=	Specifies the number of convergence iterations
NOGRIDSHUFFLE	Requests that the grid points not be shuffled
NOLOCALSEARCH	Disables local search optimization
NPARALLEL=	Specifies the number of parallel sessions
NSUBSESSIONWORKERS=	Specifies the number of workers in parallel sessions
POPSIZE=	Specifies the population size when SEARCHMETHOD=GA or SEARCHMETHOD=BAYESIAN
SAMPLESIZE=	Specifies the sample size when SEARCHMETHOD=LHS or SEARCHMETHOD=RANDOM
SEARCHMETHOD=	Specifies the search method that the optimizer uses
SELECTINITPOINT	Specifies that the tuner select the best evaluation from the lookup table

Table 37.2 *continued*

Option	Description
TUNINGPARAMETERS=	Specifies the custom tuning parameters
USEPARAMETERS=	Specifies how to handle the TUNINGPARAMETERS= option

TUNINGPARAMETERS=(*suboption* **| . . . | <** *suboption* **>)**

TUNEPARMS=(*suboption* **| . . . | <** *suboption* **>)**

specifies which parameters to tune and which ranges to tune over. If USEPARAMETERS=STANDARD, this option is ignored.

You can specify one or more of the following *suboptions*:

LEARNINGRATE (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**

specifies information about the learning rate to use for tuning the t-SNE model. For more information, see the LEARNINGRATE= option in the PROC TSNE statement.

You can specify the following additional suboptions:

LB=*number*

specifies the minimum learning rate to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=10.

UB=*number*

specifies the maximum learning rate to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=2000.

VALUES=*value-list*

specifies a list of values to consider for the learning rate during tuning, where *value-list* is a space-separated list of integer numbers greater than or equal to 1. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*

specifies the initial learning rate for the tuner to use.

By default, INIT=1000.

EXCLUDE

excludes the learning rate from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

MAXITERS (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**

specifies information about the maximum number of iterations to use for tuning the t-SNE model. For more information, see the MAXITERS= option in the PROC TSNE statement.

You can specify the following additional suboptions:

LB=*number*
specifies the minimum number of iterations to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=250.

UB=*number*
specifies the maximum number of iterations to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=5000.

VALUES=*value-list*
specifies a list of numbers of iterations to consider during tuning, where *value-list* is a space-separated list of positive integers. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial number of iterations for the tuner to use.

By default, INIT=300.

EXCLUDE
excludes the number of iterations from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

PERPLEXITY (LB=*number* **UB=***number* **VALUES=***value-list* **INIT=***number* **EXCLUDE)**
specifies information about the perplexity to use for tuning the t-SNE model. For more information, see the PERPLEXITY= option in the PROC TSNE statement.

You can specify the following additional suboptions:

LB=*number*
specifies the minimum perplexity to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=5.

UB=*number*
specifies the maximum perplexity to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, UB=50.

VALUES=*value-list*
specifies a list of perplexities to consider during tuning, where *value-list* is a space-separated list of numbers greater than 0. If you specify this suboption, you cannot specify either the LB= or UB= suboption.

INIT=*number*
specifies the initial perplexity for the tuner to use.

By default, INIT=30.

EXCLUDE
excludes the perplexity from the tuning process. If you specify this suboption, any specified LB=, UB=, VALUES=, and INIT= suboptions are ignored.

DISPLAY Statement

DISPLAY < *table-list* > < / *options* > **;**

The DISPLAY statement enables you to specify a list of display tables to display or exclude. This statement is similar to the ODS SELECT, ODS EXCLUDE, and ODS TRACE statements. However, the DISPLAY statement can improve performance when a large number of tables could be generated (such as in BY-group processing). The DISPLAY statement produces only the requested tables, but ODS statements generate all the tables and then keep the requested tables.

If you use both DISPLAY and ODS statements together, the DISPLAY statement takes precedence over the ODS statements. Note that the ODS EXCLUDE statement processes tables after they have been filtered by the DISPLAY statement. In some cases, it might appear that the ODS EXCLUDE statement is taking precedence because it can further filter the tables. For more information about ODS, see *SAS Output Delivery System: Procedures Guide*.

You can specify the *table-list* as a list of table names, paths, partial pathnames, and regular expressions.

The table names that you can specify are listed in the section "ODS Table Names" on page 1020. A path is a table name that is prefixed with dot-separated grouping information. For example, a SelectionSummary table that a procedure produces during a selection routine might have the path *Bygroup1.Summary.SelectionSummary*. A partial pathname does not include all groups; for example, *SelectionSummary* and *Summary.SelectionSummary* are partial pathnames for *Bygroup1.Summary.SelectionSummary*.

When you specify a table name or partial pathname, all display tables whose paths end in the specified name are selected for display or exclusion. For example, both *SelectionSummary* and *Summary.SelectionSummary* select *Bygroup1.Summary.SelectionSummary*.

A regular expression is enclosed in forward slashes (/). For example, specifying "/tions/" selects all pathnames that contain the substring "tions"; in particular, the *Bygroup1.Summary.SelectionSummary* table is selected. Specifying "!/tions/" selects all pathnames that do not contain the substring "tions"; in particular, the *Bygroup1.Summary.SelectionSummary* table is not selected.

You can specify the following *options* after a slash (/):

CASESENSITIVE
performs a case-sensitive comparison of table names in the *table-list* to display table names when tables are subsetted for display. To preserve case, you must enclose table names in the *table-list* in quotation marks.

EXCLUDE
displays all display tables except those that you specify in the *table-list*.

EXCLUDEALL
suppresses display of all tables. This option takes precedence over the other options.

TRACE
displays the display table names, labels, and paths.

DISPLAYOUT Statement

DISPLAYOUT *table-spec-list* < / *options* > **;**

The DISPLAYOUT statement enables you to create output tables from your displayed output. This statement is similar to the ODS OUTPUT statement. For more information about ODS, see *SAS Output Delivery System: Procedures Guide*.

The *table-spec-list* specifies a list of output tables to create. Each entry in the list has either a *key*=*value* format or a *key* format:

key=*value* specifies *key* as the ODS table name, path, or partial pathname, and specifies *value* as the output table name.

key specifies *key* as the ODS table name and also as the output table name.

The ODS table names that you can specify are listed in the section "ODS Table Names" on page 1020. You cannot specify the ODS table named OutputCasTables in the *table-spec-list*.

Table names and partial pathnames are discussed under the DISPLAY statement. The DISPLAYOUT statement does not support regular expressions.

You can specify the following *options* after a slash (/):

INCLUDEALL
creates output tables for all display tables. The name of the created output table is the same as the name of the corresponding display table. If you specify this option, the *table-spec-list* specification is ignored.

NOREPLACE
does not replace any existing output table of the same name.

INPUT Statement

INPUT *variables* **;**

The INPUT statement specifies the *variables* to use in the *t*-SNE analysis. The *variables* must be of interval type. You must specify at least one input *variable*.

OUTPUT Statement

OUTPUT OUT=*libref.data-table* < *option* > **;**

The OUTPUT statement creates an embedding data table to contain the results of the procedure run.

You must specify the following option:

OUT=*libref.data-table*
names the output data table for PROC TSNE to use. You must specify this option before any other options. *libref.data-table* is a two-level name, where

libref refers to a collection of information that is defined in the LIBNAME statement and includes the library, which includes a path to where the data table is to be stored. For more information about *libref*, see the section "Using SAS Viya Workbench" on page 1008.

data-table specifies the name of the output data table.

You can also specify the following *option*:

COPYVAR=*variable*
COPYVARS=(*variables***)**
lists one or more *variables* from the input data table to be transferred to the output data table.

WEIGHT Statement

WEIGHT *variable* **;**

The *variable* in the WEIGHT statement is used as a weight to perform a weighted analysis of the data. Observations that have nonpositive or missing weights are not included in the analysis. When the WEIGHT statement is omitted, all observations that are used in the analysis are assigned a weight of 1.

Details: TSNE Procedure

The t-SNE method for dimensionality reduction was introduced by Van der Maaten and Hinton (2008). The method estimates the embeddings y_i in two or three dimensions by minimizing the Kullback-Leibler loss, which is defined as

$$C = \mathrm{KL}(P \| Q) = \sum_{i=1}^{N} \sum_{j=1}^{N} p_{ij} \log \frac{p_{ij}}{q_{ij}}$$

where N is the number of observations and p_{ij} is the similarity between observations i and j in the data table. It is defined as

$$p_{ij} = \frac{p_{i|j} + p_{j|i}}{2N}$$

where $p_{i|j}$ and $p_{j|i}$ are conditional probabilities and q_{ij} is the estimated similarity in low-dimensional space, which follows Student's t-distribution with one degree of freedom and is given by

$$q_{ij} = \frac{(1 + \|y_i - y_j\|^2)^{-1}}{\sum_{k \neq l} (1 + \|y_k - y_l\|^2)^{-1}}$$

The conditional probability $p_{i|j}$ follows the expression

$$p_{j|i} = \frac{\exp(-\|x_i - x_j\|^2)}{2\sigma_i^2}$$

where σ_i is a bandwidth parameter that varies for each observation x_i. PROC TSNE sets σ_i in such a way that the perplexity

$$\mathrm{Perp}_i = 2^{-\sum_j p_{j|i} \log_2 p_{j|i}}$$

matches the fixed PERPLEXITY= option value that the user specifies. In general, lower perplexity yields embeddings that have smaller, more separated groupings.

To avoid the $O(N^2)$ computational complexity of the similarity computations, PROC TSNE employs the Barnes-Hut approximation that is proposed by Van der Maaten (2014).

The TSNE procedure estimates the values of y_i by minimizing C via gradient descent. This is an iterative optimization process that the user controls by specifying the MAXITERS= and LEARNINGRATE= options.

Displayed Output

The TSNE procedure displays various tables that are related to the *t*-SNE algorithm. The following sections describe the output tables in the order of their appearance when the related options are specified.

Model Information

The "Model Information" table displays basic information about the parameters that are used in the *t*-SNE algorithm. This information includes the perplexity along with the seed that is specified in the SEED= option in the PROC TSNE statement.

Number of Observations

The "Number of Observations" table displays the number of observations that are read from the input data table and used.

Iteration History

The "Iteration History" table displays the iteration history, which consists of the iteration number and the loss for every 100 iterations.

OutputCasTables Table

The OutputCasTables table is a special table that has information about each table that is created. The information for each table consists of the table name, the library in which the table resides, and the number of columns and rows in the table. Because this table is not a typical ODS table that contains analytical results, you cannot include it in the *table-spec-list* in the DISPLAYOUT statement.

ODS Table Names

Each table that the TSNE procedure creates has a name associated with it, and you must use this name to refer to the table when you use ODS statements. The name of each table and a short description of its contents are listed in Table 37.3.

Table 37.3 ODS Tables Produced by PROC TSNE

Table Name	Description	Statement	Option
BestConfiguration	Hyperparameters and objective function values for the best configuration	AUTOTUNE	Default
EvaluationHistory	Values of the hyperparameters and the objective function for all configurations	AUTOTUNE	Default
HyperparameterImportance	Relative importance of hyperparameters	AUTOTUNE	Default
IterationHistory	Tuner iteration history	AUTOTUNE	Default
ModelInfo	Model information	PROC TSNE	Default
NObs	Number of observations	PROC TSNE	Default
OptIterHistory	Iteration history	PROC TSNE	Default
OutputCASTables	See the section "OutputCasTables Table" on page 1020	PROC TSNE	Default
TunerCasOutputTables	Tables created during tuning	AUTOTUNE	Default
TunerInfo	Setup values that are used by the tuner	AUTOTUNE	Default
TunerResults	Values of the hyperparameters, the objective function for the default configuration (iteration 0), and up to 10 best configurations found	AUTOTUNE	Default
TunerSummary	Statistics about the tuning process	AUTOTUNE	Default
TunerTiming	Total time spent on different tasks during tuning	AUTOTUNE	Default

Output Data Tables

The TSNE procedure creates a data table to which it writes the embeddings. You specify the name of this data table in the OUTPUT statement. Details about the data table are shown in Table 37.4.

Table 37.4 Output Data Table Produced by PROC TSNE

Data Table	Description
TSNE_OUT	Lists the estimated embeddings

Examples: TSNE Procedure

NOTE: Input data must be in a table that is accessible in your session. You can refer to this table by using a two-level name. The first level is a libref, and the second level is the table name. For more information, see the section "Using SAS Viya Workbench" on page 6 in Chapter 2, "Shared Concepts."

Example 37.1: Tuning Hyperparameters of a *t*-SNE Model

This example shows how to use the AUTOTUNE statement in the TSNE procedure to tune hyperparameters of a *t*-SNE model from observations in a data table. The example uses the Iris data from Fisher (1936), which contain morphological measurements of 50 specimens from each of three different species of iris flowers: *Iris setosa*, *I. versicolor*, and *I. virginica*. Mezzich and Solomon (1980) discuss a variety of cluster analyses that use the Iris data. PROC TSNE returns a two-dimensional representation of each observation. The analysis uses four variables: SepalLength, SepalWidth, PetalLength, and PetalWidth. The remaining variables in the data table are not used.

The following DATA step loads the Iris data table into your session. These statements assume that your libref is named mylib, but you can substitute any appropriately defined libref.

```
data mylib.iris;
    set sashelp.iris;
    id=_n_;
run;
```

The following statements run PROC TSNE with the AUTOTUNE statement to automatically tune the hyperparameters of a *t*-SNE model:

```
proc tsne
   data         = mylib.iris
   nDimensions  = 2;
   input          SepalLength SepalWidth PetalLength PetalWidth;
   autotune
      /* Tuning Parameters
      You do not need to specify any tuning parameters for the default
      tuning process. If you want to make adjustments to the default
      tuning process, uncomment the following block of code and change
      any of the tuning parameters' attributes.

      tuningparameters=(
         PERPLEXITY        ( lb=5    ub=50   init=30   )
         LEARNINGRATE      ( lb=10   ub=2000 init=1000 )
         MAXITER           ( lb=250  ub=5000 init=300  )
      )
      */
   ;
   /* Remove this line to see all results */
   ods select BestConfiguration EvaluationHistoryPlot IterationHistoryPlot;
run;
```

The preceding statements produce the table and the plots shown in Output 37.1.1 through Output 37.1.3. The table in Output 37.1.1 displays the evaluation number, the values of the tuning parameters, and the error metric value for the best *t*-SNE model that the tuner found. Note that the ODS SELECT statement limits the displayed results to a single table and two plots. You can remove this statement to display all tables and plots. For the full list of ODS tables that PROC TSNE produces, see Table 37.3.

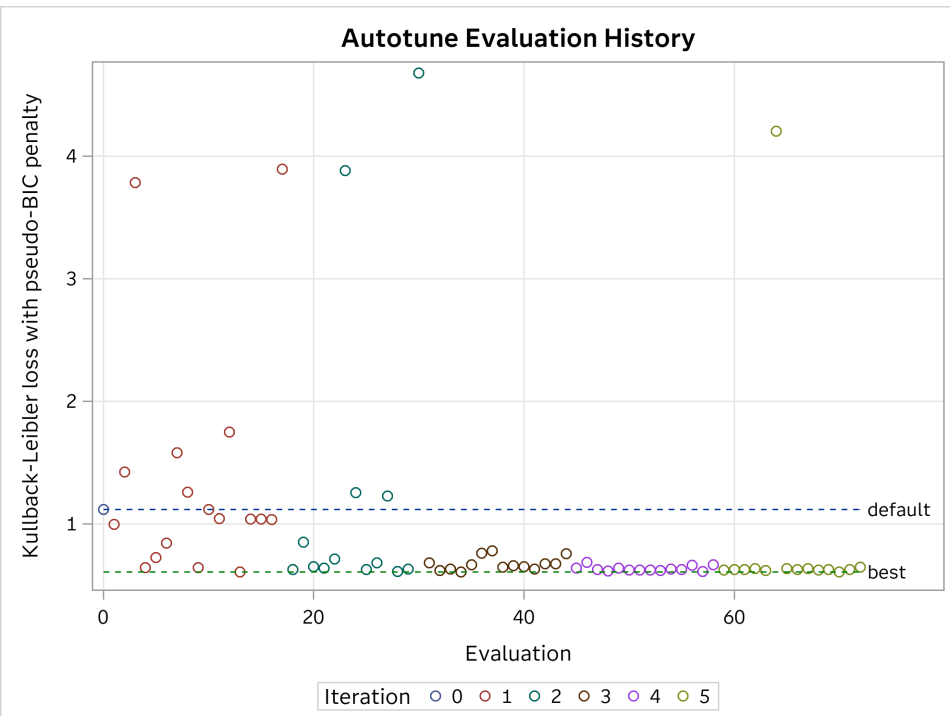
Output 37.1.1 Best Configuration Table

The TSNE Procedure

Best Configuration	
Evaluation	13
Perplexity	5
Learning Rate	1005
Maximum Number of Iterations	2625
Kullback-Leibler loss with pseudo-BIC penalty	0.6086944955

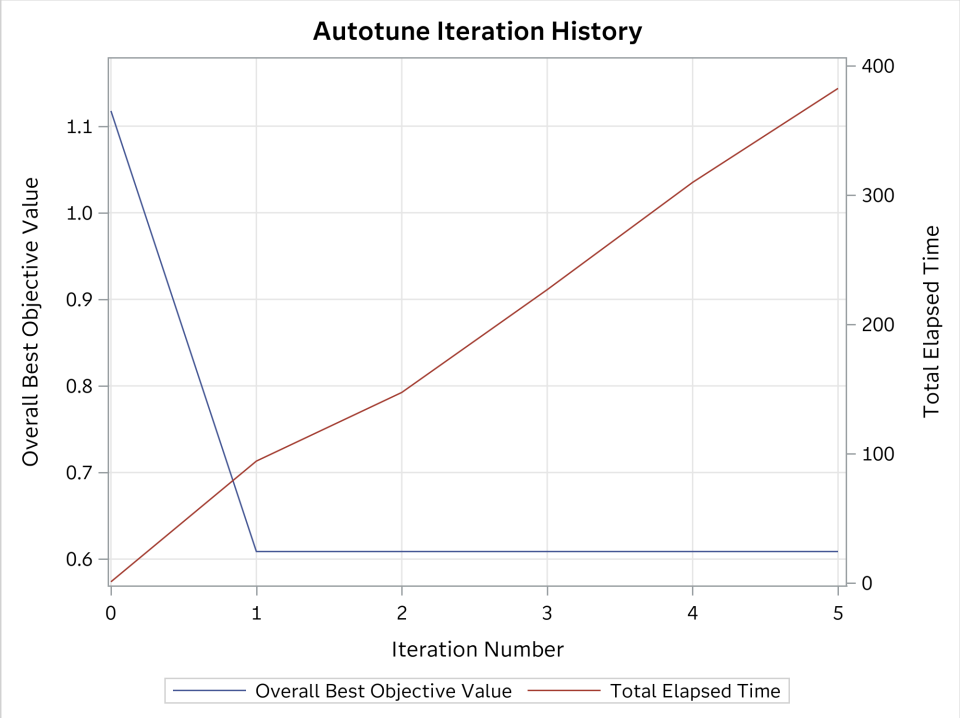
Output 37.1.2 displays a scatter plot of all configurations that the tuner tried. The objective values are shown on the Y axis, and the evaluation numbers are shown on the X axis.

Output 37.1.2 Evaluation History Plot



The plot in Output 37.1.3 displays how the best found objective value and the elapsed time changed with each iteration of the tuner.

Output 37.1.3 Iteration History Plot



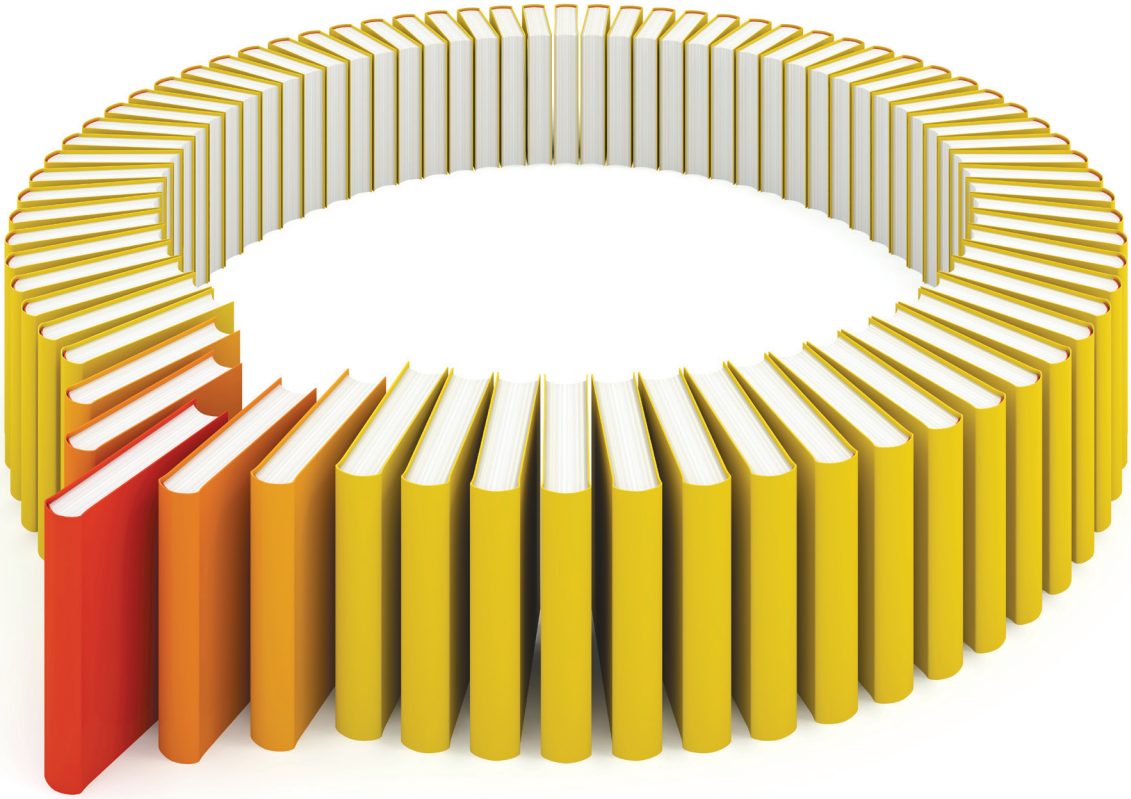
References

Fisher, R. A. (1936). "The Use of Multiple Measurements in Taxonomic Problems." *Annals of Eugenics* 7:179–188.

Mezzich, J. E., and Solomon, H. (1980). *Taxonomy and Behavioral Science*. New York: Academic Press.

Van der Maaten, L. (2014). "Accelerating t-SNE Using Tree-Based Algorithms." *Journal of Machine Learning Research* 15:3221–3245.

Van der Maaten, L., and Hinton, G. (2008). "Visualizing Data Using t-SNE." *Journal of Machine Learning Research* 9:2579–2605.



Gain Greater Insight into Your SAS® Software with SAS Books.

Discover all that you need on your journey to knowledge and empowerment.

support.sas.com/bookstore
for additional books and resources.

sas
THE POWER TO KNOW.®

SAS and all other SAS Institute Inc. product or service names are registered trademarks or trademarks of SAS Institute Inc. in the USA and other countries. ® indicates USA registration. Other brand and product names are trademarks of their respective companies. © 2013 SAS Institute Inc. All rights reserved. S107969US.0613