SAS® Visual Data Mining and Machine Learning 8.2
Procedures

The correct bibliographic citation for this manual is as follows: SAS Institute Inc. 2017. *SAS® Visual Data Mining and Machine Learning 8.2: Procedures*. Cary, NC: SAS Institute Inc.

SAS® Visual Data Mining and Machine Learning 8.2: Procedures

Copyright © 2017, 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.

U.S. Government License Rights; Restricted Rights: The Software and its documentation is commercial computer software developed at private expense and is provided with RESTRICTED RIGHTS to the United States Government. Use, duplication, or disclosure of the Software by the United States Government is subject to the license terms of this Agreement pursuant to, as applicable, FAR 12.212, DFAR 227.7202-1(a), DFAR 227.7202-3(a), and DFAR 227.7202-4, and, to the extent required under U.S. federal law, the minimum restricted rights as set out in FAR 52.227-19 (DEC 2007). If FAR 52.227-19 is applicable, this provision serves as notice under clause (c) thereof and no other notice is required to be affixed to the Software or documentation. The Government's rights in Software and documentation shall be only those set forth in this Agreement.

SAS Institute Inc., SAS Campus Drive, Cary, NC 27513-2414

June 2018

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 http://support.sas.com/thirdpartylicenses.

Contents

Chapter 1. What's New in SAS Visual Data Mining and Machine Learning 8.2 Procedures . . 1
Chapter 2. Introduction 7
Chapter 3. Shared Concepts 11
Chapter 4. The ASTORE Procedure 27
Chapter 5. The BNET Procedure 43
Chapter 6. The BOOLRULE Procedure 69
Chapter 7. The FACTMAC Procedure 99
Chapter 8. The FASTKNN Procedure 115
Chapter 9. The FISM Procedure 123
Chapter 10. The FOREST Procedure 131
Chapter 11. The GRADBOOST Procedure 167
Chapter 12. The GVARCLUS Procedure 201
Chapter 13. The MBANALYSIS Procedure 221
Chapter 14. The MWPCA Procedure 235
Chapter 15. The NNET Procedure 249
Chapter 16. The RPCA Procedure 281
Chapter 17. The SVDD Procedure 299
Chapter 18. The SVMACHINE Procedure 339
Chapter 19. The TEXTMINE Procedure 359
Chapter 20. The TMSCORE Procedure 409

Subject Index **417**

Syntax Index **423**

Chapter 1
What's New in SAS Visual Data Mining and Machine Learning 8.2 Procedures

Contents

Overview: What's New **1**
New Procedures **2**
The BNET Procedure 2
The FASTKNN Procedure 2
The FISM Procedure 2
The GVARCLUS Procedure 2
The MBANALYSIS Procedure 3
Enhancements to Procedures **3**
The ASTORE Procedure 3
The FACTMAC Procedure 3
The FOREST Procedure 4
The GRADBOOST Procedure 4
The MWPCA Procedure 5
The NNET Procedure 5
The RPCA Procedure 5
The SVDD Procedure 5
The SVMACHINE Procedure 6
References **6**

Overview: What's New

SAS Visual Data Mining and Machine Learning 8.2 contains several new procedures and several enhancements to procedures.

New Procedures

The 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 conditional dependency of the random 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, 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. It can load data from multiple nodes and perform computation in parallel.

The FASTKNN Procedure

The FASTKNN procedure implements the *k*-nearest neighbor (*k*-NN) search algorithm in SAS Viya. The *k*-NN algorithm has numerous applications, including recommendation systems, image search, fingerprint recognition and clustering. You can use the FASTKNN procedure to read and write data in distributed form, and to perform search in parallel by making full use of multicore computers or distributed computing environments.

The 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 named frequent-pattern tree (FP-tree), which retains the item set association information.

The FISM procedure is used by other procedures such as the MBANALYSIS procedure for association rule mining.

The GVARCLUS Procedure

The GVARCLUS procedure performs variable clustering and graphical modeling in SAS Viya. 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.

The 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.

Enhancements to Procedures

The ASTORE Procedure

- By specifying multiple analytic stores and DS2 scoring code, you can score an input data table and produce an output data table.
- The SETOPTION statement enables you to specify a numeric value for a named option, which is checked during scoring.
- All threads share one copy of the storage during scoring with DS2 scoring code.

The FACTMAC Procedure

- The new PRINTTARGET option in the PROC FACTMAC statement generates tables that contain naming information for the scored data.
- A new search method, Bayesian, is available in the AUTOTUNE statement.
- The new ROLE= option in the OUTPUT statement enables you to provide a name for the partition index.

The FOREST Procedure

- The new PRINTTARGET option in the PROC FOREST statement generates tables that contain naming information for the scored data.
- The new BINMETHOD= option in the PROC FOREST statement enables the use of either bucket binning or quantile binning of interval input variables.
- The new search method, Bayesian, is available in the AUTOTUNE statement.
- The new ROLE= option in the OUTPUT statement enables you to provide a name for the partition index.
- *(Experimental)* The new VII option in the PROC FOREST statement requests that the procedure calculate the variable interaction importance.

The GRADBOOST Procedure

- The new EARLYSTOP= option in the PROC GRADBOOST statement enables the use of validation data to stop model training early.
- The new ADDTREES= option in the PROC GRADBOOST statement requests that the procedure add trees to an already created gradient boosting model.
- The new RBAIMP option in the PROC GRADBOOST statement requests that the procedure calculate the random branch assignment variable importance.
- The new PRINTTARGET option in the PROC GRADBOOST statement generates tables that contain naming information for the scored data.
- The new BINMETHOD= option in the PROC GRADBOOST statement enables the use of either bucket binning or quantile binning of interval input variables.
- The new search method, Bayesian, is available in the AUTOTUNE statement.
- The new ROLE= option in the OUTPUT statement enables you to provide a name for the partition index.
- *(Experimental)* The new VII= option in the PROC GRADBOOST statement requests that the procedure calculate the variable interaction importance.

The MWPCA Procedure

- The SVD statement supports the METHOD=ITERATIVE and METHOD=RANDOM options.
- The WINSUMMARY= option in the OUTPUT statement enables you to specify a name of a windows summary table that includes information about each window, such as rank of the low-rank matrix, number of iterations, and solution status.

The NNET Procedure

- The new STAGNATION= and VALIDGOAL= early-stopping options control how to stop training by using a validation subset.
- The new TARGETSTANDARDIZE= option standardizes the interval target.
- The new PRINTTARGET option generates the "Predicted Probability Variables" and "Predicted Target Variable" tables.
- A new search method, Bayesian, is available in the AUTOTUNE statement.
- The new ROLE= option in the OUTPUT statement enables you to provide a name for the partition index.

The RPCA Procedure

- 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.
- The ID statement replaces the ROWID statement. The ID statement does not have the limitations of the ROWID statement; variables in the ID statement can be both interval and nominal.
- The SVD statement now supports the METHOD=ITERATIVE option.
- You can use PREFIX= option to specify a prefix for the principal components.

The SVDD Procedure

- The BW=MEAN option in the KERNEL statement enables you to request automatic selection of the Gaussian bandwidth parameter by using the mean criterion (Chaudhuri et al. 2017).
- The STD option in the PROC SVDD statement enables you to standardize the input training data.

- The SCORE option in the PROC SVDD statement enables you to score the training data observations and generate a frequency table of _SVDDDISTANCE_ values. You can use the NBINS option to specify the number of bins for the frequency table. The new Histogram ODS table includes the generated frequency table.
- You can include variables that have constant values for SVDD scoring by using the UCC option in the PROC SVDD statement. This option is useful in scenarios where certain variables have constant value in the training data but variation in the values of these variables is expected in the scoring data. The UCC option ensures that variables that have constant values are used for scoring.
- The default value of the STOL= option in the SOLVER statement is now 1E–4.
- The TrainingResults ODS table includes the number of dropped observations.
- The optimal value of the SVDD objective function that is reported in the OptSummary ODS table is now based on the SVDD formulation.

The SVMACHINE Procedure

- The new PRINTTARGET option generates the "Predicted Probability Variables" and "Predicted Target Variable" tables.
- A new search method, Bayesian, is available in the AUTOTUNE statement.
- The new ROLE= option in the OUTPUT statement enables you to provide a name for the partition index.

References

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.

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.

Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. San Francisco: Morgan Kaufmann.

Chapter 2
Introduction

Contents

Overview of SAS Visual Data Mining and Machine Learning Procedures **7**
Experimental Software **7**
About This Book **8**
Chapter Organization 8
Typographical Conventions 9
Options Used in Examples 9
Where to Turn for More Information **10**
Online Documentation 10
SAS Technical Support Services 10

Overview of SAS Visual Data Mining and Machine Learning Procedures

This book describes the data mining and machine learning procedures that are available in SAS Visual Data Mining and Machine Learning. These procedures provide data mining and machine learning algorithms that have been specially developed to take advantage of the distributed environment that SAS Viya 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 Visual Data Mining and Machine Learning provides procedures for sampling, data exploration, clustering, dimension reduction, model assessment, and additional supervised learning, which are described in *SAS Visual Statistics: Procedures Guide*.

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 any production release. 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 Language Reference: Concepts* and *SAS Visual Data Management and Utility 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 2, this chapter, provides an overview of the data mining and machine learning procedures that are available in SAS Visual Data Mining and Machine Learning, and it summarizes related information, products, and services.

Chapter 3 provides information about topics that are common to multiple procedures. Topics include how to use SAS Cloud Analytic Services (CAS) sessions and how to load a SAS data set onto a CAS server. 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	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 in an ODS destination statement as follows:

```
ods html style=HTMLBlue;
. . .
ods html close;

ods pdf style=PearlJ;
. . .
ods pdf close;
```

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 3
Shared Concepts

Contents

Introduction to Shared Concepts **11**
Using CAS Sessions and CAS Engine Librefs 11
Loading a SAS Data Set onto a CAS Server 12
Syntax Common to SAS Visual Data Mining and Machine Learning Procedures **14**
AUTOTUNE Statement 14
CODE Statement 18
PARTITION Statement 20
Details for SAS Visual Data Mining and Machine Learning Procedures **20**
Using Validation and Test Data 20
Multithreading 21
Hyperparameter Tuning 22
Data Partitioning 23
Hyperparameter Tuning Process 23
Determining the Number of Parallel Evaluations 24
References **25**

Introduction to Shared Concepts

This book describes data mining and machine learning procedures that run in SAS Viya. One component of SAS Viya is SAS Cloud Analytic Services (CAS), which is the analytic server and associated cloud services. The following subsections describe how to set up and use CAS sessions.

The section "Syntax Common to SAS Visual Data Mining and Machine Learning Procedures" on page 14 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 SAS Visual Data Mining and Machine Learning Procedures" on page 20 provides details that are common to some of the procedures in this book.

Using CAS Sessions and CAS Engine Librefs

SAS Cloud Analytic Services (CAS) is the analytic server and associated cloud services in SAS Viya. This section describes how to create a CAS session and set up a CAS engine libref that you can use to connect to

the CAS session. It assumes that you have a CAS server already available; contact your system administrator if you need help starting and terminating a server. This CAS server is identified by specifying the host on which it runs and the port on which it listens for communications. To simplify your interactions with this CAS server, the host information and port information for the server are stored as SAS option values that are retrieved automatically whenever this CAS server needs to be accessed. You can examine the host and port values for the server at your site by using the following statements:

```
proc options option=(CASHOST CASPORT);
run;
```

In addition to starting a CAS server, your system administrator might also have created a CAS session and a CAS engine libref for your use. You can define your own sessions and CAS engine librefs that connect to the CAS server as shown in the following statements:

```
cas mysess;
libname mycas cas sessref=mysess;
```

The CAS statement creates the CAS session named mysess, and the LIBNAME statement creates the mycas CAS engine libref that you use to connect to this session. It is not necessary to explicitly name the CASHOST and CASPORT of the CAS server in the CAS statement, because these values are retrieved from the corresponding SAS option values.

If you have created the mysess session, you can terminate it by using the TERMINATE option in the CAS statement as follows:

```
cas mysess terminate;
```

For more information about the CAS statement and the LIBNAME statement, see *SAS Cloud Analytic Services: Language Reference*. For general information about CAS and CAS sessions, see *SAS Cloud Analytic Services: Fundamentals*.

Loading a SAS Data Set onto a CAS Server

Procedures in this book require the input data to reside on a CAS server. To work with a SAS data set, you must first load the data set onto the CAS server. Data loaded on the CAS server are called *data tables*. This section lists three methods of loading a SAS data set onto a CAS server. In this section, mycas is the name of the caslib that is connected to the mysess CAS session.

- You can use a single DATA step to create a data table on the CAS server as follows:

```
data mycas.Sample;
   input y x @@;
   datalines;
.46 1 .47 2 .57 3 .61 4 .62 5 .68 6 .69 7
;
```

  Note that DATA step operations might not work as intended when you perform them on the CAS server instead of the SAS client.

- You can create a SAS data set first, and when it contains exactly what you want, you can use another DATA step to load it onto the CAS server as follows:

```
data Sample;
   input y x @@;
   datalines;
.46 1 .47 2 .57 3 .61 4 .62 5 .68 6 .69 7 .78 8
;
data mycas.Sample;
   set Sample;
run;
```

- You can use the CASUTIL procedure as follows:

```
proc casutil sessref=mysess;
   load data=Sample casout="Sample";
quit;
```

  The CASUTIL procedure can load data onto a CAS server more efficiently than the DATA step. For more information about the CASUTIL procedure, see *SAS Cloud Analytic Services: Language Reference*.

The mycas caslib stores the Sample data table, which can be distributed across many machine nodes. You must use a caslib reference in procedures in this book to enable the SAS client machine to communicate with the CAS session. For example, the following FACTMAC procedure statements use a data table that resides in the mycas caslib:

```
proc factmac data = mycas.Sample;
  ...statements...;
run;
```

You can delete your data table by using the DELETE procedure as follows:

```
proc delete data = mycas.Sample;
run;
```

The Sample data table is accessible only in the mysess session. When you terminate the mysess session, the Sample data table is no longer accessible from the CAS server. If you want your Sample data table to be available to other CAS sessions, then you must *promote* your data table. For more information about data tables, see *SAS Cloud Analytic Services: User's Guide*.

Syntax Common to SAS Visual Data Mining and Machine Learning Procedures

AUTOTUNE Statement

AUTOTUNE < *options* > **;**

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

Table 3.1 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 3.1. For more information about the TUNINGPARAMETERS= option, see the specific procedure chapter.

Table 3.1 AUTOTUNE Statement Options

Option	Description
EVALHISTORY=	Specifies how to report the evaluation history of the tuner
FRACTION=	Specifies the fraction of observations to use for validation
KFOLD=	Specifies the number of folds for *k*-fold cross validation
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
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
TARGETEVENT=	Specifies the target event for ROC-based calculations
TUNINGPARAMETERS=	Specifies the custom tuning parameters
USEPARAMETERS=	Specifies how to handle the TUNINGPARAMETERS= option

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 Evaluation-History 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

FRACTION=*number*

specifies the fraction of all data to be used 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). 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 a PARTITION statement is specified, 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.

By default, FRACTION=0.3.

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 hold-out 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 provides on average a better representation of error across the entire data table (a better generalization error).

If you do not specify either this option or the FRACTION= option, then the default of FRACTION=0.3 is used. If a PARTITION statement is specified, the validation partition defined in that statement is used, and this option is ignored. You cannot specify this option in combination with the FRACTION= option.

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.

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 is created by the tuner, with each using 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 24.

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	uses average squared error as the objective function.
AUC	uses area under the curve as the objective function (nominal type only).
F05	uses the F0.5 coefficient as the objective function (nominal type only).
F1	uses the F1 coefficient as the objective function (nominal type only).
GAMMA	uses the gamma coefficient as the objective function (nominal type only).
GINI	uses the Gini coefficient as the objective function (nominal type only).

KS uses the Kolmogorov-Smirnov coefficient as the objective function (nominal type only).

MAE uses the mean absolute error as the objective function (interval type only).

MCE uses the misclassification rate as the objective function (nominal type only).

MCLL uses the multiclass log loss as the objective function (nominal type only).

MISC uses the misclassification error percentage as the objective function (nominal type only).

MSE uses the mean squared error as the objective function (interval type only).

MSLE uses the mean squared logarithmic error as the objective function (interval type only).

RASE uses the root average squared error as the objective function.

RMAE uses the root mean absolute error as the objective function (interval type only).

RMSLE uses the root mean squared logarithmic error as the objective function (interval type only).

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

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 | 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.

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.

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 do not specify the TARGETEVENT= option, the tuner selects one of the target levels and uses it for calculating the specified objective function.

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=*tuning-parameter-option*
specifies which set of parameters to tune.

You can specify the following *tuning-parameter-options*:

STANDARD tunes using the default bounds and initial values for all parameters.

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

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

By default, USEPARAMETERS=COMBINED.

CODE Statement

CODE < *options* > **;**

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

Table 3.2 summarizes the *options* you can specify in the CODE statement.

Table 3.2 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
OUT=	Names an output CAS table in which to save the generated code

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 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 4 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.

OUT=*CAS-libref.data-table*
creates a CAS table that contains the score code. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, and *data-table* specifies the name of the output data table. For more information about this two-level name, see the section "Using CAS Sessions and CAS Engine Librefs" on page 11. You can specify this CAS table in the **runCodeTable** action in the **dataStep** action set to score another data table; for more information, see *SAS Viya: System Programming Guide*. This option cannot be specified with the FILE= option.

PCATALL
generates probabilities for all levels of categorical response variables.

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 20. 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 of the observations are 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.

Details for SAS Visual Data Mining and 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 logselect data=mycas.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 for a particular number of compute nodes. However, changing the number of compute nodes changes the initial distribution of data, resulting in different 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 logselect data=mycas.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

This section applies to the following procedures: FACTMAC, FOREST, GRADBOOST, NNET, SVMACHINE, TEXTMINE, and TMSCORE.

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 that runs in CAS is determined by your installation.

The tasks that are multithreaded by procedures that run in CAS 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: FACTMAC, FOREST, GRADBOOST, NNET, and SVMACHINE.

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 12, "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 24.

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 24). 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.

Alternate search methods can be specified: Bayesian, random, and LHS. If the search method is random or LHS, step 3 is eliminated; a single sample of candidate configurations is generated (either randomly or using a Latin hypercube sample) and evaluated as step 2.

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 23 is determined by the population size for GA or Bayesian search methods or by the sample size for random or LHS sampling. 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.

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 based on 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 GA or Bayesian search methods, or as the sampling size for the random or LHS search methods.

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. This limit can be overridden 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. This limit can be overridden 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 $2W/n$).

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 4
The ASTORE Procedure

Contents

Overview: ASTORE Procedure **27**
PROC ASTORE Features 28
Using CAS Sessions and CAS Engine Librefs 28
Getting Started: ASTORE Procedure **29**
Home Equity Data Set 29
Creating an Analytic Store 30
Using the Analytic Store 30
Syntax: ASTORE Procedure **31**
PROC ASTORE Statement 31
DESCRIBE Statement 32
DOWNLOAD Statement 32
SCORE Statement 33
SETOPTION Statement 34
UPLOAD Statement 34
Details: ASTORE Procedure **35**
ODS Table Names 35
Examples: ASTORE Procedure **36**
Example 4.1: Scoring a Single Record 36
Example 4.2: Describing the Store 37
Example 4.3: Downloading the Store to the Local File System 40
Example 4.4: Uploading the Local Store from the Local File System 40
Example 4.5: Scoring with Multiple Stores 40

Overview: ASTORE Procedure

The ASTORE procedure is an interactive procedure in which each statement runs immediately. You can use the ASTORE procedure describe an analytic store, manage it, and score new data with the analytic store. The analytic store is the result of a SAVESTATE statement from another analytic procedure; it is a binary file that contains that procedure's state after it completes the training phase of data analysis.

PROC ASTORE Features

The ASTORE procedure enables you to do the following:

- describe limited information about the analytic store
- move analytic stores between the client and the server
- produce different types of DS2 scoring code that can run locally using the DS2 procedure
- produce DS2 language scoring code that can run in SAS Viya
- 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)

Using CAS Sessions and CAS Engine Librefs

SAS Cloud Analytic Services (CAS) is the analytic server and associated cloud services in SAS Viya. This section describes how to create a CAS session and set up a CAS engine libref that you can use to connect to the CAS session. It assumes that you have a CAS server already available; contact your system administrator if you need help starting and terminating a server. This CAS server is identified by specifying the host on which it runs and the port on which it listens for communications. To simplify your interactions with this CAS server, the host information and port information for the server are stored as SAS option values that are retrieved automatically whenever this CAS server needs to be accessed. You can examine the host and port values for the server at your site by using the following statements:

```
proc options option=(CASHOST CASPORT);
run;
```

In addition to starting a CAS server, your system administrator might also have created a CAS session and a CAS engine libref for your use. You can define your own sessions and CAS engine librefs that connect to the CAS server as shown in the following statements:

```
cas mysess;
libname mycas cas sessref=mysess;
```

The CAS statement creates the CAS session named mysess, and the LIBNAME statement creates the mycas CAS engine libref that you use to connect to this session. It is not necessary to explicitly name the CASHOST and CASPORT of the CAS server in the CAS statement, because these values are retrieved from the corresponding SAS option values.

If you have created the mysess session, you can terminate it by using the TERMINATE option in the CAS statement as follows:

```
cas mysess terminate;
```

For more information about the CAS and LIBNAME statements, see the section "Introduction to Shared Concepts" on page 11 in Chapter 3, "Shared Concepts."

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 into your CAS session by specifying your CAS engine libref in the following DATA step. These statements assume that your CAS engine libref is named mycas (as in the section "Using CAS Sessions and CAS Engine Librefs" on page 28), but you can substitute any appropriately defined CAS engine 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 mycas.hmeq;
   set sampsio.hmeq;
   id = _N_;
run;
```

Because distributed computing orders distributed data differently than traditional SAS procedures orders them, one or more variables that can act as the record identifier must be in the input 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 18, "The 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 mycas.savehmeq.

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

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 CAS.

In this example, the input data table is mycas.hmeq, the output data table is mycas.scoreout1, and the analytic store is in the data table mycas.savehmeq. All the input tables must be loaded in your CAS session. The resulting output table is created in the same CAS session.

```
proc astore;
   score data=mycas.hmeq
         out=mycas.scoreout1
         rstore=mycas.savehmeq;
quit;
data scoreout1;
   set mycas.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	1.0000000088	7.2815958E-8	0.9999999272	0	
2	2	0.9999999691	9.2701136E-8	0.9999999073	0	
3	3	1.0000000096	7.2405865E-8	0.9999999276	0	
4	4	-0.846156581	0.6410258508	0.3589741492	1	M
5	5	1.0000000505	5.1990216E-8	0.999999948	0	

Syntax: ASTORE Procedure

The following statements are available in the ASTORE procedure:

PROC ASTORE ;
  SCORE *score-options* ;
  DESCRIBE *describe-options* ;
  DOWNLOAD *download-options* ;
  UPLOAD *upload-options* ;
  SETOPTION *name value* ;

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.

DESCRIBE Statement

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

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

The DESCRIBE statement specifies the name or identifier of an analytic store either in the local file system by using the STORE= option or in a data table stored in CAS by using the RSTORE= option. The DESCRIBE statement can also produce DS2 basic scoring code. 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 send 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 another procedure that processed a SAVESTATE statement.

RSTORE=*CAS-libref.data-table*
specifies the CAS table that contains the analytic store. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 28.

You can also specify the following *describe-option*:

EPCODE< =*code-file-name* **>**
sends DS2 language code either to the SAS log (if you do not specify a *code-file-name*) or to an external code file that can run in CAS and that is identified by *code-file-name*, which is either the file reference or the full path 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.

DOWNLOAD Statement

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

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

You must specify the following options in any order:

RSTORE=*CAS-libref.data-table*
specifies a data table that contains the state to be downloaded. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 28.

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.

SCORE Statement

SCORE **DATA=***CAS-libref.data-table* **OUT=***CAS-libref.data-table* **RSTORE=***CAS-libref.data-table* < . . . > < **RSTORE=***CAS-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 *score-options*:

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

CAS-libref refers to a collection of information that is defined in the LIBNAME statement and includes the caslib, which includes a path to the data, and a session identifier, which defaults to the active session but which can be explicitly defined in the LIBNAME statement. For more information about *CAS-libref*, see the section "Using CAS Sessions and CAS Engine Librefs" on page 28.

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

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

CAS-libref refers to a collection of information that is defined in the LIBNAME statement and includes the caslib, which includes a path to where the data table is to be stored, and a session identifier, which defaults to the active session but which can be explicitly defined in the LIBNAME statement. For more information about *CAS-libref*, see the section "Using CAS Sessions and CAS Engine Librefs" on page 28.

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

RSTORE=*CAS-libref.model-file-name*
specifies the data table in CAS to contain the analytic store. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 28.

You must specify at least one RSTORE= option. You can specify as many additional RSTORE= options as you need. When you specify more than one RSTORE= option, you must specify EPCODE= option.

You can also specify the following options:

COPYVAR=*variable*

COPYVARS=(*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.

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

SETOPTION Statement

SETOPTION *name value* **;**

The SETOPTION statement associates a numeric *value* to a specified option *name*. During scoring with an analytic store, the value of the named option is checked.

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

value
specifies a numeric *value* for the named option.

UPLOAD Statement

UPLOAD *upload-options* **;**

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

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

RSTORE=*CAS-libref.data-table*
specifies the CAS table to which the store is sent. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 28.

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 basic information about the store:

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

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.

The UPLOAD statement produces output that includes the store key for future reference in the DS2 language code. The code that is produced by the EPCODE= option in the DESCRIBE statement includes the same key. If you upload the store to CAS and you specify the EPCODE= option in the DESCRIBE statement again, you will observe the same key. The key of the store is dependent on the store and not on whether the store is located in the local file system or in CAS.

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 4.2.

Table 4.2 ODS Tables Produced by PROC ASTORE

Table Name	Description	Statement
Key	Key information from the UPLOAD statement	UPLOAD
	Key information from the DOWNLOAD statement	DOWNLOAD
	Key information from the DESCRIBE statement	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
Description	Name of the component that saved the state and the time it was saved	DESCRIBE
Timing	Timing details **NOTE:** Analytic stores can become very large; breaking down the different stages of scoring a data table in CAS is important and informative.	SCORE

Examples: ASTORE Procedure

NOTE: Input data must be in a CAS table that is accessible in your CAS session. You must refer to this table by using a two-level name. The first level must be a CAS engine libref, and the second level must be the table name. For more information, see the sections "Using CAS Sessions and CAS Engine Librefs" on page 11 and "Loading a SAS Data Set onto a CAS Server" on page 12 in Chapter 3, "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 30.

The following DATA step creates the mycas.hmeq1 table in your CAS session by loading the hmeq data table that was created in section "Home Equity Data Set" on page 29 and extracting the record whose ID is 100:

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

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

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

Output 4.1.1 Scoring a Single Record

Obs	id	_P_	P_BAD1	P_BAD0	I_BAD	_WARN_
1	100	1.0000000287	6.2877921E-8	0.9999999371	0	M

Example 4.2: Describing the Store

In this simple example, no DS2 language code is requested. The DESCRIBE statement produces tables that describe some of the contents store as seen in Output 4.2.1 and sends the basic code to the file *svmepcode.sas*.

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

Output 4.2.1 contains the following ODS tables:

- The “Key Information” table displays the string identifier of the store. This is the same string that is contained in the code that the EPCODE= option in 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

Key Information
14F2E0A7B9A73581F79904C6E900D60ABCC51EDB

Output 4.2.1 *continued*

Basic Information	
Analytic Engine	svmachine
Time Created	20Aug2016:20:01:08

Input variables				
Name	Length	Role	Measurement Level	Data Type
LOAN	8	Input	Interval	Num
MORTDUE	8	Input	Interval	Num
VALUE	8	Input	Interval	Num
REASON	7	Input	Classification	Char
JOB	7	Input	Classification	Char
DELINQ	8	Input	Classification	Num
NINQ	8	Input	Classification	Num
BAD	8	Target	Classification	Num
id	8	Id	Interval	Num

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	32	Char	Into: BAD
WARN	4	Char	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(32) "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 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: Downloading the Store to the Local File System

The following statements extract the analytic store that is saved in the data table mycas.savehmeq from the CAS session to the local file *svmlocalcopy* in the local file system.

```
proc astore;
  download rstore=mycas.savehmeq
           store="svmlocalcopy";
quit;
```

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

Example 4.4: Uploading the Local Store from the Local File System

The following statements send the analytic store *svmlocalcopy* from the local file system to the data table mycas.savehmeq in the CAS session.

```
proc astore;
  upload rstore=mycas.savehmeqnew
         store="svmlocalcopy";
quit;
```

The UPLOAD statement produces the store key in the listing. You can use this key to construct the embedded processing code on your own, but it is simpler for you to use the EPCODE= option in the DESCRIBE statement to produce the resulting minimal code and then edit the contents.

Example 4.5: Scoring with Multiple Stores

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, "The FOREST Procedure."

This example uses the same home equity data table (mycas.hmeq) that was loaded in the session as described in section "Home Equity Data Set" on page 29. The SAVESTATE statement in the following FOREST procedure code saves the state in an analytic store, which is stored in the table mycas.savehmeq2:

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

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

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

The contents of the *epcode2.sas* file are as follows.

```
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(32) "I_BAD" having label n'Into: BAD';
     dcl nchar(4) "_WARN_" having label n'Warnings';
     dcl double "tempP1";
     dcl double "tempP2";
     Keep
             "id"
             "_P_"
             "P_BAD1"
             "P_BAD0"
             "I_BAD"
             "_WARN_"
            ;
     varlist allvars[_all_];
     method init();
             sc.setvars(allvars);
             sc.setKey(n'14F2E0A7B9A73581F79904C6E900D60ABCC51EDB');
             sc2.setvars(allvars);
             sc2.setKey(n'FC7E82345DF91C75364F7BFD177BF93AA9129A26');
     end;
     method preScoreRecord();
     end;
     method postScoreRecord();
             P_BAD0 = 0.5*(  P_BAD0 + tempP1);
             P_BAD1 = 0.5*(  P_BAD1 + tempP2);
     end;
     method term();
     end;
     method run();
             set sasep.in;
             preScoreRecord();
             sc.scoreRecord();
```

```
                tempP1 = P_BAD0;
                tempP2 = P_BAD1;
                sc2.scoreRecord();
                postScoreRecord();
        end;
    enddata;
```

Chapter 5
The BNET Procedure

Contents

Overview: BNET Procedure **44**
PROC BNET Features 44
Using CAS Sessions and CAS Engine Librefs 44
Getting Started: BNET Procedure **45**
Structure Learning 46
Probability Table 47
Variable Selection 47
Syntax: BNET Procedure **48**
PROC BNET Statement 48
CODE Statement 53
FREQ Statement 53
ID Statement 53
INPUT Statement 54
OUTPUT Statement 54
PARTITION Statement 55
SAVESTATE Statement 55
TARGET Statement 56
Details: BNET Procedure **56**
Independence Tests 56
Variable Selection 57
Structure Learning 58
Parameter Learning 59
Displayed Output 60
ODS Table Names 60
Examples: BNET Procedure **61**
Example 5.1: Naive Bayesian Network 61
Example 5.2: Tree-Augmented Naive Bayesian Network 62
Example 5.3: Parent-Child Bayesian Network 63
Example 5.4: Markov Blanket 63
Example 5.5: Bayesian Network-Augmented Naive Bayesian Network 64
Example 5.6: Model Selection 65
References **68**

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 conditional dependency of the random 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, 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. It can load data from multiple nodes and perform computation in parallel.

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
- 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 CAS Sessions and CAS Engine Librefs

SAS Cloud Analytic Services (CAS) is the analytic server and associated cloud services in SAS Viya. This section describes how to create a CAS session and set up a CAS engine libref that you can use to connect to the CAS session. It assumes that you have a CAS server already available; contact your system administrator if you need help starting and terminating a server. This CAS server is identified by specifying the host on which it runs and the port on which it listens for communications. To simplify your interactions with this CAS server, the host information and port information for the server are stored as SAS option values that are retrieved automatically whenever this CAS server needs to be accessed. You can examine the host and port values for the server at your site by using the following statements:

```
proc options option=(CASHOST CASPORT);
run;
```

In addition to starting a CAS server, your system administrator might also have created a CAS session and a CAS engine libref for your use. You can define your own sessions and CAS engine librefs that connect to the CAS server as shown in the following statements:

```
cas mysess;
libname mycas cas sessref=mysess;
```

The CAS statement creates the CAS session named mysess, and the LIBNAME statement creates the mycas CAS engine libref that you use to connect to this session. It is not necessary to explicitly name the CASHOST and CASPORT of the CAS server in the CAS statement, because these values are retrieved from the corresponding SAS option values.

If you have created the mysess session, you can terminate it by using the TERMINATE option in the CAS statement as follows:

```
cas mysess terminate;
```

For more information about the CAS and LIBNAME statements, see the section "Introduction to Shared Concepts" on page 11 in Chapter 3, "Shared Concepts."

Getting Started: BNET Procedure

NOTE: Input data must be in a CAS table that is accessible in your CAS session. You must refer to this table by using a two-level name. The first level must be a CAS engine libref, and the second level must be the table name. For more information, see the sections "Using CAS Sessions and CAS Engine Librefs" on page 11 and "Loading a SAS Data Set onto a CAS Server" on page 12 in Chapter 3, "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 CAS session by naming your CAS engine libref in the first statement of the following DATA step:

```
data mycas.neuralgia;
   set neuralgia;
run;
```

These statements assume that your CAS engine libref is named mycas, but you can substitute any appropriately defined CAS engine 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=mycas.neuralgia numbin=3 outnetwork=mycas.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=mycas.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=mycas.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 in Figure 5.1 does not include variables Sex or Duration, because PROC BNET automatically selects those variables by using independence tests. PROC BNET produces Figure 5.3, which shows the variable selection results. 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 for the ALPHA= option, which is not specified). Sex is conditionally independent of Pain given Treatment; therefore, PROC BNET also removes it 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	

Syntax: BNET Procedure

The following statements are available in the BNET procedure:

PROC BNET < *options* > **;**
CODE FILE=*filename* **;**
FREQ *variable* **;**
ID *variables* **;**
INPUT *variables* / < **LEVEL=INTERVAL** | **NOMINAL** > **;**
OUTPUT OUT=*CAS-libref.data-table* < *option* > **;**
PARTITION *partition-option* **;**
SAVESTATE RSTORE=*CAS-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
NUMBIN=	Specifies the binning number for interval variables
PRESCREENING=	Specifies the initial screening for the input variables
VARSELECT=	Specifies the selection for the input variables
MISSINGINT=	Specifies how to handle missing values for interval variables

Table 5.1 *continued*

Option	Description
MISSINGNOM=	Specifies how to handle missing values for nominal variables
Independence Test Options	
INDEPTEST=	Specifies the methods for independence tests
ALPHA=	Specifies the significance level for independence tests by using chi-square or G-square statistics
MIALPHA=	Specifies the significant level for independence tests by using mutual information
Structure Learning Options	
STRUCTURE=	Specifies the network structure types
PARENTING=	Specifies the structure learning methods
MAXPARENTS=	Specifies the maximum number of parents allowed for each node in the network
Model Selection Options	
BESTMODEL	Requests that the best model be selected

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

requests that the best model be selected 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 on the validation data to automatically decide the best set of parameter values among these options.

By default, a best model is not selected.

DATA=*CAS-libref.data-table*

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

CAS-libref refers to a collection of information that is defined in the LIBNAME statement and includes the caslib, which includes a path to the data, and a session identifier, which defaults to the active session but which can be explicitly defined in the LIBNAME statement. For more information about *CAS-libref*, see the section "Using CAS Sessions and CAS Engine Librefs" on page 44.

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

INDEPTEST=ALL | CHIGSQUARE | CHISQUARE | GSQUARE | MI
specifies the method 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.

MAXPARENTS=*integer*
specifies the maximum number of parents that is 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 the interval variables. This option applies to training data, validation, testing data and any data 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 the nominal 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 1024, inclusive.

By default, NUMBIN=5.

OUTNETWORK=*CAS-libref.data-table*
OUTNET=*CAS-libref.data-table*
names the CAS data table to contain the network structure and the probability distributions. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 44.

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. If you specify both but you do not specify the BESTMODEL option, PROC BNET uses the first specified value, and ignores the other.

By default, PRESCREENING=1.

PRINTTARGET

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

By default, these two tables are not generated.

STRUCTURE=MB | NAIVE | PC | TAN

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 the following values:

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 PRESCREENING= and VARSELECT= options.

NAIVE assumes a naive Bayesian network structure (that is, the target has a direct link to each input variable). If MAXPARENTS=1, the structure is a naive Bayesian network (NB). If MAXPARENTS 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 PC.

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 conditionally dependent on 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 conditionally dependent on 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.

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 rolls back 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. If you specify multiple values for the VARSELECT= but you do not specify the BESTMODEL option, PROC BNET uses the first specified value, and ignores the remaining values.

By default, VARSELECT=1.

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.

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. PROC BNET does not support duplicate input variables. If the INPUT statement contains a duplicate variable, PROC BNET returns an error message and then exits. 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=*CAS-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=*CAS-libref.data-table*
names the output data table for PROC BNET to use. You must specify this option before any other options. *CAS-libref.data-table* is a two-level name, where

CAS-libref refers to a collection of information that is defined in the LIBNAME statement and includes the caslib, which includes a path to where the data table is to be stored, and a session identifier, which defaults to the active session but which can be explicitly defined in the LIBNAME statement. For more information about *CAS-libref*, see the section "Using CAS Sessions and CAS Engine Librefs" on page 44.

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 20 in Chapter 3, "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 of the observations are 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=*CAS-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, "The ASTORE Procedure."

You must specify the following option:

RSTORE=*CAS-libref.data-table*
specifies a data table in which to save the analytic store for the model. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 44.

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.

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).

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), and Markov blanket (MB). 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.

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.

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 number of observations and the number of observations 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.

Iteration Report

The "Iteration Report" table contains the variable selection results.

Validation Information

The "Validation Information" table contains the validation results. If the PARTITION statement is specified, then the misclassification errors mean misclassification errors in the validation data; if not, then the misclassification errors mean 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 5.2.

Table 5.2 ODS Tables Produced by PROC BNET

Table Name	Description	Statement	Option
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
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
VarOrder	Contains the order of the input variables	PROC BNET	Default
VarSelect	Contains the variable selection results	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 CAS session. This DATA step assumes that your CAS engine libref is named mycas, but you can substitute any appropriately defined CAS engine libref.

```
data mycas.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=mycas.iris numbin=3 structure=Naive maxparents=1
        prescreening=0 varselect=0
        outnetwork=mycas.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=mycas.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 mycas.iris;
   set sashelp.iris;
run;

 proc bnet data=mycas.iris numbin=3 structure=TAN
            prescreening=0 varselect=0
            outnetwork=mycas.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=mycas.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
SepalWidth	SepalLength
Species	SepalWidth
PetalWidth	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 mycas.iris;
   set sashelp.iris;
run;

 proc bnet data=mycas.iris numbin=3 structure=PC
             prescreening=0 varselect=0
             outnetwork=mycas.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=mycas.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 mycas.hmeq;
   set sampsio.hmeq;
run;
```

```
proc bnet data=mycas.hmeq indeptest=MI mialpha=0.2 structure=MB nbin=5
        missingint=IMPUTE missingnom=LEVEL
        outnetwork=mycas.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=mycas.network noobs label;
    var _parentnode_ _childnode_;
    where _type_="STRUCTURE";
run;
```

Output 5.4.1 Markov Blanket Network Structure

Parent Node	Child Node
BAD	DELINQ
DEROG	DELINQ
NINQ	DELINQ
BAD	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 mycas.hmeq;
   set sampsio.hmeq;
run;

proc bnet data=mycas.hmeq numbin=10 alpha=0.1 structure=Naive
            missingint=IMPUTE missingnom=LEVEL
            outnetwork=mycas.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=mycas.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: 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 mycas.dmagecr;
   set dmagecr_part;
run;

 proc bnet data=mycas.dmagecr numbin=10 alpha=0.05
          structure=Naive TAN PC MB varselect=0 1 bestmodel
          outnetwork=mycas.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 5.6.1 shows the information of number of observations: 695 observations are used for training and 305 are used for validation.

Output 5.6.1 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 5.6.2 shows the fit statistics. In the resulting network, there are 13 nodes, 13 links between the nodes, and the number of parameters is 129.

Output 5.6.2 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

Output 5.6.3 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 MAXPARENTS is greater than or equal to 2. The TAN structure has 77 misclassification errors when VARSELECT=0 and MAXPARENTS=2. The naive Bayesian network has 81 misclassification errors when VARSELECT=0 and MAXPARENTS is greater than or equal to 2. The MB Bayesian network structure has 113 misclassification errors.

Output 5.6.3 Model Selection: Validation Information

Validation Information							
Best Model	Misclassification Errors	Significance Threshold	Prescreening	Variable Selection	Structure	Parenting Method	Max N Parents
Yes	73	0.05	1	0	PC	BestSet	2
	73	0.05	1	0	PC	BestSet	3
	73	0.05	1	0	PC	BestSet	4
	73	0.05	1	0	PC	BestSet	5
	77	0.05	1	0	TAN	BestSet	2
	81	0.05	1	0	Naive	BestSet	2
	81	0.05	1	0	Naive	BestSet	3
	81	0.05	1	0	Naive	BestSet	4
	81	0.05	1	0	Naive	BestSet	5
	82	0.05	1	1	PC	BestSet	2
	82	0.05	1	1	PC	BestSet	3
	82	0.05	1	1	PC	BestSet	4
	82	0.05	1	1	PC	BestSet	5
	86	0.05	1	0	PC	BestSet	1
	86	0.05	1	0	Naive	BestSet	1
	106	0.05	1	1	PC	BestSet	1
	106	0.05	1	1	Naive	BestSet	1
	106	0.05	1	1	Naive	BestSet	2
	106	0.05	1	1	Naive	BestSet	3
	106	0.05	1	1	Naive	BestSet	4
	106	0.05	1	1	Naive	BestSet	5
	109	0.05	1	1	TAN	BestSet	2
	113	0.05	1	3	MB	BestSet	1
	113	0.05	1	3	MB	BestSet	2
	113	0.05	1	3	MB	BestSet	3
	113	0.05	1	3	MB	BestSet	4
	113	0.05	1	3	MB	BestSet	5

The following statements produce Output 5.6.4, 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=mycas.network noobs label;
    var _parentnode_ _childnode_;
    where _type_="STRUCTURE";
run;
```

Output 5.6.4 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

References

Pearl, J. (1988). *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference*. San Francisco: Morgan Kaufmann.

Chapter 6
The BOOLRULE Procedure

Contents

Overview: BOOLRULE Procedure **70**
PROC BOOLRULE Features 70
Using CAS Sessions and CAS Engine Librefs 71
Getting Started: BOOLRULE Procedure **71**
Syntax: BOOLRULE Procedure **75**
PROC BOOLRULE Statement 75
DOCINFO Statement 78
OUTPUT Statement 79
SCORE Statement 80
TERMINFO Statement 80
Details: BOOLRULE Procedure **81**
BOOLLEAR for Boolean Rule Extraction 81
Term Ensemble Process 82
Rule Ensemble Process 83
Measurements Used in BOOLLEAR 84
Precision, Recall, and the F1 Score 84
g-Score 85
Estimated Precision 85
Improvability Test 86
Shrinking the Search Space 86
Feature Selection 86
Significance Testing 86
k-Best Search 87
Improvability Test 87
Early Stop Based on the F1 Score 87
Output Data Sets 87
CANDIDATETERMS= Data Table 87
RULES= Data Table 88
RULETERMS= Data Table 89
Scoring Data Set 89
OUTMATCH= Data Table 89
Examples: BOOLRULE Procedure **90**
Example 6.1: Rule Extraction for Binary Targets 90
Example 6.2: Rule Extraction for a Multiclass Target 92
Example 6.3: Using Events in Rule Extraction 94
Example 6.4: Scoring 95
References **98**

Overview: BOOLRULE Procedure

The BOOLRULE procedure is a SAS Viya procedure that enables you to extract Boolean rules from large-scale transactional data.

The BOOLRULE procedure can automatically generate a set of Boolean rules by analyzing a text corpus that has been processed by the TEXTMINE procedure and is represented in a transactional format. For example, the following rule set is generated for documents that are related to bank interest:

```
(cut ^ rate ^ bank ^ percent ^ ~sell)       or
(market ^ money ^ ~year ^ percent ^ ~sale)  or
(repurchase ^ fee)                          or
(rate ^ prime rate)                         or
(federal ^ rate ^ maturity)
```

In this example, **^** indicates a logical "and," and **~** indicates a logical negation. The first line of the rule set says that if a document contains the terms "cut," "rate," "bank," and "percent," but does not contain the term "sell," it belongs to the bank interest category.

The BOOLRULE procedure has three advantages when you use a supervised rule-based model to analyze your large-scale transactional data. First, it focuses on modeling the positive documents in a category. Therefore, it is more robust when the data are imbalanced.[1] Second, the rules can be easily interpreted and modified by a human expert, enabling better human-machine interaction. Third, the procedure adopts a set of effective heuristics to significantly shrink the search space for search rules, and its basic operations are set operations, which can be implemented very efficiently. Therefore, the procedure is highly efficient and can handle very large-scale problems.

PROC BOOLRULE Features

The BOOLRULE procedure processes large-scale transactional data in parallel to achieve efficiency and scalability. The following list summarizes the basic features of PROC BOOLRULE:

- Boolean rules are automatically extracted from large-scale transactional data.
- The extracted rules can be easily understood and tuned by humans.
- Important features are identified for each category.
- Imbalanced data are handled robustly.
- Binary-class and multiclass categorization are supported.
- Events for defining labels for documents are supported.
- All processing phases use a high degree of multithreading.

[1] A data table is imbalanced if it contains many more negative samples than positive samples, or vice versa.

Using CAS Sessions and CAS Engine Librefs

SAS Cloud Analytic Services (CAS) is the analytic server and associated cloud services in SAS Viya. This section describes how to create a CAS session and set up a CAS engine libref that you can use to connect to the CAS session. It assumes that you have a CAS server already available; contact your system administrator if you need help starting and terminating a server. This CAS server is identified by specifying the host on which it runs and the port on which it listens for communications. To simplify your interactions with this CAS server, the host information and port information for the server are stored as SAS option values that are retrieved automatically whenever this CAS server needs to be accessed. You can examine the host and port values for the server at your site by using the following statements:

```
proc options option=(CASHOST CASPORT);
run;
```

In addition to starting a CAS server, your system administrator might also have created a CAS session and a CAS engine libref for your use. You can define your own sessions and CAS engine librefs that connect to the CAS server as shown in the following statements:

```
cas mysess;
libname mycas cas sessref=mysess;
```

The CAS statement creates the CAS session named mysess, and the LIBNAME statement creates the mycas CAS engine libref that you use to connect to this session. It is not necessary to explicitly name the CASHOST and CASPORT of the CAS server in the CAS statement, because these values are retrieved from the corresponding SAS option values.

If you have created the mysess session, you can terminate it by using the TERMINATE option in the CAS statement as follows:

```
cas mysess terminate;
```

For more information about the CAS and LIBNAME statements, see the section "Introduction to Shared Concepts" on page 11 in Chapter 3, "Shared Concepts."

Getting Started: BOOLRULE Procedure

NOTE: Input data must be in a CAS table that is accessible in your CAS session. You must refer to this table by using a two-level name. The first level must be a CAS engine libref, and the second level must be the table name. For more information, see the sections "Using CAS Sessions and CAS Engine Librefs" on page 11 and "Loading a SAS Data Set onto a CAS Server" on page 12 in Chapter 3, "Shared Concepts."

The following DATA step creates a data table that contains 20 observations that have three variables. The Text variable contains the input documents. The apple_fruit variable contains the label of documents: a value of 1 indicates that the document is related to the apple as the fruit or to the apple tree. The DID variable contains the ID of the documents. Each row in the data table represents a document for analysis.

```
data mycas.getstart;
    infile datalines delimiter='|' missover;
    length text $150;
    input text$ apple_fruit did$;
    datalines;
        Delicious and crunchy apple is one of the popular fruits | 1 |d01
        Apple was the king of all fruits. | 1 |d02
        Custard apple or Sitaphal is a sweet pulpy fruit | 1 |d03
        apples are a common tree throughout the tropics | 1 |d04
        apple is round in shape, and tasts sweet | 1 |d05
        Tropical apple trees produce sweet apple| 1| d06
        Fans of sweet apple adore Fuji because it is the sweetest of| 1 |d07
        this apple tree is small | 1 |d08
        Apple Store shop iPhone x and iPhone x Plus.| 0 |d09
        See a list of Apple phone numbers around the world.| 0 |d10
        Find links to user guides and contact Apple Support, | 0 |d11
        Apple counters Samsung Galaxy launch with iPhone gallery | 0 |d12
        Apple Smartphones - Verizon Wireless.| 0 |d13
        Apple mercurial chief executive, was furious.| 0 |d14
        Apple has upgraded the phone.| 0 |d15
        the great features of the new Apple iPhone x.| 0 |d16
        Apple sweet apple iphone.| 0 |d17
        Apple apple will make cars | 0 |d18
        Apple apple also makes watches| 0 |d19
        Apple apple makes computers too| 0 |d20
    ;
run;
```

These statements assume that your CAS engine libref is named mycas, but you can substitute any appropriately defined CAS engine libref.

The following statements use the TEXTMINE procedure to parse the input text data. The generated term-by-document matrix is stored in a data table named mycas.bow. The summary information about the terms in the document collection is stored in a data table named mycas.terms.

```
proc textmine data=mycas.getstart language="english";
    doc_id
        did;
    var
        text;
    parse
        nonoungroups
        entities    = none
        outparent   = mycas.bow
        outterms    = mycas.terms
        reducef     = 1;
run;
```

The following statements use the BOOLRULE procedure to extract rules:

```
proc boolrule
        data        =   mycas.bow
        docid       =   _document_
        termid      =   _termnum_
        docinfo     =   mycas.getstart
        terminfo    =   mycas.terms
        minsupports =   1
        mpos        =   1
        gpos        =   1;
    docinfo
        id          =   did
        targets     =   (apple_fruit);
    terminfo
        id          =   key
        label       =   term;
    output
        rules       =   mycas.rules
        ruleterms   =   mycas.ruleterms;
run;
```

The mycas.bow and mycas.terms data sets are specified as input in the DATA= and TERMINFO= options, respectively, in the PROC BOOLRULE statement. In addition, the DOCID= and TERMID= options in the PROC BOOLRULE statement specify the columns of the mycas.bow data table that contain the document ID and term ID, respectively.

The DOCINFO statement specifies the following information about the mycas.GetStart data table:

- The ID= option specifies the column that contains the document ID. The variables in this column are matched to the document ID variable that is specified in the DOCID= option in the PROC BOOLRULE statement in order to fetch target information about documents for rule extraction.
- The TARGETS= option specifies the target variables.

The TERMINFO statement specifies the following information about the mycas.terms data table:

- The ID= option specifies the column that contains the term ID. The variables in this column are matched to the term ID variable that is specified in the TERMID= option in the PROC BOOLRULE statement in order to fetch information about terms for rule extraction.
- The LABEL= option specifies the column that contains the text of the terms.

The OUTPUT statement requests that the extracted rules be stored in the data table mycas.Rules.

Figure 6.1 shows the SAS log that PROC BOOLRULE generates; the log provides information about the default configurations used by the procedure, about where the procedure runs, and about the input and output files. The log shows that the mycas.rules data table contains two observations, indicating that the BOOLRULE procedure identified two rules for the apple_fruit category.

Figure 6.1 SAS Log

```
NOTE: Neither SEQCOVER nor NOSEQCOVER is specified. SEQCOVER is used by default.
NOTE: The UUID '30484c7f-c455-834d-9db8-cd813d0e35dc' is connected using
      session MYSESS.
NOTE: The Cloud Analytic Services server processed the request in 0.045513
      seconds.
NOTE: The data set MYCAS.RULES has 2 observations and 15 variables.
```

The following statements PROC PRINT to show the contents of the mycas.rules data table that the BOOLRULE procedure generates:

```
proc print data = mycas.rules;
var target ruleid rule F1 precision recall;
run;
```

Figure 6.2 shows the output of PROC PRINT, which contains two rules. For information about the output of the RULES= option, see the section "RULES= Data Table" on page 88.

Figure 6.2 The mycas.rules Data Table

Obs	TARGET	RULEID	RULE	F1	PRECISION	RECALL
1	apple_fruit	1	be & apple	0.85714	1	0.75
2	apple_fruit	2	tree	1.00000	1	1.00

The following statements run the BOOLRULE procedure to match rules in documents and run PROC PRINT to show the results:

```
proc boolrule
        data        =   mycas.bow
        docid       =   _document_
        termid      =   _termnum_;
    score
        ruleterms   =   mycas.ruleterms
        outmatch    =   mycas.matches;
run;
proc print data=mycas.matches;
run;
```

Figure 6.3 shows the output of PROC PRINT, the mycas.matches data table. For information about the output of the OUTMATCH= option, see the section "OUTMATCH= Data Table" on page 89.

Figure 6.3 The mycas.matches Data Table

Obs	_DOCUMENT_	_TARGET_	_RULE_ID_
1	d01	1	1
2	d06	1	2
3	d09	.	0
4	d11	.	0
5	d16	.	0
6	d17	.	0
7	d04	1	2
8	d04	1	1
9	d07	1	1
10	d14	.	0
11	d15	.	0
12	d19	.	0
13	d02	1	1
14	d03	1	1
15	d05	1	1
16	d08	1	2
17	d10	.	0
18	d12	.	0
19	d13	.	0
20	d18	.	0
21	d20	.	0

Syntax: BOOLRULE Procedure

The following statements are available in the BOOLRULE procedure:

PROC BOOLRULE < *options* > **;**
DOCINFO < *options* > **;**
TERMINFO < *options* > **;**
OUTPUT < *options* > **;**
SCORE < *options* > **;**

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

PROC BOOLRULE Statement

PROC BOOLRULE < *options* > **;**

The PROC BOOLRULE statement invokes the procedure. Table 6.1 summarizes the *options* in the statement by function. The *options* are then described fully in alphabetical order.

Table 6.1 PROC BOOLRULE Statement Options

option	**Description**
Basic Options	
DATA=	Specifies the input data table (which must be in transactional format) for rule extraction
DOCID=	Specifies the variable in the DATA= data table that contains the document ID
DOCINFO=	Specifies the input data table that contains information about documents
GNEG=	Specifies the minimum g-score needed for a negative term to be considered for rule extraction
GPOS=	Specifies the minimum g-score needed for a positive term or a rule to be considered for rule extraction
MAXCANDIDATES=	Specifies the number of term candidates to be selected for each category
MAXTRIESIN=	Specifies the k_{in} value for k-best search in the term ensemble process for creating a rule
MAXTRIESOUT=	Specifies the k_{out} value for k-best search in the rule ensemble process for creating a rule set
MINSUPPORTS=	Specifies the minimum number of documents in which a term needs to appear in order for the term to be used for creating a rule
MNEG=	Specifies the m value for computing estimated precision for negative terms
MPOS=	Specifies the m value for computing estimated precision for positive terms
TERMID=	Specifies the variable in the DATA= data table that contains the term ID
TERMINFO=	Specifies the input data table that contains information about terms

You must specify the following *option*:

DATA=*CAS-libref.data-table*

DOC=*CAS-libref.data-table*

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

CAS-libref refers to a collection of information that is defined in the LIBNAME statement and includes the caslib, which includes a path to the data, and a session identifier, which defaults to the active session but which can be explicitly defined in the LIBNAME statement. For more information about *CAS-libref*, see the section "Using CAS Sessions and CAS Engine Librefs" on page 71.

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

Each row of the input data table must contain one variable for the document ID and one variable for the term ID. Both the document ID variable and the term ID variable can be either a numeric or character variable. The BOOLRULE procedure does not assume that the data table is sorted by either document ID or term ID.

You can also specify the following *options*:

DOCID=*variable*
specifies the *variable* that contains the ID of each document. The document ID can be either a number or a string of characters.

DOCINFO=*CAS-libref.data-table*
names the input data table that contains information about documents. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 71.

Each row of the input data table must contain one variable for the document ID. The BOOLRULE procedure uses the document ID in the DATA= data table to search for the document ID variable in this data table to obtain information about documents (for example, the categories of each document).

GNEG=*g-value*
specifies the minimum g-score needed for a negative term to be considered for rule extraction in the term ensemble. If you do not specify this option, the value that is specified for the GPOS= option (or its default value) is used. For more information about g-score, see the section "g-Score" on page 85.

GPOS=*g-value*
specifies the minimum g-score needed for a positive term to be considered for rule extraction in the term ensemble. A rule also needs to have a g-score that is higher than *g-value* to be considered in the rule ensemble. The *g-value* is also used in the improvability test. A rule is improvable if the g-score that is computed according to the improvability test is larger than *g-value*. By default, GPOS=8.

MAXCANDIDATES=*n*
MAXCANDS=*n*
specifies the number of term candidates to be selected for each category. Rules are built by using only these term candidates. By default, MAXCANDS=500.

MAXTRIESIN=*n*
specifies the k_{in} value for the k-best search in the term ensemble process for creating rules. For more information, see the section "k-Best Search" on page 87. By default, MAXTRIESIN=150.

MAXTRIESOUT=*n*
specifies the k_{out} value for the k-best search in the rule ensemble process for creating a rule set. For more information, see the section "k-Best Search" on page 87. By default, MAXTRIESOUT=50.

MINSUPPORTS=*n*
specifies the minimum number of documents in which a term needs to appear in order for the term to be used for creating a rule. By default, MINSUPPORTS=3.

MNEG=*m*
specifies the *m* value for computing estimated precision for negative terms. If you do not specify this option, the value specified for the MPOS= option (or its default value) is used.

MPOS=*m*
specifies the *m* value for computing estimated precision for positive terms. By default, MPOS=8.

TERMID=*variable*
specifies the *variable* that contains the ID of each term. The *variable* can be either a number or a string of characters. If the TERMINFO= option is not specified, *variable* is also used as the label of terms.

TERMINFO=*CAS-libref.data-table*
names the input data table that contains information about terms. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 71.

Each row of the input data table must contain one variable for the term ID. If you specify this option, you must use the TERMINFO statement to specify which variables in the data table contain the term ID and the term label, respectively. The BOOLRULE procedure uses the term ID in the DATA= data table to search for the term ID variable in this data table to obtain information about the terms. If you do not specify this option, the content of the TERMID= variable is also used as the label of terms.

DOCINFO Statement

DOCINFO < *options* > **;**

The DOCINFO statement specifies information about the data table that is specified in the DOCINFO= option in the PROC BOOLRULE statement.

You can specify the following *options*:

EVENTS=(*value1***,** *value2***,** . . .**)**
specifies the values of target variables that are considered as positive events or categories of interest as follows:

- When TARGETTYPE=BINARY, the *values* of each target variable that is specified in the TARGET= option correspond to positive events. All other values correspond to negative events.
- When TARGETTYPE=BINARY, for any variable specified in the TARGET= option that is a numeric variable, "1" is considered to be a positive event by default.
- When TARGETTYPE=BINARY, for any variable specified in the TARGET= option that is a character variable, "Y" is considered to be a positive event by default.
- You cannot specify this option when TARGETTYPE=MULTICLASS.

ID=*variable*
specifies the *variable* that contains the document ID. To fetch the target information about documents, the values in the *variable* are matched to the document ID variable that is specified in the DOCID= option in the PROC BOOLRULE statement. The *variable* can be either a numeric variable or a character variable. Its type must match the type of the variable that is specified in the DOCID= option in the PROC BOOLRULE statement.

TARGET=(*variable***, *variable*, . . .)**
specifies the target *variables*. A target variable can be either a numeric variable or a character variable.

- When TARGETTYPE=BINARY, you can specify multiple target *variables*, and each target *variable* corresponds to a category.
- When TARGETTYPE=MULTICLASS, you can specify only one target *variable*, and each of its levels corresponds to a category.

TARGETTYPE=BINARY | MULTICLASS
specifies the type of the target variables. You can specify the following values:

BINARY indicates that multiple target variables can be specified and each target variable corresponds to a category.

MULTICLASS indicates that only one target variable can be specified and each level of the target variable corresponds to a category.

By default, TARGETTYPE=BINARY.

OUTPUT Statement

OUTPUT < *options* > **;**

The OUTPUT statement specifies the data tables that contain the results that the BOOLRULE procedure generates.

You can specify the following *options*:

CANDIDATETERMS=*CAS-libref.data-table*
specifies a data table to contain the terms that have been selected by the BOOLRULE procedure for rule creation. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 71.

If MAXCANDIDATES=*p* in the BOOLRULE statement, the procedure selects at most *p* terms for each category to be considered for rule extraction. For more information about this data table, see the section "Output Data Sets" on page 87.

RULES=*CAS-libref.data-table*
specifies a data table to contain the rules that have been generated by the BOOLRULE procedure for each category. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 71.

For more information about this data table, see the section "Output Data Sets" on page 87.

RULETERMS=*CAS-libref.data-table*
specifies a data table to contain the terms in each rule that is generated by the BOOLRULE procedure. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 71.

For more information about this data table, see the section "Output Data Sets" on page 87.

SCORE Statement

SCORE < *options* > **;**

The SCORE statement specifies the input data table that contains the terms in rules and the output data table to contain the scoring results.

You can specify the following *options*:

OUTMATCH=*CAS-libref.data-table*
specifies a data table to contain the rule-matching results (that is, whether a document satisfies a rule). *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 71.

For more information about this data table, see the section "Scoring Data Set" on page 89.

RULETERMS=*CAS-libref.data-table*
specifies a data table that contains the terms in each rule that the BOOLRULE procedure generates. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 71.

For more information about this data table, see the section "RULETERMS= Data Table" on page 89.

TERMINFO Statement

TERMINFO < *options* > **;**

The TERMINFO statement specifies information about the data table that is specified in the TERMINFO= option in the PROC BOOLRULE statement. If you specify the TERMINFO= data table in the PROC BOOLRULE statement, you must also include this statement to specify which variables in the data table contain the term ID and the term label, respectively.

You can specify the following *options*:

ID=*variable*
specifies the variable that contains the term ID. To fetch the text of terms, the values in *variable* are matched to the term ID variable that is specified in the TERMID= option in the PROC BOOLRULE statement. The *variable* can be either a numeric variable or a character variable. Its type must match the type of the variable that is specified in the TERMID= option in the PROC BOOLRULE statement.

LABEL=*variable*
specifies the variable that contains the text of the terms, where *variable* must be a character variable.

Details: BOOLRULE Procedure

PROC BOOLRULE implements the BOOLLEAR technique for rule extraction. This section provides details about various aspects of the BOOLRULE procedure.

BOOLLEAR for Boolean Rule Extraction

Rule-based text categorization algorithms uses text rules to classify documents. Text rules are interpretable and can be effectively learned even when the number of positive documents is very limited. BOOLLEAR (Cox and Zhao 2014) is a novel technique for Boolean rule extraction. When you supply a text corpus that contains multiple categories, BOOLLEAR extracts a set of binary rules from each category and represents each rule in the form of a conjunction, where each item in the conjunction denotes the presence or absence of a particular term. The BOOLLEAR process is as follows (criteria and measurements that are used in this process are described in the next section):

1. Use an information gain criterion to form an ordered term candidate list. The term that best predicts the category is first on the list, and so on. Terms that do not have a significant relationship to the category are removed from this list. Set the current term to the first term.
2. Determine the "estimated precision" of the current term. The estimated precision is the projected percentage of the term's occurrence with the category in out-of-sample data, using additive smoothing. Create a rule that consists of that term.
3. If the "estimated precision" of the current rule could not possibly be improved by adding more terms as qualifiers, then go to step 6.
4. Starting with the next term on the list, determine whether the conjunction of the current rule with that term (via either term presence or term absence) significantly improves the information gain and also improves estimated precision.
5. If there is at least one combination that meets the criterion in step 4, choose the combination that yields the best estimated precision, and go to step 3 with that combination. Otherwise, continue to step 6.
6. If the best rule obtained in step 3 has a higher estimated precision than the current "highest precision" rule, replace the current rule with the new rule.
7. Increment the current term to the next term in the ordered candidate term list and go to step 2. Continue repeating until all terms in the list have been considered.

8. Determine whether the harmonic mean of precision and recall (the F1 score) of the current rule set is improved by adding the best rule obtained by steps 1 to 7. If it is not, then exit.

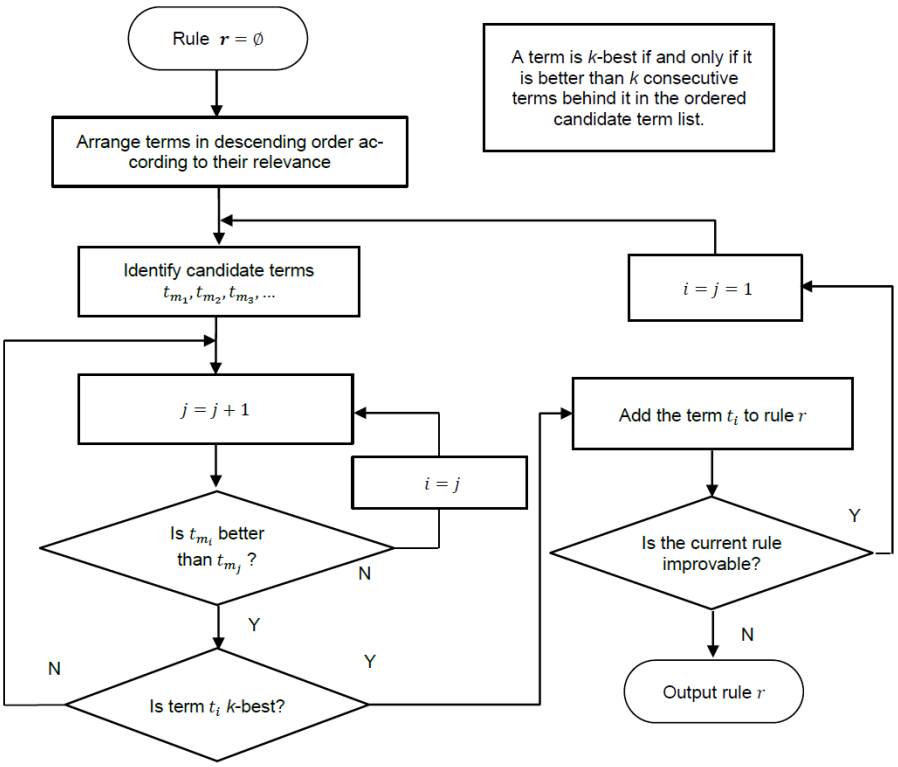
9. If so, remove from the document set all documents that match the new rule, add this rule to the rule set, and go to step 1 to start creating the next rule in the rule set.

BOOLLEAR contains two essential processes for rule extraction: a term ensemble process (steps 4–5), which creates rules by adding terms; and a rule ensemble process (steps 2–9), which creates a rule set. The rule set can then be used for either content exploration or text categorization. Both the term ensemble process and the rule ensemble process are iterative processes. The term ensemble process forms an inner loop of the rule ensemble process. Efficient heuristic search strategies and sophisticated evaluation criteria are designed to ensure state-of-the-art performance of BOOLLEAR.

Term Ensemble Process

The term ensemble process iteratively adds terms to a rule. When the process finishes, it returns a rule that can be used as a candidate rule for the rule ensemble process. Figure 6.4 shows the flowchart of the term ensemble process.

Figure 6.4 Term Ensemble Process for Creating a Rule



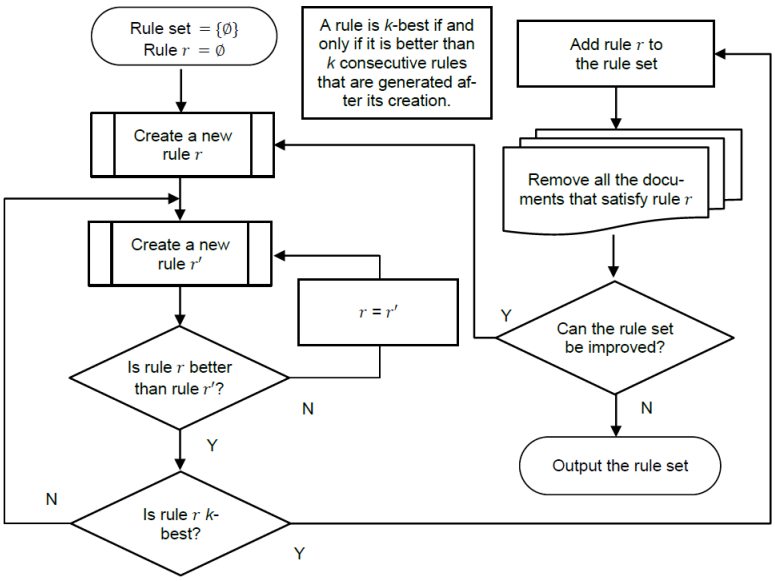
Before adding terms to a rule, BOOLLEAR first sorts the candidate terms in descending order according to their g-score with respect to the target category. It then starts to add terms to the rule iteratively. In each

iteration of the term ensemble process, BOOLLEAR takes a term t from the ordered candidate term list and determines whether adding the term to the current rule $\mathbf{r}$ can improve the rule's estimated precision. To ensure that the term is good enough, BOOLLEAR tries $k_{\text{in}} - 1$ additional terms in the term list, where k_{in} is the maximum number of terms to examine for improvement. If none of these terms is better (results in a lower g-score of the current rule $\mathbf{r}$) than term t, the term is considered to be k-best, where $k = k_{\text{in}}$, and BOOLLEAR updates the current rule $\mathbf{r}$ by adding term t to it. If one of the $k_{\text{in}} - 1$ additional terms is better than term t, BOOLLEAR sets that term as t and tries $k_{\text{in}} - 1$ additional terms to determine whether this new t is better than all of those additional terms. BOOLLEAR repeats until the current term t is k-best or until it reaches the end of the term list. After a term is added to the rule, BOOLLEAR marks the term as used and continues to identify the next k-best term from the unused terms in the sorted candidate term list. When a k-best term is identified, BOOLLEAR adds it to the rule. BOOLLEAR keeps adding k-best terms until the rule cannot be further improved. By trying to identify a k-best term instead of the global best, BOOLLEAR shrinks its search space to improve its efficiency.

Rule Ensemble Process

The rule ensemble process iteratively creates and adds new rules to a rule set. When the process finishes, it returns the rule set, which can then be used for text categorization. Figure 6.5 shows the flowchart of the rule ensemble process.

Figure 6.5 Rule Ensemble for Creating a Rule Set



In each iteration of the rule ensemble process, BOOLLEAR tries to find a rule $\mathbf{r}$ that has the highest precision in classifying the previously unclassified positive samples. For the first iteration, all samples are unclassified. To ensure that the precision of rule $\mathbf{r}$ is good enough, BOOLLEAR generates $k_{\text{out}} - 1$ additional rules, where

k_{out} is an input parameter that you specify in the MAXTRIESOUT= option in the PROC BOOLRULE statement. If one of these rules has a higher precision than rule **r**, BOOLLEAR sets that rule as the new rule **r** and generates another $k_{\text{out}} - 1$ rules to determine whether this new rule is the best among them. BOOLLEAR repeats this process until the current rule **r** is better than any of the $k_{\text{out}} - 1$ rules that are generated after it. The obtained rule **r** is called a *k*-best rule, where $k = k_{\text{out}}$. When BOOLLEAR obtains a *k*-best rule, it adds that rule to the rule set and removes from the corpus all documents that satisfy the rule. In order to reduce the possibility of generating redundant rules, BOOLLEAR then determines whether the F1 score of the rule set is improved. If the F1 score is improved, BOOLLEAR goes to the next iteration and uses the updated corpus to generate another rule. Otherwise, it treats the current rule set as unimprovable, stops the search, and outputs the currently obtained rule set. Note that to identify a "good" rule, BOOLLEAR does not go through all the potential rules to find the global "best," because doing so can be computationally intractable when the number of candidate terms is large. Also, before BOOLLEAR generates a rule, it orders the terms in the candidate term set by their correlation to the target. So it is reasonable to expect that the obtained *k*-best rule is close to a globally best rule in terms of its capability for improving the F1 score of the rule set. For information about the F1 score, see the section "Precision, Recall, and the F1 Score" on page 84.

Measurements Used in BOOLLEAR

This section provides detailed information about the measurements that are used in BOOLLEAR to evaluate terms and rules.

Precision, Recall, and the F1 Score

Precision measures the probability that the observation is actually positive when a classifier predicts it to be positive; recall measures the probability that a positive observation will be recognized; and the F1 score is the harmonic mean of precision and recall. A good classifier should be able to achieve both high precision and high recall. The precision, recall, and F1 score are defined as

$$\text{precision} = \frac{\text{TP}}{\text{TP} + \text{FP}}$$

$$\text{recall} = \frac{\text{TP}}{\text{TP} + \text{FN}}$$

$$\text{F1} = 2 \times \frac{\text{precision} \times \text{recall}}{\text{precision} + \text{recall}}$$

where TP is the true-positive (the number of documents that are predicted to be positive and are actually positive), FP is the false-positive (the number of documents that are predicted to be positive but are actually negative), TN is the true-negative (the number of documents that are predicted to be negative and are actually negative), and FN is the false-negative (the number of documents that are predicted to be negative but are actually positive). A classifier thus obtains a high F1 score if and only if it can achieve both high precision and high recall. The F1 score is a better measurement than accuracy when the data are imbalanced,[2] because a classifier can obtain very high accuracy by predicting that all samples belong to the majority category.

[2] Accuracy is defined as $\frac{\text{TP}+\text{TN}}{\text{TP}+\text{FP}+\text{TN}+\text{FN}}$.

g-Score

BOOLLEAR uses the *g*-test (which is also known as the likelihood-ratio or maximum likelihood statistical significance test) as an information gain criterion to evaluate the correlation between terms and the target. The *g*-test generates a *g*-score, which has two beneficial properties: as a form of mutual information, it is approximately equivalent to information gain in the binary case; and because it is distributed as a chi-square, it can also be used for statistical significance testing. The *g*-test is designed to compare the independence of two categorical variables. Its null hypothesis is that the proportions at one variable are the same for different values of the second variable. Given the TP, FP, FN, and TN of a term, the term's *g*-score can be computed as

$$g = 2 \times \sum_{i=\{\mathrm{TP,TN,FP,FN}\}} \mathrm{O}(i) \log\left(\frac{\mathrm{O}(i)}{\mathrm{E}(i)}\right)$$

$$\begin{aligned}
\mathrm{O(TP)} &= \mathrm{TP} \\
\mathrm{O(FP)} &= \mathrm{FP} \\
\mathrm{O(TN)} &= \mathrm{TN} \\
\mathrm{O(FN)} &= \mathrm{FN} \\
\mathrm{E(TP)} &= \frac{(\mathrm{TP}+\mathrm{FP}) \times \mathrm{P}}{\mathrm{P}+\mathrm{N}} \\
\mathrm{E(FP)} &= \frac{(\mathrm{TP}+\mathrm{FP}) \times \mathrm{N}}{\mathrm{P}+\mathrm{N}} \\
\mathrm{E(TN)} &= \frac{(\mathrm{TN}+\mathrm{FN}) \times \mathrm{N}}{\mathrm{P}+\mathrm{N}} \\
\mathrm{E(FN)} &= \frac{(\mathrm{TN}+\mathrm{FN}) \times \mathrm{P}}{\mathrm{P}+\mathrm{N}}
\end{aligned}$$

where P is the number of positive documents; N is the number of negative documents; O(TP), O(FP), O(TN), and O(FN) refer to the observed TP, FP, TN, and FN of a term; and E(TP), E(FP), E(TN), and E(FN) refer to the expected TP, FP, TN, and FN of a term. A term has a high *g*-score if it appears often in positive documents but rarely in negative documents, or vice versa.

Estimated Precision

Estimated precision helps BOOLLEAR shorten its search path and avoid generating overly specific rules. The precision is estimated by a form of additive smoothing with additional correction (err_i) to favor shorter rules over longer rules:

$$\mathrm{precision}_i^m(t) = \frac{\mathrm{TP}_{i,\mathbf{t}} + \frac{\mathrm{P}}{\mathrm{N+P}} \times m}{\mathrm{TP}_{i,\mathbf{t}} + \mathrm{FP}_{i,\mathbf{t}} + m} - \mathrm{err}_{i-1}$$

$$\mathrm{err}_i = \frac{\mathrm{TP}_{i,\mathbf{t}}}{\mathrm{TP}_{i,\mathbf{t}} + \mathrm{FP}_{i,\mathbf{t}}} - \frac{\mathrm{TP}_{i,\mathbf{t}} + \frac{\mathrm{P}}{\mathrm{N+P}} \times m}{\mathrm{TP}_{i,\mathbf{t}} + \mathrm{FP}_{i,\mathbf{t}} + m} + \mathrm{err}_{i-1}$$

In the preceding equations, $m(\leq 1)$ is a parameter that you specify for bias correction. A large m is called for when a very large number of rules are evaluated, in order to minimize selection bias. $\mathrm{TP}_{i,\mathbf{t}}$ and $\mathrm{FP}_{i,\mathbf{t}}$ are the true-positive and false-positive of rule $\mathbf{t}$ when the length of the rule is i.

Improvability Test

BOOLLEAR tests for improvability in the term ensemble step for "in-process" model pruning. To determine whether a rule is improvable, BOOLLEAR applies the g-test to a perfect confusion table that is defined as

TP	0
0	FP

In this table, TP is the true-positive of the rule and FP is the false-positive of the rule. The g-score that is computed by using this table reflects the maximum g-score that a rule could possibly obtain if a perfectly discriminating term were added to the rule. If the g-score is smaller than a number that you specify to indicate a maximum p-value for significance in the GPOS= and GNEG= options, BOOLLEAR considers the rule to be unimprovable.

Shrinking the Search Space

Exhaustively searching the space of possible rules is impractical because of the exponential number of rules that would have to be searched (2^m rules, where m is the number of candidate terms). In addition, an exhaustive search usually leads to overfitting by generating many overly specific rules. Therefore, BOOLLEAR implements the strategies described in the following sections to dramatically shrink the search space to improve its efficiency and help it avoid overfitting.

Feature Selection

BOOLLEAR uses the g-test to evaluate terms. Assume that MAXCANDIDATES=p and MINSUPPORTS=c in the PROC BOOLRULE statement. A term is added to the ordered candidate term list if and only if the following two conditions hold:

1. The term is a top p term according to its g-score.
2. The term appears in more than c documents.

The size of the candidate term list controls the size of the search space. The smaller the size, the fewer terms are used for rule extraction, and therefore the smaller the search space is.

Significance Testing

In many rule extraction algorithms, rules are built until they perform perfectly on a training set, and pruning is applied afterwards. In contrast, BOOLLEAR prunes "in-process." The following three checks are a form of in-process pruning; rules are not expanded when their expansion does not meet these basic requirements. These requirements help BOOLLEAR truncate its search path and avoid generating overly specific rules.

- **Minimum positive document coverage**: BOOLLEAR requires that a rule be satisfied by at least s positive documents, where s is the value of the MINSUPPORTS= option in the PROC BOOLRULE statement.
- **Early stop based on g-test**: BOOLLEAR stops searching when the g-score that is calculated for improving (or starting) a rule does not meet required statistical significance levels.

- **Early stop based on estimated precision**: BOOLLEAR stops building a rule when the estimated precision of the rule does not improve when the current best term is added to the rule. This strategy helps BOOLLEAR shorten its search path.

k-Best Search

In the worst case, BOOLLEAR could still examine an exponential number of rules, although the heuristics described here minimize that chance. But because the terms are ordered by predictiveness of the category beforehand, a *k*-best search is used to further improve the efficiency of BOOLLEAR: If BOOLLEAR tries unsuccessfully to expand (or start) a rule numerous times with the a priori "best" candidates, then the search can be prematurely ended. Two optional parameters, k_{in} and k_{out}, determine the maximum number of terms and rules to examine for improvement. The k_{in} parameter (which is specified in the MAXTRIESIN= option) is used in the term ensemble process: if k_{in} consecutive terms have been checked for building possible rules and none of them are superior to the best current rule, the search is terminated. The k_{out} parameter (which is specified in the MAXTRIESOUT= option) is used in the rule ensemble process: if k_{out} consecutive terms have been checked to add to a rule and they do not generate a better rule, then the search for expanding that rule is terminated. This helps BOOLLEAR shorten its search path, even with a very large number of candidate terms, with very little sacrifice in accuracy.

Improvability Test

BOOLLEAR tests whether adding a theoretical perfectly discriminating term to a particular rule could possibly have both a statistically significant result and a higher estimated precision than the current rule. If it cannot, then the current rule is recognized without additional testing as the best possible rule, and no further expansion is needed.

Early Stop Based on the F1 Score

BOOLLEAR stops building the rule set if adding the current best rule does not improve the rule set's F1 score. Thus the F1 score is treated as the objective to maximize.

Output Data Sets

This section describes the output data sets that PROC BOOLRULE produces when you specify the corresponding option in the OUTPUT statement.

CANDIDATETERMS= Data Table

The CANDIDATETERMS= option in the OUTPUT statement specifies a data table to contain the terms that have been selected by the procedure for rule creation. If MAXCANDIDATES=*p* in the PROC BOOLRULE statement, the procedure selects a maximum of *p* terms for each category.

Table 6.2 shows the fields in this data table.

Table 6.2 Fields in the CANDIDATETERMS= Data Table

Field	Description
Target	The category that the term is selected for (this field corresponds to the Target field in the RULES= data table)
Rank	The rank of the term in the ordered term list for the category (term rank starts from 1)
Term	A lowercase version of the term
Key	The term identifier of the term
GScore	The *g*-score of the term that is obtained for the target category
Support	The number of documents in which the term appears
TP	The number of positive documents in which the term appears
FP	The number of negative documents in which the term appears

RULES= Data Table

The RULES= option in the OUTPUT statement specifies the output data table to contain the rules that have been generated for each category.

Table 6.3 shows the fields in this data table.

Table 6.3 Fields in the RULES= Data Table

Field	Description
Target	The target category that the term is selected to model
Target_var	The variable that contains the target
Target_val	The value of the target variable
Ruleid	The ID of a rule (Ruleid starts from 1)
Ruleid_loc	The ID of a rule in a rule set (in each rule set, Ruleid_loc starts from 1)
Rule	The text content of the rule
TP	The number of positive documents that are satisfied by the rule set when the rule is added to the rule set
FP	The number of negative documents that are satisfied by the rule set when the rule is added to the rule set
Support	The number of documents that are satisfied by the rule set when the rule is added to the rule set
rTP	The number of positive documents that are satisfied by the rule when the rule is added to the rule set
rFP	The number of negative documents that are satisfied by the rule when the rule is added to the rule set
rSupport	The number of documents that are satisfied by the rule when the rule is added to the rule set
F1	The F1 score of the rule set when the rule is added to the rule set
Precision	The precision of the rule set when the rule is added to the rule set
Recall	The recall of the rule set when the rule is added to the rule set

This data table contains the discovered rule sets for predicting the target levels of the target variable. In each rule set, the order of the rules is important and helps you interpret the results. The first rule is trained using all the data; the second rule is trained on the data that did not satisfy the first rule; and subsequent rules are built only after the removal of observations that satisfy previous rules. The fit statistics (TP, FP, Support, F1, Precision, and Recall) of each rule are cumulative and represent totals that include using that particular rule along with all the previous rules in the rule set.

When you specify TARGETTYPE=MULTICLASS in the DOCINFO statement, each target level of the target variable defines a category and the target field contains the same content as the Target_val field. When TARGETTYPE=BINARY in the DOCINFO statement, each target variable defines a category and the target field contains the same content as the Target_var field.

RULETERMS= Data Table

The RULETERMS= option in the OUTPUT statement specifies a data table to contain the terms in the rules. The information in this data table is used in the scoring phase for scoring documents.

Table 6.4 Fields in the RULETERMS= Data Table

Field	Description
Target	The target category that the term is selected to model
Target_var	The variable that contains the target
Target_val	The value of the target variable
Ruleid	The ID of a rule (Ruleid starts from 1)
Ruleid_loc	The ID of a rule in a rule set (in each rule set, Ruleid_loc starts from 1)
Rule	The text content of the rule
termnum	The ID of a term that is used in the rule
Direction	Specifies whether the term is positive or negative (if Direction=1, the term is positive; if Direction=–1, the term is negative)
Weight	The weight of a term

Term weights are used for scoring documents. The weight of a negative term is always –1. If a positive term is in rule $\mathbf{r}$ and there are k positive terms in the rule, the weight of this positive term is $1/k + 0.000001$. If a document contains all the positive terms in the rule but none of the negative terms, the score of the document is $k \times (1/k + 0.000001) > 1$, indicating that the document satisfies the rule. Otherwise, the document's score is less than 1, indicating that the document does not satisfy the rule.

Scoring Data Set

This section describes the output data set that PROC BOOLRULE produces when you specify the corresponding option in the SCORE statement.

OUTMATCH= Data Table

The OUTMATCH= option in the SCORE statement specifies the output data table to contain the rule-matching results (that is, whether a document satisfies a rule). A document satisfies a rule (in other words, a rule is

matched in the document) if and only if all the positive terms in the rule are present in the document and none of the negative terms are present in the document. PROC BOOLRULE also outputs a special rule for which ID=0. If a document satisfies the rule for which ID=0, then the document does not satisfy any rule in the RULETERMS= table. For this special rule, the target has a missing value.

Table 6.5 shows the fields in this data table.

Table 6.5 Fields in the OUTMATCH= Data Table

Field	Description
Document	ID of the document that satisfies the rule
Target	ID of the target that the rule is generated for
_Rule_ID_	ID of the rule that the document satisfies

Examples: BOOLRULE Procedure

Example 6.1: Rule Extraction for Binary Targets

This example generates rules for a data table that contains various types of customer reviews. The following DATA step creates the mycas.reviews data table, which contains nine observations that have four variables. The text variable contains the input reviews. The positive variable contains the sentiment of the reviews: a value of 1 indicates that the review is positive, and a value of 0 indicates that the review is negative. The category variable contains the category of the reviews. The did variable contains the ID of the documents. Each row in the data table represents a document for analysis.

```
data mycas.reviews;
   infile datalines delimiter='|' missover;
   length text $300 category $20;
   input text$ positive category$ did;
   datalines;
      This is the greatest phone ever! love it!|1|electronics|1
      The phone's battery life is too short and screen resolution is low.|0|electronics|2
      The screen resolution is low, but I love this tv.|1|electronics|3
      The movie itself is great and I like it, although the resolution is low.|1|movies|4
      The movie's story is boring and the acting is poor.|0|movies|5
      I watched this movie on tv, it's not good on a small screen. |0|movies|6
      watched the movie first and loved it, the book is even better!|1|books |7
      I like the story in this book, they should put it on screen.|1|books|8
      I love the author, but this book is a waste of time, don't buy it.|0|books|9
   ;
run;
```

The following TEXTMINE procedure call parses the mycas.reviews data table, stores the term-by-document matrix in the mycas.reviews_bow data table in transactional format, and stores terms that appeared in the mycas.reviews data table in the mycas.reviews_terms data table:

```
proc textmine data=mycas.reviews;
    doc_id
        did;
    var
        text;
    parse
        nonoungroups
        notagging
        entities    = none
        outparent   = mycas.reviews_bow
        outterms    = mycas.reviews_terms
        reducef     = 1;
run;
```

The following statements run PROC BOOLRULE to extract rules from the mycas.reviews_bow data table and run PROC PRINT to show the results. By default, TARGETTYPE=BINARY. One target variable, positive, is specified; this variable indicates whether the reviews are positive or negative.

```
proc boolrule
        data        =   mycas.reviews_bow
        docid       =   _document_
        termid      =   _termnum_
        docinfo     =   mycas.reviews
        terminfo    =   mycas.reviews_terms
        minsupports =   1
        mpos        =   1
        gpos        =   1;
    docinfo
        id          =   did
        targets     =   (positive);
    terminfo
        id          =   key
        label       =   term;
    output
        ruleterms   =   mycas.ruleterms
        rules       =   mycas.rules;
run;
data rules;
set mycas.rules;
proc print data=rules;
    var target ruleid rule F1 precision recall;
run;
```

Output 6.1.1 shows that the mycas.rules data table contains rules that are generated for the "positive" categories.

Output 6.1.1 The mycas.rules Data Table

Obs	TARGET	RULEID	RULE	F1	PRECISION	RECALL
1	positive	1	like	0.57143	1.00000	0.4
2	positive	2	better	0.75000	1.00000	0.6
3	positive	3	great	0.88889	1.00000	0.8
4	positive	4	love	0.90909	0.83333	1.0

Example 6.2: Rule Extraction for a Multiclass Target

This example uses the same input table and the same TEXTMINE procedure call that are used in Example 6.1 to illustrate how you can extract rules for a multiclass target. The DATA step and procedure call are repeated here for convenience.

The following DATA step creates the mycas.reviews data table, which contains nine observations that have four variables. The text variable contains the input reviews. The positive variable contains the sentiment of the reviews: a value of 1 indicates that the review is positive, and a value of 0 indicates that the review is negative. The category variable contains the category of the reviews. The did variable contains the ID of the documents. Each row in the data table represents a document for analysis.

```
data mycas.reviews;
   infile datalines delimiter='|' missover;
   length text $300 category $20;
   input text$ positive category$ did;
   datalines;
      This is the greatest phone ever! love it!|1|electronics|1
      The phone's battery life is too short and screen resolution is low.|0|electronics|2
      The screen resolution is low, but I love this tv.|1|electronics|3
      The movie itself is great and I like it, although the resolution is low.|1|movies|4
      The movie's story is boring and the acting is poor.|0|movies|5
      I watched this movie on tv, it's not good on a small screen. |0|movies|6
      watched the movie first and loved it, the book is even better!|1|books |7
      I like the story in this book, they should put it on screen.|1|books|8
      I love the author, but this book is a waste of time, don't buy it.|0|books|9
   ;
run;
```

The following TEXTMINE procedure call parses the mycas.reviews data table, stores the term-by-document matrix in the mycas.reviews_bow data table in transactional format, and stores terms that appeared in the mycas.reviews data table in the mycas.reviews_terms data table:

```
proc textmine data=mycas.reviews;
    doc_id
        did;
    var
        text;
    parse
        nonoungroups
        notagging
        entities    = none
        outparent   = mycas.reviews_bow
        outterms    = mycas.reviews_terms
        reducef     = 1;
run;
```

The following statements run PROC BOOLRULE to extract rules from the mycas.reviews_bow data table and run PROC PRINT to show the results. TARGETTYPE=MULTICLASS is specified, and category is specified as the target variable, which contains three levels: "electronics," "movies," and "books." Each level defines a category for which the BOOLRULE procedure extracts rules.

```
proc boolrule
        data         =   mycas.reviews_bow
        docid        =   _document_
        termid       =   _termnum_
        docinfo      =   mycas.reviews
        terminfo     =   mycas.reviews_terms
        minsupports  =   1
        mpos         =   1
        gpos         =   1;
    docinfo
        id           =   did
        targettype   =   multiclass
        targets      =   (category);
    terminfo
        id           =   key
        label        =   term;
    output
        ruleterms    =   mycas.ruleterms
        rules        =   mycas.rules;
run;

data rules;
set mycas.rules;
proc print data=rules;
    var target ruleid rule F1 precision recall;
run;
```

Output 6.2.1 shows that the mycas.rules data table contains rules that are generated for the "electronics," "movies," and "books" categories.

Output 6.2.1 The mycas.rules Data Table

Obs	TARGET	RULEID	RULE	F1	PRECISION	RECALL
1	electronics	1	phone	0.80000	1.00	0.66667
2	electronics	2	resolution	0.85714	0.75	1.00000
3	movies	3	movie	0.85714	0.75	1.00000
4	books	4	book	1.00000	1.00	1.00000

Example 6.3: Using Events in Rule Extraction

This example uses the same input table and the same TEXTMINE procedure call that are used in Example 6.1 to illustrate how you can use events in rule extraction. The DATA step and procedure call are repeated here for convenience.

When TARGETTYPE=MULTICLASS, each level of the target variable defines a category for rule extraction. If you want to extract rules for only a subset of the levels of the target variable, you can use the EVENTS= option to specify the categories for which you want to extract rules.

The following DATA step creates the mycas.reviews data table, which contains nine observations that have four variables. The text variable contains the input reviews. The positive variable contains the sentiment of the reviews: a value of 1 indicates that the review is positive and a value of 0 indicates that the review is negative. The category variable contains the category of the reviews. The did variable contains the ID of the documents. Each row in the data table represents a document for analysis.

```
data mycas.reviews;
   infile datalines delimiter='|' missover;
   length text $300 category $20;
   input text$ positive category$ did;
   datalines;
      This is the greatest phone ever! love it!|1|electronics|1
      The phone's battery life is too short and screen resolution is low.|0|electronics|2
      The screen resolution is low, but I love this tv.|1|electronics|3
      The movie itself is great and I like it, although the resolution is low.|1|movies|4
      The movie's story is boring and the acting is poor.|0|movies|5
      I watched this movie on tv, it's not good on a small screen. |0|movies|6
      watched the movie first and loved it, the book is even better!|1|books |7
      I like the story in this book, they should put it on screen.|1|books|8
      I love the author, but this book is a waste of time, don't buy it.|0|books|9
   ;
run;
```

The following TEXTMINE procedure call parses the mycas.reviews data table, stores the term-by-document matrix in the mycas.reviews_bow data table in transactional format, and stores terms that appeared in the mycas.reviews data table in the mycas.reviews_terms data table:

```
proc textmine data=mycas.reviews;
    doc_id
        did;
    var
        text;
    parse
        nonoungroups
        notagging
        entities    = none
        outparent   = mycas.reviews_bow
        outterms    = mycas.reviews_terms
        reducef     = 1;
run;
```

The following statements run PROC BOOLRULE to extract rules from the mycas.reviews_bow data table and run PROC PRINT to show the results. TARGETTYPE=BINARY is specified, and category is specified as the target variable, which contains three levels: "electronics," "movies," and "books." Because the "movies" and "books" levels are specified in the EVENTS= option, PROC BOOLRULE procedure extracts rules for "movies" and "books," but not "electronics."

```
proc boolrule
        data        =   mycas.reviews_bow
        docid       =   _document_
        termid      =   _termnum_
        docinfo     =   mycas.reviews
        terminfo    =   mycas.reviews_terms
        minsupports =   1
        mpos        =   1
        gpos        =   1;
    docinfo
        id          =   did
        targettype  =   binary
        targets     =   (category)
        events      =   ("movies" "books");
    terminfo
        id          =   key
        label       =   term;
    output
        ruleterms   =   mycas.ruleterms
        rules       =   mycas.rules;
run;

data rules;
set mycas.rules;
proc print data=rules;
    var target ruleid rule F1 precision recall;
run;
```

Output 6.3.1 shows that the mycas.rules data table contains rules that are generated for the "movies" and "books" categories.

Output 6.3.1 The mycas.rules Data Table

Obs	TARGET	RULEID	RULE	F1	PRECISION	RECALL
1	category	1	movie	0.8	1	0.66667
2	category	2	book	1.0	1	1.00000

Example 6.4: Scoring

This example uses the same input table and the same TEXTMINE procedure call that are used in Example 6.1 to illustrate how you can match extracted rules in documents. Then it adds the DATA step to generate testing data. The DATA step and procedure call are repeated here for convenience.

The following DATA step creates the mycas.reviews data table, which contains nine observations that have four variables. The text variable contains the input reviews. The positive variable contains the sentiment of the reviews: a value of 1 indicates that the review is positive, and a value of 0 indicates that the review is

negative. The category variable contains the category of the reviews. The did variable contains the ID of the documents. Each row in the data table represents a document for analysis.

```
data mycas.reviews;
   infile datalines delimiter='|' missover;
   length text $300 category $20;
   input text$ positive category$ did;
   datalines;
      This is the greatest phone ever! love it!|1|electronics|1
      The phone's battery life is too short and screen resolution is low.|0|electronics|2
      The screen resolution is low, but I love this tv.|1|electronics|3
      The movie itself is great and I like it, although the resolution is low.|1|movies|4
      The movie's story is boring and the acting is poor.|0|movies|5
      I watched this movie on tv, it's not good on a small screen. |0|movies|6
      watched the movie first and loved it, the book is even better!|1|books |7
      I like the story in this book, they should put it on screen.|1|books|8
      I love the author, but this book is a waste of time, don't buy it.|0|books|9
   ;
run;
```

The following DATA step generates the testing data, which contain two observations that have two variables. The text variable contains the input reviews. The did variable contains the ID of the documents. Each row in the data table represents a document for analysis.

```
data mycas.reviews_test;
   infile datalines delimiter='|' missover;
   length text $300;
   input text$ did;
   datalines;
      love it! a great phone, even better than advertised|1
      I like the book, GREATEST in this genre|2
   ;
run;
```

The following TEXTMINE procedure call parses the mycas.reviews data table, stores the term-by-document matrix in the mycas.reviews_bow data table in transactional format, and stores terms that appeared in the mycas.reviews data table in the mycas.reviews_terms data table:

```
 proc textmine data=mycas.reviews;
     doc_id
         did;
     var
         text;
     parse
         nonoungroups
         notagging
         entities    = none
         outparent   = mycas.reviews_bow
         outterms    = mycas.reviews_terms
         outconfig   = mycas.parseconfig
         reducef     = 1;
 run;
```

The following statements run PROC BOOLRULE to extract rules from the mycas.reviews_bow data table. TARGETTYPE=BINARY is specified. One target variable, positive, is specified; this variable indicates whether the reviews are positive or negative.

```
proc boolrule
        data        =   mycas.reviews_bow
        docid       =   _document_
        termid      =   _termnum_
        docinfo     =   mycas.reviews
        terminfo    =   mycas.reviews_terms
        minsupports =   1
        mpos        =   1
        gpos        =   1;
    docinfo
        id          =   did
        targettype  =   binary
        targets     =   (positive);
    terminfo
        id          =   key
        label       =   term;
    output
        ruleterms   =   mycas.ruleterms
        rules       =   mycas.rules;
run;
```

The TMSCORE procedure uses the parsing configuration that is stored in the mycas.parseconfig data table to parse the mycas.reviews_test data table. The term-by-document matrix is stored in the mycas.reviews_test_bow data table.

```
proc tmscore
   data       = mycas.reviews_test
   terms      = mycas.reviews_terms
   config     = mycas.parseconfig
   outparent  = mycas.reviews_test_bow;
   doc_id did;
   var text;
run;
```

The following statements run PROC BOOLRULE to match rules in the testing data and run PROC PRINT to show the matching results:

```
proc boolrule
   data        =   mycas.reviews_test_bow
   docid       =   _document_
   termid      =   _termnum_;
   score
      ruleterms = mycas.ruleterms
      outmatch  = mycas.match;
run;

proc print data=mycas.match; run;
```

The mycas.match data table in Output 6.4.1 shows which documents satisfy which rules.

Output 6.4.1 The mycas.match Data Table

Obs	_DOCUMENT_	_TARGET_	_RULE_ID_
1	1	1	4
2	1	1	3
3	1	1	2
4	2	1	3
5	2	1	1

References

Cox, J., and Zhao, Z. (2014). "System for Efficiently Generating *k*-Maximally Predictive Association Rules with a Given Consequent." US Patent Number 20140337271.

Chapter 7
The FACTMAC Procedure

Contents

Overview: FACTMAC Procedure **99**
PROC FACTMAC Features 100
Using CAS Sessions and CAS Engine Librefs 100
Getting Started: FACTMAC Procedure **101**
Syntax: FACTMAC Procedure **103**
PROC FACTMAC Statement 103
AUTOTUNE Statement 105
CODE Statement 107
ID Statement 108
INPUT Statement 108
OUTPUT Statement 108
SAVESTATE Statement 109
TARGET Statement 109
Details: FACTMAC Procedure **109**
Displayed Output 110
ODS Table Names 111
Output Data Tables 111
Examples: FACTMAC Procedure **112**
Example 7.1: Running PROC FACTMAC with the MovieLens Data Set 112
References **114**

Overview: FACTMAC Procedure

The FACTMAC procedure implements the factorization machine model in SAS Viya. 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 read and write data in distributed form, and to perform factorization in parallel by making full use of multicore computers or distributed 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 statement. 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 in a distributed computing environment or on a single-machine. The following list summarizes the basic features of PROC FACTMAC:

- is highly distributed and multithreaded
- learns a factorization machine model based on a parallel implementation of the SGD optimization algorithm

Using CAS Sessions and CAS Engine Librefs

SAS Cloud Analytic Services (CAS) is the analytic server and associated cloud services in SAS Viya. This section describes how to create a CAS session and set up a CAS engine libref that you can use to connect to the CAS session. It assumes that you have a CAS server already available; contact your system administrator if you need help starting and terminating a server. This CAS server is identified by specifying the host on which it runs and the port on which it listens for communications. To simplify your interactions with this CAS server, the host information and port information for the server are stored as SAS option values that are retrieved automatically whenever this CAS server needs to be accessed. You can examine the host and port values for the server at your site by using the following statements:

```
proc options option=(CASHOST CASPORT);
run;
```

In addition to starting a CAS server, your system administrator might also have created a CAS session and a CAS engine libref for your use. You can define your own sessions and CAS engine librefs that connect to the CAS server as shown in the following statements:

```
cas mysess;
libname mycas cas sessref=mysess;
```

The CAS statement creates the CAS session named mysess, and the LIBNAME statement creates the mycas CAS engine libref that you use to connect to this session. It is not necessary to explicitly name the CASHOST and CASPORT of the CAS server in the CAS statement, because these values are retrieved from the corresponding SAS option values.

If you have created the mysess session, you can terminate it by using the TERMINATE option in the CAS statement as follows:

```
cas mysess terminate;
```

For more information about the CAS and LIBNAME statements, see the section "Introduction to Shared Concepts" on page 11 in Chapter 3, "Shared Concepts."

Getting Started: FACTMAC Procedure

NOTE: Input data must be in a CAS table that is accessible in your CAS session. You must refer to this table by using a two-level name. The first level must be a CAS engine libref, and the second level must be the table name. For more information, see the sections "Using CAS Sessions and CAS Engine Librefs" on page 11 and "Loading a SAS Data Set onto a CAS Server" on page 12 in Chapter 3, "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 CAS session by naming your CAS engine libref in the first statement of the following DATA step:

```
data mycas.cars;
   set sashelp.cars;
run;
```

These statements assume that your CAS engine libref is named mycas, but you can substitute any appropriately defined CAS engine libref.

The following statements run PROC FACTMAC and output the results to ODS tables:

```
proc factmac data=mycas.cars outmodel=mycas.factors maxiter=50
             nfactors=5  learnstep=0.002;
   input make model type /level=nominal;
   target mpg_city  /level=interval;
   output out=mycas.score_out copyvars=(make model type mpg_city);
run;

proc print data=mycas.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 mycas.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 mycas.score_out and that the make, model, type and mpg_city variables be copied from the mycas.cars data table to the mycas.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* > **;**
CODE FILE=*filename* **;**
ID *variables* **;**
INPUT *variables* < **LEVEL=NOMINAL** > **;**
OUTPUT OUT=*CAS-libref.data-table* < *options* > **;**
SAVESTATE RSTORE=*CAS-libref.data-table* **;**
TARGET *variable* < **LEVEL=INTERVAL** > **;**
AUTOTUNE < *options* > **;**

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 the PROC FACTMAC statement.

Table 7.1 PROC FACTMAC Statement Options

Option	Description
Input Data Table Options	
DATA=	Specifies the input data table
Factorization Options	
NFACTORS=	Specifies the number of factors to estimate for the model
MAXITER=	Specifies the maximum number of iterations
SEED=	Specifies the seed to be used for pseudorandom number generation
LEARNSTEP=	Specifies the learning step size for the SGD algorithm
NTHREADS=	Specifies the number of threads to use on each computation node
NONNEGATIVE	Requests nonnegative factorization

You can specify the following *options*:

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

CAS-libref refers to a collection of information that is defined in the LIBNAME statement and includes the caslib, which includes a path to the data, and a session identifier, which defaults to the active session but which can be explicitly defined in the LIBNAME statement. For more information about *CAS-libref*, see the section "Using CAS Sessions and CAS Engine Librefs" on page 100.

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 per computer.

OUTMODEL=*CAS-libref.data-table*

specifies the output model data table to contain the computed factor parameters. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 100.

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.

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 14 in Chapter 3, "Shared Concepts." The TUNINGPARAMETERS= option is described following Table 7.2.

Table 7.2 AUTOTUNE Statement Options

Option	Description
EVALHISTORY=	Specifies how to report the evaluation history of the tuner
FRACTION=	Specifies the fraction of observations to use for validation
KFOLD=	Specifies the number of folds for *k*-fold cross validation
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
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
TARGETEVENT=	Specifies the target event for ROC-based calculations
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.

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=*CAS-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=*CAS-libref.data-table*
names the output data table for PROC FACTMAC to use. You must specify this option before any other options. *CAS-libref.data-table* is a two-level name, where

CAS-libref refers to a collection of information that is defined in the LIBNAME statement and includes the caslib, which includes a path to where the data table is to be stored, and a session identifier, which defaults to the active session but which can be explicitly defined in the LIBNAME statement. For more information about *CAS-libref*, see the section "Using CAS Sessions and CAS Engine Librefs" on page 100.

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

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, "The ASTORE Procedure."

You must specify the following option:

RSTORE=*CAS-libref.data-table*
specifies a data table in which to save the analytic store for the model. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 100.

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.

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 in each computation node. The variability between runs is nevertheless expected to be small. PROC FACTMAC can still use multiple machines for the analysis even when NTHREADS=1.

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.

Output CAS Tables

When you specify the OUTPUT statement to create output tables on your CAS server, the FACTMAC procedure produces the output data table along with a table that lists the CAS library, the data table name, and the number of rows and columns in that data table.

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 7.3.

Table 7.3 ODS Tables Produced by PROC FACTMAC

Table Name	Description	Statement	Option
DescStatsInt	Descriptive statistics for interval variables	PROC FACTMAC	Default
FinalLoss	Final exact loss	PROC FACTMAC	Default
ModelInfo	Model information	PROC FACTMAC	Default
NObs	Number of observations	PROC FACTMAC	Default
OptIterHistory	Iteration history	PROC FACTMAC	Default
OutputCASTables	Summary table for output data; it contains the number of observations and the number of variables	PROC FACTMAC	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 statement. Details about the data table are listed in Table 7.4.

Table 7.4 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 CAS table that is accessible in your CAS session. You must refer to this table by using a two-level name. The first level must be a CAS engine libref, and the second level must be the table name. For more information, see the sections "Using CAS Sessions and CAS Engine Librefs" on page 11 and "Loading a SAS Data Set onto a CAS Server" on page 12 in Chapter 3, "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*. Assuming the destination directory is */data*, the following DATA step loads the data table from the directory into your CAS session:

```
proc casutil;
    load file="~/data/u.data"    /*or other user-defined location*/
    casout="movlens"
    importoptions=(filetype="CSV" delimiter="TAB" getnames="FALSE"
                   vars=("userid" "itemid" "rating" "timestamp"));
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 statements show how to use PROC FACTMAC to predict movie ratings:

```
proc factmac data=mycas.movlens nfactors=10 learnstep=0.15
             maxiter=20  outmodel=mycas.factors;

   input userid itemid /level=nominal;
   target rating /level=interval;
   output out=mycas.out1 copyvars=(userid itemid rating);
run;
```

The following statements print the first 10 observations in the mycas.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=mycas.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=mycas.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

References

Rendle, S. (2012). "Factorization Machines with libFM." *ACM Transactions on Intelligent Systems and Technology* 3:1–22.

Chapter 8
The FASTKNN Procedure

Contents

Overview: FASTKNN Procedure **115**
PROC FASTKNN Features 115
Using CAS Sessions and CAS Engine Librefs 116
Getting Started: FASTKNN Procedure **116**
Syntax: FASTKNN Procedure **117**
PROC FASTKNN Statement 118
ID Statement 119
INPUT Statement 119
OUTPUT Statement 119
Details: FASTKNN Procedure **120**
Displayed Output 120
ODS Table Names 121
Output Data Tables 121

Overview: FASTKNN Procedure

The FASTKNN procedure implements the *k*-nearest neighbor (*k*-NN) search algorithm in SAS Viya. The *k*-NN algorithm has numerous applications, including recommendation systems, image search, fingerprint recognition, and clustering. You can use the FASTKNN procedure to read and write data in distributed form, and to perform search in parallel by making full use of multicore computers or distributed computing environments.

The FASTKNN procedure operates on a query data table and an input data table. For each observation in the query data table, PROC FASTKNN returns the *k* observations in the input data table that are nearest. The FASTKNN procedure uses the Euclidean distance metric to compute distances between the input and query observations, and stores the results of the search in the output data table that is specified by the OUTPUT statement. In addition, PROC FASTKNN 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 PROC FASTKNN exclude distances that are greater than a specified threshold value.

PROC FASTKNN Features

PROC FASTKNN enables you to use parallel execution for *k*-NN search in a distributed computing environment or on a single-machine. The following list summarizes the basic features of PROC FASTKNN:

- is highly distributed and multithreaded
- returns *k*-nearest neighbors based on a parallel implementation of the *k*-NN search algorithm

Using CAS Sessions and CAS Engine Librefs

SAS Cloud Analytic Services (CAS) is the analytic server and associated cloud services in SAS Viya. This section describes how to create a CAS session and set up a CAS engine libref that you can use to connect to the CAS session. It assumes that you have a CAS server already available; contact your system administrator if you need help starting and terminating a server. This CAS server is identified by specifying the host on which it runs and the port on which it listens for communications. To simplify your interactions with this CAS server, the host information and port information for the server are stored as SAS option values that are retrieved automatically whenever this CAS server needs to be accessed. You can examine the host and port values for the server at your site by using the following statements:

```
proc options option=(CASHOST CASPORT);
run;
```

In addition to starting a CAS server, your system administrator might also have created a CAS session and a CAS engine libref for your use. You can define your own sessions and CAS engine librefs that connect to the CAS server as shown in the following statements:

```
cas mysess;
libname mycas cas sessref=mysess;
```

The CAS statement creates the CAS session named mysess, and the LIBNAME statement creates the mycas CAS engine libref that you use to connect to this session. It is not necessary to explicitly name the CASHOST and CASPORT of the CAS server in the CAS statement, because these values are retrieved from the corresponding SAS option values.

If you have created the mysess session, you can terminate it by using the TERMINATE option in the CAS statement as follows:

```
cas mysess terminate;
```

For more information about the CAS and LIBNAME statements, see the section "Introduction to Shared Concepts" on page 11 in Chapter 3, "Shared Concepts."

Getting Started: FASTKNN Procedure

NOTE: Input data must be in a CAS table that is accessible in your CAS session. You must refer to this table by using a two-level name. The first level must be a CAS engine libref, and the second level must be the table name. For more information, see the sections "Using CAS Sessions and CAS Engine Librefs" on page 11 and "Loading a SAS Data Set onto a CAS Server" on page 12 in Chapter 3, "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 CAS session by naming your CAS engine libref in the first statement of the following DATA step. These statements assume that your CAS engine libref is named mycas, but you can substitute any appropriately defined CAS engine libref.

```
data mycas.hmeq;
set sampsio.hmeq(obs=4000);
id=_n_;
run;

data mycas.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            = mycas.hmeq
   query           = mycas.query
   outdist         = mycas.dist_out
   K               = 4
   threshdist      = 10000;
   id                id;
   input             bad loan mortdue value yoj derog delinq clage ninq clno debtinc;
   output out      = mycas.knn_out;
run;
```

The K=4 option requests that the model return four neighbors, the THRESHDIST=10000 option requests that the model exclude distances that are greater than 10,000, the ID statement specifies that the id variable is to be used as a record identifier, and the INPUT statement specifies that the bad, loan, mortdue, value, yoj, derog, delinq, clage, ninq, clno, and debtinc are to be used as inputs. The OUTPUT statement requests that the neighbors be written to the data table mycas.knn_out. The OUTDIST= option requests that the distances from the query observations to the input observations be written to the data table mycas.dist_out.

Syntax: FASTKNN Procedure

The following statements are available in the FASTKNN procedure:

PROC FASTKNN < *options* > **;**
ID *variables* **;**
INPUT *variables* < **LEVEL=INTERVAL** > **;**
OUTPUT OUT=*CAS-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	
K=	Specifies the number of neighbors to return for each query
NOPRINT	Suppresses ODS output
NTHREADS=	Specifies the number of threads to use on each computation node
OUTDIST=	Specifies the output data table in which to store the computed distances
THRESHDIST=	Specifies the threshold on the distance

You can specify the following *options*:

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

CAS-libref refers to a collection of information that is defined in the LIBNAME statement and includes the caslib, which includes a path to the data, and a session identifier, which defaults to the active session but which can be explicitly defined in the LIBNAME statement. For more information about *CAS-libref*, see the section "Using CAS Sessions and CAS Engine Librefs" on page 116.

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

K=*number*
specifies the number of neighbors to return, where *number* is a positive real number.

By default, K=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 from 1 to 64, inclusive. The default value is the maximum number of available threads per computer.

OUTDIST=*CAS-libref.data-table*
specifies the output model data table to contain the computed pairwise distances between query and input observations. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 116.

QUERY=*CAS-libref.data-table*
specifies the input data table that contains the query observations. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 116.

THRESHDIST=*number*
specifies the distance threshold for the model, where *number* is an integer greater than 0. PROC FASTKNN excludes from the computation distances greater than *number*.

By default, THRESHDIST=100.

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.

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=*CAS-libref.data-table* < *options* > **;**

The OUTPUT statement creates a nearest neighbor data table to contain the results of the procedure run.

You must specify the following *option*:

OUT=*CAS-libref.data-table*
names the output data table for PROC FASTKNN to use. You must specify this option before any other options. *CAS-libref.data-table* is a two-level name, where

CAS-libref refers to a collection of information that is defined in the LIBNAME statement and includes the caslib, which includes a path to where the data table is to be stored, and a session identifier, which defaults to the active session but which can be explicitly defined in the LIBNAME statement. For more information about *CAS-libref*, see the section "Using CAS Sessions and CAS Engine Librefs" on page 116.

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

Details: FASTKNN Procedure

The distance metric in the FASTKNN procedure is defined as

$$D_{i,j} = \sqrt{\sum_{i=1}^{N}\sum_{j=1}^{M}(p_i - q_j)^2}$$

where D is the Euclidean distance metric, M is the number of observations in the QUERY= table, N is the number of input observations in the DATA= table, and p and q represent the query and the input observations respectively.

The distances are sorted in ascending order, and the k neighbors that correspond to the smallest distances are returned.

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, and the value of the distance threshold specified in the THRESHDIST= option in the PROC FASTKNN statement.

Number of Observations

The "Number of Observations" table displays the number of observations that are read from the input data table and used.

Output CAS Tables

When you specify the OUTPUT statement or the OUTDIST= option in the PROC FASTKNN statement to create output tables on your CAS server, the FASTKNN procedure produces those output data tables along with a table that lists the CAS library, the data table name, and the number of rows and columns in that data table.

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	Summary table for output data; it contains the number of observations and the number of variables	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. You specify the names of these data tables in the OUTPUT statement and in the OUTDIST= option 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
KNN_OUT	Lists the nearest neighbors
DIST_OUT	Lists the distances from query observations to the input observations

Chapter 9
The FISM Procedure

Contents

Overview: FISM Procedure **123**
PROC FISM Features 123
Using CAS Sessions and CAS Engine Librefs 124
Getting Started: FISM Procedure **124**
Syntax: FISM Procedure **125**
PROC FISM Statement 125
CUSTOMER Statement 126
OUTPUT Statement 126
TARGET Statement 127
Details: FISM Procedure **127**
FP-Growth Algorithm 127
Displayed Output 128
Results 129
Limitations 129
Examples: FISM Procedure **129**
Example 9.1: Identifying Frequent Item Sets 129
References **130**

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 named 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 127.

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:

- reads input data in parallel when the data source is on a distributed system
- 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 in a distributed environment for faster performance
- stores large databases in compact FP-tree format, which uses memory efficiently.

Using CAS Sessions and CAS Engine Librefs

SAS Cloud Analytic Services (CAS) is the analytic server and associated cloud services in SAS Viya. This section describes how to create a CAS session and set up a CAS engine libref that you can use to connect to the CAS session. It assumes that you have a CAS server already available; contact your system administrator if you need help starting and terminating a server. This CAS server is identified by specifying the host on which it runs and the port on which it listens for communications. To simplify your interactions with this CAS server, the host information and port information for the server are stored as SAS option values that are retrieved automatically whenever this CAS server needs to be accessed. You can examine the host and port values for the server at your site by using the following statements:

```
proc options option=(CASHOST CASPORT);
run;
```

In addition to starting a CAS server, your system administrator might also have created a CAS session and a CAS engine libref for your use. You can define your own sessions and CAS engine librefs that connect to the CAS server as shown in the following statements:

```
cas mysess;
libname mycas cas sessref=mysess;
```

The CAS statement creates the CAS session named mysess, and the LIBNAME statement creates the mycas CAS engine libref that you use to connect to this session. It is not necessary to explicitly name the CASHOST and CASPORT of the CAS server in the CAS statement, because these values are retrieved from the corresponding SAS option values.

If you have created the mysess session, you can terminate it by using the TERMINATE option in the CAS statement as follows:

```
cas mysess terminate;
```

For more information about the CAS and LIBNAME statements, see the section "Introduction to Shared Concepts" on page 11 in Chapter 3, "Shared Concepts."

Getting Started: FISM Procedure

NOTE: Input data must be in a CAS table that is accessible in your CAS session. You must refer to this table by using a two-level name. The first level must be a CAS engine libref, and the second level must be the table

name. For more information, see the sections "Using CAS Sessions and CAS Engine Librefs" on page 11 and "Loading a SAS Data Set onto a CAS Server" on page 12 in Chapter 3, "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 CAS session by specifying your CAS engine libref in the first statement in the following DATA step:

```
data mycas.assocs;
   set sampsio.assocs;
run;
```

These statements assume that your CAS engine libref is named mycas, as in the section "Using CAS Sessions and CAS Engine Librefs" on page 124, but you can substitute any appropriately defined CAS engine libref.

The following statements execute the FISM procedure on the mycas.assocs data table and generate output in mycas.out data table, which is specified in the OUTPUT statement:

```
proc fism data=mycas.assocs items=3 support=100;
 output out=mycas.out outfreq=mycas.outfreq;
 customer Customer;
 target product;
 run;
```

The PROC FISM statement specifies that input data are contained in the mycas.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 *variables* **;**
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=*CAS-libref.data-table*
names the input data table for PROC FISM to use. *CAS-libref.data-table* is a two-level name, where

CAS-libref refers to a collection of information that is defined in the LIBNAME statement and includes the caslib, which includes a path to the data, and a session identifier, which defaults to the active session but which can be explicitly defined in the LIBNAME statement. For more information about *CAS-libref*, see the section "Using CAS Sessions and CAS Engine Librefs" on page 124.

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 0. 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 *option*:

ITEMS=*number*
specifies the number of items in a frequent item set, where *number* must be an integer between 1 and 100. By default, ITEMS=2 when the OUT= option is specified in the OUTPUT statement; otherwise, ITEMS=1 by default.

CUSTOMER Statement

CUSTOMER *variable* **;**

The CUSTOMER statement groups one or more specified target *variables* into a basket. You can also specify this statement as ID *variables*.

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=*CAS-libref.data-table*
creates a data table that contains information about frequent item sets with their transaction counts and support. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 124.

OUTFREQ=*CAS-libref.data-table*
creates a data table that contains information about the unique frequent items with their transaction counts and support. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 124.

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 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.

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 hangs.

The maximum number of rules that are generated on each node is one million.

Examples: FISM Procedure

NOTE: Input data must be in a CAS table that is accessible in your CAS session. You must refer to this table by using a two-level name. The first level must be a CAS engine libref, and the second level must be the table name. For more information, see the sections "Using CAS Sessions and CAS Engine Librefs" on page 11 and "Loading a SAS Data Set onto a CAS Server" on page 12 in Chapter 3, "Shared Concepts."

Example 9.1: Identifying Frequent Item Sets

NOTE: Input data must be in a CAS table that is accessible in your CAS session. You must refer to this table by using a two-level name. The first level must be a CAS engine libref, and the second level must be the table name. For more information, see the sections "Using CAS Sessions and CAS Engine Librefs" on page 11 and "Loading a SAS Data Set onto a CAS Server" on page 12 in Chapter 3, "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 CAS session by specifying your CAS engine libref in the first statement in the following DATA step:

```
data mycas.assocs;
   set sampsio.assocs;
run;
```

These statements assume that your CAS engine libref is named mycas, but you can substitute any appropriately defined CAS engine libref.

The following statements run the FP-growth algorithm on the mycas.assocs data table:

```
proc fism data=mycas.assocs items=5 support=200;
 output out=mycas.out outfreq=mycas.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*. New York: ACM Press. http://crono.dei.unipd.it/~lds/MATERIALE/AgrawalIS93.pdf.

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
The FOREST Procedure

Contents

Overview: FOREST Procedure **132**
PROC FOREST Features 133
Using CAS Sessions and CAS Engine Librefs 133
Getting Started: FOREST Procedure **133**
Syntax: FOREST Procedure **138**
PROC FOREST Statement 139
AUTOTUNE Statement 143
CODE Statement 147
CROSSVALIDATION Statement 147
GROW Statement 148
ID Statement 149
INPUT Statement 149
OUTPUT Statement 149
PARTITION Statement 150
SAVESTATE Statement 151
TARGET Statement 151
WEIGHT Statement 151
Details: FOREST Procedure **152**
Bagging the Data 152
Training a Decision Tree 152
Loh Method 153
Predicting an Observation 153
Measuring Prediction Error 153
Handling Missing Values 154
Strategies 154
Specifics 155
Handling Values That Are Absent from Training Data 155
Measuring Variable Importance 155
Residual Sum of Squares Importance Method 156
Random Branch Assignment Importance Method 158
Hyperparameter Tuning 158
k-fold Cross Validation 158
Displayed Output 159
Model Information 159
Number of Observations 159
Variable Importance 159

RBA Variable Importance 159
Fit Statistics 159
Tuner Information 159
Tuner Summary 160
Tuner Timing 160
Best Configuration 160
Tuner Results 160
Cross Validation Results 160
Evaluation History 160
Output CAS Tables 160
Predicted Probability Names 160
ODS Table Names 160
Example: FOREST Procedure **162**
Example 10.1: Scoring New Data by Using a Previous Forest Model 162
References **164**

Overview: FOREST Procedure

The FOREST procedure creates a predictive model called a *forest* (which consists of several decision trees) in SAS Viya. 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 CAS Sessions and CAS Engine Librefs

SAS Cloud Analytic Services (CAS) is the analytic server and associated cloud services in SAS Viya. This section describes how to create a CAS session and set up a CAS engine libref that you can use to connect to the CAS session. It assumes that you have a CAS server already available; contact your system administrator if you need help starting and terminating a server. This CAS server is identified by specifying the host on which it runs and the port on which it listens for communications. To simplify your interactions with this CAS server, the host information and port information for the server are stored as SAS option values that are retrieved automatically whenever this CAS server needs to be accessed. You can examine the host and port values for the server at your site by using the following statements:

```
proc options option=(CASHOST CASPORT);
run;
```

In addition to starting a CAS server, your system administrator might also have created a CAS session and a CAS engine libref for your use. You can define your own sessions and CAS engine librefs that connect to the CAS server as shown in the following statements:

```
cas mysess;
libname mycas cas sessref=mysess;
```

The CAS statement creates the CAS session named mysess, and the LIBNAME statement creates the mycas CAS engine libref that you use to connect to this session. It is not necessary to explicitly name the CASHOST and CASPORT of the CAS server in the CAS statement, because these values are retrieved from the corresponding SAS option values.

If you have created the mysess session, you can terminate it by using the TERMINATE option in the CAS statement as follows:

```
cas mysess terminate;
```

For more information about the CAS and LIBNAME statements, see the section "Introduction to Shared Concepts" on page 11 in Chapter 3, "Shared Concepts."

Getting Started: FOREST Procedure

NOTE: Input data must be in a CAS table that is accessible in your CAS session. You must refer to this table by using a two-level name. The first level must be a CAS engine libref, and the second level must be the table name. For more information, see the sections "Using CAS Sessions and CAS Engine Librefs" on page 11 and "Loading a SAS Data Set onto a CAS Server" on page 12 in Chapter 3, "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 mycas.hmeq data into your CAS session. For this example, the statements assume that your CAS engine libref is named mycas, but you can substitute any appropriately defined CAS engine libref.

```
/* Convert variable names to mixed case */
data mycas.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=mycas.hmeq(obs=10); run;
```

Output 10.1 shows the first 10 observations of mycas.hmeq.

Figure 10.1 Partial Listing of the mycas.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 mycas.savedModel:

```
proc forest data=mycas.hmeq outmodel=mycas.savedModel;
   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	2059146326
Bootstrap Percentage	60
Number of Bins	20
Number of Input Variables	12
Maximum Number of Tree Nodes	321
Minimum Number of Tree Nodes	123
Maximum Number of Branches	2
Minimum Number of Branches	2
Maximum Depth	20
Minimum Depth	20
Maximum Number of Leaves	161
Minimum Number of Leaves	62
Maximum Leaf Size	2579
Minimum Leaf Size	5
OOB Misclassification Rate	0.10251678

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
DebtInc	274.92	83.5246
Delinq	90.6345	13.3056
Value	73.9882	24.1221
Derog	53.3015	7.7220
nInq	42.7026	3.3223
CLNo	30.3675	3.8418
CLAge	26.3095	4.9389
Job	24.7731	2.4681
MortDue	23.1489	3.0192
YoJ	21.7237	2.2514
Loan	21.1239	6.0751
Reason	5.0869	1.5725

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 152. 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	0.0939	0.0802	0.117	0.1023	0.725	0.497
2	0.1709	0.0711	0.122	0.0936	0.819	0.279
3	0.2149	0.0684	0.120	0.0901	0.840	0.243
4	0.2387	0.0656	0.118	0.0878	0.854	0.231
5	0.2566	0.0661	0.116	0.0846	0.849	0.233
6	0.2685	0.0646	0.114	0.0827	0.842	0.228
7	0.2760	0.0644	0.112	0.0820	0.826	0.227
8	0.2811	0.0652	0.112	0.0807	0.810	0.231
9	0.2849	0.0651	0.109	0.0785	0.817	0.231
10	0.2875	0.0648	0.107	0.0789	0.819	0.230
.	.	.	.	.	.	.
.	.	.	.	.	.	.
.	.	.	.	.	.	.
91	0.3052	0.0628	0.103	0.0827	0.852	0.222
92	0.3054	0.0627	0.103	0.0829	0.853	0.221
93	0.3053	0.0628	0.103	0.0829	0.852	0.222
94	0.3053	0.0627	0.102	0.0824	0.853	0.221
95	0.3054	0.0627	0.102	0.0826	0.853	0.222
96	0.3054	0.0627	0.103	0.0824	0.853	0.221
97	0.3056	0.0626	0.102	0.0824	0.853	0.221
98	0.3055	0.0626	0.102	0.0826	0.853	0.221
99	0.3055	0.0626	0.102	0.0827	0.853	0.221
100	0.3055	0.0626	0.103	0.0826	0.853	0.221

Syntax: FOREST Procedure

The following statements are available in the FOREST procedure:

PROC FOREST < *options* > ;
AUTOTUNE < *options* > ;
CODE < *options* > ;
CROSSVALIDATION < **KFOLD=***number* > ;
GROW *criterion* ;
ID *variables* ;
INPUT *variables* < / **LEVEL=NOMINAL** | **INTERVAL** > ;
OUTPUT **OUT**=*CAS-libref.data-table* < *option* > ;
PARTITION *partition-option* ;
SAVESTATE **RSTORE=***CAS-libref.data-table* ;
TARGET *variable* < / **LEVEL=NOMINAL** | **INTERVAL** > ;
WEIGHT *variable* ;

The PROC FOREST, INPUT, and TARGET statements are required. 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 the PROC FOREST 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 to score a new table
LOH=	Specifies the number of variables to preselect using the Loh method
NOPRINT	Suppresses ODS output
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	Create tables that contain information about columns in the output
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 variable interaction importance for the specified number of 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

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 missing interval inputs to the leftmost branch of the split, and unknown and missing nominal levels to the largest branch in the split.

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=BUCKET.

DATA=*CAS-libref.data-table*

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

CAS-libref refers to a collection of information that is defined in the LIBNAME statement and includes the caslib, which includes a path to the data, and a session identifier, which defaults to the active session but which can be explicitly defined in the LIBNAME

statement. For more information about *CAS-libref*, see the section "Using CAS Sessions and CAS Engine Librefs" on page 133.

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

INBAGFRACTION=*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.

By default, INBAGFRACTION=0.6. This value can be tuned with the AUTOTUNE statement.

INMODEL=< *CAS-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. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 133.

When you use the INMODEL= option, the OUTPUT statement is required and any other options in the PROC FOREST statement, except for NOPRINT, are ignored.

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 one.

By default, MAXDEPTH=20.

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, MINLEAFSIZE=5.

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.

NOPRINT
suppresses ODS output.

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*
specifies the *number* of bins in which to bin 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=20.

OUTMODEL=< *CAS-libref.***>***data-table*
specifies the data table to which to save the forest model. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 133.

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.

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 155.

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 from 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 (Experimental)

INTERACTIONIMP=2 | 3

calculates the variable interaction importance, which is described in the section "Variable Interaction Importance (Experimental)" on page 157. You can specify the following values:

2	calculates all two-way variable importance interactions.
3	calculates all three-way and all two-way variable importance 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.

By default, VOTE=PROBABILITY.

AUTOTUNE Statement

AUTOTUNE < *options* > **;**

The AUTOTUNE statement searches for the best combination of values of the INBAGFRACTION=, MAXDEPTH=, 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 14 in Chapter 3, "Shared Concepts." The TUNINGPARAMETERS= option is described following Table 10.3.

Table 10.3 AUTOTUNE Statement Options

Option	Description
EVALHISTORY=	Specifies how to report the evaluation history of the tuner
FRACTION=	Specifies the fraction of observations to use for validation
KFOLD=	Specifies the number of folds for *k*-fold cross validation
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
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
TARGETEVENT=	Specifies the target event for ROC-based calculations
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.

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 a catalog entry, or to a CAS table. To score new data, you can then include the file or the catalog entry in a DATA step, or you can specify the CAS table in the **`runCodeTable`** action in the **`dataStep`** action set (for more information, see *SAS Viya: System Programming Guide*).

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
OUT=	Names an output CAS table in which to save the generated code

For more information about the syntax of the CODE statement, see the section "CODE Statement" on page 18 in Chapter 3, "Shared Concepts."

CROSSVALIDATION Statement

CROSSVALIDATION < **KFOLD=***number* > **;**

The CROSSVALIDATION statement performs a *k*-fold cross validation process to find the average estimated validation error. You cannot specify the CROSSVALIDATION statement if you specify either the AUTOTUNE statement or the PARTITION statement.

You can specify the following option:

KFOLD=*number*
specifies the number of partition folds in the cross validation process, where *number* must be between 2 and 20, inclusive.

By default, KFOLD=5.

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=NOMINAL** | **INTERVAL** > **;**

The INPUT statement names input *variables* that share common options. The INPUT statement can be repeated.

You can specify the following option:

LEVEL=NOMINAL | **INTERVAL**
specifies the level of measurement of two variables. 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.

OUTPUT Statement

OUTPUT OUT=*CAS-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=*CAS-libref.data-table*
names the output data table for PROC FOREST to use. You must specify this option before any other options. *CAS-libref.data-table* is a two-level name, where

CAS-libref refers to a collection of information that is defined in the LIBNAME statement and includes the caslib, which includes a path to where the data table is to be stored, and a session identifier, which defaults to the active session but which can be explicitly defined in the LIBNAME statement. For more information about *CAS-libref*, see the section "Using CAS Sessions and CAS Engine Librefs" on page 133.

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 20 in Chapter 3, "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 of the observations are 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=*CAS-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, "The ASTORE Procedure."

You must specify the following option:

RSTORE=*CAS-libref.data-table*
specifies a data table in which to save the analytic store for the model. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 133.

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.

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. If a WEIGHT statement is not included, 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.

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 153. 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.

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.

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

$$\text{ASE}_{\text{int}} = \sum_{i=1}^{N} \frac{(y_i - \hat{y}_i)^2}{N}$$

$$\text{ASE}_{\text{cat}} = \sum_{i=1}^{N} \sum_{j=1}^{J} \frac{(\delta_{ij} - \hat{p}_{ij})^2}{JN}$$

$$\text{Misc} = \sum_{i=1}^{N} \frac{1(y_i \neq \hat{y}_i)}{N}$$

$$\text{LogLoss} = -\sum_{i=1}^{N} \sum_{j=1}^{J} \frac{\delta_{ij} \log(\ddot{p}_{ij})}{N}$$

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 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 (Experimental)

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, with node d being the parent of node e
- v and w are variables, with v being the splitting variable for node d and w being 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, with node d being the parent of node e, which is the parent of node f
- v, w, and x are variables, with v being the splitting variable for node d, w being the splitting variable for node e, and x being 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 only compare a two-way interaction with another two-way interaction, or a three-way interaction 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.

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.

Hyperparameter Tuning

For more information about hyperparameter tuning, see the section "Hyperparameter Tuning" on page 22 in Chapter 3, "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 a k-fold cross validation process to find the average estimated validation error (misclassification error for nominal targets or average square error for interval targets) for

the trained 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 (hold-out) fold. The validation error estimates are then averaged over each set of training and scoring executions to obtain a single value. The CROSSVALIDATION statement returns a table that contains a single data row that shows the average validation error.

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 155. 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 158. 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 153. 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 while 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 best found configurations. This table is produced by the AUTOTUNE statement.

Cross Validation Results

The "Cross Validation Results" table contains the average error rate (misclassification error or average square error) 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.

Output CAS Tables

When you specify the OUTPUT statement or the OUTMODEL= option in the PROC FOREST statement to create output tables on your CAS server, the FOREST procedure produces the output data table along with a table that lists the CAS library, the data table name, and the number of rows and columns in that data 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.

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
CrossValidationResults	Average error rate (misclassification error or average square error) 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 FOREST	Default
ModelInfo	Model information	PROC FOREST	Default
Nobs	Number of observations	PROC FOREST	Default
OutCASTbl	Summary table for output data; it contains the number of observations and the number of variables	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
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
TunerTiming	Total time spent on different tasks while tuning	AUTOTUNE	Default
VariableImportance	Residual sum of squares variable importance	PROC FOREST	Default

Example: FOREST Procedure

NOTE: Input data must be in a CAS table that is accessible in your CAS session. You must refer to this table by using a two-level name. The first level must be a CAS engine libref, and the second level must be the table name. For more information, see the sections "Using CAS Sessions and CAS Engine Librefs" on page 11 and "Loading a SAS Data Set onto a CAS Server" on page 12 in Chapter 3, "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 CAS session by naming your CAS engine libref in the first statement of the following DATA step:

```
data mycas.junkmail;
   set sashelp.junkmail;
run;
```

These statements assume that your CAS engine libref is named mycas, but you can substitute any appropriately defined 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=mycas.junkmail outmodel=mycas.forest_model;
   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=mycas.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	0.114	0.1052	0.148	0.1448	0.538	0.430
2	0.192	0.0926	0.140	0.1274	0.769	0.337
3	0.226	0.0857	0.133	0.1150	0.787	0.282
4	0.254	0.0849	0.132	0.1121	0.823	0.280
5	0.275	0.0831	0.132	0.1089	0.849	0.277
6	0.284	0.0803	0.122	0.1054	0.844	0.270
7	0.293	0.0807	0.118	0.1000	0.834	0.272
8	0.298	0.0797	0.115	0.0963	0.841	0.271
9	0.302	0.0794	0.114	0.0982	0.843	0.271
10	0.306	0.0795	0.114	0.0976	0.850	0.272
.	.	.	.	.	.	.
.	.	.	.	.	.	.
.	.	.	.	.	.	.
91	0.332	0.0788	0.103	0.0878	0.892	0.274
92	0.332	0.0787	0.103	0.0876	0.892	0.274
93	0.331	0.0787	0.103	0.0880	0.892	0.274
94	0.331	0.0787	0.102	0.0880	0.892	0.274
95	0.331	0.0787	0.102	0.0878	0.891	0.274
96	0.331	0.0787	0.102	0.0885	0.891	0.274
97	0.331	0.0787	0.103	0.0887	0.892	0.274
98	0.331	0.0787	0.103	0.0887	0.892	0.273
99	0.331	0.0788	0.103	0.0887	0.891	0.274
100	0.331	0.0787	0.104	0.0887	0.892	0.273

The following statements use a previously saved model to score new data:

```
proc forest data=mycas.junkmail inmodel=mycas.forest_model;
     output out=mycas.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	0.1052	0.1448	0.430
2	0.0926	0.1274	0.337
3	0.0857	0.1150	0.282
4	0.0849	0.1121	0.280
5	0.0831	0.1089	0.277
6	0.0803	0.1054	0.270
7	0.0807	0.1000	0.272
8	0.0797	0.0963	0.271
9	0.0794	0.0982	0.271
10	0.0795	0.0976	0.272
.	.	.	.
.	.	.	.
.	.	.	.
91	0.0788	0.0878	0.274
92	0.0787	0.0876	0.274
93	0.0787	0.0880	0.274
94	0.0787	0.0880	0.274
95	0.0787	0.0878	0.274
96	0.0787	0.0885	0.274
97	0.0787	0.0887	0.274
98	0.0787	0.0887	0.273
99	0.0788	0.0887	0.274
100	0.0787	0.0887	0.273

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 mycas.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.

References

Breiman, L. (1996). "Bagging Predictors." *Machine Learning* 24:123–140.

Breiman, L. (2001). "Random Forests." *Machine Learning* 45:5–32.

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.

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
The GRADBOOST Procedure

Contents

Overview: GRADBOOST Procedure **168**
Using CAS Sessions and CAS Engine Librefs 168
Getting Started: GRADBOOST Procedure **169**
Syntax: GRADBOOST Procedure **173**
PROC GRADBOOST Statement 174
AUTOTUNE Statement 179
CODE Statement 183
CROSSVALIDATION Statement 184
ID Statement 184
INPUT Statement 185
OUTPUT Statement 185
PARTITION Statement 186
SAVESTATE Statement 186
TARGET Statement 187
WEIGHT Statement 187
Details: GRADBOOST Procedure **187**
Subsampling the Data 187
Training a Decision Tree 188
Boosting 188
Early Stopping 188
Measuring Prediction Error 189
Handling Missing Values 190
Strategies 190
Specifics 190
Measuring Variable Importance 190
Residual Sum of Squares Importance Method 191
Random Branch Assignment Importance Method 193
Hyperparameter Tuning 193
k-fold Cross Validation 194
Displayed Output 194
Model Information 194
Number of Observations 194
Variable Importance 194
RBA Variable Importance 194
Fit Statistics 194
Tuner Information 195

Tuner Summary 195
Tuner Timing 195
Best Configuration 195
Tuner Results 195
Cross Validation Results 195
Evaluation History 195
Output Table 195
Predicted Probability Names 195
ODS Table Names 196
Example: GRADBOOST Procedure **197**
Example 11.1: Scoring New Data by Using a Previous Boosting Model 197
References **200**

Overview: GRADBOOST Procedure

The GRADBOOST procedure creates a predictive model called a gradient boosting model in SAS Viya. 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 188.

Using CAS Sessions and CAS Engine Librefs

SAS Cloud Analytic Services (CAS) is the analytic server and associated cloud services in SAS Viya. This section describes how to create a CAS session and set up a CAS engine libref that you can use to connect to the CAS session. It assumes that you have a CAS server already available; contact your system administrator if you need help starting and terminating a server. This CAS server is identified by specifying the host on which it runs and the port on which it listens for communications. To simplify your interactions with this CAS server, the host information and port information for the server are stored as SAS option values that are retrieved automatically whenever this CAS server needs to be accessed. You can examine the host and port values for the server at your site by using the following statements:

```
proc options option=(CASHOST CASPORT);
run;
```

In addition to starting a CAS server, your system administrator might also have created a CAS session and a CAS engine libref for your use. You can define your own sessions and CAS engine librefs that connect to the CAS server as shown in the following statements:

```
cas mysess;
libname mycas cas sessref=mysess;
```

The CAS statement creates the CAS session named mysess, and the LIBNAME statement creates the mycas CAS engine libref that you use to connect to this session. It is not necessary to explicitly name the CASHOST and CASPORT of the CAS server in the CAS statement, because these values are retrieved from the corresponding SAS option values.

If you have created the mysess session, you can terminate it by using the TERMINATE option in the CAS statement as follows:

```
cas mysess terminate;
```

For more information about the CAS and LIBNAME statements, see the section "Introduction to Shared Concepts" on page 11 in Chapter 3, "Shared Concepts."

Getting Started: GRADBOOST Procedure

NOTE: Input data must be in a CAS table that is accessible in your CAS session. You must refer to this table by using a two-level name. The first level must be a CAS engine libref, and the second level must be the table name. For more information, see the sections "Using CAS Sessions and CAS Engine Librefs" on page 11 and "Loading a SAS Data Set onto a CAS Server" on page 12 in Chapter 3, "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 11.1 describes the variables in Hmeq.

Table 11.1 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 mycas.hmeq data into your CAS session. For this example, the statements assume that your CAS engine libref is named mycas, but you can substitute any appropriately defined CAS engine libref.

```
data mycas.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=mycas.hmeq(obs=10); run;
```

Output 11.1 shows the first 10 observations of mycas.hmeq.

Figure 11.1 Partial Listing of the mycas.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 mycas.savedModel:

```
proc gradboost data=mycas.hmeq outmodel=mycas.savedModel;
   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 11.2 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 11.2 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	20
Number of Input Variables	12
Maximum Number of Tree Nodes	63
Minimum Number of Tree Nodes	31
Maximum Number of Branches	2
Minimum Number of Branches	2
Maximum Depth	5
Minimum Depth	5
Maximum Number of Leaves	32
Minimum Number of Leaves	16
Maximum Leaf Size	2851
Minimum Leaf Size	5
Seed	123818132

Figure 11.3 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 11.3 Number of Observations

	Training
Number of Observations Read	5960
Number of Observations Used	5960

Figure 11.4 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 11.4 Variable Importance

Variable Importance		
Variable	**Importance**	**Std Dev Importance**
DebtInc	26.9257	53.6874
Delinq	7.3450	5.1979
nInq	6.1775	1.8946
Job	5.4479	1.7540
Derog	5.3054	3.0248
CLAge	5.0686	2.6648
Value	4.9606	3.2630
CLNo	4.9280	1.4812
YoJ	4.8388	1.5184
MortDue	4.7727	1.6993
Loan	3.2080	1.5420
Reason	1.1742	1.3284

Figure 11.5 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 11.5 Fit Statistics

Fit Statistics			
Number of Trees	Training Average Square Error	Training Misclassification Rate	Training Log Loss
1	0.1443	0.1995	0.455
2	0.1324	0.1995	0.424
3	0.1222	0.1995	0.399
4	0.1147	0.1961	0.380
5	0.1084	0.1763	0.364
6	0.1033	0.1473	0.351
7	0.0981	0.1352	0.337
8	0.0941	0.1201	0.326
9	0.0908	0.1124	0.316
10	0.0877	0.1059	0.307
.	.	.	.
.	.	.	.
.	.	.	.
91	0.0338	0.0436	0.126
92	0.0336	0.0436	0.125
93	0.0333	0.0433	0.125
94	0.0330	0.0421	0.123
95	0.0326	0.0416	0.122
96	0.0324	0.0421	0.122
97	0.0322	0.0421	0.121
98	0.0320	0.0418	0.120
99	0.0319	0.0423	0.119
100	0.0317	0.0409	0.119

Syntax: GRADBOOST Procedure

The following statements are available in the GRADBOOST procedure:

PROC GRADBOOST < *options* > **;**
AUTOTUNE < *options* > **;**
CODE < *options* > **;**
CROSSVALIDATION < **KFOLD=***number* > **;**
ID *variables* **;**
INPUT *variables* < / **LEVEL=NOMINAL** | **INTERVAL** > **;**
OUTPUT OUT=*CAS-libref.data-table* < *option* > **;**
PARTITION *partition-option* **;**
SAVESTATE RSTORE=*CAS-libref.data-table* **;**
TARGET *variable* < / **LEVEL=NOMINAL** | **INTERVAL** > **;**
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 11.2 summarizes the options in the PROC GRADBOOST statement.

Table 11.2 PROC GRADBOOST Statement Options

Option	Description
Basic Options	
ADDTREES	Add trees to an already generated boosting model, which is specified in the INMODEL= option
BINMETHOD=	Specifies how to bin interval inputs prior to training
EARLYSTOP=	Specifies how to perform early stopping during model training
INMODEL=	Specifies a saved gradient boosting model to use to score new a table
LASSO=	Specifies the L1 norm regularization parameter
LEARNINGRATE=	Specifies the learning rate for each tree
NOPRINT	Suppresses ODS output
NTREES=	Specifies the number of trees to grow in the boosting model
NUMBIN=	Specifies the number of bins for continuous variables
OUTMODEL=	Specifies the data table to which the gradient boosting model is to be saved
PRINTTARGET	Create 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 variable interaction importance for a specified number of 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

You also specify the following *options*:

ADDTREES
ADDTREE
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 188. During model scoring, observations are assigned to a node in a tree based upon 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 leftmost branch of the split, and assigns observations that have unknown and missing nominal levels to the largest branch in the split.

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=BUCKET.

DATA=*CAS-libref.data-table*

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

CAS-libref refers to a collection of information that is defined in the LIBNAME statement and includes the caslib, which includes a path to the data, and a session identifier, which defaults to the active session but which can be explicitly defined in the LIBNAME statement. For more information about *CAS-libref*, see the section "Using CAS Sessions and CAS Engine Librefs" on page 168.

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

EARLYSTOP(*suboption* **. . .)**

specifies options for stopping the gradient boosting model training early. For more information, see the section "Early Stopping" on page 188. You can specify one or more of the following *suboptions*:

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 no early stopping is performed. If you want to use autotuning by specifying AUTOTUNE statement, but you do not want autotuning to use early stopping, then specify STAGNATION=0.

By default, STAGNATION=0. When the AUTOTUNE statement is specified, STAGNATION=4 by default.

TOLERANCE=*number*

specifies the *number* to be used as the tolerance for early stopping, where *number* must be nonnegative.

By default TOLERANCE=0.

INMODEL=*CAS-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. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 168.

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.

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 one.

By default, MAXDEPTH=5.

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, MINLEAFSIZE=5.

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, MINUSERINSEARCH=1.

NOPRINT

suppresses ODS output.

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*

specifies the *number* of bins in which to bin 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=20.

OUTMODEL=< *CAS-libref.* **>***data-table*

specifies the data table to which to save the gradient boosting model. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 168.

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 190.

RIDGE=*number*

L2=*number*

specifies the L2 norm regularization parameter on prediction. The *number* must be nonnegative.

By default, RIDGE=0. 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 from 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 (Experimental)

INTERACTIONIMP=2 | 3

calculates the variable interaction importance, which is described in the section "Interaction Importance (Experimental)" on page 192. You can specify the following values:

2 calculates all two-way variable importance interactions.

3 calculates all three-way and all two-way variable importance 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=, NTREES=, 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 11.3 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 14 in Chapter 3, "Shared Concepts." The TUNINGPARAMETERS= option is described following Table 11.3.

Table 11.3 AUTOTUNE Statement Options

Option	Description
EVALHISTORY=	Specifies how to report the evaluation history of the tuner
FRACTION=	Specifies the fraction of observations to use for validation
KFOLD=	Specifies the number of folds for *k*-fold cross validation
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
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
TARGETEVENT=	Specifies the target event for ROC-based calculations
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.

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.

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 a catalog entry, or to a CAS table. To score new data, you can then include the file or the catalog entry

in a DATA step, or you can specify the CAS table in the **runCodeTable** action in the **dataStep** action set (for more information, see *SAS Viya: System Programming Guide*).

Table 11.4 summarizes the *options* available in the CODE statement.

Table 11.4 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
OUT=	Names an output CAS table in which to save the generated code

For more information about the syntax of the CODE statement, see the section "CODE Statement" on page 18 in Chapter 3, "Shared Concepts."

CROSSVALIDATION Statement

CROSSVALIDATION < **KFOLD=***number*> **;**

The CROSSVALIDATION statement performs a *k*-fold cross validation process to find the average estimated validation error. You cannot specify the CROSSVALIDATION statement if you specify either the AUTOTUNE statement or the PARTITION statement.

You can specify the following option:

KFOLD=*number*

specifies the number of partition folds in the cross validation process, where *number* must be between 2 and 20, inclusive.

By default, KFOLD=5.

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=NOMINAL | INTERVAL** > **;**

The INPUT statement names input *variables* that share common options. The INPUT statement can be repeated.

You can specify the following option:

LEVEL=NOMINAL | INTERVAL
specifies the level of measurement of two variables. 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.

OUTPUT Statement

OUTPUT OUT=*CAS-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=*CAS-libref.data-table*
names the output data table for PROC GRADBOOST to use. You must specify this option before any other options. *CAS-libref.data-table* is a two-level name, where

CAS-libref refers to a collection of information that is defined in the LIBNAME statement and includes the caslib, which includes a path to where the data table is to be stored, and a session identifier, which defaults to the active session but which can be explicitly defined in the LIBNAME statement. For more information about *CAS-libref*, see the section "Using CAS Sessions and CAS Engine Librefs" on page 168.

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 11.5.

Table 11.5 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 20 in Chapter 3, "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 of the observations are 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=*CAS-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, "The ASTORE Procedure."

You must specify the following option:

RSTORE=*CAS-libref.data-table*
specifies a data table in which to save the analytic store for the model. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 168.

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.

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. If a WEIGHT statement is not included, 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 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. 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 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.

Boosting

A description of gradient boosting for decision trees can be found in Hastie, Tibshirani, and Friedman (2001) and Friedman (2001).

PROC GRADBOOST 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 188. The sequence of trees and how each tree affects a subsequent tree are discussed in Hastie, Tibshirani, and Friedman (2001) and Friedman (2001).

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.

The GRADBOOST procedure performs early stopping by considering a number of consecutive decreases. Two suboptions of the EARLYSTOP option in the PROC GRADBOOST statement dictate how the early stopping is performed: STAGNATION=*number* and TOLERANCE=τ (τ can be any number greater than or equal to 0).

If the relative change in error from iteration *i–1* to iteration *i* is less than τ,

$$\left|\frac{Err_{i-1} - Err_i}{Err_{i-1}}\right| < \tau,$$

for *number* iterations, then 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. If $\tau = 0$, then the stopping criterion is determined by consecutive increases in the error.

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, 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 (Experimental)

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, with node d being the parent of node e
- v and w are variables, with v being the splitting variable for node d and w being 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, with node d being the parent of node e, which is the parent of node f
- v, w, and x are variables, with v being the splitting variable for node d, w being the splitting variable for node e, and x being 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 only compare a two-way interaction with another two-way interaction, or a three-way interaction 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.

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.

Hyperparameter Tuning

For more information about hyperparameter tuning, see the section "Hyperparameter Tuning" on page 22 in Chapter 3, "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
- NTREES= option for the number of trees
- 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 a *k*-fold cross validation process to find the average estimated validation error (misclassification error for nominal targets or average square error for interval targets) for the trained 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 (hold-out) fold. The validation error estimates are then averaged over each set of training and scoring executions to obtain a single value. The CROSSVALIDATION statement returns a table that contains a single data row that shows the average validation error.

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 190. 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 193. 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 189. 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 while 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 best found configurations. This table is produced by the AUTOTUNE statement.

Cross Validation Results

The "Cross Validation Results" table contains the average error rate (misclassification error or average square error) 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.

Output Table

The "Output Table" table describes tables that are created either when you specify the OUTMODEL option in the PROC GRADBOOST statement or when you specify the OUTPUT statement.

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.

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 11.6.

Table 11.6 ODS Tables Produced by PROC GRADBOOST

Table Name	Description	Statement	Option
BestConfiguration	Hyperparameters and objective function values for the best configuration	AUTOTUNE	Default
CrossValidationResults	Average error rate (misclassification error or average square error) 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
ModelInfo	Model information	PROC GRADBOOST	Default
Nobs	Number of observations	PROC GRADBOOST	Default
OutputCASTables	Output table	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
PredIntoName	Predicted name information for nominal targets	PROC GRADBOOST	PRINTTARGET
RBAImportance	Random branch assignment variable importance	PROC GRADBOOST	RBAIMP
TunerInfo	Setup values used by the tuner	AUTOTUNE	Default

Table 11.6 *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 found configurations	AUTOTUNE	Default
TunerSummary	Statistics about the tuning process	AUTOTUNE	Default
TunerTiming	Total time spent on different tasks while tuning	AUTOTUNE	Default
VariableImportance	Residual sum of squares variable importance	PROC GRADBOOST	Default

Example: GRADBOOST Procedure

NOTE: Input data must be in a CAS table that is accessible in your CAS session. You must refer to this table by using a two-level name. The first level must be a CAS engine libref, and the second level must be the table name. For more information, see the sections "Using CAS Sessions and CAS Engine Librefs" on page 11 and "Loading a SAS Data Set onto a CAS Server" on page 12 in Chapter 3, "Shared Concepts."

Example 11.1: 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 CAS session by naming your CAS engine libref in the first statement of the following DATA step:

```
data mycas.junkmail;
   set sashelp.junkmail;
run;
```

These statements assume that your CAS engine libref is named mycas, but you can substitute any appropriately defined 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=mycas.junkmail outmodel=mycas.gradboost_model;
   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=mycas.score_at_runtime;
   ods output FitStatistics=fit_at_runtime;
run;
```

The preceding statements produce the table shown in Output 11.1.1. The table shows the training statistics.

Output 11.1.1 Fit Statistics, Fit at Run Time

Fit Statistics

Number of Trees	Training Average Square Error	Training Misclassification Rate	Training Log Loss
1	0.2144	0.3940	0.620
2	0.1953	0.2467	0.580
3	0.1800	0.2039	0.548
4	0.1669	0.1506	0.520
5	0.1563	0.1361	0.496
6	0.1472	0.1384	0.475
7	0.1391	0.1326	0.456
8	0.1326	0.1308	0.441
9	0.1270	0.1302	0.427
10	0.1217	0.1221	0.413
.	.	.	.
.	.	.	.
.	.	.	.
91	0.0570	0.0706	0.199
92	0.0569	0.0706	0.199
93	0.0568	0.0702	0.199
94	0.0565	0.0698	0.198
95	0.0564	0.0698	0.197
96	0.0561	0.0700	0.196
97	0.0559	0.0702	0.196
98	0.0557	0.0702	0.195
99	0.0555	0.0704	0.194
100	0.0553	0.0702	0.194

The following statements use a previously saved model to score new data:

```
proc gradboost data=mycas.junkmail inmodel=mycas.gradboost_model;
   output out=mycas.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 11.1.2 match those previously seen in Output 11.1.1.

Output 11.1.2 Fit Statistics, Fit Later

Fit Statistics			
Number of Trees	Average Square Error	Misclassification Rate	Log Loss
1	0.2144	0.3940	0.620
2	0.1953	0.2467	0.580
3	0.1800	0.2039	0.548
4	0.1669	0.1506	0.520
5	0.1563	0.1361	0.496
6	0.1472	0.1384	0.475
7	0.1391	0.1326	0.456
8	0.1326	0.1308	0.441
9	0.1270	0.1302	0.427
10	0.1217	0.1221	0.413
.	.	.	.
.	.	.	.
.	.	.	.
91	0.0570	0.0706	0.199
92	0.0569	0.0706	0.199
93	0.0568	0.0702	0.199
94	0.0565	0.0698	0.198
95	0.0564	0.0698	0.197
96	0.0561	0.0700	0.196
97	0.0559	0.0702	0.196
98	0.0557	0.0702	0.195
99	0.0555	0.0704	0.194
100	0.0553	0.0702	0.194

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 mycas.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.

References

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.

Van der Laan, M. J. (2006). "Statistical Inference for Variable Importance." *International Journal of Biostatistics* 2:1–31. Article 2.

Chapter 12
The GVARCLUS Procedure

Contents

Overview: GVARCLUS Procedure **201**
PROC GVARCLUS Features 202
PROC GVARCLUS Compared to PROC VARCLUS 203
Using CAS Sessions and CAS Engine Librefs 203
Getting Started: GVARCLUS Procedure **204**
Syntax: GVARCLUS Procedure **206**
PROC GVARCLUS Statement 206
DISPLAY Statement 209
DISPLAYOUT Statement 210
FREQ Statement 211
INPUT Statement 211
Details: GVARCLUS Procedure **211**
Missing Values 212
Graphical Lasso 212
Graphical Representation of Precision Matrix 213
Preprocessing 213
Displayed Output 213
Number of Observations 214
Cluster Summary 214
ODS Table Names 214
Examples: GVARCLUS Procedure **214**
Example 12.1: Analyzing United States Senate Voting Data 214
Example 12.2: Output a Covariance Matrix to a SAS Data File 216
Example 12.3: Output the Covariance Matrix in LIL Format 217
References **219**

Overview: GVARCLUS Procedure

The GVARCLUS procedure performs variable clustering and graphical modeling in SAS Viya. 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

Because the GVARCLUS procedure runs on CAS, it also does the following:

- enables you to run on a cluster of machines that distribute the data and the computations
- enables you to run in single-machine mode
- exploits all the available cores and concurrent threads. For information about how PROC GVARCLUS uses threads, see the section "Multithreading" on page 21 in Chapter 3, "Shared Concepts."

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 CAS Sessions and CAS Engine Librefs

SAS Cloud Analytic Services (CAS) is the analytic server and associated cloud services in SAS Viya. This section describes how to create a CAS session and set up a CAS engine libref that you can use to connect to the CAS session. It assumes that you have a CAS server already available; contact your system administrator if you need help starting and terminating a server. This CAS server is identified by specifying the host on which it runs and the port on which it listens for communications. To simplify your interactions with this CAS server, the host information and port information for the server are stored as SAS option values that are retrieved automatically whenever this CAS server needs to be accessed. You can examine the host and port values for the server at your site by using the following statements:

```
proc options option=(CASHOST CASPORT);
run;
```

In addition to starting a CAS server, your system administrator might also have created a CAS session and a CAS engine libref for your use. You can define your own sessions and CAS engine librefs that connect to the CAS server as shown in the following statements:

```
cas mysess;
libname mycas cas sessref=mysess;
```

The CAS statement creates the CAS session named mysess, and the LIBNAME statement creates the mycas CAS engine libref that you use to connect to this session. It is not necessary to explicitly name the CASHOST and CASPORT of the CAS server in the CAS statement, because these values are retrieved from the corresponding SAS option values.

If you have created the mysess session, you can terminate it by using the TERMINATE option in the CAS statement as follows:

```
cas mysess terminate;
```

For more information about the CAS and LIBNAME statements, see the section "Introduction to Shared Concepts" on page 11 in Chapter 3, "Shared Concepts."

Getting Started: GVARCLUS Procedure

NOTE: Input data must be in a CAS table that is accessible in your CAS session. You must refer to this table by using a two-level name. The first level must be a CAS engine libref, and the second level must be the table name. For more information, see the sections "Using CAS Sessions and CAS Engine Librefs" on page 11 and "Loading a SAS Data Set onto a CAS Server" on page 12 in Chapter 3, "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 and loads the table into your CAS session. 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 mycas.getStarted;
    array x{20}  x1-x20;
    do i = 1 to 100;
        x1=rannor(12334); x2=0.9*x1+0.2; x3=0.7*x1+0.6; x4=x3*x2*x1; x5=x1*x1*x1;
        x6=rannor(56757); 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} = ranuni(10);
        end;
        output;
    end;
run;
```

These statements assume that your CAS engine libref is named mycas, but you can substitute any appropriately defined CAS engine libref.

The following statements execute the GVARCLUS algorithm on the mycas.getStarted data table:

```
proc gvarclus data=mycas.getStarted mincluster=2 maxsteps=8
              rho=0.8 outtree=mycas.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 $[0.8^1, 0.8^2, 0.8^3, ...]$ that are used in each step sequentially. The OUTTREE= option creates the mycas.dendrogram table to contain information about the tree structure of hierarchical clustering.

The output from this analysis is presented in Figure 12.1 and Figure 12.2.

Figure 12.1 displays the "Number of Observations" tables. This table shows that all 100 observations in the data table are used in the analysis.

Figure 12.1 Number of Observations

The GVARCLUS Procedure

Number of Observations Read	100
Number of Observations Used	100

Figure 12.2 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 12.2 Cluster Summary

Cluster Summary

Parameter	Step 1	Step 2	Step 3	Step 4	Step 5	Step 6	Step 7	Step 8
x1	1	19	36	52	68	82	96	108
x2	2	19	36	52	68	82	96	108
x3	3	20	36	52	68	82	96	108
x4	4	21	37	53	69	83	96	108
x5	5	22	38	54	68	82	96	108
x6	6	23	39	55	70	84	97	109
x7	6	23	39	55	70	84	97	109
x8	6	23	39	55	70	84	97	109
x9	7	24	40	56	71	85	97	109
x10	8	25	41	57	70	84	97	109
x11	9	26	42	58	72	86	98	110
x12	10	27	43	59	73	87	99	111
x13	11	28	44	60	74	88	100	112
x14	12	29	45	61	75	89	101	113
x15	13	30	46	62	76	90	102	114
x16	14	31	47	63	77	91	103	115
x17	15	32	48	64	78	92	104	116
x18	16	33	49	65	79	93	105	117
x19	17	34	50	66	80	94	106	118
x20	18	35	51	67	81	95	107	119

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 mycas.dendrogram into the Work library:

```
data outtree;
    set mycas.dendrogram;
run;
```

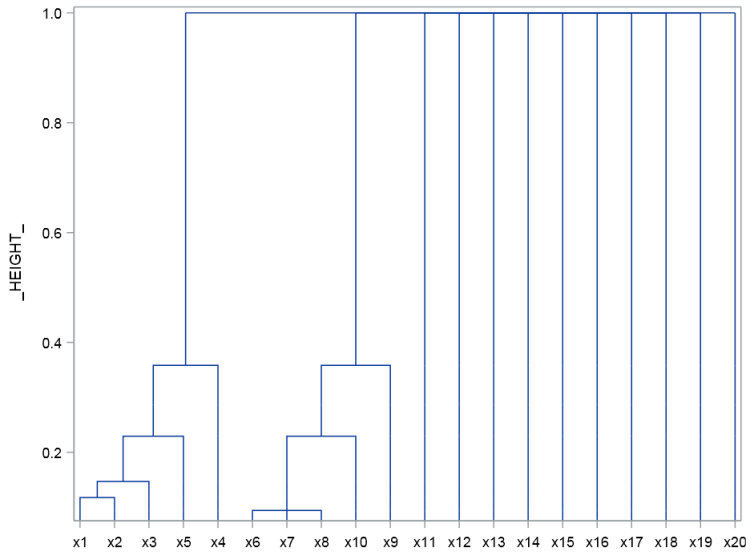
The following code defines the parent node, child node and the height in the dendrogram through the DENDROGRAM statement in GTL. The dendrogram is then plotted using the SGRENDER procedure.

```
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 12.3 shows the dendrogram.

Figure 12.3 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 IPUT 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 12.1 summarizes the important options in the PROC GVARCLUS statement by function. The options are then described fully in alphabetical order.

Table 12.1 PROC GVARCLUS Statement Options

Option	Description
Basic Options	
DATA=	Specifies the input data table
Options Related to Variable Clustering	
MINCLUSTER=	Specifies the minimum number of clusters
MAXSTEP=	Specifies the maximum number of clustering steps
MAXMEMBER=	Specifies the maximum number of members within each final cluster
MAXITER=	Specifies the maximum number of iterations for each estimation step
RHO=	Specifies the initial value of the regularization parameter
TOL=	Specifies the threshold of convergence
EXACT	Performs graphical variable clustering without preprocessing
STOP=	Requests that the clustering process stop if the clustering structure does not change in the previous number of specified steps
Options for Output Tables	
OUTCP=	Outputs the covariance matrix from the last step
OUTTREE=	Outputs the hierarchical clustering results in a tree structure
OUTEDGE=	Outputs the edges for defining an undirected network
OUTVERT=	Outputs the vertices and their size for defining an undirected network

You can specify the following *options*:

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

CAS-libref refers to a collection of information that is defined in the LIBNAME statement and includes the caslib, which includes a path to the data, and a session identifier, which defaults to the active session but which can be explicitly defined in the LIBNAME statement. For more information about *CAS-libref*, see the section "Using CAS Sessions and CAS Engine Librefs" on page 203.

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=*CAS-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. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 203.

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=*CAS-libref.data-table*
creates a data set for use with the hypergroup action. This table contains the information that defines the edges in the network, _FROM_, _TO_, and _WEIGHT_. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 203.

OUTTREE=*CAS-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. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 203.

OUTVERT=*CAS-libref.data-table*
creates a data set for use with the hypergroup action. This table contains the vertices in the network and their size. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 203.

RHO=*fraction*
specifies the value of ρ that determines the sequence of regularization parameters, $\{\rho^1, \rho^2, \rho^3, ...\}$, 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.

STOP=*n*
requests that the action 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 procedure processes the DISPLAY statement on a CAS server and thus sends only a subset of ODS tables to the SAS client. Because ODS statements are processed on a SAS client, first all the generated display tables are sent to the client, and then the client creates a subset.

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 that are sent to the client 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 214. 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 CAS 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 CAS 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 CAS output table name.

key specifies *key* as the ODS table name and also as the CAS output table name.

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 CAS tables for all display tables. The name of the created output CAS table is the same as the corresponding display table name. If you specify this option, the *table-spec-list* specification is ignored.

NOREPLACE
does not replace any existing CAS output table of the same name.

REPEATED
replicates all CAS output tables on all nodes.

FREQ Statement

FREQ *variable* **;**

The *variable* in the FREQ statement identifies a numeric variable in the data set 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 the FREQ statement is not specified, 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 $j = 1, 2, ...p, ...$, 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.

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 12.2.

Table 12.2 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
ClassInfo	Level information for classification variables	PROC GVARCLUS	Default

Examples: GVARCLUS Procedure

NOTE: Input data must be in a CAS table that is accessible in your CAS session. You must refer to this table by using a two-level name. The first level must be a CAS engine libref, and the second level must be the table name. For more information, see the sections "Using CAS Sessions and CAS Engine Librefs" on page 11 and "Loading a SAS Data Set onto a CAS Server" on page 12 in Chapter 3, "Shared Concepts."

Example 12.1: 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 load the Work.Senate data set into your CAS session by naming your CAS engine libref in the first statement of the following DATA step:

```
data mycas.senate;
    set work.senate;
run;
```

These statements assume that your CAS engine libref is named mycas, but you can substitute any appropriately defined CAS engine libref.

The following statements invoke the GVARCLUS procedure:

```
proc gvarclus data=mycas.senate rho=0.8 maxstep=5 outtree=mycas.tree
              outedge=mycas.edge outvert=mycas.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 CAS 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 CAS table named mycas.edge, and the OUTVERT= option saves the vertices and their sizes in the network to CAS table named mycas.vert.

The following statements downloads the tree table to the Work library:

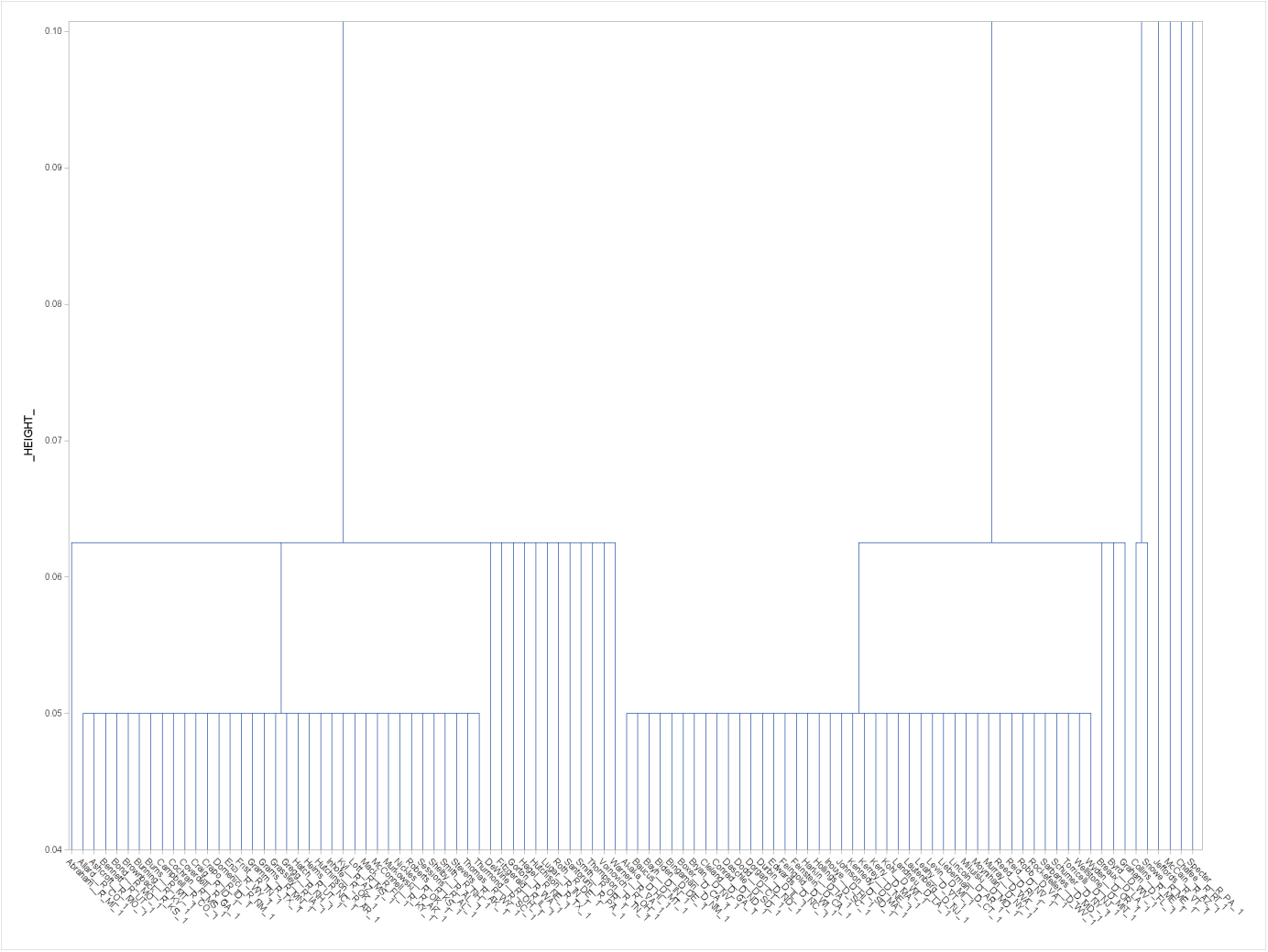
```
data work.tree (drop = last);
    set mycas.tree;
    last = scan(_CHILD_, -1);
    if last=0   then delete;
run;
```

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.tree1 template=dendrogram;
run;
```

Output 12.1.1 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 12.1.1 Senate Voting Records from the 106th Congress



Example 12.2: 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 mycas.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 mycas.data1;
    array x{2};
    array c{2};
    do i=1 to 2000;
        a=int(ranuni(1)*2);
        do j=1 to 2;
            x{j}=ranuni(1);
            c{j}=int(ranuni(1)*2);
       end;
       output;
    end;
run;
```

These statements assume that your CAS engine libref is named mycas, but you can substitute any appropriately defined CAS engine libref.

The following statements invoke the GVARCLUS procedure:

```
title  "Output the Covariance Matrix";

proc gvarclus data=mycas.data1 maxiter=4 outcp=mycas.cov;
    input c1 c2 /level=nominal;
    input x1 x2 /level=interval;
run;

proc print data=mycas.cov;
run;
```

Output 12.2.1 shows the content of the data file that PROC GVARCLUS generates.

Output 12.2.1 Output the Covariance Matrix

Output the Covariance Matrix

Obs	_ID_	_TYPE_	_VAR_	_vID_	v1	v2	v3	v4	v5	v6
1	1	MEAN/FREQ			0.50	0.50	999.00	1001.00	975.00	1025.00
2	2	N			2000.00	2000.00	2000.00	2000.00	2000.00	2000.00
3	3	COV	x1	v1	165.55	-1.49	0.32	-0.32	5.24	-5.24
4	4	COV	x2	v2	-1.49	166.41	1.31	-1.31	1.32	-1.32
5	5	COV	c1 0	v3	0.32	1.31	500.00	-500.00	2.99	-2.99
6	6	COV	c1 1	v4	-0.32	-1.31	-500.00	500.00	-2.99	2.99
7	7	COV	c2 0	v5	5.24	1.32	2.99	-2.99	499.69	-499.69
8	8	COV	c2 1	v6	-5.24	-1.32	-2.99	2.99	-499.69	499.69

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 12.3: 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 mycas.data2;
    array x{2};
    do i=1 to 2000;
        a=int(ranuni(1)*2);
        do j=1 to 2;
            x{j}=ranuni(1);
        end;
        output;
    end;
run;
```

These statements assume that your CAS engine libref is named mycas, but you can substitute any appropriately defined CAS engine libref.

The following statements invoke the GVARCLUS procedure:

```
title  "Output the Covariance Matrix in LIL Format";

proc gvarclus data=mycas.data2 maxiter=4 outcp=mycas.cov_lil/list eps=0.01;
    input a /level=nominal;
    input x1-x2 /level=interval;
run;

proc print data=mycas.cov_lil;
run;
```

The OUTCP= option creates an output data table named mycas.cov_lil, which contains the covariance matrix. Output 12.3.1 shows the correlation matrix in LIL format.

Output 12.3.1 Output the Correlation Matrix in LIL Format

Output the 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		0.50
6	F	3	a 0		979.00
7	F	4	a 1		1021.00
8	V	1	1	1	168.27
9	V	2	2	1	0.58
10	V	3	2	2	166.44
11	V	4	3	1	8.37
12	V	5	3	2	0.46
13	V	6	3	3	499.78
14	V	7	4	1	-8.37
15	V	8	4	2	-0.46
16	V	9	4	3	-499.78
17	V	10	4	4	499.78

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 13
The MBANALYSIS Procedure

Contents

Overview: MBANALYSIS Procedure **221**
PROC MBANALYSIS Features 222
Using CAS Sessions and CAS Engine Librefs 222
Getting Started: MBANALYSIS Procedure **223**
Syntax: MBANALYSIS Procedure **223**
PROC MBANALYSIS Statement 224
CUSTOMER statement 225
HIERARCHY Statement 225
OUTPUT Statement 226
SAVESTATE Statement 226
TARGET Statement 227
Details: MBANALYSIS Procedure **227**
FP-Growth Algorithm 228
Scoring with PROC ASTORE 229
Displayed Output 230
Results 230
Limitations 230
Examples: MBANALYSIS Procedure **231**
Example 13.1: Training a Model 231
References **232**

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 228.

PROC MBANALYSIS Features

The MBANALYSIS procedure has the following features:

- reads input data in parallel when the data source is on a distributed system
- processes data in a single thread on a single node or on multiple nodes
- 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 CAS Sessions and CAS Engine Librefs

SAS Cloud Analytic Services (CAS) is the analytic server and associated cloud services in SAS Viya. This section describes how to create a CAS session and set up a CAS engine libref that you can use to connect to the CAS session. It assumes that you have a CAS server already available; contact your system administrator if you need help starting and terminating a server. This CAS server is identified by specifying the host on which it runs and the port on which it listens for communications. To simplify your interactions with this CAS server, the host information and port information for the server are stored as SAS option values that are retrieved automatically whenever this CAS server needs to be accessed. You can examine the host and port values for the server at your site by using the following statements:

```
proc options option=(CASHOST CASPORT);
run;
```

In addition to starting a CAS server, your system administrator might also have created a CAS session and a CAS engine libref for your use. You can define your own sessions and CAS engine librefs that connect to the CAS server as shown in the following statements:

```
cas mysess;
libname mycas cas sessref=mysess;
```

The CAS statement creates the CAS session named mysess, and the LIBNAME statement creates the mycas CAS engine libref that you use to connect to this session. It is not necessary to explicitly name the CASHOST and CASPORT of the CAS server in the CAS statement, because these values are retrieved from the corresponding SAS option values.

If you have created the mysess session, you can terminate it by using the TERMINATE option in the CAS statement as follows:

```
cas mysess terminate;
```

For more information about the CAS and LIBNAME statements, see the section "Introduction to Shared Concepts" on page 11 in Chapter 3, "Shared Concepts."

Getting Started: MBANALYSIS Procedure

NOTE: Input data must be in a CAS table that is accessible in your CAS session. You must refer to this table by using a two-level name. The first level must be a CAS engine libref, and the second level must be the table name. For more information, see the sections "Using CAS Sessions and CAS Engine Librefs" on page 11 and "Loading a SAS Data Set onto a CAS Server" on page 12 in Chapter 3, "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 CAS session by specifying your CAS engine libref in the first statement in the following DATA step:

```
data mycas.assocs;
   set sampsio.assocs;
run;
```

These statements assume that your CAS engine libref is named mycas, as in the section "Using CAS Sessions and CAS Engine Librefs" on page 222, but you can substitute any appropriately defined CAS engine libref.

The following statements execute the MBANALYSIS procedure on the mycas.assocs data table and generate output in the mycas.assocs data table:

```
proc mbanalysis data=mycas.assocs items=3 support=100;
 output out=mycas.out outfreq=mycas.outfreq outrule=mycas.outrule;
 customer Customer;
 target product;
 run;
```

The PROC MBANALYSIS statement specifies mycas.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 mycas.outfreq table, the frequent item sets in the mycas.out table, and the generated rules in the mycas.outrule table. The mycas.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 *variables* **;**
HIERARCHY DATA=*CAS-libref.data-table* **<** *CAS-libref.data-table...* **> ;**
OUTPUT *options* **;**
SAVESTATE RSTORE=*CAS-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=*CAS-libref.data-table*
names the input data table for PROC MBANALYSIS to use. *CAS-libref.data-table* is a two-level name, where

CAS-libref refers to a collection of information that is defined in the LIBNAME statement and includes the caslib, which includes a path to the data, and a session identifier, which defaults to the active session but which can be explicitly defined in the LIBNAME statement. For more information about *CAS-libref*, see the section "Using CAS Sessions and CAS Engine Librefs" on page 222.

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 0. 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*:

CONF=*number*
specifies the minimum confidence for the rules, where *number* must be a real number between 0 and 100.

By default, CONF=50

ITEMS=*number*
specifies the number of items in a rule, where *number* must be an integer between 1 and 100. By default, ITEMS=2 when either an OUT= or OUTRULE= option is specified 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 100, 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.

NORM=TRUE | FALSE
specifies whether to normalize the values of the items in the output tables which are specified in the OUTPUT statement.

By default, NORM=FALSE.

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* **;**

CUSTOMER statement groups one or more specified target *variables* into a basket. You can also specify this statement as ID *variables*.

HIERARCHY Statement

HIERARCHY DATA=*CAS-libref.data-table* < *CAS-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=*CAS-libref.data-table < CAS-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. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 222.

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=*CAS-libref.data-table*
creates a data table that contains information about frequent item sets with their transaction counts and support. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 222.

OUTFREQ=*CAS-libref.data-table*
creates a data table that contains information about the unique frequent items with their transaction counts and support. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 222.

OUTRULE=*CAS-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. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 222.

SAVESTATE Statement

SAVESTATE RSTORE=*CAS-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, "The ASTORE Procedure".

You must specify the following option:

RSTORE=*CAS-libref.data-table*

specifies a data table in which to save the analytic store for the model. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 222.

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.

The following statements provide the mycas.assocs data table as input to the ASTORE procedure. The SETOPTION ARM_SCORING_ALGO statement in PROC ASTORE specifies the algorithm to be used for scoring.

```
proc astore;
 setoption ARM_SCORING_ALGO 1;
 score data=mycas.assocs out=mycas.armScore rstore=mycas.mystateArm;
 run;
```

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 13.1.

Table 13.1 Results Tables Produced by the MBANALYSIS Procedure

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
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 hangs.

When one item has multiple parents, only the last parent is considered for generating rules. The last parent item might vary if the data are distributed.

The maximum number of rules that are generated on each node is one million.

Examples: MBANALYSIS Procedure

NOTE: Input data must be in a CAS table that is accessible in your CAS session. You must refer to this table by using a two-level name. The first level must be a CAS engine libref, and the second level must be the table name. For more information, see the sections "Using CAS Sessions and CAS Engine Librefs" on page 11 and "Loading a SAS Data Set onto a CAS Server" on page 12 in Chapter 3, "Shared Concepts."

Example 13.1: Training a Model

NOTE: Input data must be in a CAS table that is accessible in your CAS session. You must refer to this table by using a two-level name. The first level must be a CAS engine libref, and the second level must be the table name. For more information, see the sections "Using CAS Sessions and CAS Engine Librefs" on page 11 and "Loading a SAS Data Set onto a CAS Server" on page 12 in Chapter 3, "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 CAS session by specifying your CAS engine libref in the first statement in the following DATA step. These statements assume that your CAS engine libref is named mycas, but you can substitute any appropriately defined CAS engine libref.

```
data mycas.assocs;
   set sampsio.assocs;
run;
```

The following DATA steps create two hierarchies, mycas.firstLevel and mycas.secondLevel:

```
data mycas.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 mycas.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 mycas.assocs data table:

```
proc mbanalysis data=mycas.assocs items=5 support=200;
   output out=mycas.out outfreq=mycas.outfreq outrule=mycas.outrule;
   customer Customer;
   target product;
   hierarchy data = mycas.firstLevel mycas.secondLevel;
   savestate rstore=mycas.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 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*. New York: ACM Press. http://crono.dei.unipd.it/~lds/MATERIALE/AgrawalIS93.pdf.

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 14
The MWPCA Procedure

Contents

Overview: MWPCA Procedure **235**
PROC MWPCA Features 236
Using CAS Sessions and CAS Engine Librefs 236
Getting Started: MWPCA Procedure **237**
Syntax: MWPCA Procedure **240**
PROC MWPCA Statement 241
DISPLAY Statement 242
DISPLAYOUT Statement 243
ID Statement 243
INPUT Statement 244
OUTPUT Statement 244
RPCA Statement 245
SVD Statement 246
Details: MWPCA Procedure **246**
Displayed Output 247
ODS Table Names 247
References **248**

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:

- 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 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

Using CAS Sessions and CAS Engine Librefs

SAS Cloud Analytic Services (CAS) is the analytic server and associated cloud services in SAS Viya. This section describes how to create a CAS session and set up a CAS engine libref that you can use to connect to the CAS session. It assumes that you have a CAS server already available; contact your system administrator if you need help starting and terminating a server. This CAS server is identified by specifying the host on which it runs and the port on which it listens for communications. To simplify your interactions with this CAS server, the host information and port information for the server are stored as SAS option values that are retrieved automatically whenever this CAS server needs to be accessed. You can examine the host and port values for the server at your site by using the following statements:

```
proc options option=(CASHOST CASPORT);
run;
```

In addition to starting a CAS server, your system administrator might also have created a CAS session and a CAS engine libref for your use. You can define your own sessions and CAS engine librefs that connect to the CAS server as shown in the following statements:

```
cas mysess;
libname mycas cas sessref=mysess;
```

The CAS statement creates the CAS session named mysess, and the LIBNAME statement creates the mycas CAS engine libref that you use to connect to this session. It is not necessary to explicitly name the CASHOST and CASPORT of the CAS server in the CAS statement, because these values are retrieved from the corresponding SAS option values.

If you have created the mysess session, you can terminate it by using the TERMINATE option in the CAS statement as follows:

```
cas mysess terminate;
```

For more information about the CAS and LIBNAME statements, see the section “Introduction to Shared Concepts” on page 11 in Chapter 3, “Shared Concepts.”

Getting Started: MWPCA Procedure

NOTE: Input data must be in a CAS table that is accessible in your CAS session. You must refer to this table by using a two-level name. The first level must be a CAS engine libref, and the second level must be the table name. For more information, see the sections "Using CAS Sessions and CAS Engine Librefs" on page 11 and "Loading a SAS Data Set onto a CAS Server" on page 12 in Chapter 3, "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.

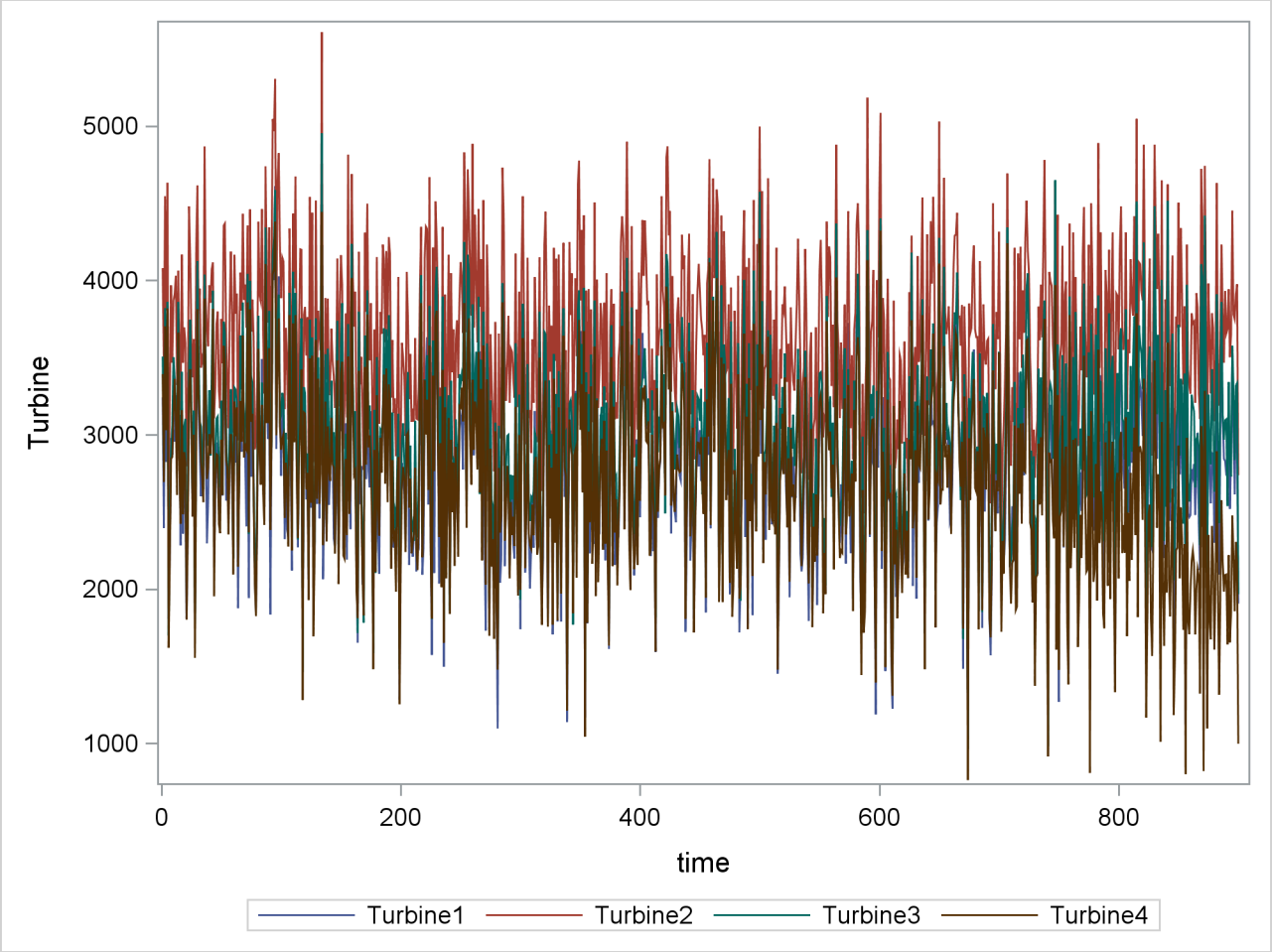
The following DATA step creates the Turbines data set:

```
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 14.1 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 14.1 Energy Generated (Kilowatts)



You can load the Turbines data set into your CAS session by specifying your CAS engine libref in the following DATA step. These statements assume that your CAS engine libref is named mycas, but you can substitute any appropriately defined CAS engine libref.

```
data mycas.turbines;
set turbines;
run;
```

The following statements run PROC MWPCA:

```
proc mwpca data=mycas.turbines windowsize=200 stepsize=1;
id time;
output out=mycas.windowpcs standardpc;
run;
```

Note that the STANDARDPC option is used in the OUTPUT statement. Using this option is recommended.

Figure 14.2 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 mycas.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 RPCA/PCA algorithm ran successfully for all windows.

Figure 14.2 ODS Tables

Model Information	
Data Source	TURBINES
Window Size	200
Step Size	1
RPCA Method	Augmented Lagrange Multiplier
Lambda	0.0707107
Lambda Weight	1
SVD Method	Eigenvalue Decomposition

Number of Observations	900
Number of Variables	4

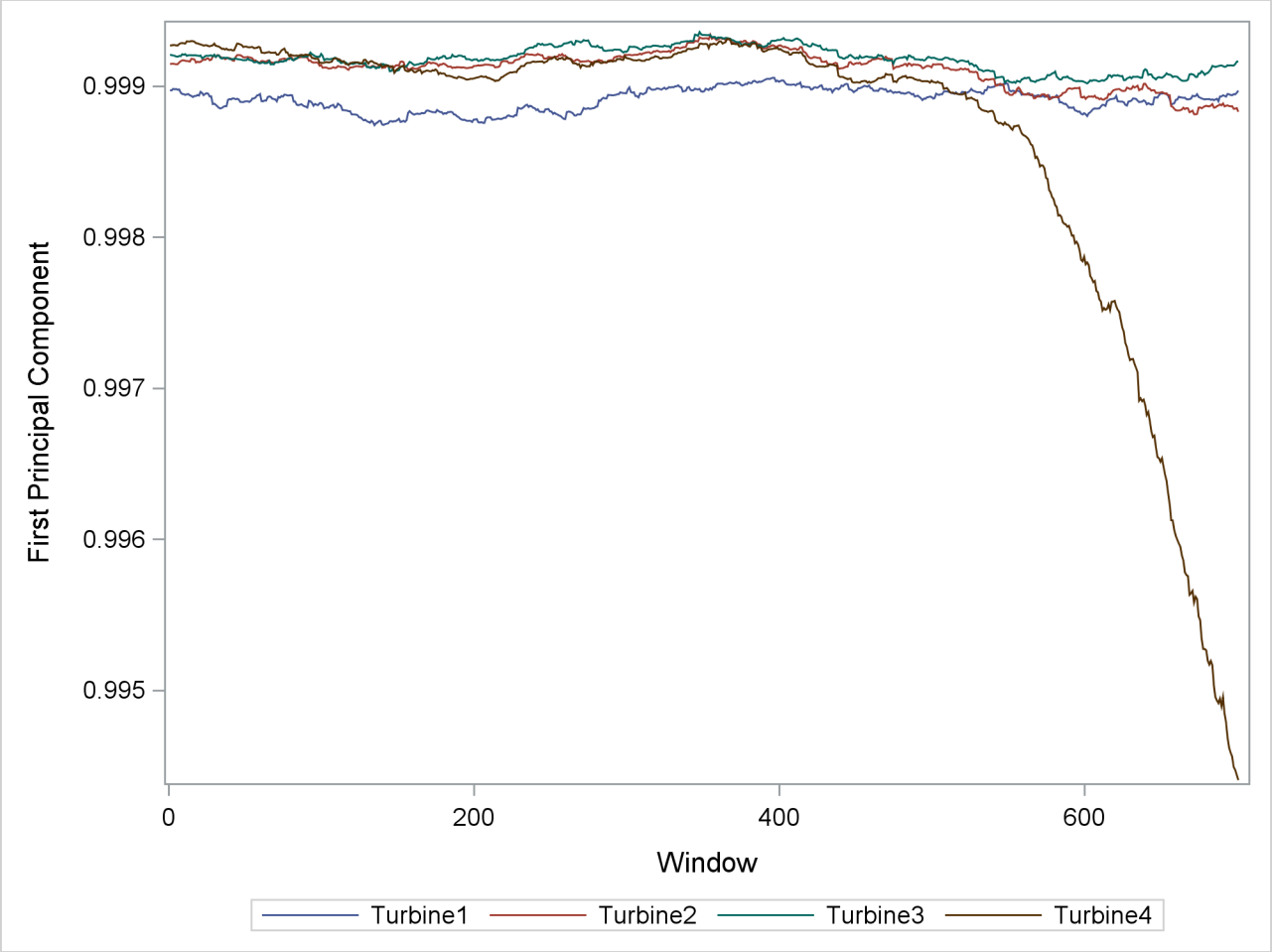
Results Summary	
Number of Windows	701
Solution Status	Optimal
Run Time (Seconds)	0.52

You can see the output table windowpcs in your mycas CAS engine 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. Output 14.3 shows the resulting plot.

```
proc sgplot data =mycas.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 Output 14.3, 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 14.3 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=*CAS-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 14.1 summarizes the *options* available in the PROC MWPCA statement.

Table 14.1 other Options

Option	Description
Input Data Table Option	
DATA=	Specifies the input data table
Other Options	
CENTER	Centers the input data
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.

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

CAS-libref refers to a collection of information that is defined in the LIBNAME statement and includes the caslib, which includes a path to the data, and a session identifier, which defaults to the active session but which can be explicitly defined in the LIBNAME statement. For more information about *CAS-libref*, see the section "Using CAS Sessions and CAS Engine Librefs" on page 236.

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 16, "The RPCA Procedure." If you do not specify this option, the regular PCA method is used on each window.

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 procedure processes the DISPLAY statement on a CAS server and thus sends only a subset of ODS tables to the SAS client. Because ODS statements are processed on a SAS client, first all the generated display tables are sent to the client, and then the client creates a subset.

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 that are sent to the client 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 247. 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 CAS 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 CAS 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 CAS output table name.
key	specifies *key* as the ODS table name and also as the CAS output table name.

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 CAS tables for all display tables. The name of the created output CAS table is the same as the corresponding display table name. If you specify this option, the *table-spec-list* specification is ignored.

NOREPLACE
does not replace any existing CAS output table of the same name.

REPEATED
replicates all CAS output tables on all nodes.

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=*CAS-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=*CAS-libref.data-table*
names the output data table for PROC MWPCA to use. You must specify this option before any other options. *CAS-libref.data-table* is a two-level name, where

CAS-libref refers to a collection of information that is defined in the LIBNAME statement and includes the caslib, which includes a path to where the data table is to be stored, and a session identifier, which defaults to the active session but which can be explicitly defined in the LIBNAME statement. For more information about *CAS-libref*, see the section "Using CAS Sessions and CAS Engine Librefs" on page 236.

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

You can also specify the following *options*:

NPC=*number*
specifies the number of principal components to display for each window in the output table.

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

$$P_{ij}^S = \frac{P_{ij} \times \sigma_i}{(W^T W)_{jj}}$$

where σ_i is the ith largest eigenvalue of matrix W.

WINSUMMARY=*CAS-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, and solution status.

CAS-libref.data-table is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 236.

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 input variables in the input data set. The value of λ 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 16, "The 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 294.

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*:

METHOD=EIGEN | RANDOM
specifies the type of the SVD solver. You can specify the following values:

EIGEN uses the eigenvalue decomposition method.

RANDOM uses the iterative SVD method.

MAXRANK=*number*
specifies the maximum value for rank considered by the SVD solver. By default, *number* is the number of input variables.

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.

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 Not Successful in One or More Windows."

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 14.2.

Table 14.2 ODS Tables Produced by PROC MWPCA

Table Name	Description	Statement	Option
ModelInfo	Model information	PROC MWPCA	Default
Dimensions	Dimensions of the input table	PROC MWPCA	Default
Summary	Summary of the results	PROC MWPCA	Default

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.

Chapter 15
The NNET Procedure

Contents

Overview: NNET Procedure **250**
PROC NNET Features 250
Using CAS Sessions and CAS Engine Librefs 251
Getting Started: NNET Procedure **251**
Syntax: NNET Procedure **253**
PROC NNET Statement 253
ARCHITECTURE Statement 255
AUTOTUNE Statement 256
CODE Statement 260
CROSSVALIDATION Statement 261
HIDDEN Statement 261
INPUT Statement 262
OPTIMIZATION Statement 262
OUTPUT Statement 264
PARTITION Statement 265
TARGET Statement 266
TRAIN Statement 267
WEIGHT Statement 269
Details: NNET Procedure **269**
Computational Method 269
Hyperparameter Tuning 270
k-fold Cross Validation 271
Autoencoder 271
Displayed Output 271
Iteration History 271
Convergence Status 271
Model Information 271
Score Information 272
Tuner Information 272
Tuner Results 272
Evaluation History 272
Best Configuration 272
Tuner Summary 272
TunerTiming 272
ROCInfo 272
FitStat 272

Cross Validation Results 272
PredProbName 272
PredIntoName 273
PredName 273
ODS Table Names 273
Examples: NNET Procedure **274**
Example 15.1: Binary Target Classification with Partition 275
Example 15.2: Finding the Best Neural Network Configuration 277
References **279**

Overview: NNET Procedure

The NNET procedure trains a multilayer perceptron neural network in SAS Viya. 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. Thus, even problems with smaller numbers of variables and training observations can benefit from the use of distributed mode.

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:

- ability to train and score using distributed mode
- parallel reading of input data and parallel writing of output data
- high degree of 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
- automatic searching for best hidden layers and key parameters such as L1 and L2 regularization norm, learning rate, and so on
- *k*-fold cross validation to estimate average validation error

Using CAS Sessions and CAS Engine Librefs

SAS Cloud Analytic Services (CAS) is the analytic server and associated cloud services in SAS Viya. This section describes how to create a CAS session and set up a CAS engine libref that you can use to connect to the CAS session. It assumes that you have a CAS server already available; contact your system administrator if you need help starting and terminating a server. This CAS server is identified by specifying the host on which it runs and the port on which it listens for communications. To simplify your interactions with this CAS server, the host information and port information for the server are stored as SAS option values that are retrieved automatically whenever this CAS server needs to be accessed. You can examine the host and port values for the server at your site by using the following statements:

```
proc options option=(CASHOST CASPORT);
run;
```

In addition to starting a CAS server, your system administrator might also have created a CAS session and a CAS engine libref for your use. You can define your own sessions and CAS engine librefs that connect to the CAS server as shown in the following statements:

```
cas mysess;
libname mycas cas sessref=mysess;
```

The CAS statement creates the CAS session named mysess, and the LIBNAME statement creates the mycas CAS engine libref that you use to connect to this session. It is not necessary to explicitly name the CASHOST and CASPORT of the CAS server in the CAS statement, because these values are retrieved from the corresponding SAS option values.

If you have created the mysess session, you can terminate it by using the TERMINATE option in the CAS statement as follows:

```
cas mysess terminate;
```

For more information about the CAS and LIBNAME statements, see the section "Introduction to Shared Concepts" on page 11 in Chapter 3, "Shared Concepts."

Getting Started: NNET Procedure

NOTE: Input data must be in a CAS table that is accessible in your CAS session. You must refer to this table by using a two-level name. The first level must be a CAS engine libref, and the second level must be the table name. For more information, see the sections "Using CAS Sessions and CAS Engine Librefs" on page 11 and "Loading a SAS Data Set onto a CAS Server" on page 12 in Chapter 3, "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 CAS session by naming your CAS engine libref in the first statement of the following DATA step:

```
data mycas.iris;
   set sashelp.iris;
run;
```

These statements assume that your CAS engine libref is named mycas, but you can substitute any appropriately defined CAS engine libref.

The following statements run PROC NNET and output the results to ODS tables:

```
proc nnet data=mycas.iris;
   input SepalLength SepalWidth PetalLength PetalWidth;
   target Species / level=nominal;
   hidden 2;
   train outmodel=mycas.nnetModel_gs seed=635117188;
   partition fraction(validate=0.3 seed=103873735);
run;
```

Figure 15.1 shows the model information for the neural network.

Figure 15.1 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 15.2 shows the misclassification rate for the training sample.

Figure 15.2 Score Information for Training Data

Score Information for Training	
Number of Observations Read	93
Number of Observations Used	93
Misclassification Rate	0.0215

Figure 15.3 shows the misclassification rate for the validation sample.

Figure 15.3 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* > **;**
INPUT *variables* < / **LEVEL=INTERVAL** | **NOMINAL** > **;**
HIDDEN *number* < */options* > **;**
TARGET *variables* < */options* > **;**
TRAIN OUTMODEL=*CAS-libref.data-table* < *options* > **;**
ARCHITECTURE *architecture-options* **;**
WEIGHT *variable* **;**
PARTITION < *partition-options* > **;**
OPTIMIZATION < *options* > **;**
AUTOTUNE < *options* > **;**
CROSSVALIDATION < **KFOLD**=*number* > **;**
OUTPUT OUT=*CAS-libref.data-table* < *option* > **;**
CODE < *options* > **;**

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

CAS-libref refers to a collection of information that is defined in the LIBNAME statement and includes the caslib, which includes a path to the data, and a session identifier, which defaults to the active session but which can be explicitly defined in the LIBNAME statement. For more information about *CAS-libref*, see the section "Using CAS Sessions and CAS Engine Librefs" on page 251.

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

NOTE: The data set options WHERE, TEMPNAMES, and TEMPEXPRESS are not supported.

INMODEL=*CAS-libref.data-table*
specifies a model for stand-alone scoring or coding. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 251.

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 15.3. This option is valid when TARGET statement is specified with OUTPUT statement or CODE statement.

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

TARGETSTDE=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 271):

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 suggested by the tuning optimization. As a result, it might use a different architecture from the one 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, 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 USEPARAMETERS= and TUNINGPARAMETERS= options. You cannot specify both the AUTOTUNE statement and the CROSSVALIDATION statement in the same procedure run.

Table 15.1 summarizes the *options* you can specify in the AUTOTUNE statement. For more information about all options except the TUNINGPARAMETERS= option, see the section "AUTOTUNE Statement" on page 14 in Chapter 3, "Shared Concepts." The TUNINGPARAMETERS= option is described in detail following Table 15.1.

Table 15.1 AUTOTUNE Statement Options

Option	Description
EVALHISTORY=	Specifies how to report the evaluation history of the tuner
FRACTION=	Specifies the fraction of observations to use for validation
KFOLD=	Specifies the number of folds for *k*-fold cross validation
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
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
TARGETEVENT=	Specifies the target event for ROC-based calculations
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 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 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 < **KFOLD**=*number*> **;**

The CROSSVALIDATION statement performs a *k*-fold cross validation process to find the average estimated validation error. You cannot specify the CROSSVALIDATION statement if you specify either the AUTOTUNE statement or the PARTITION statement.

You can specify the following option:

KFOLD=*number*
specifies the *number* of partition folds in the cross validation process, where *number* must be between 2 and 20, inclusive.

By default, KFOLD=5.

HIDDEN Statement

HIDDEN *number* < */options*> **;**

The HIDDEN statement specifies the number of neurons or units in a hidden layer, which must be a positive integer. You can specify multiple HIDDEN statements; each HIDDEN statement represents a hidden layer.

You can 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.

HIDDEN statements are ignored when you specify the AUTOTUNE statement.

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.

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=LBFGS | SGD < *sgd-options* > | HF

ALG=LBFGS | SGD < *sgd-options* > | HF

specifies the optimization algorithm to use during training. You can specify the following optimization algorithms:

HF (*Experimental*) specifies 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 specifies the limited-memory Broyden-Fletcher-Goldfarb-Shanno algorithm.

SGD specifies the stochastic gradient descent algorithm.

When ALGORITHM=SGD, 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 SGD progresses, causing smaller steps as SGD 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 SGD 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 SGD. New iterates for SGD 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 with SGD, especially with small data set, it is likely that the learning rate is set too large.

By default, LEARNINGRATE=0.001. The *number* must be a nonnegative double.

MINIBATCHSIZE=*number*

specifies the size of the minibatches used in SGD.

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 SGD algorithm. If *number* is less than or equal to 0 or not specified, a random seed from the computer clock is used.

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. For LBFGS, the algorithm stops after MAXITER= iterations if convergence has not been achieved. For SGD, *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=*CAS-libref.data-table* <*option*> **;**

SCORE OUT=*CAS-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=*CAS-libref.data-table*
names the output data table for PROC NNET to use. You must specify this option before any other options. *CAS-libref.data-table* is a two-level name, where

CAS-libref refers to a collection of information that is defined in the LIBNAME statement and includes the caslib, which includes a path to where the data table is to be stored, and a session identifier, which defaults to the active session but which can be explicitly defined in the LIBNAME statement. For more information about *CAS-libref*, see the section "Using CAS Sessions and CAS Engine Librefs" on page 251.

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 15.2 shows how this *variable* is interpreted for each observation.

Table 15.2 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.

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 271.

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=*CAS-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=*CAS-libref.data-table*
specifies the final model from training. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 251.

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 ALGORITHM=SGD and 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 ALGORITHM=SGD and 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=*CAS-libref.data-table*
specifies a separate data table for validation during training. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 251.

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 from the computer 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), and 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 22 in Chapter 3, "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 optimizer when you specify ALGORITHM=SGD in the OPTIMIZATION statement
- LEARNINGRATE= option for the learning rate parameter, which is used by the training algorithm of the SGD optimizer when you specify ALGORITHM=SGD in the OPTIMIZATION statement

k-fold Cross Validation

The CROSSVALIDATION statement uses a *k*-fold cross validation process to find the average estimated validation error (misclassification error for nominal targets or average square error for interval targets) for the trained model. During cross validation, all data are divided into *k* partitions (folds), where *k* is the value of the KFOLD= option. For each of the folds, a new model is trained on the (KFOLD–1) folds and then validated using the selected (holdout) fold. The validation error estimates are then averaged over each set of training and scoring executions to obtain a single value. The CROSSVALIDATION statement returns a table that contains a single data row that shows the average validation 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.

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.

Iteration History

The "Iteration History" table contains the iteration history from the optimization solver. This table is suppressed if the NUMTRIES option is greater than 1 in the TRAIN statement.

Convergence Status

The "Convergence Status" table contains the returning status of optimization solver. This table is suppressed if the NUMTRIES option is greater than 1 in the TRAIN statement.

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.

Tuner Information

The "Tuner Information" table contains the values of options used by tuner.

Tuner Results

The "Tuner Results" table contains the values of hyperparameters, objective function for the default configuration (Iteration 0), and up to 10 best configurations found from tuner.

Evaluation History

The "Evaluation Results" table contains values of all tuning parameters and the objective function for all model configurations evaluated by the tuner.

Best Configuration

The "Best Configuration" table contains the values of hyperparameters and objective function for the best configuration from tuner.

Tuner Summary

The "Tuner Summary" table contains statistics for the tuning process.

TunerTiming

The "TunerTiming" table contains the run time break down of different tasks during tuning.

ROCInfo

The "ROCInfo" table contains the ROC curve information for the specified event in the nominal target. For more information, see Chapter 17, "The ASSESS Procedure" (*SAS Visual Statistics: Procedures Guide*).

FitStat

The "FitStat" table contains various error metrics for supervised learning models. For more information, see Chapter 17, "The ASSESS Procedure" (*SAS Visual Statistics: Procedures Guide*).

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.

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 15.3.

Table 15.3 ODS Tables Produced by PROC NNET

Table Name	Description	Statement	Option
BestConfiguration	Hyperparameters and objective function values for the best configuration	AUTOTUNE	Default
ConvergenceStatus	Convergence status	PROC NNET	Default
CrossValidationResults	Average error rate (misclassification error or average square error) of *k*-fold cross validation	CROSSVALIDATION	Default
ModelInfo	Information about the modeling environment	PROC NNET	Default
OptIterHistory	Iteration history information	PROC NNET	Default
ScoreInfo	Misclassification rate or mean square error for partition sets	SCORE, PARTITION	VALIDATION=
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

Table 15.3 *continued*

Table Name	Description	Statement	Option
EvaluationHistory	Values of the hyperparameters, the objective function for the all model configurations evaluated by the tuner	AUTOTUNE	Default
TunerSummary	Statistics about the tuning process	AUTOTUNE	Default
TunerTiming	Total time spent on different tasks during tuning	AUTOTUNE	Default
ROCInfo	ROC curve information for the specified event in the nominal target	AUTOTUNE, TARGET	LEVEL=NOMINAL
FitStat	Error metrics for supervised learning models	AUTOTUNE, TARGET	Default
PredProbName	The name mapping between the score variables and the corresponding events in the nominal target	SCORE, CODE, TARGET	PRINTTARGET LEVEL=NOMINAL
PredIntoName	The name mapping between the predicted target variable and the original target variable when the target is nominal	SCORE, CODE, TARGET	PRINTTARGET LEVEL=NOMINAL
PredName	The name mapping between the predicted target variable and the original target variable when the target is interval	SCORE, CODE, TARGET	PRINTTARGET LEVEL=INTERVAL

Examples: NNET Procedure

NOTE: Input data must be in a CAS table that is accessible in your CAS session. You must refer to this table by using a two-level name. The first level must be a CAS engine libref, and the second level must be the table name. For more information, see the sections "Using CAS Sessions and CAS Engine Librefs" on page 11 and "Loading a SAS Data Set onto a CAS Server" on page 12 in Chapter 3, "Shared Concepts."

Example 15.1: 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 CAS session by naming a CAS engine libref in the first statement of each step:

```
data mycas.hmeq;
   set sampsio.hmeq;
run;
```

These statements assume that the CAS engine libref is named mycas, but you can substitute any appropriately defined CAS engine libref.

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=mycas.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=mycas.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 15.1.1 shows the model information for the neural network.

Output 15.1.1 Model Information

The NNET Procedure

Model Information	
Model	Neural Net
Number of Observations Used	4184
Number of Observations Read	4184
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	2.0429266604
Misclassification Rate for Validation	0.1831

Output 15.1.2 shows the misclassification rate of the training data.

Output 15.1.2 Score Information for Training Data

Score Information for Training	
Number of Observations Read	4184
Number of Observations Used	4184
Misclassification Rate	0.2029

Output 15.1.3 shows the misclassification rate of the validation data.

Output 15.1.3 Score Information for Validation Data

Score Information for Validation	
Number of Observations Read	1174
Number of Observations Used	1174
Misclassification Rate	0.1831

Output 15.1.4 shows the misclassification rate of the testing data.

Output 15.1.4 Score Information for Testing Data

Score Information for Testing	
Number of Observations Read	602
Number of Observations Used	602
Misclassification Rate	0.2076

Example 15.2: 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 as is used in the section "Getting Started: NNET Procedure" on page 251. 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 CAS session by naming your CAS engine libref in the first statement of the following DATA step. These statements assume that your CAS engine libref is named mycas, but you can substitute any appropriately defined CAS engine libref.

```
data mycas.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=mycas.iris;
   input SepalLength SepalWidth PetalLength PetalWidth;
   target Species / level=nominal;
   train outmodel=mycas.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 15.2.1 shows the setup values used by the tuner.

Output 15.2.1 Tuner Information

The NNET Procedure

Tuner Information	
Model Type	Neural Net
Tuner Objective Function	Misclassification
Search Method	GA
Maximum Evaluations	50
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

Output 15.2.2 shows the results reported by the NNET procedure. The first row displays results from the default settings, the second row displays the best results found by the tuner, and the third row displays the second-best results found.

Output 15.2.2 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	Time In Seconds
0	1	1	0	0.001000	0.001000	0.3111	1.48
3	1	7	0	0.010000	0.000775	0.0222	1.60
4	2	2	13	0.000100	0.001292	0.0222	1.76
8	1	8	0	0.000278	0.000278	0.0222	0.46
10	1	4	0	0.005995	0.000167	0.0222	0.49
19	1	8	0	0.000278	0.000279	0.0222	0.45
22	1	8	0	0.000266	0.000298	0.0222	0.54
28	1	8	0	0.001330	0.000344	0.0222	0.68
39	1	8	0	0.000987	0.000321	0.0222	0.60
17	1	8	0	0.001496	0.000450	0.0444	0.66
18	2	10	5	0.000774	0.000100	0.0444	0.63

Output 15.2.3 shows the best values of the tuning parameters from the tuning process.

Output 15.2.3 Best Configuration

Best Configuration	
Evaluation	3
Hidden Layers	1
Neurons in Hidden Layer 1	7
Neurons in Hidden Layer 2	0
L1 Regularization	0.01
L2 Regularization	0.0007746
Misclassification	0.0222222222

Output 15.2.4 shows the tuner summary.

Output 15.2.4 Tuner Summary

Tuner Summary	
Initial Configuration Objective Value	0.3111
Best Configuration Objective Value	0.02222
Worst Configuration Objective Value	0.3333
Initial Configuration Evaluation Time in Seconds	1.4818
Best Configuration Evaluation Time in Seconds	1.5961
Number of Improved Configurations	3
Number of Evaluated Configurations	46
Total Tuning Time in Seconds	17.1790
Parallel Tuning Speedup	2.0796

Output 15.2.5 shows the run time for each task during the searching process. It is evident that the tuner spent the vast majority of 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 15.2.5 Tuner Timing

Tuner Task Timing

Task	Seconds	Percent
Model Training	17.16	48.04
Model Scoring	11.34	31.75
Total Objective Evaluations	28.51	79.81
Tuner	7.21	20.19
Total CPU Time	35.73	100.00

References

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 16
The RPCA Procedure

Contents

Overview: RPCA Procedure **281**
PROC RPCA Features 282
Using CAS Sessions and CAS Engine Librefs 282
Getting Started: RPCA Procedure **283**
Syntax: RPCA Procedure **286**
PROC RPCA Statement 286
CODE Statement 289
DISPLAY Statement 290
DISPLAYOUT Statement 291
ID Statement 291
INPUT Statement 292
OUTDECOMP Statement 292
SAVESTATE Statement 293
SVD Statement 293
Details: RPCA Procedure **294**
Augmented Lagrange Multiplier Method 294
Accelerated Proximal Gradient Method 295
Scaling and Centering 295
Displayed Output 295
ODS Table Names 296
Output Data Tables 297
References **297**

Overview: RPCA Procedure

The RPCA procedure implements robust principal component analysis (RPCA) in SAS Viya. 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 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:

- 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 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

Using CAS Sessions and CAS Engine Librefs

SAS Cloud Analytic Services (CAS) is the analytic server and associated cloud services in SAS Viya. This section describes how to create a CAS session and set up a CAS engine libref that you can use to connect to the CAS session. It assumes that you have a CAS server already available; contact your system administrator if you need help starting and terminating a server. This CAS server is identified by specifying the host on which it runs and the port on which it listens for communications. To simplify your interactions with this CAS server, the host information and port information for the server are stored as SAS option values that are retrieved automatically whenever this CAS server needs to be accessed. You can examine the host and port values for the server at your site by using the following statements:

```
proc options option=(CASHOST CASPORT);
run;
```

In addition to starting a CAS server, your system administrator might also have created a CAS session and a CAS engine libref for your use. You can define your own sessions and CAS engine librefs that connect to the CAS server as shown in the following statements:

```
cas mysess;
libname mycas cas sessref=mysess;
```

The CAS statement creates the CAS session named mysess, and the LIBNAME statement creates the mycas CAS engine libref that you use to connect to this session. It is not necessary to explicitly name the CASHOST and CASPORT of the CAS server in the CAS statement, because these values are retrieved from the corresponding SAS option values.

If you have created the mysess session, you can terminate it by using the TERMINATE option in the CAS statement as follows:

```
cas mysess terminate;
```

For more information about the CAS and LIBNAME statements, see the section "Introduction to Shared Concepts" on page 11 in Chapter 3, "Shared Concepts."

Getting Started: RPCA Procedure

NOTE: Input data must be in a CAS table that is accessible in your CAS session. You must refer to this table by using a two-level name. The first level must be a CAS engine libref, and the second level must be the table name. For more information, see the sections "Using CAS Sessions and CAS Engine Librefs" on page 11 and "Loading a SAS Data Set onto a CAS Server" on page 12 in Chapter 3, "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 table 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 table:

```
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 CAS session by naming your CAS engine libref in the first statement of the following DATA step:

```
data mycas.rpcaData;
set rpcaData;
run;
```

This statement assumes your CAS engine libref is named mycas, but you can substitute any appropriately defined CAS engine libref.

The following statement runs PROC RPCA and outputs the decomposition results to the mycas CAS library engine:

```
proc rpca data=mycas.rpcaData
          decomp=svd
          outlowrank=mycas.lowrankmat
          outsparse=mycas.sparsemat;
id index;
outdecomp svdleft=mycas.svdleft
          svddiag=mycas.svddiag
          svdright=mycas.svdright;
run;
```

These statements produce the mycas.lowrankmat and mycas.sparsemat tables, which are the decompositions of mycas.rpcaData. They also produce the mycas.svdleft, mycas.svddiag, and mycas.svdright tables,

which are the decompositions of the mycas.lowrankmat table. The DECOMP= option produces the SVD decomposition of the low-rank matrix.

Figure 16.1 ODS Tables

Model Information	
Data Source	RPCADATA
RPCA Method	Augmented Lagrange Multiplier
SVD Method	Eigenvalue Decomposition
Lambda	0.1414214
Lambda Weight	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)	0.59

Figure 16.1 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 16.2 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=mycas.rpcaData
          lambdaweight = 3.5
          outsparse=mycas.sparsemat2;
id index;
run;

proc print data=mycas.sparsemat2;
run;
```

Figure 16.2 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)	0.60

If you look at mycas.sparsemat2 in the CAS engine library, 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).

Syntax: RPCA Procedure

The following statements are available in the RPCA procedure:

PROC RPCA < *options* > **;**
ID *variables* **;**
INPUT *variables* **;**
SVD < *options* > **;**
OUTDECOMP < *options* > **;**
CODE FILE=*filename* **;**
DISPLAY < *table-list* > < / *options* > **;**
DISPLAYOUT *table-spec-list* < / *options* > **;**
SAVESTATE RSTORE=*CAS-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 16.1 summarizes the *options* available in the PROC RPCA statement.

Table 16.1 PROC RPCA Statement Options

Option	Description
Input Data Table Option	
DATA=	Specifies the input data table
RPCA Options	
METHOD=	Specifies the method of solving RPCA
DECOMP=	Specifies the low-rank decomposition method

Table 16.1 *continued*

Option	Description
MAXITER=	Specifies the maximum number of iterations
LAMBDA=	Specifies the parameter λ
LAMBDAWEIGHT=	Specifies the parameter λ_{weight}
MU=	Specifies the parameter μ
FIXEDMU	Uses a fixed value for μ
TOLERANCE=	Specifies the convergence tolerance
SCALE	Scales the input data
CENTER	Centers the input data
NTHREADS=	Specifies the number of threads to use on each computation node
PREFIX=	Specifies a prefix for naming the principal components
Decomposition Output Table Options	
OUTLOWRANK=	Specifies the name of the low-rank output table
OUTSPARSE=	Specifies the name of the sparse output table
OUTERROR=	Specifies the name of the error 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 295.

COLSTATISTICS=*CAS-libref.data-table*
specifies the name of the output table to contain simple statistics for the variables of the input data table.

CAS-libref.data-table is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 282.

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

CAS-libref refers to a collection of information that is defined in the LIBNAME statement and includes the caslib, which includes a path to the data, and a session identifier, which defaults to the active session but which can be explicitly defined in the LIBNAME statement. For more information about *CAS-libref*, see the section "Using CAS Sessions and CAS Engine Librefs" on page 282.

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 none of the above two analysis.

By default, DECOMP=NONE.

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 number of input 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 294.

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=*CAS-libref.data-table*
specifies the name of the output table for the error matrix, which contains the noise in the input data.

CAS-libref.data-table is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 282.

OUTLOWRANK=*CAS-libref.data-table*
specifies the name of the output table for the low-rank matrix.

CAS-libref.data-table is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 282.

OUTSPARSE=*CAS-libref.data-table*
specifies the name of the output table for the sparse matrix.

CAS-libref.data-table is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 282.

PREFIX=*name*
specifies a prefix for naming the principal components. By default, the components are named Prin1, Prin2, . . . , Prin*n*. If you specify PREFIX=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 295.

TOLERANCE=*number*
specifies the convergence criterion for the RPCA algorithm. The default value is 10^{-7}.

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—that is, for projecting the new

observations onto the principal components. 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 filename 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.

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 procedure processes the DISPLAY statement on a CAS server and thus sends only a subset of ODS tables to the SAS client. Because ODS statements are processed on a SAS client, first all the generated display tables are sent to the client, and then the client creates a subset.

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 that are sent to the client 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 296. 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 CAS 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 CAS 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 CAS output table name.
key	specifies *key* as the ODS table name and also as the CAS output table name.

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 CAS tables for all display tables. The name of the created output CAS table is the same as the corresponding display table name. If you specify this option, the *table-spec-list* specification is ignored.

NOREPLACE
does not replace any existing CAS output table of the same name.

REPEATED
replicates all CAS output tables on all nodes.

ID Statement

ID *variables* **;**

The ID statement lists one or more variables to be copied from the input data table to the OUTLOWRANK=, OUTSPARSE=, and OUTERROR= 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 RSTORE= file).

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.

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=*CAS-libref.data-table*
specifies the name of the output table to contain the SVD diagonal vector.

CAS-libref.data-table is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 282.

SVDLEFT=*CAS-libref.data-table*
specifies the name of the output table to contain the SVD left matrix.

CAS-libref.data-table is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 282.

SVDRIGHT=*CAS-libref.data-table*
specifies the name of the output table to contain the SVD right matrix.

CAS-libref.data-table is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 282.

If you specify DECOMP=PCA in the PROC RPCA statement, then you can specify the following two *options*:

PCLOADINGS=*CAS-libref.data-table*
specifies the name of the output table to contain the matrix of principal component loadings.

CAS-libref.data-table is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 282.

PCSCORES=*CAS-libref.data-table*
specifies the name of the output table to contain the matrix of principal component scores.

CAS-libref.data-table is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 282.

SAVESTATE Statement

SAVESTATE RSTORE=*CAS-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, "The ASTORE Procedure."

You must specify the following option:

RSTORE=*CAS-libref.data-table*
specifies the filename of the file in which to save the analytic store. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 282.

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.

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

$$\begin{aligned} &\text{minimize} \quad ||L||_* + \lambda||S||_1 \\ &\text{subject to} \quad L + S = M \end{aligned}$$

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: augmented Lagrange multiplier and accelerated proximal gradient.

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

[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.

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.

Scaling and Centering

If the CENTER or SCALE option (or both) are specified, 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:

$$L(i,j) = L(i,j) * (\sigma_j) + \mu_j$$

$$S(i,j) = S(i,j) * (\sigma_j)$$

$$N(i,j) = N(i,j) * (\sigma_j)$$

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.

[2]The singular value thresholding operator is based on the singular value decomposition.

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 created by the RPCA procedure 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 16.2.

Table 16.2 ODS Tables Produced by PROC RPCA

Table Name	Description	Statement	Option
ModelInfo	Model information	PROC RPCA	Default
Dimensions	Dimensions of the input table	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 16.3 lists details about these data tables.

Table 16.3 Output Data Table Produced by PROC RPCA

Data Table	Content	Dimensions
OUTLOWRANK	Low-rank matrix	Number of observations × number of variables
OUTSPARSE	Sparse matrix	Number of observations × number of variables
OUTERROR	Error matrix	Number of observations × number of variables
SVDLEFT	Left-singular vectors	Number of observations × number of singular values
SVDDIAG	Singular values in the decreasing order	Number of singular values
SVDRIGHT	Right-singular vectors (and an extra column for variable names)	Number of variables × (number of singular values +1)
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 × diagonal matrix of singular values	Number of observations × number of singular values
COLSTATISTICS	Mean and standard deviation for each variable of the input data table	Number of variables × 3

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.

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 17
The SVDD Procedure

Contents

Overview: SVDD Procedure **299**
PROC SVDD Features 300
Using CAS Sessions and CAS Engine Librefs 301
Getting Started: SVDD Procedure **301**
Syntax: SVDD Procedure **305**
PROC SVDD Statement 306
CODE Statement 307
ID Statement 307
INPUT Statement 308
KERNEL Statement 308
SAVESTATE Statement 309
SOLVER Statement 309
WEIGHT Statement 310
Details: SVDD Procedure **310**
Mathematical Formulation 310
Radial Basis Function (RBF) Bandwidth Parameter 313
Fraction Outlier Parameter 315
SVDD Formulation with a Weight Variable 316
Comparison with One-Class Support Vector Machines 317
SVDD Solvers 317
ODS Table Names 318
Examples: SVDD Procedure **319**
Example 17.1: Analysis of Aircraft Engine Degradation 319
Example 17.2: Automatic RBF Kernel Bandwidth Parameter Value Selection 323
Example 17.3: Stochastic Subset Solver 328
References **335**

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 the majority of the data belong to one class. Applications of SVDD include the following:

- multivariate process control (Sun and Tsung 2003; Sukchotrat, Kim, and Tsung 2009)
- 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)
- 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 CAS Sessions and CAS Engine Librefs

SAS Cloud Analytic Services (CAS) is the analytic server and associated cloud services in SAS Viya. This section describes how to create a CAS session and set up a CAS engine libref that you can use to connect to the CAS session. It assumes that you have a CAS server already available; contact your system administrator if you need help starting and terminating a server. This CAS server is identified by specifying the host on which it runs and the port on which it listens for communications. To simplify your interactions with this CAS server, the host information and port information for the server are stored as SAS option values that are retrieved automatically whenever this CAS server needs to be accessed. You can examine the host and port values for the server at your site by using the following statements:

```
proc options option=(CASHOST CASPORT);
run;
```

In addition to starting a CAS server, your system administrator might also have created a CAS session and a CAS engine libref for your use. You can define your own sessions and CAS engine librefs that connect to the CAS server as shown in the following statements:

```
cas mysess;
libname mycas cas sessref=mysess;
```

The CAS statement creates the CAS session named mysess, and the LIBNAME statement creates the mycas CAS engine libref that you use to connect to this session. It is not necessary to explicitly name the CASHOST and CASPORT of the CAS server in the CAS statement, because these values are retrieved from the corresponding SAS option values.

If you have created the mysess session, you can terminate it by using the TERMINATE option in the CAS statement as follows:

```
cas mysess terminate;
```

For more information about the CAS and LIBNAME statements, see the section "Introduction to Shared Concepts" on page 11 in Chapter 3, "Shared Concepts."

Getting Started: SVDD Procedure

NOTE: Input data must be in a CAS table that is accessible in your CAS session. You must refer to this table by using a two-level name. The first level must be a CAS engine libref, and the second level must be the table name. For more information, see the sections "Using CAS Sessions and CAS Engine Librefs" on page 11 and "Loading a SAS Data Set onto a CAS Server" on page 12 in Chapter 3, "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;
   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
2.05387 25.6756

   ... more lines ...
```

The following code standardizes variables y4 and y5:

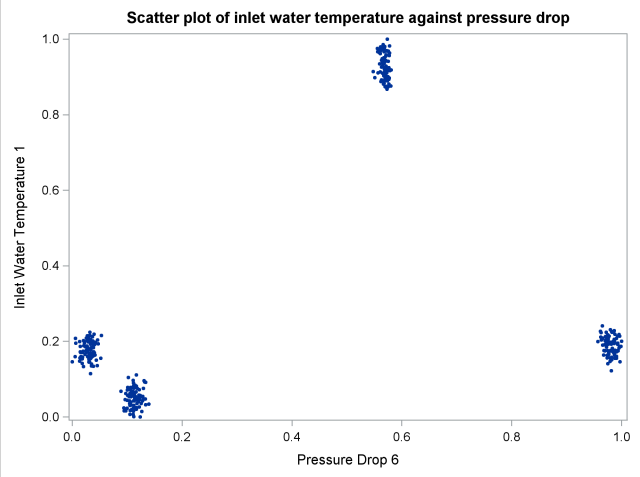
```
proc sql noprint;
   select min(y4), min(y5), max(y4), max(y5)
   into :miny4, :miny5, :maxy4, :maxy5
   from  Refrigerant;
quit;


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 17.1 shows the results. The scatter plot indicates four distinct clusters.

Figure 17.1 Scatter Plot of Inlet Water Temperature against Pressure Drop



You can load the work.refrigerant data set into your CAS session by specifying your CAS engine libref in the second statement in the following DATA step:

```
data mycas.refrigerant;
   set refrigerant;
run;
```

These statements assume that your CAS engine libref is named mycas, as in the section "Using CAS Sessions and CAS Engine Librefs" on page 301, but you can substitute any appropriately defined CAS engine libref.

The following statements execute the SVDD algorithm on the mycas.refrigerant data table and produce Figure 17.2 through Figure 17.5.

```
proc svdd data=mycas.Refrigerant;
   id y4 y5;
   input y4 y5/level=interval;
   kernel rbf / bw=0.021;
   savestate rstore=mycas.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 *mycas.state* data table. You can use the analytic store later in the ASTORE procedure to score new data. The "Model Information" table in Figure 17.2 summarizes the key options and the input data variables.

Figure 17.2 Refrigerant Analysis Model Information

Model Information	
Optimization Method	Active set
Kernel Type	RBF
RBF Kernel Bandwidth	0.021
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 17.3 shows the number of support vectors and the R^2 threshold value.

Figure 17.3 Refrigerant Analysis Training Results

Training Results	
Number of Support Vectors	38
Number of Support Vectors on Boundary	38
Number of Dropped Observations	0
Threshold R^2 Value	0.94341
Constant (C_r) Value	0.05659
Run Time (Seconds)	0.00779

Figure 17.4 displays the number of observations in the training data set.

Figure 17.4 Refrigerant Analysis Training Observations

Number of Observations Read	360
Number of Observations Used	360

The "Optimization Summary" table in Figure 17.5 shows whether the solution is optimal, the number of iterations that were required, and the objective function value.

Figure 17.5 Refrigerant Analysis Optimization Summary

Optimization Summary	
Number of Iterations	1
Objective Value	0.0565942724
Infeasibility	0.0000295791
Optimization Status	Optimal
Degenerate	False

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 CAS session by specifying your CAS engine libref in the second statement in the following DATA step:

```
data mycas.scoreds;
   set scoreds;
run;
```

The following code scores the mycas.scoreds data by using the score analytic store that was created during training:

```
 proc astore;
    score data=mycas.scoreds
    out=mycas.score_results
    rstore=mycas.state;
 quit;
```

The following code copies the score_results into the Work library:

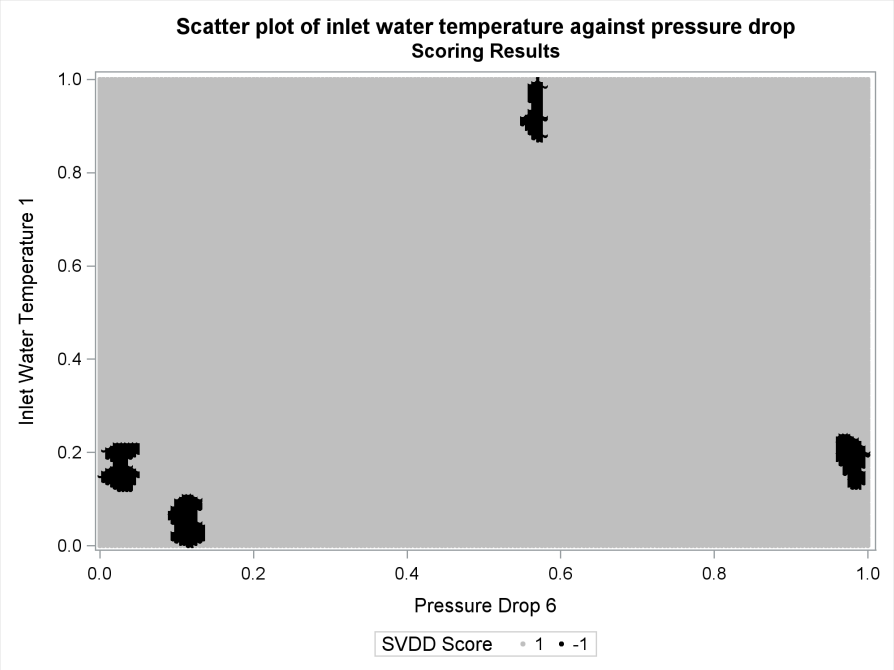
```
data work.score_results;
   set mycas.score_results;
run;
```

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 17.6 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 17.1 and the training data in Figure 17.6 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 17.6 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 17.6 Scoring Results Scatter Plot



.

Syntax: SVDD Procedure

The following statements are available in the SVDD procedure:

```
PROC SVDD < options > ;
   CODE FILE=filename ;
   ID variables ;
   INPUT variables / < LEVEL=INTERVAL | NOMINAL > ;
   WEIGHT variable ;
   KERNEL kernel-type / < kernel-parameter> ;
   SOLVER solver-type / < options > ;
   SAVESTATE RSTORE=CAS-libref.data-table ;
```

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

CAS-libref refers to a collection of information that is defined in the LIBNAME statement and includes the caslib, which includes a path to the data, and a session identifier, which defaults to the active session but which can be explicitly defined in the LIBNAME statement. For more information about *CAS-libref*, see the section "Using CAS Sessions and CAS Engine Librefs" on page 301.

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* must be 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=*CAS-libref.data-table*
SV=*CAS-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. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 301.

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 filename of the file to write the SAS score code to.

The CODE statement is optional.

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-parameter*:

RBF / BW=*s*

RBF / BW=MEAN (DELTA=*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, y are two 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 is decreased. At very small values of s, the processing time can be very high.
- You can instruct the SVDD procedure 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 section "Mean Criterion " on page 315.

SAVESTATE Statement

SAVESTATE RSTORE=*CAS-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, "The ASTORE Procedure."

You must specify the following option:

RSTORE=*CAS-libref.data-table*
specifies the filename of the file in which to save the analytic store. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 301.

SOLVER Statement

SOLVER *solver-type* < */options* > **;**

The SOLVER statement specifies the type of optimization solver for SVDD training. You can specify following *solver-types*:

ACTSET
uses the active-set solver for SVDD training. For more information about this solver, see the section "Active-Set Solver " on page 318.

STOCHS
uses a stochastic subset solver for SVDD training. For more information about this solver, see the section "Stochastic Subset Solver " on page 318.

You can specify the following *options* after a slash (/) for either solver-type:

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.

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.

You can specify the following options after a slash(/) only when the *solver-type* is 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.

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.

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 0. If the radius and center values converge for *number* of consecutive iterations, then convergence is declared. By default, NMATCH=5. To obtain a solution more quickly, specify a lower value for *number*.

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 312). 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
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 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 311) 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

$$\sum_{i=1}^{n} \alpha_i = 1$$

$$0 \leq \alpha_i \leq C \text{ for all } i = 1, \ldots, n$$

The results for the center, inside, boundary, and outside positions, which are described in the section "Duality Information" on page 311, 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 $\text{dist}^2(z)$ is calculated as follows:

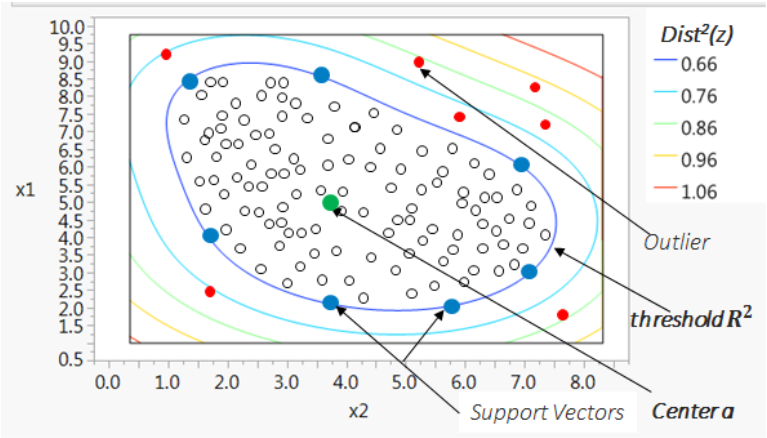
$$\text{dist}^2(z) = K(z,z) - 2\sum_i \alpha_i K(x_i, z) + \sum_{i,j} \alpha_i \alpha_j K(x_i, x_j)$$

Observations in the scoring data set for which $\text{dist}^2(z) > R^2$ are designated as outliers.

Visual Explanation of SVDD

Figure 17.7 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 17.7 Key SVDD Statistics

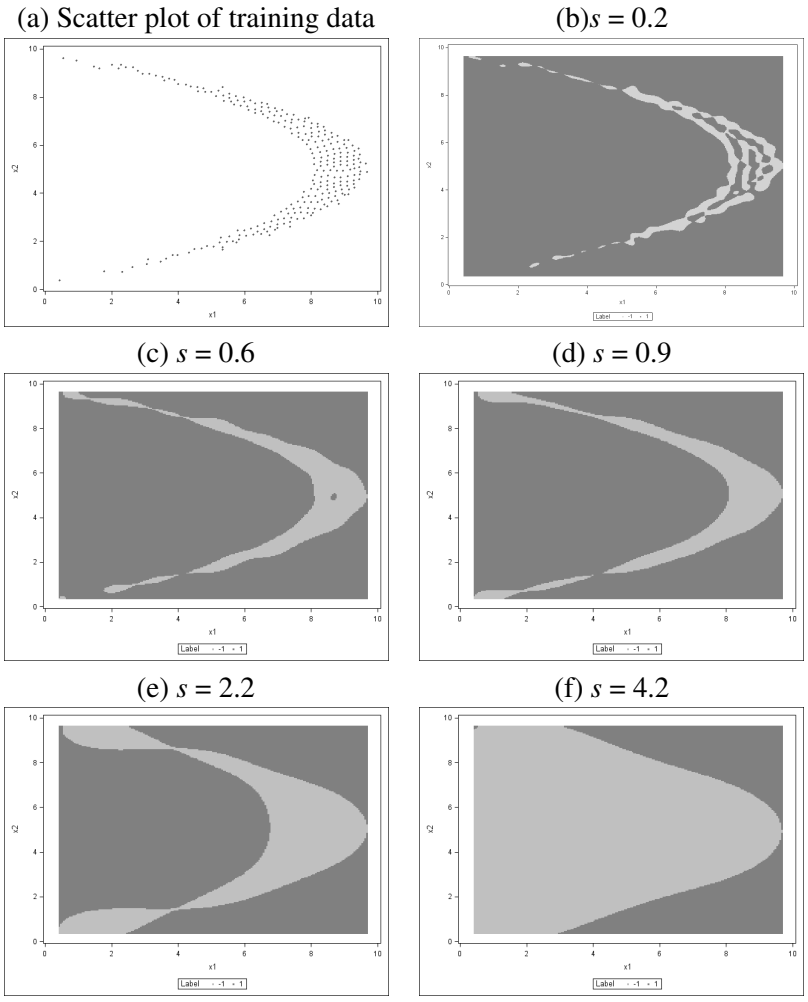


Radial Basis Function (RBF) Bandwidth Parameter

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 high.

Figure 17.8 illustrates the effect of s on the data boundary.

Figure 17.8 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 often involves experimenting with several values until a good data boundary is obtained. If labeled data that belong to multiple classes are available, cross validation can be used to select value of s. If training data observations belong to a single class, cross validation cannot be used. In such a scenario, the distance to the farthest neighbor (DFN) method (Xiao et al. 2014), the peak criterion method (Kakde et al. 2017), or the mean or median criterion (Chaudhuri et al. 2017) can be used to choose an appropriate value of s. The SVDD procedure supports the mean criterion for selecting an appropriate value of s. You can use the mean criterion by specifying BW=MEAN, as described in the KERNEL statement.

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.

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. The bandwidth relative scale value is reported in the model information ODS table. If your analysis is taking long time to complete or your model is overfitting, the bandwidth relative scale value can be used to determine if resubmitting the analysis with a different value of bandwidth is warranted. A very small value of bandwidth relative scale which is less than one indicates that user specified bandwidth value is too low for the training data set. In such scenario, improvements in performance or quality of training results can be achieved by resubmitting the analysis with a higher bandwidth value.

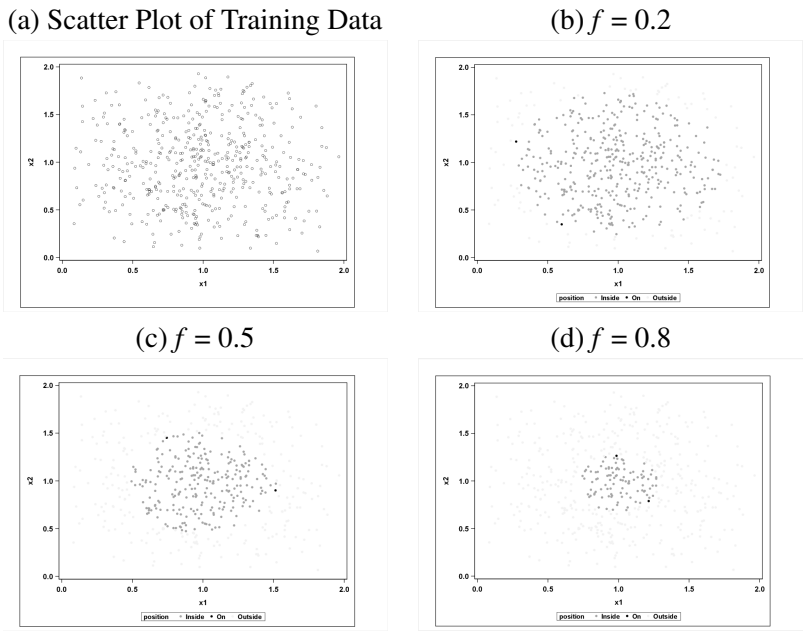
There are multiple factors (such as, but not limited to, solver type, training data, and solver tolerances) that can affect the processing time and quality of results. Hence, there might be situations where adjusting the bandwidth parameter value alone might not lead to an improvement in performance or quality of results.

Fraction Outlier Parameter

As outlined in the section "Mathematical Formulation" on page 310, 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 17.9 illustrates the effect of f on the data boundary.

Figure 17.9 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 more 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

$$\sum_{i=1}^{n} \alpha_i = 1$$

$$0 \leq \alpha_i \leq \frac{1}{\nu n} \text{ for all } i = 1, \ldots, n$$

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 312 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 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 option 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.

Once convergence is obtained, the resulting master set of support vectors SV^* is set as an input to the active-set solver and solution is obtained. For more information about the stochastic subset solver, see Chaudhuri et al. (2018).

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 17.1.

Table 17.1 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	Default
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
TrainingResults	Displays the training results	PROC SVDD	Default

Examples: SVDD Procedure

NOTE: Input data must be in a CAS table that is accessible in your CAS session. You must refer to this table by using a two-level name. The first level must be a CAS engine libref, and the second level must be the table name. For more information, see the sections "Using CAS Sessions and CAS Engine Librefs" on page 11 and "Loading a SAS Data Set onto a CAS Server" on page 12 in Chapter 3, "Shared Concepts."

Example 17.1: 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 $\text{dist}^2(z)$ is computed and compared to the threshold R^2 value. Observations for which $\text{dist}^2(z) > R^2$ 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 CAS session by specifying your CAS engine libref in the second statement in the following DATA step. These statements assume that your CAS engine libref is named mycas, as in the section "Using CAS Sessions and CAS Engine Librefs" on page 301, but you can substitute any appropriately defined CAS engine libref.

```
data mycas.train;
   set train;
run;
```

You can load the work.score data set into your CAS session by specifying your CAS engine libref in the second statement in the following DATA step:

```
data mycas.score;
   set score;
run;
```

The following statements execute the SVDD algorithm on the mycas.train data table and produce Figure 17.1.1 through Figure 17.1.2.

```
 proc svdd data=mycas.train;
    id engine cycle;
    input x1-x24/level=interval;
    kernel rbf / bw=94;
    savestate rstore=mycas.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 *mycas.state* data table. You can use the analytic store later in the ASTORE procedure to score new data. The "Model Information" table in Output 17.1.1 summarizes the key options and input data variables.

Output 17.1.1 Aircraft Engine Degradation Analysis Model Information

Model Information	
Optimization Method	Active set
Kernel Type	RBF
RBF Kernel Bandwidth	94
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 17.1.2 shows the number of support vectors and the threshold R^2 value.

Output 17.1.2 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^2 Value	0.80378
Constant (C_r) Value	0.19634
Run Time (Seconds)	0.00487

Output 17.1.3 displays the number of observations in the training data set.

Output 17.1.3 Aircraft Engine Degradation Analysis Training Observations

Number of Observations Read	225
Number of Observations Used	225

The "Optimization Summary" table in Output 17.1.4 shows whether the solution is optimal, the number of iterations that were required and the objective function value.

Output 17.1.4 Aircraft Engine Degradation Analysis Optimization Summary

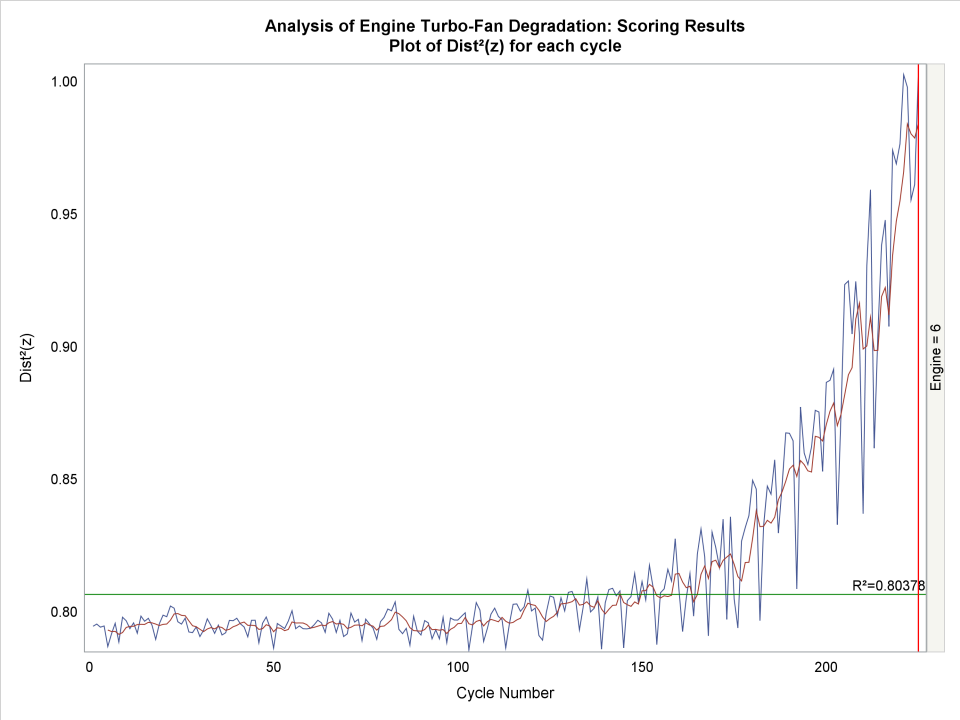
Optimization Summary	
Number of Iterations	1
Objective Value	0.1963444804
Infeasibility	0.0000478938
Optimization Status	Optimal
Degenerate	False

The following code scores the *mycas.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=mycas.score
   out=mycas.test_out
   rstore=mycas.state;
quit;
```

Output 17.1.5 shows the scoring results for Engine 6. The plot indicates the $\mathrm{dist}^2(z)$ value 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. The Output 17.1.5 shows that the $\mathrm{dist}^2(z)$ value serves as a good proxy for engine degradation.

Output 17.1.5 Scoring Results Scatter Plot



.

Example 17.2: 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. 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 301, where the value of RBF kernel bandwidth parameter was predetermined by the specification of BW=0.021 in the KERNEL statement. In this example, the SVDD procedure computes the RBF kernel bandwidth parameter value using the mean criterion, alleviating the need to explicitly specify a value. 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 data set (which is stored in the Work library by default):

```
data Refrigerant;
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
2.05387 25.6756

   ... more lines ...
```

The following code standardizes variables y4 and y5:

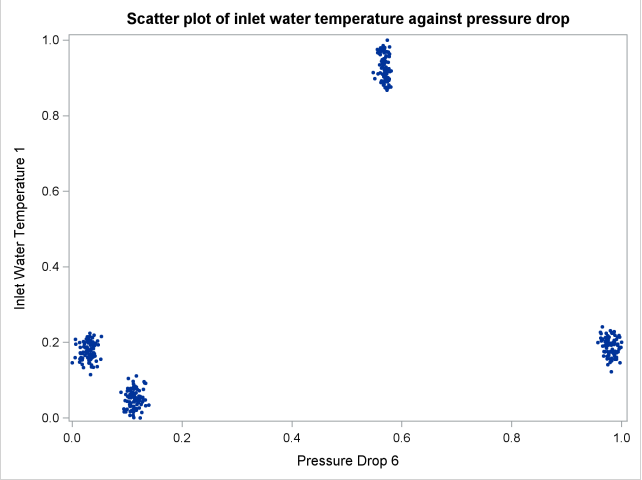
```
proc sql noprint;
   select min(y4), min(y5), max(y4), max(y5)
   into :miny4, :miny5, :maxy4, :maxy5
   from  Refrigerant;
quit;


   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 17.2.1 shows the results. The scatter plot indicates four distinct clusters.

Output 17.2.1 Scatter Plot of Inlet Water Temperature against Pressure Drop



You can load the Work.Refrigerant data set into your CAS session by specifying your CAS engine libref in the second statement in the following DATA step:

```
data mycas.Refrigerant;
      set Refrigerant;
run;
```

These statements assume that your CAS engine libref is named mycas, as in the section "Using CAS Sessions and CAS Engine Librefs" on page 301, but you can substitute any appropriately defined CAS engine libref.

The following statements execute the SVDD algorithm on the Mycas.Refrigerant data table and produce Output 17.2.2 through Output 17.2.5.

```
proc svdd data=mycas.Refrigerant score nbins=10;
   id y4 y5;
   input y4 y5/level=interval;
   kernel rbf / bw=mean;
   savestate rstore=mycas.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 mycas.state data table. You can use the analytic store later in the ASTORE procedure to score new data. The "Model Information" table in Output 17.2.2 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 17.2.2 Refrigerant Analysis Model Information

Model Information	
Optimization Method	Active set
Kernel Type	RBF
RBF Kernel Bandwidth	0.1262714382
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 17.2.3 shows the number of support vectors and the R^2 threshold value.

Output 17.2.3 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^2 Value	0.76304
Constant (C_r) Value	0.23696
Run Time (Seconds)	0.00472

Output 17.2.4 displays the number of observations in the training data set.

Output 17.2.4 Refrigerant Analysis Training Observations

Number of Observations Read	360
Number of Observations Used	360

The "Optimization Summary" table in Output 17.2.5 shows whether the solution is optimal, the number of iterations that were required, and the objective function value.

Output 17.2.5 Refrigerant Analysis Optimization Summary

Optimization Summary	
Number of Iterations	1
Objective Value	0.2369581441
Infeasibility	1.897082E-11
Optimization Status	Optimal
Degenerate	False

The "Histogram" table in Output 17.2.6 shows the frequency table of _SVDDDISTANCE_ values of observations in the mycas.Refrigerant training data set.

Output 17.2.6 Refrigerant Analysis Histogram Table

Histogram					
Min Point	Mid Point	Max Point	Count	Percent (%)	Cumulative Percent (%)
0.713638	0.716108	0.718578	28	7.777778	7.777778
0.718578	0.721048	0.723518	45	12.500000	20.277778
0.723518	0.725989	0.728459	73	20.277778	40.555556
0.728459	0.730929	0.733399	62	17.222222	57.777778
0.733399	0.735869	0.738340	48	13.333333	71.111111
0.738340	0.740810	0.743280	37	10.277778	81.388889
0.743280	0.745750	0.748221	30	8.333333	89.722222
0.748221	0.750691	0.753161	12	3.333333	93.055556
0.753161	0.755631	0.758101	15	4.166667	97.222222
0.758101	0.760572	0.763042	10	2.777778	100.000000

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 CAS session by specifying your CAS engine libref in the first statement in the following DATA step:

```
data mycas.scoreds;
   set scoreds;
run;
```

The following code scores the mycas.scoreds data table by using the score analytic store that was created during training:

```
 proc astore;
    score data=mycas.scoreds
    out=mycas.score_results
    rstore=mycas.state;
 quit;
```

The following code copies the score_results into the Work library:

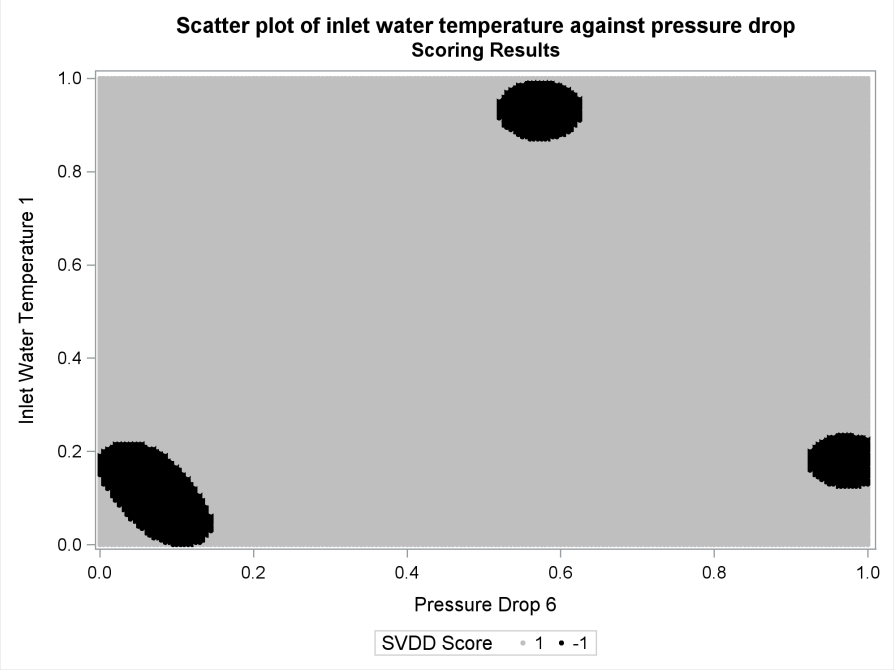
```
data work.score_results;
   set mycas.score_results;
run;
```

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;
```

Output 17.2.7 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. In this example, the mean criterion was used to compute the RBF kernel bandwidth parameter value. Comparison between Output 17.2.7 and the training data in Output 17.2.1 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 a one cluster. In addition, the description of clusters obtained in this example is more circular. This is expected because the RBF bandwidth parameter value of 0.1262 computed using the mean criterion is higher than the value 0.021 that was specified in the section "Getting Started: SVDD Procedure" on page 301. Despite these shortcomings, the mean criterion provides a reasonable approximation of the training data. The scatter plot of the scoring data set in Output 17.2.7 indicates that PROC SVDD was able to correctly identify a majority of the outliers.

Output 17.2.7 Scoring Results Scatter Plot: Bandwidth Computation Using the Mean Criterion



.

Example 17.3: 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 318. 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:

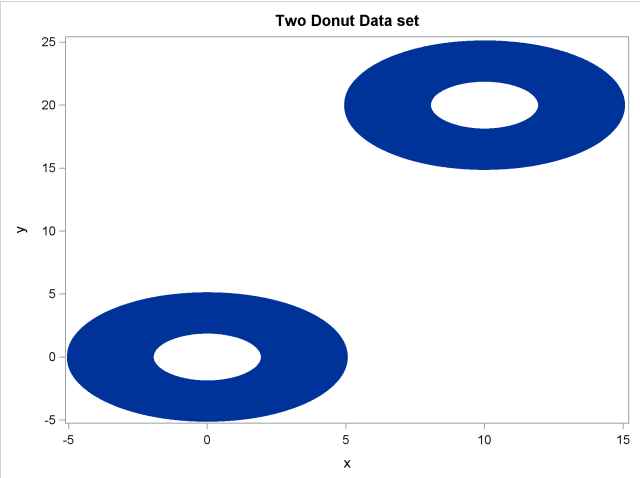
```
%let r_min=2;
%let r_max=5;

   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 17.3.1 shows the results.

Output 17.3.1 Scatter Plot of Two-Donut Data



You can load the Work.two_donut data set into your CAS session by specifying your CAS engine libref in the first statement in the following DATA step:

```
data mycas.two_donut;
  set two_donut;
run;
```

These statements assume that your CAS engine libref is named mycas, as in the section "Using CAS Sessions and CAS Engine Librefs" on page 301, but you can substitute any appropriately defined CAS engine libref. The following statements execute the SVDD algorithm on the mycas.two_donut data table by using the active-set solver and produce Output 17.3.2 through Output 17.3.5:

```
proc svdd data=mycas.two_donut std;
   id x y;
   input x y/level=interval;
   kernel rbf/bw=MEAN ;
   solver actset/stol=1e-6;
   savestate rstore=mycas.actset;
run;
```

The STD option in the PROC SVDD statement standardizes the mycas.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 mycas.actset data table. You can use the analytic store later in the ASTORE procedure to score new data.

The "Model Information" table in Output 17.3.2 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 17.3.2 Two Donut Model Information: Active-Set Solver

Model Information	
Optimization Method	Active set
Kernel Type	RBF
RBF Kernel Bandwidth	0.305591793
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 17.3.3 shows the number of support vectors and the R^2 threshold value.

Output 17.3.3 Two Donut Analysis Training Results: Active-Set Solver

Training Results	
Number of Support Vectors	54
Number of Support Vectors on Boundary	54
Number of Dropped Observations	0
Threshold R^2 Value	0.90931
Constant (C_r) Value	0.09069
Run Time (Seconds)	18.56936

Output 17.3.4 displays the number of observations in the training data set.

Output 17.3.4 Two Donut Training Observations: Active-Set Solver

Number of Observations Read	4000000
Number of Observations Used	4000000

The "Optimization Summary" table in Output 17.3.5 shows whether the solution is optimal, the number of iterations that were required, and the objective function value.

Output 17.3.5 Two Donut Optimization Summary: Active-Set Solver

Optimization Summary	
Number of Iterations	1
Objective Value	0.0906857275
Infeasibility	6.5580118E-7
Optimization Status	Optimal
Degenerate	False

The following statements execute the SVDD algorithm on the mycas.two_donut data table by using the STOCHS solver and produce Output 17.3.6 through Output 17.3.9.

```
proc svdd data=mycas.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=mycas.stochs;
run;
```

The STD option in PROC SVDD statement standardizes the mycas.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 mycas.stochs data table. You can use the analytic store later in the ASTORE procedure to score new data.

The "Model Information" table in Output 17.3.6 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 17.3.6 Two Donut Model Information: Stochastic Subset Solver

Model Information	
Optimization Method	Stochastic Subset
Kernel Type	RBF
RBF Kernel Bandwidth	0.305591793
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 17.3.7 shows the number of support vectors and the R^2 threshold value.

Output 17.3.7 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^2 Value	0.90849
Constant (C_r) Value	0.09151
Run Time (Seconds)	0.00408

Output 17.3.8 displays the number of observations in the training data set.

Output 17.3.8 Two Donut Training Observations: Stochastic Subset Solver

Number of Observations Read	4000000
Number of Observations Used	4000000

The "Optimization Summary" table in Output 17.3.9 shows whether the solution is optimal, the number of iterations that were required, and the objective function value.

Output 17.3.9 Two Donut Optimization Summary: Stochastic Subset Solver

Optimization Summary	
Number of Iterations	200
Objective Value	0.0915147103
Infeasibility	2.740319E-11
Optimization Status	Optimal
Degenerate	False

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 17.3.2 and Output 17.3.6. 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 mycas.actset and mycas.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 CAS session by specifying your CAS engine libref in the first statement in the following DATA step:

```
   data mycas.scoreds;
      set scoreds;
   run;
```

The following code scores the mycas.scoreds using the score analytic store that was created using the ACTSET solver during training.

```
proc astore;
   score data=mycas.scoreds
   out=mycas.scoreact
   rstore=mycas.actset;
run;
```

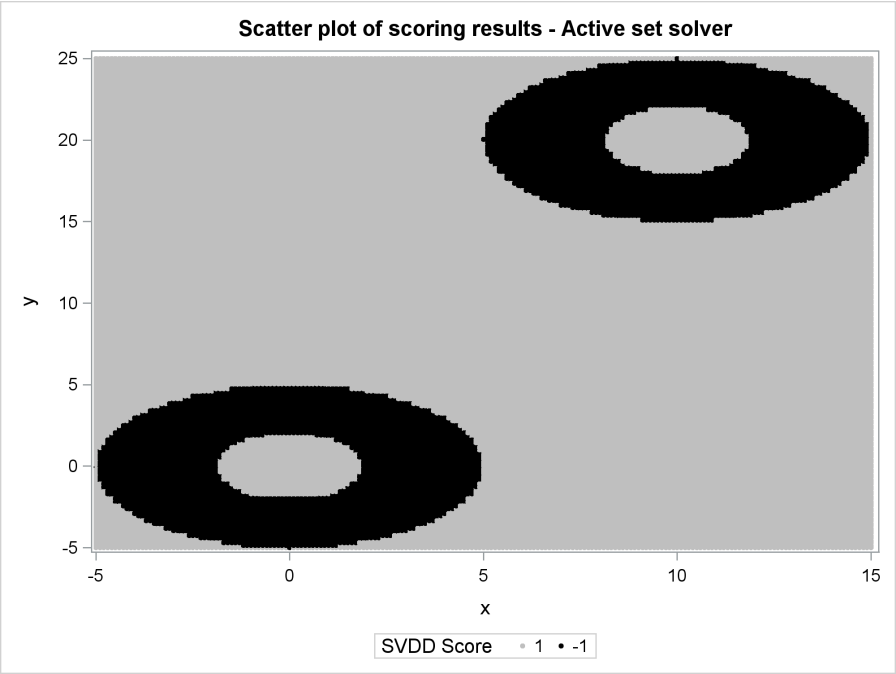
The following code copies the mycas.scoreact data table into the Work library:

```
   data work.scoreact;
      set mycas.scoreact;
   run;
```

The following code plots the scoring results, as shown in Output 17.3.10:

```
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 17.3.10 Scoring Results Scatter Plot: Active-Set Solver



The following code scores the mycas.scoreds data table by using the score analytic store that was created using the STOCHS solver during training:

```
proc astore;
   score data=mycas.scoreds
   out=mycas.scorestochs
   rstore=mycas.stochs;
run;
```

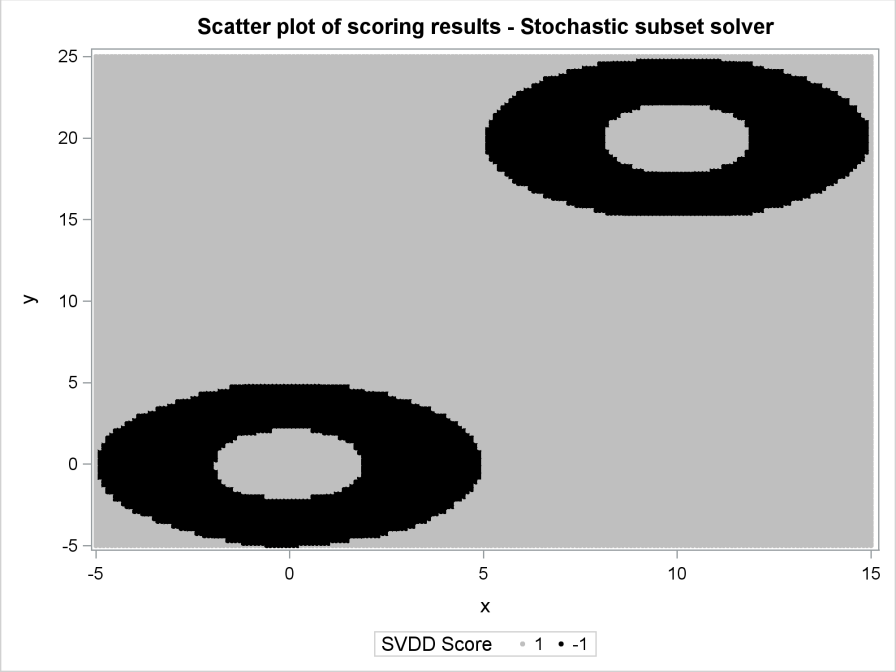
The following code copies the scorestochs data table into the Work library:

```
   data Work.scorestochs;
     set mycas.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 17.3.11 Scoring Results Scatter Plot: Stochastic Subset Solver



Output 17.3.10 and Output 17.3.11 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 17.3.10 and Output 17.3.11 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.

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.

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.

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.

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.

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://ti.arc.nasa.gov/tech/dash/groups/pcoe/prognostic-data-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 18
The SVMACHINE Procedure

Contents

Overview: SVMACHINE Procedure **339**
PROC SVMACHINE Features 340
Using CAS Sessions and CAS Engine Librefs 340
Getting Started: SVMACHINE Procedure **341**
Syntax: SVMACHINE Procedure **343**
PROC SVMACHINE Statement 343
AUTOTUNE Statement 344
CODE Statement 347
ID Statement 347
INPUT Statement 347
KERNEL Statement 348
OUTPUT Statement 348
PARTITION Statement 349
SAVESTATE Statement 350
TARGET Statement 350
Details: SVMACHINE Procedure **350**
Interior-Point Method Optimization Technique 351
Scoring Process 352
Displayed Output 352
ODS Table Names 353
Examples: SVMACHINE Procedure **354**
Example 18.1: Home Equity Loan Case 354
Example 18.2: Large Simulated Data Table 357
References **358**

Overview: SVMACHINE Procedure

The SVMACHINE procedure implements the support vector machines (SVM) algorithm in SAS Viya. A popular data mining area classification method, the SVM algorithm computes support vector machine learning classifiers for the binary pattern recognition problem; it has been broadly used in the fields such as image classification, handwriting recognition, financial decision, text mining, and so on.

Like other predictive modeling tools, the SVMACHINE procedure uses input data to train a model and provides information about the model. The SVMACHINE procedure executes the SVM algorithm (applying

the interior-point optimization technique during training) and can generate SAS code for scoring future data. PROC SVMACHINE uses both linear and low-degree polynomial kernels to conduct computation, and it can run on multiple threads in a single machine or on multiple threads on multiple machines. It can load data from multiple nodes and perform computation in parallel.

PROC SVMACHINE Features

The SVMACHINE procedure has the following features:

- reads input data in parallel when the data source is on a distributed system
- is highly multithreaded during all phases of analytic execution
- supports large-scale training data
- supports both continuous and categorical inputs
- supports classification of a binary target
- supports the interior-point method

Using CAS Sessions and CAS Engine Librefs

SAS Cloud Analytic Services (CAS) is the analytic server and associated cloud services in SAS Viya. This section describes how to create a CAS session and set up a CAS engine libref that you can use to connect to the CAS session. It assumes that you have a CAS server already available; contact your system administrator if you need help starting and terminating a server. This CAS server is identified by specifying the host on which it runs and the port on which it listens for communications. To simplify your interactions with this CAS server, the host information and port information for the server are stored as SAS option values that are retrieved automatically whenever this CAS server needs to be accessed. You can examine the host and port values for the server at your site by using the following statements:

```
proc options option=(CASHOST CASPORT);
run;
```

In addition to starting a CAS server, your system administrator might also have created a CAS session and a CAS engine libref for your use. You can define your own sessions and CAS engine librefs that connect to the CAS server as shown in the following statements:

```
cas mysess;
libname mycas cas sessref=mysess;
```

The CAS statement creates the CAS session named mysess, and the LIBNAME statement creates the mycas CAS engine libref that you use to connect to this session. It is not necessary to explicitly name the CASHOST and CASPORT of the CAS server in the CAS statement, because these values are retrieved from the corresponding SAS option values.

If you have created the mysess session, you can terminate it by using the TERMINATE option in the CAS statement as follows:

```
cas mysess terminate;
```

For more information about the CAS and LIBNAME statements, see the section "Introduction to Shared Concepts" on page 11 in Chapter 3, "Shared Concepts."

Getting Started: SVMACHINE Procedure

NOTE: Input data must be in a CAS table that is accessible in your CAS session. You must refer to this table by using a two-level name. The first level must be a CAS engine libref, and the second level must be the table name. For more information, see the sections "Using CAS Sessions and CAS Engine Librefs" on page 11 and "Loading a SAS Data Set onto a CAS Server" on page 12 in Chapter 3, "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 CAS session by specifying your CAS engine libref in the second statement in the following DATA step:

```
data mycas.dmagecr;
   set sampsio.dmagecr;
run;
```

These statements assume that your CAS engine libref is named mycas, as in the section "Using CAS Sessions and CAS Engine Librefs" on page 340, but you can substitute any appropriately defined CAS engine libref.

The following statements execute the SVM algorithm on the mycas.dmagecr data table and produce Figure 18.1 through Figure 18.3.

```
proc svmachine data=mycas.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 18.1 shows that the inner product of weights is 11.6121718, the bias is –2.1296773, and the number of support vectors is 531, of which 481 are on the margin.

Figure 18.1 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 18.2 shows that among the total 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%, which is indicated in the "Fit Statistics" table in Figure 18.3.

Figure 18.2 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 18.3 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* **;**
ID *variables* **;**
INPUT *variables* / < **LEVEL=INTERVAL** | **NOMINAL** > **;**
KERNEL *kernel-type* / < *kernel-parameter* > **;**
OUTPUT OUT=*CAS-libref.data-table* < *option* > **;**
PARTITION *partition-option* **;**
SAVESTATE RSTORE=*CAS-libref.data-table* **;**
TARGET *variable* < /*option* > **;**

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* must be a real number greater than 0.

By default, C=1.0.

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

CAS-libref refers to a collection of information that is defined in the LIBNAME statement and includes the caslib, which includes a path to the data, and a session identifier, which defaults to the active session but which can be explicitly defined in the LIBNAME statement. For more information about *CAS-libref*, see the section "Using CAS Sessions and CAS Engine Librefs" on page 340.

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

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.

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. In some cases, you can obtain a good model in less than five iterations.

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 that are used in the computation. The default value is the number of CPUs available in the machine.

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

By default, these two tables are not generated.

SCALE
scales the input variables to between 0 and 1 during training.

By default, all numerical data are scaled before the training.

TOLERANCE=*number*
specifies the minimal absolute tolerance at which the iteration stops. The tolerance *number* must be equal to or greater than 1.0E-12.

By default, TOLERANCE=1.0E-6. In some cases, you can obtain a good model with TOLERANCE $\geq$ 0.01.

USEMISS
uses missing values for input variables. Missing 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= option in the PROC SVMACHINE statement and the DEGREE= suboption of the POLYNOMIAL option in the KERNEL statement.

Table 18.1 summarizes the *options* 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 14 in Chapter 3, "Shared Concepts." The TUNINGPARAMETERS option is described following table Table 18.1.

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

NOTE: If you specify both AUTOTUNE statement and OUTPUT statement, the PROC SVMACHINE exits with error message.

Table 18.1 AUTOTUNE Statement Options

Option	Description
EVALHISTORY=	Specifies how to report the evaluation history of the tuner
FRACTION=	Specifies the fraction of observations to use for validation
KFOLD=	Specifies the number of folds for *k*-fold cross validation
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
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
TARGETEVENT=	Specifies the target event for ROC-based calculations
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*:

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* must be 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 number to consider during tuning. If you specify this suboption, you cannot specify the VALUES= suboption.

By default, LB=1E-10.

UB=*number*
specifies the maximum penalty number 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 penalty numbers 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 penalty value for the tuner to use.

By default, INIT=1.

EXCLUDE
excludes the penalty value 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 linear (when *number*=1) or polynomial (when *number*=2 or 3) kernel during training, where *number* and values in *value-list* can be 1, 2 or 3. For more information, see the POLYNOMIAL option in the KERNEL statement.

You can specify the following additional suboptions:

LB=*number*
specifies a lower bound of the degree 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 of the degree to consider during tuning. 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.

By default, INIT=1.

EXCLUDE
excludes the degree 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 done by the OUTPUT statement.

You must specify the following option:

FILE=*filename*
specifies the filename of the file to write the SAS score code to.

The CODE statement is optional.

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* must be 2 or 3 (the default is 2). 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.

By default, the kernel type is LINEAR.

OUTPUT Statement

OUTPUT OUT=*CAS-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=*CAS-libref.data-table*
names the output data table for PROC SVMACHINE to use. You must specify this option before any other options. *CAS-libref.data-table* is a two-level name, where

CAS-libref refers to a collection of information that is defined in the LIBNAME statement and includes the caslib, which includes a path to where the data table is to be stored, and a session identifier, which defaults to the active session but which can be explicitly defined in the LIBNAME statement. For more information about *CAS-libref*, see the section "Using CAS Sessions and CAS Engine Librefs" on page 340.

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 18.2 shows how this *variable* is interpreted for each observation.

Table 18.2 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 20 in Chapter 3, "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 of the observations are 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=*CAS-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, "The ASTORE Procedure."

You must specify the following option:

RSTORE=*CAS-libref.data-table*
specifies a data table in which to save the analytic store for the model. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 340.

TARGET Statement

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

The TARGET statement names the target *variable* whose values PROC SVMACHINE predicts. The target *variable* must be binary and must be different from the variables in the INPUT statement.

You can specify following *options* after a slash (/):

ASC | ASCENDING
levelizes the target values in ascending order.

DESC | DESCENDING
levelizes the target values in descending order.

By default, the order is DESCENDING.

Details: SVMACHINE Procedure

PROC SVMACHINE uses linear or nonlinear kernels to compute support vector machine (SVM) learning classifiers for the binary pattern recognition problem. For more information about the theory and use of SVM learning, see Vapnik (1995); Burges (1998); Cristianini and Shawe-Taylor (2000).

In the linear kernel case, PROC SVMACHINE computes the parameters w and β that define the model function,

$$f(x) = w^T x + \beta$$

by solving the following quadratic optimization problem

$$\begin{aligned} \underset{w,\beta,z}{\text{minimize}} \quad & \frac{1}{2} w^T w + Ce^T z \\ \text{subject to} \quad & \mathbf{DX}w + \beta d \geq e - z \\ & z \geq 0 \end{aligned}$$

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, z denotes the slack variables that relax the classification constraints, and C denotes the penalty term. The corresponding dual optimization problem is

$$\begin{aligned} \underset{\alpha}{\text{minimize}} \quad & \frac{1}{2} \alpha^T \mathbf{Q} \alpha - e^T \alpha \\ \text{subject to} \quad & d^T \alpha = 0 \\ & 0 \leq \alpha \leq C \end{aligned}$$

where $\mathbf{Q} = \mathbf{DXX}^T\mathbf{D}$ and α is the Lagrange multiplier. 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 348. The corresponding model 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 infinite. Polynomial kernels are a special case in that the primal problem definition can be formed explicitly; in this case, the matrix $\mathbf{X}$ corresponds to explicitly projected observations.

Interior-Point Method Optimization Technique

A popular approach for accurately obtaining the global solution for an SVM optimization problem is the interior-point method. 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 exists an underlying structure that can be exploited. For example, interior-point methods can be extremely efficient for problems where the number of variables are small in comparison to the number of constraints. When both the number of variables and number of constraints is 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, $\mathbf{X}$ is obtained by explicitly projecting each observation in 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, KERNEL POLYNOMIAL/DEGREE=2 is not recommended when the number of columns in the design matrix is much greater than 100. Similarly, KERNEL POLYNOMIAL/DEGREE=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 for 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.

Scoring Process

The scoring process for the interior-point 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. If the decision function is less than or equal to 0, then the prediction is an event; otherwise, it is a nonevent.

For the interior-point method, the score code is provided for a linear kernel and for a polynomial kernel of degree 2 and 3.

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.

Training Results

The "Training Results" table shows the model information. It includes but is not limited to the inner product of weights, bias, and the number of support vectors.

Iteration History

The "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).

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.

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.

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_.

Predicted Target Variable

The "Predicted Target Variable" table contains the predicted target variable, whose name begins with I_.

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 18.3.

Table 18.3 ODS Tables Produced by PROC SVMACHINE

Table Name	Description	Statement	Option
FitStatistics	Accuracy information about the training	PROC SVMACHINE	Default

Table 18.3 *continued*

Table Name	Description	Statement	Option
IterationReport	Accuracy report for each iteration	PROC SVMACHINE	ITERATIONREPORT
IterHistory	Iteration history	PROC SVMACHINE	Default
Misclassification	Misclassification matrix table	PROC SVMACHINE	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
PredProbName	Predicted probability variables	PROC SVMACHINE	PRINTTARGET
TrainingResult	Displays the training results	PROC SVMACHINE	Default

Examples: SVMACHINE Procedure

NOTE: Input data must be in a CAS table that is accessible in your CAS session. You must refer to this table by using a two-level name. The first level must be a CAS engine libref, and the second level must be the table name. For more information, see the sections "Using CAS Sessions and CAS Engine Librefs" on page 11 and "Loading a SAS Data Set onto a CAS Server" on page 12 in Chapter 3, "Shared Concepts."

Example 18.1: 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 18.4 describes the variables in Hmeq.

Table 18.4 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 CAS session by specifying your CAS engine libref in the second statement in the following DATA step:

```
data mycas.hmeq;
   set sampsio.hmeq;
run;
```

The following statements execute the SVM algorithm on the mycas.hmeq data table:

```
filename codefile temp;
proc svmachine data=mycas.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 18.1.1 through Output 18.1.5.

The "Model Information" table in Output 18.1.1 shows that the kernel function is linear, and the penalty parameter value is 1 (both of which are default values).

Output 18.1.1 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 18.1.2 shows that the total number of observations is 5,960 and the number of observations used in the training is 3,364.

Output 18.1.2 Number of Observations

Number of Observations Read	5960
Number of Observations Used	3364

The "Training Results" table in Output 18.1.3 shows the inner product of weights, bias, total slack, and so on.

Output 18.1.3 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 18.1.4 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 18.1.4 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 18.1.5 shows information about the accuracy, error, sensitivity, and specificity.

Output 18.1.5 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 "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 18.2: Large Simulated Data Table

This example uses a large simulated data table to demonstrate how PROC SVMACHINE can handle relatively large data. The following DATA step generates 10 million observations in the CAS table mycas.bigdata:

```
data mycas.bigdata ;
   array x{5} x1-x5;
   drop i n;
   do n=1 to 10000000;
      do i=1 to dim(x);
         x{i} = ranbin(10816, 12, 0.6);
         x6 = sum(x2-x4) + ranuni(6068);
      end;
      if x6 > 0.5 then y = 1;
      else if x6 < -0.5 then y = 0;
      else y = ranbin(6084, 1, 0.4);
      output;
   end;
run;
```

The following statements execute the SVM algorithm on the table mycas.bigdata:

```
proc svmachine data=mycas.bigdata;
   input x1-x6 / level=interval;
   target y;
run;
```

The "Misclassification Matrix" table in Output 18.2.1 shows the classification result. The total number of observations in which $y = 1$ is 5,631,506, and the total number of observations in which $y = 0$ is 4,368,494.

Output 18.2.1 Misclassification Matrix

The SVMACHINE Procedure

Misclassification Matrix

	Training Prediction		
Observed	**1**	**0**	**Total**
1	5164803	466703	5631506
0	248256	4120238	4368494
Total	5413059	4586941	10000000

The "Fit Statistics" table in Output 18.2.2 shows the accuracy (92.85%) and the error (7.15%) of the model.

Output 18.2.2 Fit Statistics

Fit Statistics

Statistic	Training
Accuracy	0.9285
Error	0.0715
Sensitivity	0.9171
Specificity	0.9432

References

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. http://dx.doi.org/10.1007/s10589-008-9228-z.

Vapnik, V. N. (1995). *The Nature of Statistical Learning Theory*. New York: Springer-Verlag.

Chapter 19
The TEXTMINE Procedure

Contents

Overview: TEXTMINE Procedure **360**
PROC TEXTMINE Features 360
Using CAS Sessions and CAS Engine Librefs 361
Getting Started: TEXTMINE Procedure **361**
Syntax: TEXTMINE Procedure **365**
PROC TEXTMINE Statement 365
DOC_ID Statement 367
PARSE Statement 367
SAVESTATE Statement 371
SELECT Statement 371
SVD Statement 372
TARGET Statement 376
VARIABLES Statement 376
Details: TEXTMINE Procedure **377**
Natural Language Processing 377
Stemming 377
Part-of-Speech Tagging 377
Noun Group Extraction 378
Entity Identification 378
Multiword Terms Handling 381
Language Support 381
Term and Cell Weighting 381
Sparse Format 382
Coordinate List (COO) Format 382
Singular Value Decomposition 382
Applications in Text Mining 382
Computation 383
SVD-Only Mode 383
Topic Discovery 383
Output Data Tables 384
The OUTCHILD= Data Table 384
The OUTCONFIG= Data Table 384
The OUTDOCPRO= Data Table 385
The OUTPARENT= Data Table 385
The OUTPOS= Data Table 386
The OUTTERMS= Data Table 386

The OUTTOPICS= Data Table 387
System Configuration 388
Prerequisites for Running PROC TEXTMINE 388
Configuring for Language Binary Files 388
Deploying Language Binary Files on the Grid 388
Language Binary Files 388
The GRID_TEXTANALYTICS_BIN_LOC Macro 389
Examples: TEXTMINE Procedure **389**
Example 19.1: Parsing with No Options Turned On 389
Example 19.2: Parsing with Stemming 392
Example 19.3: Adding Entities and Noun Groups 394
Example 19.4: Adding Part-of-Speech Tagging 396
Example 19.5: Adding Synonyms 398
Example 19.6: Adding a Stop List 400
Example 19.7: Adding a Multiterm List 403
Example 19.8: Selecting Parts of Speech and Entities to Ignore 406

Overview: TEXTMINE Procedure

The TEXTMINE procedure integrates natural language processing and statistical analysis to analyze large-scale textual data in SAS Viya. PROC TEXTMINE supports a wide range of fundamental text analysis features, which include tokenizing, stemming, part-of-speech tagging, noun group extraction, default or customized stop lists and start lists, entity parsing, multiword tokens, synonym lists, term weighting, term-by-document matrix creation, dimension reduction with singular value decomposition (SVD), and topic discovery.

PROC TEXTMINE Features

The TEXTMINE procedure processes large-scale textual data in parallel in order to achieve efficiency and scalability. The following list summarizes the basic features of PROC TEXTMINE:

- Functionalities that are related to document parsing, term-by-document matrix creation, and dimension reduction are integrated into one procedure in order to process data more efficiently.
- Parsing supports essential natural language processing (NLP) features, which include tokenizing, stemming, part-of-speech tagging, noun group extraction, default or customized stop lists and start lists, entity parsing, multiword tokens, and synonym lists.
- Term weighting and filtering are supported for term-by-document matrix creation.
- Parsing and term-by-document matrix creation are processed in parallel.
- Computation of singular value decomposition (SVD) is parallelized.

- Topic discovery is integrated into the procedure.
- All phases of processing use a high degree of multithreading.

Using CAS Sessions and CAS Engine Librefs

SAS Cloud Analytic Services (CAS) is the analytic server and associated cloud services in SAS Viya. This section describes how to create a CAS session and set up a CAS engine libref that you can use to connect to the CAS session. It assumes that you have a CAS server already available; contact your system administrator if you need help starting and terminating a server. This CAS server is identified by specifying the host on which it runs and the port on which it listens for communications. To simplify your interactions with this CAS server, the host information and port information for the server are stored as SAS option values that are retrieved automatically whenever this CAS server needs to be accessed. You can examine the host and port values for the server at your site by using the following statements:

```
proc options option=(CASHOST CASPORT);
run;
```

In addition to starting a CAS server, your system administrator might also have created a CAS session and a CAS engine libref for your use. You can define your own sessions and CAS engine librefs that connect to the CAS server as shown in the following statements:

```
cas mysess;
libname mycas cas sessref=mysess;
```

The CAS statement creates the CAS session named mysess, and the LIBNAME statement creates the mycas CAS engine libref that you use to connect to this session. It is not necessary to explicitly name the CASHOST and CASPORT of the CAS server in the CAS statement, because these values are retrieved from the corresponding SAS option values.

If you have created the mysess session, you can terminate it by using the TERMINATE option in the CAS statement as follows:

```
cas mysess terminate;
```

For more information about the CAS and LIBNAME statements, see the section "Introduction to Shared Concepts" on page 11 in Chapter 3, "Shared Concepts."

Getting Started: TEXTMINE Procedure

The input data must be a table on your CAS server, and a CAS session must be set up. For more information, see the sections "Using CAS Sessions and CAS Engine Librefs" on page 11 and "Loading a SAS Data Set onto a CAS Server" on page 12 in Chapter 3, "Shared Concepts."

The following DATA step creates the getstart data table, which contains 16 observations that have two variables, in your CAS session. The text variable contains the input documents, and the did variable contains the ID of the documents. Each row in the data table represents a document for analysis.

```
data mycas.getstart;
    infile datalines delimiter='|' missover;
    length text $150;
    input text$ did;
    datalines;
      Reduces the cost of maintenance. Improves revenue forecast.        | 1
      Analytics holds the key to unlocking big data.                     | 2
      The cost of updates between different environments is eliminated.  | 3
      Ensures easy deployment in the cloud or on-site.                   | 4
      Organizations are turning to SAS for business analytics.           | 5
      This removes concerns about maintenance and hidden costs.          | 6
      Service-oriented and cloud-ready for many cloud infrastructures.   | 7
      Easily apply machine learning and data mining techniques to data.  | 8
      SAS Viya will address data analysis, modeling and learning.        | 9
      Helps customers reduce cost and make better decisions faster.      | 10
      Simple, powerful architecture ensures easy deployment in the cloud.| 11
      SAS is helping industries glean insights from data.                | 12
      Solve complex business problems faster than ever.                  | 13
      Shatter the barriers associated with data volume with SAS Viya.    | 14
      Casual business users, data scientists and application developers. | 15
      Serves as the basis for innovation causing revenue growth.         | 16
run;
```

These statements assume that your CAS engine libref is named mycas, but you can substitute any appropriately defined CAS engine libref.

The following DATA step uses the default stop list to eliminate noisy, noninformative terms:

```
data mycas.engstop;
    set sashelp.engstop;
run;
```

The following statements parse the input collection and use singular value decomposition followed by a rotation to discover topics that exist in the sample collection. The statements specify that only the terms that appear at least twice in the document collection are to be kept for generating the term-by-document matrix. The summary information about the terms in the document collection is stored in a data table named mycas.terms. The SVD statement requests that the first three singular values and singular vectors be computed. The topic assignments of the documents are stored in a data table named mycas.docpro, and the descriptive terms that define each topic are stored in a data table named mycas.topics.

```
proc textmine data=mycas.getstart;
doc_id      did;
variables   text;
parse
            outterms  = mycas.terms
            reducef   = 1
            stop      = mycas.engstop;
svd
            k         = 3
            outdocpro = mycas.docpro
            outtopics = mycas.topics
            numLabels = 4;
run;
```

The output from this analysis is presented in Figure 19.2, Figure 19.3 and Figure 19.4.

Figure 19.1 shows the SAS log that is generated by PROC TEXTMINE; the log provides information about the default configurations used by the procedure and about the input and output files. The log shows that the mycas.terms data table contains 24 observations. This means that the TEXTMINE procedure identified 24 individual terms in the input document collection. Because K=3 in the SVD statement, the mycas.docpro data table contains four variables: the first variable is the document ID, and the remaining three variables are obtained by projecting the original document onto the three left-singular vectors that have been rotated with the default orthogonal (varimax) rotation. The small collection and the use of the default cutoff threshold causes the values in the table to all be either 0 or 1.

Figure 19.1 SAS Log

```
NOTE: Stemming will be used in parsing.
NOTE: Tagging will be used in parsing.
NOTE: Noun groups will be used in parsing.
NOTE: No TERMWGT option is specified. TERMWGT=ENTROPY will be run by default.
NOTE: No CELLWGT option is specified. CELLWGT=LOG will be run by default.
NOTE: No ENTITIES option is specified. ENTITIES=NONE will be run by default.
NOTE: The dense SVD solver was used for this calculation.
NOTE: The Cloud Analytic Services server processed the request in 1.679806
      seconds.
NOTE: The data set MYCAS.TERMS has 129 observations and 11 variables.
NOTE: The data set MYCAS.DOCPRO has 16 observations and 4 variables.
NOTE: The data set MYCAS.TOPICS has 3 observations and 3 variables.
```

The following statements use PROC PRINT in Base SAS to show the contents of the first 10 rows of the sorted mycas.docpro data table that is generated by the TEXTMINE procedure:

```
data docpro;
   set mycas.docpro;
run;
proc sort data=docpro;
   by did;
run;
proc print data = docpro (obs=10);
run;
```

Figure 19.2 shows the output of PROC PRINT. For information about the output of the OUTDOCPRO= option, see the section "The OUTDOCPRO= Data Table" on page 385.

Figure 19.2 The mycas.docpro Data Table

Obs	did	COL1	COL2	COL3
1	1	0	0	1
2	2	0.6340795882	0	0.7732677905
3	3	0	0	0
4	4	0	1	0
5	5	0.8099208183	0	0.5865392297
6	6	0	0	0
7	7	0	1	0
8	8	0.6340795882	0	0.7732677905
9	9	0.1727144191	0	0.9849719435
10	10	0	0	1

The following statements use a DATA step and PROC PRINT to show the contents of the mycas.topics data table that is generated by the TEXTMINE procedure:

```
data topics; set mycas.topics; run;
proc print data = topics;
run;
```

Figure 19.3 shows the output of PROC PRINT. The three discovered topics are listed with four descriptive terms to characterize each topic.

Figure 19.3 The mycas.topics Data Table

Obs	_topicid	_name	_termCutOff
1	1	+application developer, +data scientist, +business user, application	0.145
2	2	easy deployment, deployment, +ensure, easy	0.139
3	3	+reduce, +help, +better decision, +decision	0.152

The following statements use a DATA step and the SORT and PRINT procedures to show the first 10 observations of the mycas.terms data table that is generated by the TEXTMINE procedure:

```
data terms; set mycas.terms; run;
proc sort data = terms; by key; run;
proc print data = terms (obs=10);
var term role freq numdocs key parent;
run;
```

Figure 19.4 shows the output of PROC PRINT, which provides details about the terms that are identified by the TEXTMINE procedure. Only the values of the variables term, role, freq, numdocs, key, and parent are displayed. For example, the output shows that the key of "ensures" is 19 and its parent's key is 4, which is the term "ensure." The TEXTMINE procedure also identified that "sas" is a proper noun and that "easy deployment" is a noun group. For information about the output of the OUTTERMS= option, see the section "The OUTTERMS= Data Table" on page 386.

Figure 19.4 The mycas.terms Data Table

Obs	Term	Role	Freq	numdocs	Key	Parent
1	simple	A	1	1	1	.
2	revenue forecast	nlpNounGroup	1	1	2	.
3	technique	N	1	1	3	.
4	different environment	nlpNounGroup	1	1	4	.
5	decision	N	1	1	5	.
6	cloud infrastructure	nlpNounGroup	1	1	6	.
7	hold	V	1	1	7	.
8	application developer	nlpNounGroup	1	1	8	.
9	analysis	N	1	1	9	.
10	analytics	N	2	2	10	.

Syntax: TEXTMINE Procedure

The following statements are available in the TEXTMINE procedure:

PROC TEXTMINE DATA=*CAS-libref.data-table* < *options* > **;**
VARIABLES *variable* **;**
TARGET *variable* **;**
DOC_ID *variable* **;**
PARSE < *parse-options* > **;**
SELECT *label-list* /< **GROUP=***group-option* > **KEEP** | **IGNORE** **;**
SVD < *svd-options* > **;**
SAVESTATE RSTORE=*CAS-libref.data-model* **;**

The PROC TEXTMINE statement, the VARIABLES statement, and the DOC_ID statement are required.

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

PROC TEXTMINE Statement

PROC TEXTMINE DATA=*CAS-libref.data-table* < *options* > **;**

The PROC TEXTMINE statement invokes the procedure. Table 19.1 summarizes the *options* in the statement by function. The *options* are then described fully in alphabetical order.

Table 19.1 PROC TEXTMINE Statement Options

option	**Description**
Basic Options	
DATA | DOC=	Specifies the input document data table
LANGUAGE=	Specifies the language that the input data table of documents uses

Table 19.1 *continued*

Option	Description
NEWVARNAMES	Specifies that the new-style variable names should be used on tables
Multithreading Options	
NTHREADS=	Specifies number of threads

You must specify the following option:

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

CAS-libref refers to a collection of information that is defined in the LIBNAME statement and includes the caslib, which includes a path to the data, and a session identifier, which defaults to the active session but which can be explicitly defined in the LIBNAME statement. For more information about *CAS-libref*, see the section "Using CAS Sessions and CAS Engine Librefs" on page 361.

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

Each row of the input data table must contain one text variable and one ID variable that correspond to the text and the unique ID of a document, respectively.

When you specify the SVD statement but not the PARSE statement, PROC TEXTMINE runs in SVD-only mode. In this mode, the DATA= option names the input SAS data table that contains the term-by-document matrix that is generated by the OUTPARENT= option in the PARSE statement.

You can also specify the following *options*:

LANGUAGE=*language*
names the language that is used by the documents in the input SAS data table. Languages supported in the current release are Arabic, Chinese, Croatian, Czech, Danish, Dutch, English, Finnish, French, German, Greek, Hebrew, Indonesian, Italian, Japanese, Korean, Norwegian, Polish, Portuguese, Russian, Slovak, Slovene, Spanish, Swedish, Thai, Turkish and Vietnamese. By default, LANGUAGE=ENGLISH.

NEWVARNAMES
adds leading and trailing blanks to variable names in the input and output tables.

NTHREADS=*nthreads*
specifies the number of threads to be used. By default, the number of threads is the same as the number of CPUs on the CAS server.

DOC_ID Statement

DOC_ID *variable* **;**

The DOC_ID statement specifies the *variable* that contains the ID of each document. In the input data table, each row corresponds to one document. The ID of each document must be unique; it can be either a number or a string of characters.

PARSE Statement

PARSE < *parse-options* > **;**

The PARSE statement specifies the options for parsing the input documents and creating the term-by-document matrix. Table 19.2 summarizes the *parse-options* in the statement by function. The *parse-options* are then described fully in alphabetical order.

Table 19.2 PARSE Statement Options

parse-option	**Description**
Parsing Options	
ENTITIES=	Specifies whether to extract entities in parsing
MULTITERM=	Specifies the multiword term list
NONOUNGROUPS | NONG	Suppresses noun group extraction in parsing
NOSTEMMING	Suppresses stemming in parsing
NOTAGGING	Suppresses part-of-speech tagging in parsing
SHOWDROPPEDTERMS=	Includes dropped terms in the OUTTERMS= data table
START=	Specifies the start list
STOP=	Specifies the stop list
SYNONYM | SYN=	Specifies the synonym list
Term-by-Document Matrix Creation Options	
CELLWGT=	Specifies how cells are weighted
REDUCEF=	Specifies the frequency for term filtering
TERMWGT=	Specifies how terms are weighted
Output Options	
OUTCHILD=	Specifies the data table to contain the raw term-by-document matrix. All kept terms, whether or not they are child terms, are represented in this data table along with their corresponding frequency.
OUTCONFIG=	Specifies the data table to contain the option settings that PROC TEXTMINE uses in the current run
OUTPARENT=	Specifies the data table to contain the term-by-document matrix. Child terms are not represented in this data table. The frequencies of child terms are attributed to their corresponding parents.
OUTTERMS=	Specifies the data table to contain the summary information about the terms in the document collection
OUTPOS=	Specifies the data table to contain the position information about the child terms' occurrences in the document collection

You can specify the following *parse-options*.

CELLWGT=LOG | NONE
specifies how the elements in the term-by-document matrix are weighted. You can specify the following values:

LOG weights cells by using the log formulation. For information about the log formulation for cell weighting, see the section "Term and Cell Weighting" on page 381.

NONE specifies that no cell weight be applied.

ENTITIES=STD | NONE
determines whether to use the standard LITI file for entity extraction. You can specify the following values:

STD uses the standard LITI file for entity extraction. A term such as "George W. Bush" is recognized as an entity and given the corresponding entity role and attribute. For this term, the entity role is PERSON and the attribute is Entity. Although the entity is treated as the single term, "george w. bush," the individual tokens "george," "w," and "bush" are also included.

NONE does not use the standard LITI file for entity extraction.

By default, ENTITIES=NONE.

MULTITERM=*CAS-libref.data-table*
specifies the input SAS data table that contains a list of multiword terms. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 361. The multiword terms are case-sensitive and are treated as a single entry by the TEXTMINE procedure. Thus, the terms "Thank You" and "thank you" are processed differently. Consequently, you must convert all text strings to lowercase or add each of the multiterm's case variations to the list before using the TEXTMINE procedure to create consistent multiword terms. The multiterm data table must have a variable Multiterm and each of its values must be formatted in the following manner:

```
multiterm:  3:  pos
```

Specifically, the first item is the multiword term itself followed by a colon, the second item is a number that represents the token type followed by a colon, and the third item is the part of speech that the multiword term represents. **NOTE:** The token type 3 is the most common token type for multiterm lists; it represents compound words.

NONOUNGROUPS
NONG
suppresses standard noun group extraction. By default, the TEXTMINE procedure extracts noun groups, returns noun phrases without determiners or prepositions, and (unless the NOSTEMMING option is specified) stems noun group elements.

NOSTEMMING
suppresses stemming of words. By default, words are stemmed; that is, terms such as "advises" and "advising" are mapped to the parent term "advise." The TEXTMINE procedure uses dictionary-based stemming (also known as lemmatization).

NOTAGGING
suppresses tagging of terms. By default, terms are tagged and the TEXTMINE procedure identifies a term's part of speech based on context clues. The identified part of speech is provided in the Role variable of the OUTTERMS= data table.

OUTCHILD=*CAS-libref.data-table*
specifies the output data table to contain a compressed representation of the sparse term-by-document matrix. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 361. The term counts are not weighted. The data table saves only the kept, representative terms. The child frequencies are not attributed to their corresponding parent (as they are in the OUTPARENT= data table). For more information about the compressed representation of the sparse term-by-document matrix, see the section "The OUTCHILD= Data Table" on page 384.

OUTCONFIG=*CAS-libref.data-table*
specifies the output data table to contain configuration information that is used for the current run of PROC TEXTMINE. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 361. The primary purpose of this data table is to relay the configuration information from the TEXTMINE procedure to the TMSCORE procedure. The TMSCORE procedure uses options that are consistent with the TEXTMINE procedure. Thus, the data table that is created by using the OUTCONFIG= option becomes an input data table for PROC TMSCORE and ensures that the parsing options are consistent between the two runs. For more information about this data table, see the section "The OUTCONFIG= Data Table" on page 384.

OUTPARENT=*CAS-libref.data-table*
specifies the output data table to contain a compressed representation of the sparse term-by-document matrix. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 361. The term counts can be weighted, if requested. The data table contains only the kept, representative terms, and the child frequencies are attributed to the corresponding parent. To obtain information about the children, use the OUTCHILD= option. For more information about the compressed representation of the sparse term-by-document matrix, see the section "The OUTPARENT= Data Table" on page 385.

OUTPOS=*CAS-libref.data-table*
specifies the output data table to contain the position information about the child terms' occurrences in the document collection. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 361. For more information about this data table, see the section "The OUTPOS= Data Table" on page 386.

OUTTERMS=*CAS-libref.data-table*

specifies the output data table to contain the summary information about the terms in the document collection. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 361. For more information about this data table, see the section "Output Data Tables" on page 384.

REDUCEF=*n*

removes terms that are not in at least *n* documents. The value of *n* must be a positive integer. By default, REDUCEF=4.

SHOWDROPPEDTERMS

includes the terms that have a keep status of N in the OUTTERMS= data table and the OUTCHILD= data table.

START=*CAS-libref.data-table*

specifies the input data table that contains the terms that are to be kept for the analysis. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 361. These terms are displayed in the OUTTERMS= data table with a keep status of Y. All other terms are displayed with a keep status of N if the SHOWDROPPEDTERMS option is specified or not displayed if the SHOWDROPPEDTERMS option is not specified. The START= data table must have a Term variable and can also have a Role variable. You cannot specify both the START= and STOP= options.

STOP=*CAS-libref.data-table*

specifies the input data table that contains the terms to exclude from the analysis. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 361. These terms are displayed in the OUTTERMS= data table with a keep status of N if the SHOWDROPPEDTERMS option is specified. The terms are not identified as parents or children. The STOP= data table must have a Term variable and can also have a Role variable. You cannot specify both the START= and STOP= options.

SYNONYM=*CAS-libref.data-table*

SYN=*CAS-libref.data-table*

specifies the input data table that contains user-defined synonyms to be used in the analysis. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 361. The data table specifies parent-child relationships that enable you to map child terms to a representative parent. The synonym relationship is indicated in the data table that is specified in the OUTTERMS= option and is also reflected in the term-by-document data table that is specified in the OUTPARENT= option. The input synonym data table must have either the two variables Term and Parent or the four variables Term, Parent, Termrole, and Parentrole. This data table overrides any relationships that are identified when terms are stemmed. (Terms are stemmed by default; you can suppress stemming by specifying the NOSTEMMING option.)

TERMWGT=ENTROPY | MI | NONE

specifies how terms are weighted. You can specify the following values:

ENTROPY	uses the entropy formulation to weight terms.
MI	uses the mutual information formulation to weight terms (you must also specify the TARGET statement).
NONE	requests that no term weight be applied.

For more information about the entropy formulation and the mutual information formulation for term weighting, see the section "Term and Cell Weighting" on page 381.

SAVESTATE Statement

SAVESTATE RSTORE=*CAS-libref.data-model* **;**

The SAVESTATE statement saves a text mining model to a binary object contained in a data table. The object is referred to as the analytic store and contains the necessary information for scoring a text mining model by the ASTORE procedure. Only complete text models consisting of both parsing and document projections can be saved to the analytic store by the TEXTMINE procedure. The SELECT statement is not currently supported in an analytic store.

You must specify the following option:

RSTORE=*CAS-libref.data-model*

specifies a data table in which to save the text mining model. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 361.

SELECT Statement

SELECT *label-list* /< **GROUP=***group-option* > **KEEP | IGNORE ;**

The SELECT statement enables you to specify the parts of speech or entities or attributes that you want to include in or exclude from your analysis. Exclusion by the SELECT statement is different from exclusion that is indicated by the _keep variable in the OUTTERMS= data table. Terms that are excluded by the SELECT statement cannot be included in the OUTTERMS= data table, whereas terms that have _keep=N can be included in the OUTTERMS= data table if the SHOWDROPPEDTERMS option is specified. Terms excluded by the SELECT statement are excluded from the OUTPOS= data table, but terms that have _keep=N are included in OUTPOS= data table. Table 19.3 summarizes the options you can specify in the SELECT statement. The options are then described fully in syntactic order.

Table 19.3 SELECT Statement Options

Option	Description
label-list	Specifies one or more labels of terms that are to be ignored or kept in your analysis
GROUP=	Specifies whether the labels are parts of speech, entities, or attributes
IGNORE	Ignores terms whose labels are specified in the *label-list*
KEEP	Keeps terms whose labels are specified in the *label-list*

You must specify a *label-list* and either the IGNORE or KEEP option:

label-list
specifies one or more labels that are either parts of speech or entities or attributes. Each label must be surrounded by double quotation marks and separated by spaces from other labels. Labels are case-insensitive. Terms that have these labels are either ignored during parsing (when the IGNORE option is specified) or kept in the parsing results in the OUTPOS= and OUTTERMS= data tables (when the KEEP option is specified). Table 19.5 shows all possible part-of-speech tags. Table 19.6 shows all valid English entities. The attribute variable in Table 19.12 shows all possible attributes.

IGNORE
ignores during parsing all terms whose labels are specified in the *label-list*, but keeps all other terms in the parsing results (the OUTPOS= and OUTTERMS= data tables).

KEEP
keeps in the parsing results (the OUTPOS= and OUTTERMS= data tables) only the terms whose labels are specified in the *label-list*.

You can also specify the following option:

GROUP="ATTRIBUTES" | "ENTITIES" | "POS"
specifies whether the labels are attributes, entities, or parts of speech. The group type must be surrounded by double quotation marks and is case-insensitive. All labels that are specified in the *label-list* in the same SELECT statement should belong to the specified group. If you need to select labels from more than one group, you can use multiple SELECT statements (one for each group that you need to select from). You cannot specify multiple SELECT statements for the same group. By default, Num and Punct in the "ATTRIBUTES" group are ignored, but this default is overridden by a SELECT statement that specifies GROUP="ATTRIBUTES". By default, GROUP="POS".

SVD Statement

SVD < *svd-options* > **;**

The SVD statement specifies the options for calculating a truncated singular value decomposition (SVD) of the large, sparse term-by-document matrix that is created during the parsing phase of PROC TEXTMINE. Table 19.4 summarizes the *svd-options* in the statement by function. The *svd-options* are then described fully in alphabetical order.

Table 19.4 SVD Statement Options

svd-option	**Description**
Input Options	
COL=	Specifies the column variable, which contains the column indices of the term-by-document matrix, which is stored in coordinate list (COO) format
ROW=	Specifies the row variable, which contains the row indices of the term-by-document matrix, which is stored in COO format
ENTRY=	Specifies the entry variable, which contains the entries of the term-by-document matrix, which is stored in COO format
SVD Computation Options	
K=	Specifies the number of dimensions to be extracted
MAX_K=	Specifies the maximum number of dimensions to be extracted
TOL=	Specifies the maximum allowable tolerance for the singular value
RESOLUTION | RES=	Specifies the recommended number of dimensions (resolution) to be extracted by SVD, when the MAX_K= option is specified
Topic Discovery Options	
NUMLABELS=	Specifies the number of terms to be used in the descriptive label for each topic
ROTATION=	Specifies the type of rotation to be used for topic discovery
IN_TERMS=	Specifies the data table that contains the terms for topic discovery in SVD-only mode
EXACTWEIGHT	Prevents rounding of the topic weights
NOCUTOFFS	Prevents setting term weights to 0 when they are below the threshold
Output Options	
SVDU=	Specifies the U matrix, which contains the left singular vectors
SVDV=	Specifies the V matrix, which contains the right singular vectors
SVDS=	Specifies the S matrix, whose diagonal elements are the singular values
OUTDOCPRO=	Specifies the data table to contain the projections of the documents
OUTTOPICS=	Specifies the data table to contain the topics that have been discovered

You can specify the following *svd-options*:

COL=*variable*
specifies the *variable* that contains the column indices of the term-by-document matrix. You must specify this option when you run PROC TEXTMINE in SVD-only mode (that is, when you specify the SVD statement but not the PARSE statement).

ENTRY=*variable*
specifies the *variable* that contains the entries of the term-by-document matrix. You must specify this option when you run PROC TEXTMINE in SVD-only mode (that is, when you specify the SVD statement but not the PARSE statement).

EXACTWEIGHT
requests that the weights aggregated during topic derivation not be rounded. By default, the calculated weights are rounded to the nearest 0.001.

IN_TERMS=*CAS-libref.data-table*
specifies the input data table that contains information about the terms in the document collection. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 361. The data table should have the variables that are described in Table 19.12. The terms are required to generate topic names in the OUTTOPICS= data table. This option is only for topic discovery in SVD-only mode. This option conflicts with the PARSE statement, and only one of the two can be specified. If you want to run SVD-only mode without topic discovery, then you do not need to specify this option.

K=*k*
specifies the number of columns in the matrices **U**, **V**, and **S**. This value is the number of dimensions of the data table after SVD is performed. If the value of *k* is too large, then the TEXTMINE procedure runs for an unnecessarily long time. This option takes precedence over the MAX_K= option. This option also controls the number of topics that are extracted from the text corpus when the ROTATION= option is specified.

MAX_K=*n*
specifies the maximum value that the TEXTMINE procedure should return as the recommended value of *k* (the number of columns in the matrices **U**, **V**, and **S**) when the RESOLUTION= option is specified (to recommend the value of *k*). The TEXTMINE procedure attempts to calculate *k* dimensions (as opposed to recommending it) when it performs SVD. This option is ignored if the K= option has been specified. This option also controls the number of topics that are extracted from the text corpus when the ROTATION= option is specified.

NOCUTOFFS
uses all weights in the **U** matrix to form the document projections. When topics are requested, weights below the term cutoff (as calculated in the OUTTOPICS= data table) are set to 0 before the projection is formed.

NUMLABELS=*n*
specifies the number of terms to use in the descriptive label for each topic. The descriptive label provides a quick synopsis of the discovered topics. The labels are stored in the OUTTOPICS= data table. By default, NUMLABELS=5.

OUTDOCPRO=*CAS-libref.data-table* **<KEEPVARIABLES=***variable-list***><NONORMDOC>**
OUTDOCPRO=*CAS-libref.data-table* **<KEEPVARS=***variable-list***><NONORMDOC>**
specifies the output data table to contain the projections of the columns of the term-by-document matrix onto the columns of **U**. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 361. Because each column of the term-by-document matrix corresponds to a document, the output forms a new representation of the input documents in a space that has much lower dimensionality.

You can copy the variables from the data table that is specified in the DATA= option in the PROC TEXTMINE statement to the data table that is specified in this option. You can specify the following suboptions:

KEEPVARIABLES=*variable-list*
attaches the content of the variables that are specified in the *variable-list* to the output. These variables must appear in the data table that is specified in the DATA= option in the PROC TEXTMINE statement.

NONORMDOC
suppresses normalization of the columns that contain the projections of documents to have a unit norm.

OUTTOPICS=*CAS-libref.data-table*
specifies the output data table to contain the topics that are discovered. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 361.

RESOLUTION=LOW | MED | HIGH
RES=LOW | MED | HIGH
specifies how to calculate the recommended number of dimensions (resolution) for the singular value decomposition. If you specify this option, you must also specify the MAX_K= option. A low-resolution singular value decomposition returns fewer dimensions than a high-resolution singular value decomposition. This option recommends the value of k (the number of columns in the matrices $\mathbf{U}$, $\mathbf{V}$, and $\mathbf{S}$) heuristically based on the value specified in the MAX_K= option. Assume that the MAX_K= option is set to n and a singular value decomposition that has n dimensions accounts for $t\%$ of the total variance. You can specify the following values:

HIGH always recommends the maximum number of dimensions; that is, $k = n$.

MED recommends a k that explains $(5/6) * t\%$ of the total variance.

LOW recommends a k that explains $(2/3) * t\%$ of the total variance.

By default, RESOLUTION=HIGH.

ROTATION=VARIMAX | PROMAX
specifies the type of rotation to be used in order to maximize the explanatory power of each topic. You can specify the following values:

PROMAX does an oblique rotation on the original left singular vectors and generates topics that might be correlated.

VARIMAX does an orthogonal rotation on the original left singular vectors and generates uncorrelated topics.

By default, ROTATION=VARIMAX.

ROW=*variable*
specifies the *variable* that contains the row indices of the term-by-document matrix. You must specify this option when you run PROC TEXTMINE in SVD-only mode (that is, when you specify the SVD statement but not the PARSE statement).

SVDS=*CAS-libref.data-table*
specifies the output data table to contain the calculated singular values. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 361.

SVDU=*CAS-libref.data-table*
specifies the data table to contain the calculated left singular vectors. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 361.

SVDV=*CAS-libref.data-table*
specifies the data table to contain the calculated right singular vectors. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 361.

TOL=ϵ
specifies the maximum allowable tolerance for the singular value. Let $\mathbf{A}$ be a matrix. Suppose λ_i is the ith singular value of $\mathbf{A}$ and ξ_i is the corresponding right singular vector. The SVD computation terminates when for all $i \in \{1, \ldots, k\}$, λ_i and $_{0i}$ satisfy $\|\mathbf{A}^\top\mathbf{A}\xi - \lambda\xi\|^2 \leq \epsilon$. The default value of ϵ is 10^{-6}, which is more than adequate for most text mining problems.

TARGET Statement

TARGET *variable* **;**

This statement specifies the *variable* that contains the information about the category that a document belongs to. The target *variable* can be any nominal or ordinal variable; it is used in calculating mutual information term weighting.

VARIABLES Statement

VARIABLES *variable* **;**

VAR *variable* **;**

This statement specifies the *variable* that contains the text to be processed.

Details: TEXTMINE Procedure

Natural Language Processing

Natural language processing (NLP) techniques can be used to extracting meaningful information from natural language input. The following sections describe features from SAS linguistic technologies that the TEXTMINE procedure implements to support natural language processing.

Stemming

Stemming (a special case of morphological analysis) identifies the possible root form of an inflected word. For example, the word "talk" is the stem of the words "talk," "talks," "talking," and "talked." In this case "talk" is the parent, and "talk," "talks," "talking," and "talked" are its children. The TEXTMINE procedure uses dictionary-based stemming (also known as lemmatization), which unlike tail-chopping stemmers, produces only valid words as stems. When part-of-speech tagging is on (that is, the NOTAGGING option is not specified), the stem selection process restricts the stem to be of the same part-of-speech as the original term.

Part-of-Speech Tagging

Part-of-speech tagging uses SAS linguistic technologies to identify or disambiguate the grammatical category of a word by analyzing it within its context. For example:

```
I like to bank at the local branch of my bank.
```

In this case, the first "bank" is tagged as a verb (V), and the second "bank" is tagged as a noun (N). Table 19.5 shows all possible part-of-speech tags.

Table 19.5 All Part-of-Speech Tags

Part-of-Speech Tag	Description
ABBR	Abbreviation
ADJ	Adjective
ADV	Adverb
AUX	Auxiliary or modal term
CONJ	Conjunction
DET	Determiner
INTERJ	Interjection
NOUN	Noun
NUM	Number or numeric expression
PART	Infinitive marker, negative participle, or possessive marker
PREF (Korean only)	Prefix
PREP	Preposition
PRON	Pronoun
PROP	Proper noun
PUNCT	Punctuation
VERB	Verb
VERBADJ	Verbal adjective

Noun Group Extraction

Noun groups provide more relevant information than simple nouns. A noun group is defined as a sequence of nouns and their modifiers. Noun group extraction uses part-of-speech tagging to identify nouns and their adjacent noun and adjective modifiers that together form a noun group. Examples of noun groups are "weeklong cruises" and "Middle Eastern languages."

Entity Identification

Entity identification uses SAS linguistic technologies to classify sequences of words into predefined classes. These classes are assigned as roles for the corresponding sequences. For example, "Person," "Location," "Company," and "Measurement" are identified as classes for "George W. Bush," "Boston," "SAS Institute," "2.5 inches," respectively. Table 19.6 shows all valid entities for English. Not all languages support all entities. Table 19.7, Table 19.8, and Table 19.9 indicate the languages that are available for each entity.

Table 19.6 All Valid English Entities

Entities	Description
ADDRESS	Postal address or number and street name
COMPANY	Company name
CURRENCY	Currency or currency expression
INTERNET	Email address or URL
LOCATION	City, county, state, political, or geographical place or region
MEASURE	Measurement or measurement expression
NOUN_GROUP	Phrases that contain multiple words
ORGANIZATION	Government, legal, or service agency
PERCENT	Percentage or percentage expression
PERSON	Person's name
PHONE	Telephone number
PROP_MISC	Proper noun with an ambiguous classification
SSN	Social Security number
TIME	Time or time expression
TIME_PERIOD	Measure of time expressions
TITLE	Person's title or position
VEHICLE	Motor vehicle, including color, year, make, and model

Table 19.7 Supported Language-Entity Pairs, Part 1

Language	Address	Company	Currency	Date	Internet	Location
Arabic	✓		✓	✓	✓	✓
Chinese	✓	✓	✓	✓	✓	✓
Croatian		✓	✓			✓
Czech			✓	✓		✓
Danish	✓	✓	✓	✓	✓	✓
Dutch	✓	✓	✓	✓	✓	✓
English	✓	✓	✓	✓	✓	✓

Table 19.7 *continued*

Language	Address	Company	Currency	Date	Internet	Location
Finnish		✓	✓	✓	✓	✓
French	✓	✓	✓	✓	✓	✓
German	✓	✓	✓	✓	✓	✓
Greek	✓	✓	✓	✓	✓	✓
Hebrew						
Indonesian		✓				
Italian	✓	✓	✓	✓	✓	✓
Japanese	✓		✓	✓	✓	✓
Korean	✓	✓	✓	✓	✓	✓
Norwegian						
Polish	✓	✓	✓	✓	✓	✓
Portuguese	✓	✓	✓	✓	✓	✓
Russian			✓	✓		✓
Slovak			✓	✓		✓
Slovene		✓				✓
Spanish	✓	✓	✓	✓	✓	✓
Swedish	✓	✓	✓	✓	✓	✓
Thai						✓
Turkish		✓				✓
Vietnamese				✓	✓	

Table 19.8 Supported Language-Entity Pairs, Part 2

Language	Measure	Noun_Group	Organization	Percent	Person	Phone
Arabic	✓	✓	✓	✓	✓	✓
Chinese	✓			✓	✓	✓
Croatian	✓	✓			✓	✓
Czech		✓	✓	✓	✓	
Danish	✓	✓	✓	✓	✓	✓
Dutch	✓	✓	✓	✓	✓	✓
English	✓	✓	✓	✓	✓	✓
Finnish		✓	✓		✓	
French	✓	✓	✓	✓	✓	✓
German	✓	✓	✓	✓	✓	✓
Greek	✓	✓	✓	✓	✓	✓
Hebrew		✓				
Indonesian		✓				
Italian	✓	✓		✓	✓	✓
Japanese	✓		✓	✓	✓	✓
Korean	✓		✓	✓	✓	✓
Norwegian		✓				
Polish	✓	✓	✓	✓	✓	✓
Portuguese	✓	✓		✓	✓	✓

Table 19.8 *continued*

Language	Measure	Noun_Group	Organization	Percent	Person	Phone
Russian		✓	✓	✓	✓	
Slovak		✓	✓	✓	✓	
Slovene	✓	✓	✓		✓	
Spanish	✓	✓	✓	✓	✓	✓
Swedish	✓	✓	✓	✓	✓	✓
Thai			✓		✓	
Turkish		✓	✓		✓	
Vietnamese						

Table 19.9 Supported Language-Entity Pairs, Part 3

Language	Prop_Misc	SSN	Time	Time_Period	Title	Vehicle
Arabic			✓	✓	✓	
Chinese	✓		✓	✓	✓	
Croatian					✓	
Czech			✓			
Danish	✓	✓	✓	✓	✓	✓
Dutch	✓	✓	✓	✓	✓	✓
English	✓	✓	✓	✓	✓	✓
Finnish			✓			
French	✓	✓	✓	✓	✓	
German	✓	✓	✓	✓	✓	
Greek	✓		✓	✓		
Hebrew						
Indonesian						
Italian	✓	✓	✓	✓	✓	
Japanese			✓	✓	✓	
Korean			✓	✓	✓	
Norwegian						
Polish	✓		✓	✓	✓	
Portuguese	✓	✓	✓	✓	✓	
Russian			✓			
Slovak			✓			
Slovene	✓					✓
Spanish	✓	✓	✓	✓	✓	
Swedish	✓	✓	✓	✓	✓	✓
Thai						
Turkish						
Vietnamese	✓		✓	✓		

Multiword Terms Handling

By default, SAS linguistic technologies tokenize the text to individual words and operate at the word level. Multiword terms provide a control that enables you to specify sequences of words to be interpreted as individual units. For example, "greater than," "in spite of," and "as well as" can be defined as multiword terms.

Language Support

Languages supported in the current release are Arabic, Chinese, Croatian, Czech, Danish, Dutch, English, Finnish, French, German, Greek, Hebrew, Indonesian, Italian, Japanese, Korean, Norwegian, Polish, Portuguese, Russian, Slovak, Slovene, Spanish, Swedish, Thai, Turkish and Vietnamese. By turning off some of the advanced parsing functionality, you might be able to use PROC TEXTMINE effectively with other space-delimited languages.

Term and Cell Weighting

The TERMWGT= option and the CELLWGT= option control how to weight the frequencies in the compressed term-by-document matrix. The term weight is a positive number that is assigned to each term based on the distribution of that term in the document collection. This weight can be interpreted as an indication of the importance of that term to the document collection. The cell weight is a function that is applied to every entry in the term-by-document matrix; it moderates the effect of a term that is repeated within a document.

Let $f_{i,j}$ be the entry in the ith row and jth column of the term-by-document matrix, which indicates the time of appearance of term i in document j. Assuming that the term weight of term i is w_i and the cell weight function is $g(x)$, the weighted frequency of each entry in the term-by-document matrix is given by $w_i \times g(f_{i,j})$.

When the CELLWGT=LOG option is specified, the following equation is used to weight cells:

$$g(x) = \log_2(f_{i,j} + 1)$$

The equation reduces the influence of highly frequent terms by applying the log function.

When the TERMWGT=ENTROPY option is specified, the following equation is used to weight terms:

$$w_i = 1 + \sum_j \frac{p_{i,j} \log_2(p_{i,j})}{\log_2(n)}$$

In this equation, n is the number of documents, and $p_{i,j}$ is the probability that term i appears in document j, which can be estimated by $p_{i,j} = \frac{f_{i,j}}{g_i}$, where g_i is the global term frequency for term i.

When the TERMWGT=MI option is specified, the following equation is used to weight terms:

$$w_i = \max_{C_k} \left(\log \left(\frac{P\left(t_i, C_k\right)}{P\left(t_i\right) P\left(C_k\right)} \right) \right)$$

In this equation, C_k is the set of documents that belong to category k, $P\left(C_k\right)$ is the percentage of documents that belong to category k, and $P\left(t_i, C_k\right)$ is the percentage of documents that contain term t_i and belong to category k. Let d_i be the number of documents that term i appears in. Then $P\left(t_i\right) = \frac{d_i}{n}$.

Sparse Format

A matrix is sparse when most of its elements are 0. The term-by-document matrix that the TEXTMINE procedure generates is a sparse matrix. To save storage space, the TEXTMINE procedure supports the COO format for storing a sparse matrix.

Coordinate List (COO) Format

The COO is also known as the transactional format. In this format, the matrix is represented as a set of triples $(i,\ j,\ x)$, where x is an entry in the matrix and i and j denote its row and column indices, respectively. When the transactional style is used, all 0 entries in the matrix are ignored in the output, thereby saving storing space when the matrix is sparse. The COO format is good for incremental matrix construction. For example, it is easy to add new rows and new columns to the matrix by inserting more tuples in the list.

Singular Value Decomposition

Singular value decomposition (SVD) of a matrix $\mathbf{A}$ factors $\mathbf{A}$ into three matrices such that $\mathbf{A} = \mathbf{U}\Sigma\mathbf{V}^{\top}$. Singular value decomposition also requires that the columns of $\mathbf{U}$ and $\mathbf{V}$ be orthogonal and that Σ be a real-valued diagonal matrix that contains monotonically decreasing, nonnegative entries. The entries of Σ are called singular values. The columns of $\mathbf{U}$ and $\mathbf{V}$ are called left and right singular vectors, respectively. A truncated singular value decomposition calculates only the first k singular values and their corresponding left and right singular vectors. In information retrieval, singular value decomposition of a term-by-document matrix is also known as latent semantic indexing (LSI).

Applications in Text Mining

Let $\mathbf{A} \in \mathbb{R}^{m \times n}$ be a term-by-document matrix, where m is the number of terms and n is the number of documents. The SVD statement has two main functions: to calculate a truncated singular value decomposition (SVD) of $\mathbf{A}$, and to project the columns of $\mathbf{A}$ onto the left singular vectors to generate a new representation of the documents that has a much lower dimensionality. The output of the SVD statement is a truncated singular value decomposition of $\mathbf{A}$, for which the parameter k defines how many singular values and singular vectors to compute. Singular value decomposition reduces the dimension of the term-by-document matrix and reveals themes that are present in the document collection.

In general, the value of k must be large enough to capture the meaning of the document collection, yet small enough to ignore the noise. You can specify this value explicitly in the K= option or accept a value that is recommended by the TEXTMINE procedure. A value between 50 and 200 should work well for a document collection that contains thousands of documents.

An important purpose of singular value decomposition is to reduce a high-dimensional term-by-document matrix into a low-dimensional representation that reveals information about the document collection. The columns of the $\mathbf{A}$ form the coordinates of the document space, and the rows form the coordinates of the term space. Each document in the collection is represented as a vector in m-dimensional space and each term as a vector in n-dimensional space. The singular value decomposition captures this same information by using a smaller number of basis vectors than would be necessary if you analyzed $\mathbf{A}$ directly.

For example, consider the columns of $\mathbf{A}$, which represent the document space. By construction, the columns of $\mathbf{U}$ also reside in m-dimensional space. If $\mathbf{U}$ has only one column, the line between that vector and the

origin would form the best fit line, in a least squares sense, to the original document space. If **U** has two columns, then these columns would form the best fit plane to the original document space. In general, the first k columns of **U** form the best fit k-dimensional subspace for the document space. Thus, you can project the columns of **A** onto the first k columns of **U** in order to optimally reduce the dimension of the document space from m to k.

The projection of a document $\mathbf{d}$ (one column of **A**) onto **U** results in k real numbers that are defined by the inner product $\mathbf{d}$ with each column of **U**. That is, $p_i = \mathbf{d}^\top \mathbf{u}_i$. With this representation, each document forms a k-dimensional vector that can be considered a theme in the document collection. You can then calculate the Euclidean distance between each document and each column of **U** to determine the documents that are described by this theme.

In a similar fashion, you can repeat the previous process by using the rows of **A** and the first k columns of **V**. This generates a best fit k-dimensional subspace for the term space. This representation is used to group terms into similar clusters. These clusters also represent concepts that are prevalent in the document collection. Thus, singular value decomposition can be used to cluster both the terms and the documents into meaningful representations of the entire document collection.

Computation

The computation of the singular vector decomposition is fully parallelized in PROC TEXTMINE via multithreading and distributed computing. In the current release, computing singular value decomposition requires the input data to contain at least 25 documents and at least as many documents as there are nodes in the grid. If p nodes are used for computing singular value decomposition in a distributed computing environment, then the input data must contain at least $\max(p, 25)$ documents. Computing singular value decomposition is an iterative process that involves considerable communication among the computer nodes in a distributed computing environment. Therefore, adding more computer nodes for computing singular value decomposition might not always improve efficiency. Conversely, when the data size is not large enough, adding too many computer nodes for computation might lead to a noticeable increase in communication time and sometimes might even slow down the overall computation.

SVD-Only Mode

If you run PROC TEXTMINE without a PARSE statement (called SVD-only mode), PROC TEXTMINE directly takes the term-by-document matrix as input and computes singular value decomposition (SVD). This functionality enables you to parse documents and compute the SVD separately in two procedure calls. This approach is useful when you want to try different parameters for SVD computation after document parsing. When you run PROC TEXTMINE in SVD-only mode, the DATA= option in the PROC TEXTMINE statement names the data table that contains the term-by-document matrix.

Topic Discovery

You can use the TEXTMINE procedure to discover topics that exist in your collection. In PROC TEXTMINE, topics are calculated as a "rotation" of the SVD dimensions in order to maximize the sum of squares of the term loadings in the V matrix. This rotation preserves the spatial information that the SVD provides, but it also allows the newly rotated SVD dimensions to become semantically interpretable. Topics are characterized by a set of weighted terms. Documents that contain many of these weighted terms are highly associated with the topic, and documents that contain few of them are less associated with the topic. The term scores are found in the **U** matrix that has been rotated to maximize the explanatory power of each topic. The columns

of the **V** matrix characterize the strength of the association of each document with each topic. Finally, the TEXTMINE procedure can output a topic table that contains the best set of descriptor terms for each topic.

Because topic discovery is derived from the **U** matrix of SVD (each column of the **U** matrix is rotated and corresponds to a topic), topic discovery options are specified in the SVD statement.

Output Data Tables

This section describes the output data tables that PROC TEXTMINE produces when you specify the corresponding option.

The OUTCHILD= Data Table

The OUTCHILD= option in the PARSE statement specifies the data table to contain a compressed representation of the sparse term-by-document matrix, which is usually very sparse. To save space, this matrix is stored in COO format.

If you do not specify the SHOWDROPPEDTERMS option in the PARSE statement, this data table saves only the kept terms.[1]

The child frequencies are not attributed to their corresponding parent (as they are in the data table specified in the OUTPARENT= option). Using the example in the previous section, the data table that is generated by the OUTCHILD= option will have two entries:

```
t1    d1    8
t2    d1    1
```

The term count of "said" in **d1** is not attributed to its parent, "say." The data table that is specified in the OUTCHILD= option can be combined with the data table that is specified in the OUTTERMS= option to construct the data table that is specified in the OUTPARENT= option.

When you specify the SHOWDROPPEDTERMS option in the PARSE statement, the data table saves all the terms that appear in the data table that is specified in the OUTTERMS= option in the PARSE statement.

The OUTCONFIG= Data Table

The OUTCONFIG= option in the PARSE statement specifies a SAS data table to contain the configuration that PROC TEXTMINE uses in the current run. The primary purpose of this data table is to relay the configuration information from the TEXTMINE procedure to the TMSCORE procedure so that the TMSCORE procedure can use options that are consistent with the TEXTMINE procedure during scoring.

Table 19.10 shows the configuration information that is contained in this data table.

Table 19.10 Variables in the OUTCONFIG= Data Table

Variable	Indicates
Language	Source language of the documents
Stemming	Whether stemming is used: "Y" indicates that stemming is used, and "N" indicates that it is not used

[1] Kept terms are terms that are marked as kept in the data table specified in the OUTTERMS= option in the PARSE statement.

Table 19.10 *continued*

Variable	Description
Tagging	Whether tagging is used: "Y" indicates that tagging is used, and "N" indicates that it is not used
NG	Whether noun grouping is used: "Y" indicates that noun grouping is used, and "N" indicates that it is not used
Entities	Whether entities should be extracted: "STD" indicates that entities should be extracted, and "N" indicates that entities should not be extracted. When the SELECT statement is specified, "K" indicates that entities are kept, and "D" indicates that entities are ignored.
Multiterm	The name of the multiterm SAS data table
Cellwgt	How the cells of the term-by-document matrix are weighted

The contents of this data table are case-sensitive.

The OUTDOCPRO= Data Table

The OUTDOCPRO= option in the SVD statement specifies a SAS data table to contain the projections of the columns of the term-by-document matrix onto the columns of U. Because each column of the term-by-document matrix corresponds to a document, the output forms a new representation of the input documents in a space that has much lower dimensionality. If the K= option in the SVD statement is set to k and the input data table contains n documents, the output will have n rows and $k + 1$ columns. Each row of the output corresponds to a document. The first column of the output contains the ID of the documents, and the name of the column is the same as the variable that is specified in the DOC_ID statement. The remaining k columns are the projections and are named "COL1" to "COLk."

The OUTPARENT= Data Table

The OUTPARENT= option in the PARSE statement specifies a SAS data table to contain a compressed representation of the sparse term-by-document matrix. The term-by-document matrix is usually very sparse.[2] To save space, this matrix is stored in COO format.

This data table contains three columns: _TERMNUM_, _DOCUMENT_, and _COUNT_. The _TERMNUM_ column contains the ID of the terms (which corresponds to the "Key" column of the data table that is generated by the OUTTERMS= option), the _DOCUMENT_ column contains the ID of the documents, and the _COUNT_ column contains the term counts. For example, (**`t1 d1 k`**) means that term **`t1`** appears **`k`** times in document **`d1`**.

The term counts can be weighted, if requested. The data table saves only the terms that are marked as kept in the data table that is specified in the OUTTERMS= option in the PARSE statement. In the data table, the child frequencies are attributed to the corresponding parent. For example, assume that "said" has term ID **`t1`** and appears eight times in document **`d1`**, "say" has term ID **`t2`** and appears one time in document **`d1`**, "say" is the parent of "said", and neither cell weighting nor term weighting is applied. Then the data table that is specified in the OUTPARENT= option will contain the following entry:

```
t2    d1    9
```

The term count of "said" in **`d1`** is attributed to its parent, "say."

[2] Many elements of the matrix are 0.

The OUTPOS= Data Table

The OUTPOS= option in the PARSE statement specifies a SAS data table to contain the position information about the child terms' occurrences in the document collection. Table 19.11 shows the variables in this data table.

Table 19.11 Variables in the OUTPOS= Data Table

Variable	Description
Term	A lowercase version of the term
Role	The term's part of speech (this variable is empty if the NOTAGGING option is specified in the PARSE statement)
Parent	A lowercase version of the parent term
Start	The starting position of the term's occurrence (the first position is 0)
End	The ending position of the term's occurrence
Sentence	The sentence where the occurrence appears
Paragraph	The paragraph where the occurrence appears (this has not been implemented in the current release, and the value is always set to 0)
Document	The ID of the document where the occurrence appears
Target	The value of the target variable that is associated with the document ID if a variable is specified in the TARGET statement

If you exclude terms by specifying the IGNORE option in the SELECT statement, then those terms are excluded from the OUTPOS= data table. No synonym lists, start lists, or stop lists are used when generating the OUTPOS= data table.

The OUTTERMS= Data Table

The OUTTERMS= option in the PARSE statement specifies a SAS data table to contain the summary information about the terms in the document collection. Table 19.12 shows the variables in this data table.

Table 19.12 Variables in the OUTTERMS= Data Table

Variable	Description	
Term	A lowercase version of the term	
Role	The term's part of speech (this variable is empty if the NOTAGGING option is specified in the PARSE statement)	
Attribute	An indication of the characters that compose the term. Possible attributes are as follows:	
	Alpha	only alphabetic characters
	Mixed	a combination of attributes
	Num	only numbers
	Punct	punctuation characters
	Entity	an identified entity

Table 19.12 *continued*

Variable	Description
Freq	The frequency of a term in the entire document collection
Numdocs	The number of documents that contain the term
_keep	The keep status of the term: "Y" indicates that the term is kept for analysis, and "N" indicates that the term should be dropped in later stages of analysis. To ensure that the OUTTERMS= data table is of a reasonable size, only terms that have _keep=Y are kept in the OUTTERMS= data table by default.
Key	The assigned term number (each unique term in the parsed documents and each unique parent term has a unique Key value)
Parent	The Key value of the term's parent or a "." (period): • If a term has a parent, this variable contains the term number of that parent. • If a term does not have a parent, this value is a "." (period). • If the values of Key, Parent, and Parent_id are identical, the parent occurs as itself. • If the values of Parent and Parent_id are identical but differ from Key, the observation is a child.
Parent_id	Another description of the term's parent: Parent contains the parent's term number if a term is a child, but Parent_id contains this value for all terms.
_ispar	An indication of term's status as a parent, child, or neither: • A "+" (plus sign) indicates that the term is a parent. • A "." (period) indicates that the term is a child. • A missing value indicates that the term is neither a parent nor a child.
Weight	The weights of the terms

If you do not specify the SHOWDROPPEDTERMS option in the PARSE statement, this data table saves only the terms that have _keep=Y. This helps ensure that the OUTTERMS= data table is of a reasonable size. When you specify the SHOWDROPPEDTERMS option, the data table also saves terms that have _keep=N.

The OUTTOPICS= Data Table

If you specify the ROTATION= option in the SVD statement, the OUTTOPICS= option specifies the data table for storing the topics that have been discovered. This data table contains three columns: _topicid, _termCutoff, and _name. If the K= option in the SVD statement is set to k, the _topicid column contains the topic index, which is an integer from 1 to k. The _termCutoff column contains the cutoff value that is

recommended in order to determine which terms actually belong to the topic. The weights for the terms and topics are contained in **V** matrix, which is stored in the data table that is specified in the SVDV= option in the SVD statement. The _name column contains the generated topic name, which is the descriptive label for each topic and provides a synopsis of the discovered topics. The generated topic name contains the terms that have the highest term loadings after the rotation has been performed. The number of terms that are used in the generated name is determined by the NUMLABELS= option in the SVD statement.

System Configuration

Prerequisites for Running PROC TEXTMINE

To use the TEXTMINE procedure, you must have a valid SAS Text Miner license, and the language binary files that are provided under that license must be available on the grid for parsing text.

Configuring for Language Binary Files

PROC TEXTMINE needs to find the grid location of the language binary files that are used in parsing text. These binary files must be deployed to the grid. The GRID_TEXTANALYTICS_BIN_LOC macro variable can be specified to indicate the location of the binary files on the grid. If the macro variable is not specified, then PROC TEXTMINE uses the default installation location.

Deploying Language Binary Files on the Grid

The language binary files must be deployed to the grid. The SAS grid installation script can automatically install the language binary files.

If you choose to manually deploy the binary files, you must copy the binary files from the *$SASROOT* folder to the grid either by placing the files in a location that is accessible by all nodes of the grid or by placing a copy of the files on each node of the grid. The binary files are very large, and a shared location requires less space to store them. However, using a shared location means that the files must be distributed to the grid when PROC TEXTMINE runs, and this can be time-consuming. Consult your grid administrator for a recommended binary location on the grid.

Language Binary Files

The language binary files are originally installed in the following location: *$SASROOT/misc/tktg*.

When you manually deploy the language binary files, you need to copy to the grid only the binary files that correspond to the language you plan to use. The following binary files are used for English processing:

- *en-compounds.txt*
- *en-ne.li*
- *en-sastags.txt*
- *en-utf8.stkzo*
- *en-utf8.tkzo*

- *en-utf8-AnaInfSinFas.mdic*
- *en-utf8-std.htagger*
- *case_mapping.bin*
- *chlangid.bin*
- *chmap.bin*
- *tix.config*
- *language-data.yml*

If you have licensed SAS Text Miner languages other than English, you will see other files in your installation directory. The filenames begin with the corresponding two-letter language codes. For any language that you want to use on the grid, you need to copy its corresponding language binary files to the grid.

The GRID_TEXTANALYTICS_BIN_LOC Macro

You can use the GRID_TEXTANALYTICS_BIN_LOC macro to tell the TEXTMINE procedure where to find the language binary files. If the macro variable is not specified, then the default installation location will be used.

Assume that the grid administrator has installed the language binary files to a directory named */global_dir/tktg/misc*, which is accessible to all nodes of the grid. To tell PROC TEXTMINE the location of the language binary files, insert the following statement before calling the procedure:

```
%let GRID_TEXTANALYTICS_BIN_LOC=/global_dir/tktg/misc;
```

When storage space permits, you can ensure optimal performance by placing a copy of the language binary files on each node and using a relative pathname for the GRID_TEXTANALYTICS_BIN_LOC macro variable. For example, the grid administrator can create a directory whose pathname is */local_dir/tktg/misc* on each node and can store the language binary files in that directory. When the following statement is specified, the TEXTMINE procedure goes to the directory */local_dir/tktg/misc* on each node to load the binary files:

```
%let GRID_TEXTANALYTICS_BIN_LOC=/local_dir/tktg/misc;
```

Examples: TEXTMINE Procedure

Example 19.1: Parsing with No Options Turned On

This example parses five documents, which are in a generated data table. The following DATA step generates the five documents:

```
/* 1)  create data table */

data mycas.CarNominations;
infile datalines delimiter='|' missover;
```

```
length text $70 ;
input text$ i;
datalines;
   The Ford Taurus is the World Car of the Year. |1
   Hyundai won the award last year. |2
   Toyota sold the Toyota Tacoma in bright green. |3
   The Ford Taurus is sold in all colors except for lime green. |4
   The Honda Insight was World Car of the Year in 2008. |5
   ;
run;
```

The following statements run PROC TEXTMINE to parse the documents.

```
/* 2) starting code */
proc textmine data=mycas.CarNominations;
doc_id i;
var text;
parse
   nostemming notagging nonoungroups
   termwgt        = none
   cellwgt        = none
   reducef        = 1
   entities       = none
   outparent      = mycas.outparent
   outterms       = mycas.outterms
   outchild       = mycas.outchild
   outconfig      = mycas.outconfig
   ;
run;

/* 3) print outterms data table */
data outterms; set mycas.outterms; run;
proc print data=outterms; run;
```

Output 19.1.1 shows the content of the mycas.outterms data table. In this example, stemming, part-of-speech tagging, and noun group extraction are suppressed and NONE is specified for entity identification, term and cell weighting, and term filtering. No synonym list, multiterm list, or stop list is specified. As a result of this configuration, there is no child term in the mycas.outterms data table. Also, the mycas.outparent data table and the mycas.outchild data table are exactly the same. The TEXTMINE procedure automatically drops punctuation and numbers.

Output 19.1.1 The mycas.outterms Data Table

Obs	Term	Role	Attribute	Freq	numdocs	_keep	Key	Parent	Parent_id	_ispar	Weight
1	all		Alpha	1	1	Y	1	.	1		1
2	toyota		Alpha	2	1	Y	2	.	2		1
3	ford		Alpha	2	2	Y	3	.	3		1
4	tacoma		Alpha	1	1	Y	4	.	4		1
5	year		Alpha	3	3	Y	5	.	5		1
6	taurus		Alpha	2	2	Y	6	.	6		1
7	won		Alpha	1	1	Y	7	.	7		1
8	honda		Alpha	1	1	Y	8	.	8		1
9	bright		Alpha	1	1	Y	9	.	9		1
10	sold		Alpha	2	2	Y	10	.	10		1
11	colors		Alpha	1	1	Y	11	.	11		1
12	lime		Alpha	1	1	Y	12	.	12		1
13	except		Alpha	1	1	Y	13	.	13		1
14	hyundai		Alpha	1	1	Y	14	.	14		1
15	in		Alpha	3	3	Y	15	.	15		1
16	is		Alpha	2	2	Y	16	.	16		1
17	for		Alpha	1	1	Y	17	.	17		1
18	world		Alpha	2	2	Y	18	.	18		1
19	green		Alpha	2	2	Y	19	.	19		1
20	the		Alpha	8	5	Y	20	.	20		1
21	of		Alpha	2	2	Y	21	.	21		1
22	award		Alpha	1	1	Y	22	.	22		1
23	was		Alpha	1	1	Y	23	.	23		1
24	car		Alpha	2	2	Y	24	.	24		1
25	insight		Alpha	1	1	Y	25	.	25		1
26	last		Alpha	1	1	Y	26	.	26		1

Example 19.2: Parsing with Stemming

This example uses the data table that is generated in Example 19.1. The following statements run PROC TEXTMINE to parse the documents. Because the NOSTEMMING option is not specified in the PARSE statement, words are stemmed (the default).

```
/* create data table */
data mycas.CarNominations;
infile datalines delimiter='|' missover;
length text $70 ;
input text$ i;
datalines;
   The Ford Taurus is the World Car of the Year. |1
   Hyundai won the award last year. |2
   Toyota sold the Toyota Tacoma in bright green. |3
   The Ford Taurus is sold in all colors except for lime green. |4
   The Honda Insight was World Car of the Year in 2008. |5
   ;
run;

proc textmine data=mycas.CarNominations;
doc_id i;
var text;
parse
   notagging nonoungroups
   termwgt  = none
   cellwgt  = none
   reducef  = 1
   entities = none
   outparent= mycas.outparent
   outterms = mycas.outterms
   outchild = mycas.outchild
   outconfig= mycas.outconfig
   ;
run;
data outterms; set mycas.outterms; run;
proc print data = outterms; run;
```

Output 19.2.1 shows the content of the mycas.outterms data table. In this example, words are stemmed. You can see that the term "sold" now stems to the parent term "sell." Also, the mycas.outparent data table and the mycas.outchild data table are different. The parent term "sell" shows up in mycas.outparent (key=11), but not the child term "sold" (key=27). Only "sold" appears in the mycas.outchild data table, and "sell" does not appear.

Output 19.2.1 The mycas.outterms Data Table with Stemming

Obs	Term	Role	Attribute	Freq	numdocs	_keep	Key	Parent	Parent_id	_ispar	Weight
1	all		Alpha	1	1	Y	1	.	1		1
2	win		Alpha	1	1	Y	2	.	2	+	1
3	toyota		Alpha	2	1	Y	3	.	3		1
4	ford		Alpha	2	2	Y	4	.	4		1
5	tacoma		Alpha	1	1	Y	5	.	5		1
6	year		Alpha	3	3	Y	6	.	6		1
7	taurus		Alpha	2	2	Y	7	.	7		1
8	won		Alpha	1	1	Y	26	2	2	.	1
9	honda		Alpha	1	1	Y	8	.	8		1
10	bright		Alpha	1	1	Y	9	.	9		1
11	be		Alpha	3	3	Y	10	.	10	+	1
12	sold		Alpha	2	2	Y	27	11	11	.	1
13	sell		Alpha	2	2	Y	11	.	11	+	1
14	colors		Alpha	1	1	Y	28	23	23	.	1
15	lime		Alpha	1	1	Y	12	.	12		1
16	except		Alpha	1	1	Y	13	.	13		1
17	hyundai		Alpha	1	1	Y	14	.	14		1
18	in		Alpha	3	3	Y	15	.	15		1
19	is		Alpha	2	2	Y	29	10	10	.	1
20	for		Alpha	1	1	Y	16	.	16		1
21	world		Alpha	2	2	Y	17	.	17		1
22	green		Alpha	2	2	Y	18	.	18		1
23	the		Alpha	8	5	Y	19	.	19		1
24	of		Alpha	2	2	Y	20	.	20		1
25	award		Alpha	1	1	Y	21	.	21		1
26	was		Alpha	1	1	Y	30	10	10	.	1
27	car		Alpha	2	2	Y	22	.	22		1
28	color		Alpha	1	1	Y	23	.	23	+	1
29	insight		Alpha	1	1	Y	24	.	24		1
30	last		Alpha	1	1	Y	25	.	25		1

Example 19.3: Adding Entities and Noun Groups

This example uses the data table that is generated in Example 19.1. The following statements run PROC TEXTMINE to parse the documents. Because the NONOUNGROUPS option is not specified in the PARSE statement, noun groups are extracted, and because the ENTITIES=STD option is specified, entities are identified.

```
/* create data table */
data mycas.CarNominations;
infile datalines delimiter='|' missover;
length text $70 ;
input text$ i;
datalines;
   The Ford Taurus is the World Car of the Year. |1
   Hyundai won the award last year. |2
   Toyota sold the Toyota Tacoma in bright green. |3
   The Ford Taurus is sold in all colors except for lime green. |4
   The Honda Insight was World Car of the Year in 2008. |5
   ;
run;

proc textmine data=mycas.CarNominations;
doc_id i;
var text;
PARSE
   notagging
   termwgt   = none
   cellwgt   = none
   reducef   = 1
   entities  = std
   outparent = mycas.outparent
   outterms  = mycas.outterms
   outchild  = mycas.outchild
   outconfig = mycas.outconfig
   ;
run;
data outterms; set mycas.outterms; run;
proc print data=outterms; run;
```

Output 19.3.1 shows the content of the mycas.outterms data table. Compared to Output 19.2.1, the mycas.outterms data table is longer, because it contains entities and noun groups. For example, “honda insight” is included in the mycas.outterms data table as an entity with Role=Vehicle, and “bright green” is also included in the mycas.outterms data table as a noun group.

Output 19.3.1 The mycas.outterms Data Table with Noun Group Extraction and Entity Identification

Obs	Term	Role	Attribute	Freq	numdocs	_keep	Key	Parent	Parent_id	_ispar	Weight
1	all		Alpha	1	1	Y	1	.	1		1
2	win		Alpha	1	1	Y	2	.	2	+	1
3	tacoma		Alpha	1	1	Y	3	.	3		1
4	year		Alpha	3	3	Y	4	.	4		1
5	taurus		Alpha	2	2	Y	5	.	5		1
6	lime green	nlpNounGroup	Alpha	1	1	Y	6	.	6		1
7	won		Alpha	1	1	Y	30	2	2	.	1
8	honda		Alpha	1	1	Y	7	.	7		1
9	bright		Alpha	1	1	Y	8	.	8		1
10	in 2008	nlpDate	Entity	1	1	Y	9	.	9		1
11	be		Alpha	3	3	Y	10	.	10	+	1
12	sold		Alpha	2	2	Y	31	12	12	.	1
13	bright green	nlpNounGroup	Alpha	1	1	Y	11	.	11		1
14	sell		Alpha	2	2	Y	12	.	12	+	1
15	colors		Alpha	1	1	Y	32	27	27	.	1
16	lime		Alpha	1	1	Y	13	.	13		1
17	hyundai	nlpOrganization	Entity	1	1	Y	14	.	14		1
18	except		Alpha	1	1	Y	15	.	15		1
19	in		Alpha	3	3	Y	16	.	16		1
20	is		Alpha	2	2	Y	33	10	10	.	1
21	toyota	nlpOrganization	Entity	2	1	Y	17	.	17		1
22	last year	nlpDate	Entity	1	1	Y	18	.	18		1
23	ford	nlpOrganization	Entity	2	2	Y	19	.	19		1
24	for		Alpha	1	1	Y	20	.	20		1
25	world		Alpha	2	2	Y	21	.	21		1
26	green		Alpha	2	2	Y	22	.	22		1
27	the		Alpha	8	5	Y	23	.	23		1
28	of		Alpha	2	2	Y	24	.	24		1
29	award		Alpha	1	1	Y	25	.	25		1
30	was		Alpha	1	1	Y	34	10	10	.	1
31	car		Alpha	2	2	Y	26	.	26		1
32	color		Alpha	1	1	Y	27	.	27	+	1
33	insight		Alpha	1	1	Y	28	.	28		1
34	last		Alpha	1	1	Y	29	.	29		1

Example 19.4: Adding Part-of-Speech Tagging

This example uses the data table that is generated in Example 19.1. The following statements run PROC TEXTMINE to parse the documents. Because the NOTAGGING option is not specified in the PARSE statement, PROC TEXTMINE uses context clues to determine a term's part of speech.

```
/* create data table */
data mycas.CarNominations;
infile datalines delimiter='|' missover;
length text $70 ;
input text$ i;
datalines;
   The Ford Taurus is the World Car of the Year. |1
   Hyundai won the award last year. |2
   Toyota sold the Toyota Tacoma in bright green. |3
   The Ford Taurus is sold in all colors except for lime green. |4
   The Honda Insight was World Car of the Year in 2008. |5
   ;
run;

proc textmine data=mycas.CarNominations;
doc_id i;
var text;
parse
   termwgt   = none
   cellwgt   = none
   reducef   = 1
   entities  = std
   outparent = mycas.outparent
   outterms  = mycas.outterms
   outchild  = mycas.outchild
   outconfig = mycas.outconfig
   ;
run;
data outterms; set mycas.outterms; run;
proc print data= outterms; run;
```

Output 19.4.1 shows the content of the mycas.outterms data table. Compared to Output 19.3.1, the mycas.outterms data table also contains the part-of-speech tag for the terms.

Output 19.4.1 The mycas.outterms Data Table with Part-of-Speech Tagging

Obs	Term	Role	Attribute	Freq	numdocs	_keep	Key	Parent	Parent_id	_ispar	Weight
1	is	V	Alpha	2	2	Y	30	26	26	.	1
2	was	V	Alpha	1	1	Y	31	26	26	.	1
3	bright	A	Alpha	1	1	Y	1	.	1		1
4	taurus	N	Alpha	2	2	Y	2	.	2		1
5	won	V	Alpha	1	1	Y	32	17	17	.	1
6	for	PPOS	Alpha	1	1	Y	3	.	3		1
7	lime green	nlpNounGroup	Alpha	1	1	Y	4	.	4		1
8	lime	A	Alpha	1	1	Y	5	.	5		1
9	in 2008	nlpDate	Entity	1	1	Y	6	.	6		1
10	the	DET	Alpha	8	5	Y	7	.	7		1
11	except	V	Alpha	1	1	Y	8	.	8		1
12	bright green	nlpNounGroup	Alpha	1	1	Y	9	.	9		1
13	color	N	Alpha	1	1	Y	10	.	10	+	1
14	in	PPOS	Alpha	3	3	Y	11	.	11		1
15	hyundai	nlpOrganization	Entity	1	1	Y	12	.	12		1
16	sold	V	Alpha	2	2	Y	33	28	28	.	1
17	toyota	nlpOrganization	Entity	2	1	Y	13	.	13		1
18	last year	nlpDate	Entity	1	1	Y	14	.	14		1
19	ford	nlpOrganization	Entity	2	2	Y	15	.	15		1
20	all	A	Alpha	1	1	Y	16	.	16		1
21	win	V	Alpha	1	1	Y	17	.	17	+	1
22	car	PN	Alpha	2	2	Y	18	.	18		1
23	colors	N	Alpha	1	1	Y	34	10	10	.	1
24	award	N	Alpha	1	1	Y	19	.	19		1
25	insight	PN	Alpha	1	1	Y	20	.	20		1
26	of	PPOS	Alpha	2	2	Y	21	.	21		1
27	honda	PN	Alpha	1	1	Y	22	.	22		1
28	world	PN	Alpha	2	2	Y	23	.	23		1
29	last	A	Alpha	1	1	Y	24	.	24		1
30	green	N	Alpha	2	2	Y	25	.	25		1
31	be	V	Alpha	3	3	Y	26	.	26	+	1
32	tacoma	PN	Alpha	1	1	Y	27	.	27		1
33	sell	V	Alpha	2	2	Y	28	.	28	+	1
34	year	N	Alpha	3	3	Y	29	.	29		1

Example 19.5: Adding Synonyms

This example uses the data table that is generated in Example 19.1. So far, by looking at the mycas.outterms data tables that are generated by Example 19.1 to Example 19.4, you can see that the data are very "vehicle focused." But suppose what you really care about are the companies. You can use a synonym list in parsing in order to map each vehicle to the company that produces it. The following DATA step generates the synonym list, and the following statements show this mapping:

```
/* create data table */
data mycas.CarNominations;
infile datalines delimiter='|' missover;
length text $70 ;
input text$ i;
datalines;
   The Ford Taurus is the World Car of the Year. |1
   Hyundai won the award last year. |2
   Toyota sold the Toyota Tacoma in bright green. |3
   The Ford Taurus is sold in all colors except for lime green. |4
   The Honda Insight was World Car of the Year in 2008. |5
   ;
run;
/* create synonym list */
data mycas.synds;
infile datalines delimiter=',';
length Term $13;
input Term $ TermRole $ Parent $ ParentRole$;
datalines;
   honda insight, VEHICLE , honda,  COMPANY,
   ford taurus,   VEHICLE,  ford,   COMPANY,
   toyota tacoma, VEHICLE,  toyota, COMPANY,
;
run;

proc textmine data=mycas.CarNominations;
doc_id i;
var text;
parse
   termwgt   = none
   cellwgt   = none
   reducef   = 1
   entities  = std
   synonym   = mycas.synds
   outparent = mycas.outparent
   outterms  = mycas.outterms
   outchild  = mycas.outchild
   outconfig = mycas.outconfig
   ;
RUN;

data outterms; set mycas.outterms; run;
proc print data= outterms; run;
```

Output 19.5.1 shows the content of the mycas.outterms data table. You can see that the term "honda insight" (key=39) is assigned the parent term "honda" (key=6). Only the term "honda" appears in the mycas.outparent data table.

Output 19.5.1 The mycas.outterms Data Table with Synonym Mapping

Obs	Term	Role	Attribute	Freq	numdocs	_keep	Key	Parent	Parent_id	_ispar	Weight
1	is	V	Alpha	2	2	Y	30	26	26	.	1
2	was	V	Alpha	1	1	Y	31	26	26	.	1
3	bright	A	Alpha	1	1	Y	1	.	1		1
4	taurus	N	Alpha	2	2	Y	2	.	2		1
5	won	V	Alpha	1	1	Y	32	17	17	.	1
6	for	PPOS	Alpha	1	1	Y	3	.	3		1
7	lime green	nlpNounGroup	Alpha	1	1	Y	4	.	4		1
8	lime	A	Alpha	1	1	Y	5	.	5		1
9	in 2008	nlpDate	Entity	1	1	Y	6	.	6		1
10	the	DET	Alpha	8	5	Y	7	.	7		1
11	except	V	Alpha	1	1	Y	8	.	8		1
12	bright green	nlpNounGroup	Alpha	1	1	Y	9	.	9		1
13	color	N	Alpha	1	1	Y	10	.	10	+	1
14	in	PPOS	Alpha	3	3	Y	11	.	11		1
15	hyundai	nlpOrganization	Entity	1	1	Y	12	.	12		1
16	sold	V	Alpha	2	2	Y	33	28	28	.	1
17	toyota	nlpOrganization	Entity	2	1	Y	13	.	13		1
18	last year	nlpDate	Entity	1	1	Y	14	.	14		1
19	ford	nlpOrganization	Entity	2	2	Y	15	.	15		1
20	all	A	Alpha	1	1	Y	16	.	16		1
21	win	V	Alpha	1	1	Y	17	.	17	+	1
22	car	PN	Alpha	2	2	Y	18	.	18		1
23	colors	N	Alpha	1	1	Y	34	10	10	.	1
24	award	N	Alpha	1	1	Y	19	.	19		1
25	insight	PN	Alpha	1	1	Y	20	.	20		1
26	of	PPOS	Alpha	2	2	Y	21	.	21		1
27	honda	PN	Alpha	1	1	Y	22	.	22		1
28	world	PN	Alpha	2	2	Y	23	.	23		1
29	last	A	Alpha	1	1	Y	24	.	24		1
30	green	N	Alpha	2	2	Y	25	.	25		1
31	be	V	Alpha	3	3	Y	26	.	26	+	1
32	tacoma	PN	Alpha	1	1	Y	27	.	27		1
33	sell	V	Alpha	2	2	Y	28	.	28	+	1
34	year	N	Alpha	3	3	Y	29	.	29		1

Example 19.6: Adding a Stop List

This example uses the data table that is generated in Example 19.1. If you want to eliminate the entity TOYOTA from the analysis, you can enter the parent term "Toyota" with role=COMPANY in a stop list. When this stop list is an input, PROC TEXTMINE drops the term "Toyota" (role=COMPANY) and all its children terms in the mycas.outterms data table by marking _keep=N for these terms. The following DATA steps generate the synonym list and the stop list:

```
/* create data table */
data mycas.CarNominations;
infile datalines delimiter='|' missover;
length text $70 ;
input text$ i;
datalines;
   The Ford Taurus is the World Car of the Year. |1
   Hyundai won the award last year. |2
   Toyota sold the Toyota Tacoma in bright green. |3
   The Ford Taurus is sold in all colors except for lime green. |4
   The Honda Insight was World Car of the Year in 2008. |5
   ;
run;

/* create synonym list */
data mycas.synds;
infile datalines delimiter=',';
length Term $13;
input Term $ TermRole $ Parent $ ParentRole$;
datalines;
   honda insight, VEHICLE , honda,  COMPANY,
   ford taurus,   VEHICLE,  ford,   COMPANY,
   toyota tacoma, VEHICLE,  toyota, COMPANY,
;
run;

 /* create stop list*/
data mycas.stopList;
infile datalines delimiter='|' missover;
length term $25 role $40;
input term$ role$ ;
datalines;
   toyota| COMPANY
;
run;

proc textmine data=mycas.CarNominations;
doc_id i;
var text;
parse
   termwgt    = none
   cellwgt    = none
   reducef    = 1
   entities   = std
```

```
    synonym    = mycas.synds
    stop       = mycas.stopList
    outparent  = mycas.outparent
    outterms   = mycas.outterms
    outchild   = mycas.outchild
    outconfig  = mycas.outconfig
    ;
run;

data outterms; set mycas.outterms; run;
proc print data= outterms; run;
```

Output 19.6.1 shows the content of the mycas.outterms data table. You can see that the term "Toyota, COMPANY" and the term "Toyota Tacoma, VEHICLE" are removed from the mycas.outterms data table. However, the term "Toyota, Prop" remains in the data table because its role is not COMPANY. The mycas.outparent data table is shorter than the one generated in Example 19.5.

Output 19.6.1 The mycas.outterms Data Table Filtered Using Stop List

Obs	Term	Role	Attribute	Freq	numdocs	_keep	Key	Parent	Parent_id	_ispar	Weight
1	is	V	Alpha	2	2	Y	30	26	26	.	1
2	was	V	Alpha	1	1	Y	31	26	26	.	1
3	bright	A	Alpha	1	1	Y	1	.	1		1
4	taurus	N	Alpha	2	2	Y	2	.	2		1
5	won	V	Alpha	1	1	Y	32	17	17	.	1
6	for	PPOS	Alpha	1	1	Y	3	.	3		1
7	lime green	nlpNounGroup	Alpha	1	1	Y	4	.	4		1
8	lime	A	Alpha	1	1	Y	5	.	5		1
9	in 2008	nlpDate	Entity	1	1	Y	6	.	6		1
10	the	DET	Alpha	8	5	Y	7	.	7		1
11	except	V	Alpha	1	1	Y	8	.	8		1
12	bright green	nlpNounGroup	Alpha	1	1	Y	9	.	9		1
13	color	N	Alpha	1	1	Y	10	.	10	+	1
14	in	PPOS	Alpha	3	3	Y	11	.	11		1
15	hyundai	nlpOrganization	Entity	1	1	Y	12	.	12		1
16	sold	V	Alpha	2	2	Y	33	28	28	.	1
17	toyota	nlpOrganization	Entity	2	1	Y	13	.	13		1
18	last year	nlpDate	Entity	1	1	Y	14	.	14		1
19	ford	nlpOrganization	Entity	2	2	Y	15	.	15		1
20	all	A	Alpha	1	1	Y	16	.	16		1
21	win	V	Alpha	1	1	Y	17	.	17	+	1
22	car	PN	Alpha	2	2	Y	18	.	18		1
23	colors	N	Alpha	1	1	Y	34	10	10	.	1
24	award	N	Alpha	1	1	Y	19	.	19		1
25	insight	PN	Alpha	1	1	Y	20	.	20		1
26	of	PPOS	Alpha	2	2	Y	21	.	21		1
27	honda	PN	Alpha	1	1	Y	22	.	22		1
28	world	PN	Alpha	2	2	Y	23	.	23		1
29	last	A	Alpha	1	1	Y	24	.	24		1
30	green	N	Alpha	2	2	Y	25	.	25		1
31	be	V	Alpha	3	3	Y	26	.	26	+	1
32	tacoma	PN	Alpha	1	1	Y	27	.	27		1
33	sell	V	Alpha	2	2	Y	28	.	28	+	1
34	year	N	Alpha	3	3	Y	29	.	29		1

Example 19.7: Adding a Multiterm List

You can specify a multiterm list to define terms that consist of multiple words. This example uses the data table that is generated in Example 19.1 to show how to use the MULTITERM= option. The following DATA steps generate a synonym list, a stop list, and a multiterm list:

```
/* create data table */
data mycas.CarNominations;
infile datalines delimiter='|' missover;
length text $70 ;
input text$ i;
datalines;
   The Ford Taurus is the World Car of the Year. |1
   Hyundai won the award last year. |2
   Toyota sold the Toyota Tacoma in bright green. |3
   The Ford Taurus is sold in all colors except for lime green. |4
   The Honda Insight was World Car of the Year in 2008. |5
   ;
run;
/* create synonym list */
data mycas.synds;
infile datalines delimiter=',';
length Term $13;
input Term $ TermRole $ Parent $ ParentRole$;
datalines;
   honda insight, VEHICLE , honda,  COMPANY,
   ford taurus,   VEHICLE,  ford,   COMPANY,
   toyota tacoma, VEHICLE,  toyota, COMPANY,
;
run;
/* create stop list */
data mycas.stopList;
infile datalines delimiter='|' missover;
length term $25 role $40;
input term$ role$ ;
datalines;
   toyota| COMPANY
;
run;
/* create multiterm list */
data mycas.multiterms;
infile datalines delimiter='|';
length multiterm $64;
input multiterm$;
datalines;
   except for :3:Prep
;
run;

proc textmine data=mycas.CarNominations;
doc_id i;
var text;
```

```
parse
   termwgt    = none
   cellwgt    = none
   reducef    = 1
   entities   = std
   synonym    = mycas.synds
   stop       = mycas.stopList
   multiterm  = mycas.multiterms
   outparent  = mycas.outparent
   outterms   = mycas.outterms
   outchild   = mycas.outchild
   outconfig  = mycas.outconfig
   ;
run;

data outterms; set mycas.outterms; run;
proc print data= outterms; run;
```

Output 19.7.1 shows the content of the mycas.outterms data table. In the preceding statements, "except for" is defined as an individual term in the third DATA step. In the mycas.outterms data table, you can see that the two terms "except" and "for" have become one term, "except for."

Output 19.7.1 The mycas.outterms Data Table Using a Multiterm List

Obs	Term	Role	Attribute	Freq	numdocs	_keep	Key	Parent	Parent_id	_ispar	Weight
1	is	V	Alpha	2	2	Y	29	25	25	.	1
2	was	V	Alpha	1	1	Y	30	25	25	.	1
3	bright	A	Alpha	1	1	Y	1	.	1		1
4	taurus	N	Alpha	2	2	Y	2	.	2		1
5	won	V	Alpha	1	1	Y	31	16	16	.	1
6	lime green	nlpNounGroup	Alpha	1	1	Y	3	.	3		1
7	except for	PPOS	Alpha	1	1	Y	4	.	4		1
8	lime	A	Alpha	1	1	Y	5	.	5		1
9	in 2008	nlpDate	Entity	1	1	Y	6	.	6		1
10	the	DET	Alpha	8	5	Y	7	.	7		1
11	bright green	nlpNounGroup	Alpha	1	1	Y	8	.	8		1
12	color	N	Alpha	1	1	Y	9	.	9	+	1
13	in	PPOS	Alpha	3	3	Y	10	.	10		1
14	hyundai	nlpOrganization	Entity	1	1	Y	11	.	11		1
15	sold	V	Alpha	2	2	Y	32	27	27	.	1
16	toyota	nlpOrganization	Entity	2	1	Y	12	.	12		1
17	last year	nlpDate	Entity	1	1	Y	13	.	13		1
18	ford	nlpOrganization	Entity	2	2	Y	14	.	14		1
19	all	A	Alpha	1	1	Y	15	.	15		1
20	win	V	Alpha	1	1	Y	16	.	16	+	1
21	car	PN	Alpha	2	2	Y	17	.	17		1
22	colors	N	Alpha	1	1	Y	33	9	9	.	1
23	award	N	Alpha	1	1	Y	18	.	18		1
24	insight	PN	Alpha	1	1	Y	19	.	19		1
25	of	PPOS	Alpha	2	2	Y	20	.	20		1
26	honda	PN	Alpha	1	1	Y	21	.	21		1
27	world	PN	Alpha	2	2	Y	22	.	22		1
28	last	A	Alpha	1	1	Y	23	.	23		1
29	green	N	Alpha	2	2	Y	24	.	24		1
30	be	V	Alpha	3	3	Y	25	.	25	+	1
31	tacoma	PN	Alpha	1	1	Y	26	.	26		1
32	sell	V	Alpha	2	2	Y	27	.	27	+	1
33	year	N	Alpha	3	3	Y	28	.	28		1

Example 19.8: Selecting Parts of Speech and Entities to Ignore

This example uses the data table that is generated in Example 19.1. If you want to eliminate prepositions, determiners, and proper nouns from your analysis, you can add a SELECT statement that lists these part-of-speech labels. If you also want to eliminate entities that are labeled "PROP_MISC," you can add another SELECT statement that includes "PROP_MISC" in the label list.

```
/* create data table */
data mycas.CarNominations;
infile datalines delimiter='|' missover;
length text $70 ;
input text$ i;
datalines;
   The Ford Taurus is the World Car of the Year. |1
   Hyundai won the award last year. |2
   Toyota sold the Toyota Tacoma in bright green. |3
   The Ford Taurus is sold in all colors except for lime green. |4
   The Honda Insight was World Car of the Year in 2008. |5
   ;
run;

/* create synonym list */
data mycas.synds;
infile datalines delimiter=',';
length Term $13;
input Term $ TermRole $ Parent $ ParentRole$;
datalines;
   honda insight, VEHICLE , honda,  COMPANY,
   ford taurus,   VEHICLE,  ford,   COMPANY,
   toyota tacoma, VEHICLE,  toyota, COMPANY,
;
run;

 /* create stop list*/
data mycas.stopList;
infile datalines delimiter='|' missover;
length term $25 role $40;
input term$ role$ ;
datalines;
   toyota| COMPANY
;
run;

proc textmine data=mycas.CarNominations;
doc_id i;
var text;
parse
   termwgt    = none
   cellwgt    = none
   reducef    = 1
   entities   = std
   synonym    = mycas.synds
```

```
    stop       = mycas.stopList
    outparent  = mycas.outparent
    outterms   = mycas.outterms
    outchild   = mycas.outchild
    outconfig  = mycas.outconfig
    ;
select "prep" "det" "prop"/ignore;
select "prop_misc"/group="entities" ignore;
run;

data outterms; set mycas.outterms; run;
proc print data= outterms; run;
```

Output 19.8.1 shows the content of the mycas.outterms data table. You can see that prepositions, determiners, and proper nouns are excluded. Terms that are labeled “PROP_MISC” are also excluded.

Output 19.8.1 The mycas.outterms Data Table Ignoring Specified Parts of Speech and Entities

Obs	Term	Role	Attribute	Freq	numdocs	_keep	Key	Parent	Parent_id	_ispar	Weight
1	is	V	Alpha	2	2	Y	29	25	25	.	1
2	was	V	Alpha	1	1	Y	30	25	25	.	1
3	bright	A	Alpha	1	1	Y	1	.	1		1
4	taurus	N	Alpha	2	2	Y	2	.	2		1
5	won	V	Alpha	1	1	Y	31	16	16	.	1
6	for	PPOS	Alpha	1	1	Y	3	.	3		1
7	lime green	nlpNounGroup	Alpha	1	1	Y	4	.	4		1
8	lime	A	Alpha	1	1	Y	5	.	5		1
9	in 2008	nlpDate	Entity	1	1	Y	6	.	6		1
10	except	V	Alpha	1	1	Y	7	.	7		1
11	bright green	nlpNounGroup	Alpha	1	1	Y	8	.	8		1
12	color	N	Alpha	1	1	Y	9	.	9	+	1
13	in	PPOS	Alpha	3	3	Y	10	.	10		1
14	hyundai	nlpOrganization	Entity	1	1	Y	11	.	11		1
15	sold	V	Alpha	2	2	Y	32	27	27	.	1
16	toyota	nlpOrganization	Entity	2	1	Y	12	.	12		1
17	last year	nlpDate	Entity	1	1	Y	13	.	13		1
18	ford	nlpOrganization	Entity	2	2	Y	14	.	14		1
19	all	A	Alpha	1	1	Y	15	.	15		1
20	win	V	Alpha	1	1	Y	16	.	16	+	1
21	car	PN	Alpha	2	2	Y	17	.	17		1
22	colors	N	Alpha	1	1	Y	33	9	9	.	1
23	award	N	Alpha	1	1	Y	18	.	18		1
24	insight	PN	Alpha	1	1	Y	19	.	19		1
25	of	PPOS	Alpha	2	2	Y	20	.	20		1
26	honda	PN	Alpha	1	1	Y	21	.	21		1
27	world	PN	Alpha	2	2	Y	22	.	22		1
28	last	A	Alpha	1	1	Y	23	.	23		1
29	green	N	Alpha	2	2	Y	24	.	24		1
30	be	V	Alpha	3	3	Y	25	.	25	+	1
31	tacoma	PN	Alpha	1	1	Y	26	.	26		1
32	sell	V	Alpha	2	2	Y	27	.	27	+	1
33	year	N	Alpha	3	3	Y	28	.	28		1

Chapter 20
The TMSCORE Procedure

Contents

Overview: TMSCORE Procedure **409**
PROC TMSCORE Features 409
Using CAS Sessions and CAS Engine Librefs 410
Getting Started: TMSCORE Procedure **410**
Syntax: TMSCORE Procedure **414**
PROC TMSCORE Statement 414
DOC_ID Statement 416
VARIABLES Statement 416
Details: TMSCORE Procedure **416**
System Configuration 416
Prerequisites for Running PROC TMSCORE 416
Configuration for Language Binary Files 416

Overview: TMSCORE Procedure

The TMSCORE procedure scores textual data in SAS Viya. In text mining, scoring is the process of applying parsing and singular value decomposition (SVD) projections to new textual data. The TMSCORE procedure performs this scoring of new documents, and its primary outputs are the Outparent data table (which holds the parsing results of the term-by-document matrix) and the Outdocpro data table (which holds the reduced-dimensional representation of the score collection). PROC TMSCORE uses some of the output data tables of the TEXTMINE procedure as input data to ensure consistency between scoring and training. During scoring, the new textual data must be parsed using the same settings that the training data were parsed with, indexed using only the subset of terms that were used during training, and projected onto the reduced-dimensional subspace of the singular value decomposition that was derived from the training data. To facilitate this process, you specify the CONFIG=, TERMS=, and SVDU= options in PROC TEXTMINE to create three data tables (Outconfig, Outterms, and Svdu, respectively), and then you specify those three data tables as inputs to PROC TMSCORE. For more information about these data tables, see the CONFIG=, TERMS=, and SVDU= options, respectively, in the section "PROC TMSCORE Statement" on page 414.

PROC TMSCORE Features

The TMSCORE procedure processes large-scale textual data in parallel to achieve efficiency and scalability. The following list summarizes the basic features of PROC TMSCORE:

- Functionalities that are related to document parsing, term-by-document matrix creation, and dimension reduction are integrated into one procedure to process data more efficiently.
- Parsing and term-by-document matrix creation are performed in parallel.
- Computation of document projection is performed in parallel.
- All phases of processing use a high degree of multithreading.

Using CAS Sessions and CAS Engine Librefs

SAS Cloud Analytic Services (CAS) is the analytic server and associated cloud services in SAS Viya. This section describes how to create a CAS session and set up a CAS engine libref that you can use to connect to the CAS session. It assumes that you have a CAS server already available; contact your system administrator if you need help starting and terminating a server. This CAS server is identified by specifying the host on which it runs and the port on which it listens for communications. To simplify your interactions with this CAS server, the host information and port information for the server are stored as SAS option values that are retrieved automatically whenever this CAS server needs to be accessed. You can examine the host and port values for the server at your site by using the following statements:

```
proc options option=(CASHOST CASPORT);
run;
```

In addition to starting a CAS server, your system administrator might also have created a CAS session and a CAS engine libref for your use. You can define your own sessions and CAS engine librefs that connect to the CAS server as shown in the following statements:

```
cas mysess;
libname mycas cas sessref=mysess;
```

The CAS statement creates the CAS session named mysess, and the LIBNAME statement creates the mycas CAS engine libref that you use to connect to this session. It is not necessary to explicitly name the CASHOST and CASPORT of the CAS server in the CAS statement, because these values are retrieved from the corresponding SAS option values.

If you have created the mysess session, you can terminate it by using the TERMINATE option in the CAS statement as follows:

```
cas mysess terminate;
```

For more information about the CAS and LIBNAME statements, see the section "Introduction to Shared Concepts" on page 11 in Chapter 3, "Shared Concepts."

Getting Started: TMSCORE Procedure

NOTE: Input data must be in a CAS table that is accessible in your CAS session. You must refer to this table by using a two-level name. The first level must be a CAS engine libref, and the second level must be the table

name. For more information, see the sections "Using CAS Sessions and CAS Engine Librefs" on page 11 and "Loading a SAS Data Set onto a CAS Server" on page 12 in Chapter 3, "Shared Concepts."

The following DATA steps generate two data tables: the mycas.getstart data table contains 36 observations, and the mycas.getstart_score data table contains 31 observations. Both data tables have two variables: the text variable contains the input documents, and the did variable contains the ID of the documents. Each row in each data table represents a "document" for analysis.

```
data mycas.getstart;
   infile datalines delimiter='|' missover;
   length text $150;
   input text$ did;
   datalines;
      High-performance analytics hold the key to |1
      unlocking the unprecedented business value of big data.|2
      Organizations looking for optimal ways to gain insights|3
      from big data in shorter reporting windows are turning to SAS.|4
      As the gold-standard leader in business analytics |5
      for more than 36 years,|6
      SAS frees enterprises from the limitations of |7
      traditional computing and enables them |8
      to draw instant benefits from big data.|9
      Faster Time to Insight.|10
      From banking to retail to health care to insurance, |11
      SAS is helping industries glean insights from data |12
      that once took days or weeks in just hours, minutes or seconds.|13
      It's all about getting to and analyzing relevant data faster.|14
      Revealing previously unseen patterns, sentiments and relationships.|15
      Identifying unknown risks.|16
      And speeding the time to insights.|17
      High-Performance Analytics from SAS Combining industry-leading |18
      analytics software with high-performance computing technologies|19
      produces fast and precise answers to unsolvable problems|20
      and enables our customers to gain greater competitive advantage.|21
      SAS In-Memory Analytics eliminate the need for disk-based processing|22
      allowing for much faster analysis.|23
      SAS In-Database executes analytic logic into the database itself |24
      for improved agility and governance.|25
      SAS Grid Computing creates a centrally managed,|26
      shared environment for processing large jobs|27
      and supporting a growing number of users efficiently.|28
      Together, the components of this integrated, |29
      supercharged platform are changing the decision-making landscape|30
      and redefining how the world solves big data business problems.|31
      Big data is a popular term used to describe the exponential growth,|32
      availability and use of information,|33
      both structured and unstructured.|34
      Much has been written on the big data trend and how it can |35
      serve as the basis for innovation, differentiation and growth.|36
run;
```

```
data mycas.getstart_score;
   infile datalines delimiter='|' missover;
   length text $150;
   input text$ did;
   datalines;
      Big data according to SAS|1
      At SAS, consider two other dimensions|2
      when thinking about big data:|3
      Variability. In addition to the|4
      increasing velocities and varieties of data, data|5
      flows can be highly inconsistent with periodic peaks.|6
      Is something big trending in the social media?|7
      Perhaps there is a high-profile IPO looming.|8
      Maybe swimming with pigs in the Bahamas is suddenly|9
      the must-do vacation activity. Daily, seasonal and|10
      event-triggered peak data loads can be challenging|11
      to manage - especially with social media involved.|12
      Complexity. When you deal with huge volumes of data,|13
      it comes from multiple sources. It is quite an|14
      undertaking to link, match, cleanse and|15
      transform data across systems. However,|16
      it is necessary to connect and correlate|17
      relationships, hierarchies and multiple data|18
      linkages or your data can quickly spiral out of|19
      control. Data governance can help you determine|20
      how disparate data relates to common definitions|21
      and how to systematically integrate structured|22
      and unstructured data assets to produce|23
      high-quality information that is useful,|24
      appropriate and up-to-date.|25
      Ultimately, regardless of the factors involved,|26
      I believe that the term big data is relative|27
      it applies (per Gartner's assessment)|28
      whenever an organization's ability|29
      to handle, store and analyze data|30
      exceeds its current capacity.|31
run;
```

The following statements use PROC TEXTMINE for processing the input text data table mycas.getstart and create three data tables (mycas.outconfig, mycas.outterms, and mycas.svdu), which can be used in PROC TMSCORE for scoring:

```
proc textmine data    = mycas.getstart;
doc_id      did;
variables   text;
parse
            outterms  = mycas.outterms
            outconfig = mycas.outconfig
            reducef   = 2;
svd
            k         = 5
            svdu      = mycas.svdu;
run;
```

The following statements then use PROC TMSCORE to score the input text data table mycas.getstart_score. The statements take the three data tables that are generated by PROC TEXTMINE as input and create a data table named mycas.docpro, which contains the projection of the documents in the input data table mycas.getstart_score.

```
proc tmscore
             data      = mycas.getstart_score
             terms     = mycas.outterms
             config    = mycas.outconfig
             svdu      = mycas.svdu
             svddocpro = mycas.docpro;
doc_id       did;
variables    text;
run;
```

The output from this analysis is presented in Figure 20.1.

The following statements use PROC PRINT to show the content of the first 10 rows of the sorted mycas.docpro data table, which is generated by the TMSCORE procedure:

```
data docpro;
   set mycas.docpro;
run;
proc sort data=docpro;
by did;
run;
proc print data = docpro (obs=10);
run;
```

Figure 20.1 shows the output of PROC PRINT.

Figure 20.1 The mycas.docpro Data Table

Obs	did	COL1	COL2	COL3	COL4	COL5
1	1	0.8618192721	-0.167546011	0.037379386	0.4703235489	-0.081206017
2	2	0.3275352424	0.5970467719	-0.046820597	0.7262257288	0.081607841
3	3	0.8604238893	-0.412231055	0.0599438871	0.2665897873	-0.122771753
4	4	0.7741694143	0.4670908635	0.2844247769	-0.207037529	-0.242334173
5	5	0.9576866114	-0.265146311	0.0265294315	0.0244271372	0.1059872269
6	6	0.8078123292	-0.198309231	-0.399949692	0.3182364359	-0.216514445
7	7	0.9183024782	-0.099013401	0.3165905611	0.1756583621	-0.125823135
8	8	0.8321211924	-0.161579007	-0.291096477	0.4392268744	0.0617182277
9	9	0.8905692292	0.1358612507	0.3702567322	0.1577544227	-0.162639672
10	10	0.6387842303	0.1643248339	0.1598343808	-0.572869995	0.4595922056

Syntax: TMSCORE Procedure

The following statements are available in the TMSCORE procedure:

PROC TMSCORE DATA=*CAS-libref.data-table* < *options* > **;**
VARIABLES *variable* **;**
DOC_ID *variable* **;**

PROC TMSCORE Statement

PROC TMSCORE DATA=*CAS-libref.data-table* < *options* > **;**

The PROC TMSCORE statement invokes the procedure. Table 20.1 summarizes the *options* in the statement by function. The *options* are then described fully in alphabetical order.

Table 20.1 PROC TMSCORE Statement Options

option	**Description**
Basic Options	
DATA | DOC=	Specifies the input document data table
TERMS=	Specifies the data table that contains the terms to be used for scoring
CONFIG=	Specifies the data table that contains the configuration information
SVDU=	Specifies the data table that contains the U matrix whose columns are the left singular vectors
Output Options	
OUTPARENT=	Specifies the data table that contains the term-by-document frequency matrix that is used to model the document collection. In this matrix, the child terms are not represented and child terms' frequencies are attributed to their corresponding parents.
SVDDOCPRO=	Specifies the data table that contains the projections of the documents

You must specify the following option:

DATA=*CAS-libref.data-table*

DOC=*CAS-libref.data-table*

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

CAS-libref refers to a collection of information that is defined in the LIBNAME statement and includes the caslib, which includes a path to the data, and a session identifier, which defaults to the active session but which can be explicitly defined in the LIBNAME statement. For more information about *CAS-libref*, see the section "Using CAS Sessions and CAS Engine Librefs" on page 410.

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

The input data table contains documents for PROC TMSCORE to score. Each row of the input data table must contain one text variable and one ID variable, which correspond to the text and the unique ID of a document, respectively.

You can also specify the following *options*:

CONFIG=*CAS-libref.data-table*
specifies the input data table that contains configuration information for PROC TMSCORE. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 410. Specify the table that was generated by the OUTCONFIG= option in the PARSE statement of the TEXTMINE procedure during training. For more information about this data table, see the section "The OUTCONFIG= Data Table" on page 384 of Chapter 19, "The TEXTMINE Procedure."

OUTPARENT=*CAS-libref.data-table*
specifies the output data table to contain a compressed representation of the sparse term-by-document frequency matrix. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 410. The data table contains only the kept representative terms, and the child frequencies are attributed to the corresponding parent. For more information about the compressed representation of the sparse term-by-document frequency matrix, see the section "The OUTPARENT= Data Table" on page 385 of Chapter 19, "The TEXTMINE Procedure."

SVDDOCPRO=*CAS-libref.data-table*
specifies the output data table to contain the reduced dimensional projections for each document. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 410. The contents of this data table are formed by multiplying the term-by-document frequency matrix by the input data table that is specified in the SVDU= option and then normalizing the result.

SVDU=*CAS-libref.data-table*
specifies the input data table that contains the **U** matrix, which is created during training by PROC TEXTMINE. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 410. The data table contains the information that is needed to project each document into the reduced dimensional space. For more information about the contents of this data table, see the SVDU= option in Chapter 19, "The TEXTMINE Procedure."

TERMS=*CAS-libref.data-table*
specifies the input data table of terms to be used by PROC TMSCORE. *CAS-libref.data-table* is a two-level name, where *CAS-libref* refers to the caslib and session identifier, 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 CAS Sessions and CAS Engine Librefs" on page 410. Specify the table that was generated by the OUTTERMS= option in the PARSE statement of the TEXTMINE procedure during training. This data table conveys to PROC TMSCORE which terms should be used in the analysis and whether they should be mapped to a parent. The data table also assigns to each term

a key that corresponds to the key that is used in the input data table that is specified by the SVDU= option. For more information about this data table, see the section "The OUTTERMS= Data Table" on page 386 of Chapter 19, "The TEXTMINE Procedure."

DOC_ID Statement

DOC_ID *variable* **;**

This statement specifies the *variable* that contains the ID of each document. The ID of each document must be unique; it can be either a number or a string of characters.

VARIABLES Statement

VARIABLES *variable* **;**

VAR *variable* **;**

This statement specifies the *variable* that contains the text to be processed.

Details: TMSCORE Procedure

For information about the techniques that are used for nature language processing, term processing, and singular value decomposition, see the section "Details: TEXTMINE Procedure" on page 377 of Chapter 19, "The TEXTMINE Procedure."

System Configuration

Prerequisites for Running PROC TMSCORE

To use the TMSCORE procedure, the language binary files that are provided under that license must be available on the grid for parsing text.

Configuration for Language Binary Files

PROC TMSCORE needs to find the grid location of the language binary files that are used in parsing text. These binary files must be deployed to the grid. The GRID_TEXTANALYTICS_BIN_LOC macro can be specified to indicate the location of the binary files on your grid. If the macro variable is not set, then the default installation location *$SASROOT/misc/tktg* will be used. For more information, see the section "Configuring for Language Binary Files" on page 388 of Chapter 19, "The TEXTMINE Procedure."

Subject Index

k-fold cross validation
  FOREST procedure, 158
  GRADBOOST procedure, 194
  NNET procedure, 271

active-set solver
  SVDD procedure, 318
ASTORE procedure
  computational method, 21
  hyperparameter tuning, 22
  input data tables, 33
  multithreading, 21
  output data tables, 33
  test data, 20
  validation, 20
autoencoder
  NNET procedure, 271
AUTOTUNE statement
  syntax (Shared Concepts), 14

bagging the data
  FOREST procedure, 152
Bandwidth Relative Scale
  SVDD procedure, 315
best configuration
  FOREST procedure, 160
  GRADBOOST procedure, 195
  NNET procedure, 272
binning
  BNET procedure, 51
BNET procedure, 43
  binning, 51
  displayed output, 60
  fit statistics table, 60
  independence tests, 49, 50
  input data tables, 49
  iteration report table, 60
  missing values, 50
  model information table, 60
  model selection, 49
  number of observations table, 60
  ODS table names, 60
  output data tables, 54
  output network data set, 51
  PROC BNET features, 44
  PROC BNET syntax, 48
  structure learning, 50, 51
  validation information table, 60
  variable information table, 60
  variable level table, 60
  variable order table, 60
  variable selection, 51, 52
BOOLRULE procedure
  computational method, 21
  hyperparameter tuning, 22
  multithreading, 21
  test data, 20
  validation, 20
boosting
  GRADBOOST procedure, 188

cluster summary
  GVARCLUS procedure, 214
CODE statement
  syntax (Shared Concepts), 18
comparison with existing methods
  SVDD procedure, 317
computational method
  ASTORE procedure, 21
  BOOLRULE procedure, 21
  FACTMAC procedure, 21
  FOREST procedure, 21
  GRADBOOST procedure, 21
  NNET procedure, 21, 269
  SVMACHINE procedure, 21
  TEXTMINE procedure, 21
  TMSCORE procedure, 21
convergence status
  NNET procedure, 271
cross validation results
  FOREST procedure, 160
  GRADBOOST procedure, 195
  NNET procedure, 272

details
  MWPCA procedure, 246
  RPCA procedure, 294
displayed output
  BNET procedure, 60
  FACTMAC procedure, 110
  FASTKNN procedure, 120
  FISM procedure, 128
  FOREST procedure, 159
  GRADBOOST procedure, 194
  GVARCLUS procedure, 213
  MBANALYSIS procedure, 230
  NNET procedure, 271
  RPCA procedure, 295

SVMACHINE procedure, 352

early stopping
GRADBOOST procedure, 188
evaluation history
FOREST procedure, 160
GRADBOOST procedure, 195
NNET procedure, 272

FACTMAC procedure, 99
computational method, 21
displayed output, 110
final exact loss, 111
hyperparameter tuning, 22
input data tables, 103
interval variables, 111
iteration history, 110
model information, 110
multithreading, 21
number of observations, 110
ODS table names, 111
output CAS tables, 111
output data tables, 108, 111
test data, 20
validation, 20
FASTKNN procedure, 115
displayed output, 120
input data tables, 118
model information, 120
number of observations, 120
ODS table names, 121
output CAS tables, 120
output data tables, 119, 121
final exact loss
FACTMAC procedure, 111
FISM procedure, 127
displayed output, 128
FP-growth algorithm, 127
input data tables, 126
limitations, 129
fit statistics
FOREST procedure, 159
GRADBOOST procedure, 194
fit statistics table
BNET procedure, 60
SVMACHINE procedure, 353
FitStat table
NNET procedure, 272
FOREST procedure, 132
k-fold cross validation, 158
bagging the data, 152
best configuration, 160
computational method, 21
cross validation results, 160
displayed output, 159
evaluation history, 160
fit statistics, 159
handling missing values, 154
handling values that are absent from training data, 155
hyperparameter tuning, 22, 158
input data tables, 140
measuring variable importance, 155
model information, 159
multithreading, 21
number of observations, 159
ODS table names, 160
output data tables, 149
output table, 160
predicted probability names, 160
preselection, 153
splitting criteria, 148
test data, 20
training a decision tree, 152
tuner information, 159
tuner results, 160
tuner summary, 160
tuner timing, 160
validation, 20
variable importance, 159
FP-growth algorithm
FISM procedure, 127
MBANALYSIS procedure, 228
fraction outlier parameter
SVDD procedure, 315

GRADBOOST procedure, 168
k-fold cross validation, 194
best configuration, 195
boosting, 188
computational method, 21
cross validation results, 195
displayed output, 194
early stopping, 188
evaluation history, 195
fit statistics, 194
handling missing values, 190
hyperparameter tuning, 22, 193
input data tables, 176
measuring prediction error, 189
measuring variable importance, 190
model information, 194
multithreading, 21
number of observations, 194
ODS table names, 196
output data tables, 185
output table, 195
predicted probability names, 195

subsampling the data, 187
test data, 20
training a decision tree, 188
tuner information, 195
tuner results, 195
tuner summary, 195
tuner timing, 195
validation, 20
variable importance, 194
graphical lasso
GVARCLUS procedure, 212
graphical representation of precision matrix
GVARCLUS procedure, 213
GVARCLUS procedure, 201
cluster summary, 214
displayed output, 213
graphical lasso, 212
graphical representation of precision matrix, 213
input data tables, 207
maximum iterations, 208
maximum members, 208
maximum steps, 208
minimum clusters, 208
missing values, 212
number of observations, 214
ODS table names, 214
preprocessing, 213
rho, 209
stop, 209
tolerance, 209

handling missing values
FOREST procedure, 154
GRADBOOST procedure, 190
handling values that are absent from training data
FOREST procedure, 155
hyperparameter tuning
ASTORE procedure, 22
BOOLRULE procedure, 22
FACTMAC procedure, 22
FOREST procedure, 22, 158
GRADBOOST procedure, 22, 193
NNET procedure, 22
SVMACHINE procedure, 22
TEXTMINE procedure, 22
TMSCORE procedure, 22

independence tests, 56
BNET procedure, 49, 50
input variables
NNET procedure, 262
interval variables
FACTMAC procedure, 111
iteration history
NNET procedure, 271
iteration history table
SVMACHINE procedure, 353
iteration report table
BNET procedure, 60
SVMACHINE procedure, 353

limitations
FISM procedure, 129
MBANALYSIS procedure, 230
linear kernels
SVDD procedure, 310
SVMACHINE procedure, 350

mathematical formulation
SVDD procedure, 310
MBANALYSIS procedure, 227
displayed output, 230
FP-growth algorithm, 228
input data tables, 224
limitations, 230
scoring with PROC ASTORE, 229
Mean Criteria
SVDD procedure, 315
measuring prediction error
GRADBOOST procedure, 189
measuring variable importance
FOREST procedure, 155
GRADBOOST procedure, 190
misclassification matrix table
SVMACHINE procedure, 353
missing values
BNET procedure, 50
GVARCLUS procedure, 212
model information
FOREST procedure, 159
GRADBOOST procedure, 194
NNET procedure, 271
model information table
BNET procedure, 60
SVMACHINE procedure, 352
model selection
BNET procedure, 49
multithreading
ASTORE procedure, 21
BOOLRULE procedure, 21
FACTMAC procedure, 21
FOREST procedure, 21
GRADBOOST procedure, 21
NNET procedure, 21
SVMACHINE procedure, 21
TEXTMINE procedure, 21
TMSCORE procedure, 21
MWPCA procedure

details, 246
dimensions, 247
displayed output, 247
input data tables, 241
model information, 247
ODS table names, 247
output data tables, 244
results summary, 247

NNET procedure, 250
k-fold cross validation, 271
autoencoder, 271
best configuration, 272
computational method, 21, 269
convergence status, 271
cross validation results, 272
displayed output, 271
evaluation history, 272
FitStat table, 272
hyperparameter tuning, 22
input data tables, 253
iteration history, 271
model information, 271
multithreading, 21
ODS table names, 273
output data tables, 265
parameter tuning, 270
PredIntoName, 273
PredName, 273
PredProbName, 272
ROC curve, 272
score information, 272
test data, 20
tuner information, 272
tuner results, 272
tuner summary, 272
tuner timing, 272
validation, 20
nonlinear kernels
SVDD procedure, 310
SVMACHINE procedure, 350
number of hidden neurons
NNET procedure, 261
number of observations
FOREST procedure, 159
GRADBOOST procedure, 194
GVARCLUS procedure, 214
number of observations table
BNET procedure, 60
SVMACHINE procedure, 352

optimization
SVMACHINE procedure, 351
options summary
PARSE statement, 367
PROC FACTMAC statement, 103
PROC FASTKNN statement, 118
PROC FOREST statement, 139
PROC GRADBOOST statement, 174
PROC GVARCLUS statement, 207
PROC MWPCA statement, 241
PROC RPCA statement, 286
PROC TEXTMINE statement, 365
PROC TMSCORE statement, 414
SELECT statement, 371
SVD statement, 372
output CAS tables
FACTMAC procedure, 111
FASTKNN procedure, 120
output data tables
FACTMAC procedure, 111
FASTKNN procedure, 121
RPCA procedure, 297
output network data set
BNET procedure, 51
output table
FOREST procedure, 160
GRADBOOST procedure, 195

parameter learning, 47, 59
parameter tuning
NNET procedure, 270
PARTITION statement
syntax (Shared Concepts), 19, 20
predicted probability names
FOREST procedure, 160
GRADBOOST procedure, 195
Predicted probability variables table
SVMACHINE procedure, 353
Predicted target variable table
SVMACHINE procedure, 353
PredIntoName
NNET procedure, 273
PredName
NNET procedure, 273
PredProbName
NNET procedure, 272
preprocessing
GVARCLUS procedure, 213
preselection
FOREST procedure, 153
PROC ASTORE, 27
PROC ASTORE features, 28
PROC BNET features
BNET procedure, 44
PROC BNET syntax
BNET procedure, 48
PROC FISM, 123

PROC FISM features, 123
PROC MBANALYSIS, 221
PROC MBANALYSIS features, 222
PROC MWPCA, 235
PROC MWPCA features, 236
PROC RPCA, 281
PROC RPCA features, 282
PROC SVDD, 299
PROC SVMACHINE, 339
PROC SVMACHINE features, 340

radial basis function bandwidth parameter
    SVDD procedure, 313
rba variable importance
    variable importance, 194
ROC curve
    NNET procedure, 272
RPCA procedure
    details, 294
    dimensions, 296
    displayed output, 295
    input data tables, 287
    model information, 296
    ODS table names, 296
    output data tables, 297
    results summary, 296

score information
    NNET procedure, 272
scoring process
    SVMACHINE procedure, 352
scoring with PROC ASTORE
    MBANALYSIS procedure, 229
Shared Concepts
    AUTOTUNE statement, 14
    CODE statement, 18
    PARTITION statement, 19, 20
solvers
    SVDD procedure, 317
sparse matrix
    TEXTMINE procedure, 385
splitting criteria
    FOREST procedure, 148
stochastic subset solver
    SVDD procedure, 318
structure learning, 46, 58
    BNET procedure, 50, 51
subsampling the data
    GRADBOOST procedure, 187
SVDD formulation with weight variable
    SVDD procedure, 316
SVDD procedure
    active-set solver, 318
    Bandwidth Relative Scale, 315
    comparison with existing methods, 317
    fraction outlier parameter, 315
    input data tables, 306
    linear kernels, 310
    mathematical formulation, 310
    Mean Criteria, 315
    nonlinear kernels, 310
    ODS table names, 318
    radial basis function bandwidth parameter, 313
    solvers, 317
    stochastic subset solver, 318
    SVDD formulation with weight variable, 316
SVMACHINE procedure
    computational method, 21
    displayed output, 352
    fit statistics table, 353
    hyperparameter tuning, 22
    input data tables, 343
    iteration history table, 353
    iteration report table, 353
    linear kernels, 350
    misclassification matrix table, 353
    model information table, 352
    multithreading, 21
    nonlinear kernels, 350
    number of observations table, 352
    ODS table names, 353
    optimization, 351
    output data tables, 348
    Predicted probability variables table, 353
    Predicted target variable table, 353
    scoring process, 352
    test data, 20
    training results table, 353
    validation, 20

target variables
    NNET procedure, 266
test data
    ASTORE procedure, 20
    BOOLRULE procedure, 20
    FACTMAC procedure, 20
    FOREST procedure, 20
    GRADBOOST procedure, 20
    NNET procedure, 20
    SVMACHINE procedure, 20
    TEXTMINE procedure, 20
    TMSCORE procedure, 20
TEXTMINE procedure, 360
    cell weight, 368
    computational method, 21
    coordinate list (COO) format, 382
    entity, 368
    filtering term by frequency, 370

hyperparameter tuning, 22
input data tables, 366
language used by input data tables, 366
multiterm words list, 368
multithreading, 21
noun groups, 368
number of threads, 366
show dropped terms, 370
sparse format, 382
sparse matrix, 385
start list, 370
stemming, 369
stop list, 370
SVD, singular value decomposition, 382
synonym list, 370
system configuration, 388
tagging, 369
term weight, 371
test data, 20
transactional style, 385
validation, 20
variable name style, 366
TEXTMINE procedure, system configuration
configuration for language binary files, 388
deploying language binary files, 388
GRID_TEXTANALYTICS_BIN_LOC macro, 389
language binary files, 388
prerequisite, 388
TMSCORE procedure, 409
computational method, 21
hyperparameter tuning, 22
input data tables, 414
multithreading, 21
system configuration, 416
test data, 20
validation, 20
TMSCORE procedure, system configuration
configuration for language binary files, 416
prerequisite, 416
training a decision tree
FOREST procedure, 152
GRADBOOST procedure, 188
training results table
SVMACHINE procedure, 353
transactional style
TEXTMINE procedure, 385
tuner information
FOREST procedure, 159
GRADBOOST procedure, 195
NNET procedure, 272
tuner results
FOREST procedure, 160
GRADBOOST procedure, 195
NNET procedure, 272
tuner summary
FOREST procedure, 160
GRADBOOST procedure, 195
NNET procedure, 272
tuner timing
FOREST procedure, 160
GRADBOOST procedure, 195
NNET procedure, 272

validation
ASTORE procedure, 20
BOOLRULE procedure, 20
FACTMAC procedure, 20
FOREST procedure, 20
GRADBOOST procedure, 20
NNET procedure, 20
SVMACHINE procedure, 20
TEXTMINE procedure, 20
TMSCORE procedure, 20
validation information table
BNET procedure, 60
variable importance
FOREST procedure, 159
GRADBOOST procedure, 194
rba variable importance, 194
variable information table
BNET procedure, 60
variable level table
BNET procedure, 60
variable order table
BNET procedure, 60
variable selection, 47, 57
BNET procedure, 51, 52

weight variable
NNET procedure, 269

Syntax Index

ACT= option
  HIDDEN statement, 261
  TARGET statement, 266
ACTSET option
  SOLVER statement, 309
ADDTREES option
  PROC GRADBOOST statement, 175
ALGORITHM= option
  OPTIMIZATION statement, 263
ANNEALINGRATE=option
  OPTIMIZATION statement, 263
ARCHITECTURE statement
  NNET procedure, 255
ASCENDING option
  TARGET statement, 350
ASSIGNMISSING= option
  PROC FOREST statement, 139
  PROC GRADBOOST statement, 175
ASTORE procedure
  PROC ASTORE statement, 31
ASTORE procedure, CODE statement
  COMMENT option, 19
  FILE= option, 19
  FORMATWIDTH= option, 19
  INDENTSIZE= option, 19
  LABELID= option, 19
  LINESIZE= option, 19
  NOTRIM option, 19
  OUT= option, 19
  PCATALL option, 19
ASTORE procedure, DESCRIBE statement, 32
  EPCODE= option, 32
  RSTORE= option, 32
  STORE= option, 32
ASTORE procedure, DOWNLOAD statement, 32
  RSTORE= option, 32
  STORE= option, 33
ASTORE procedure, PROC ASTORE statement, 31
  DATA= option, 33
ASTORE procedure, SCORE statement, 33
  COPYVAR= option, 34
  OUT= option, 33
ASTORE procedure, SETOPTION statement, 34
ASTORE procedure, syntax, 31
ASTORE procedure, UPLOAD statement, 34
  RSTORE= option, 34
  STORE= option, 34
AUTOTUNE statement
  ASTORE procedure, 14
  BOOLRULE procedure, 14
  FACTMAC procedure, 14
  FOREST procedure, 14, 143
  GRADBOOST procedure, 14, 105, 179
  NNET procedure, 14, 256
  SVMACHINE procedure, 14, 344
  TEXTMINE procedure, 14
  TMSCORE procedure, 14

BINMETHOD = option
  PROC FOREST statement, 140
  PROC GRADBOOST statement, 175
BNET procedure, 48
  CODE statement, 53
  FREQ statement, 53
  ID statement, 53
  INPUT statement, 54
  OUTPUT statement, 54
  PROC BNET statement, 48
  SAVESTATE statement, 55
  syntax, 48
  TARGET statement, 56
BNET procedure, OUTPUT statement
  COPYVARS= option, 55
  OUT= option, 54
  ROLE= option, 55
BNET procedure, PARTITION statement
  FRACTION option, 55
  ROLEVAR= option, 55
BNET procedure, PROC BNET statement
  DATA= option, 49
  NTHREADS= option, 51
BNET procedure, SAVESTATE statement
  RSTORE= option, 56
BOOLRULE procedure, 75
  DOCINFO statement, 78
  OUTPUT statement, 79
  PROC BOOLRULE statement, 75
  SCORE statement, 80
  syntax, 75
  TERMINFO statement, 80
BOOLRULE procedure, CODE statement
  COMMENT option, 19
  FILE= option, 19
  FORMATWIDTH= option, 19
  INDENTSIZE= option, 19
  LABELID= option, 19

LINESIZE= option, 19
NOTRIM option, 19
OUT= option, 19
PCATALL option, 19
BOOLRULE procedure, DOCINFO statement, 78
EVENTS= option, 78
ID= option, 78
TARGET= option, 79
TARGETTYPE= option, 79
BOOLRULE procedure, OUTPUT statement, 79
CANDIDATETERMS= option, 79
RULES= option, 79
RULETERMS= option, 80
BOOLRULE procedure, PROC BOOLRULE statement, 75
DATA= option, 76
DOC= option, 76
DOCID= option, 77
DOCINFO= option, 77
GNEG= option, 77
GPOS= option, 77
MAXCANDIDATES= option, 77
MAXCANDS= option, 77
MAXTRIESIN= option, 77
MAXTRIESOUT= option, 77
MINSUPPORTS= option, 77
MNEG= option, 78
MPOS= option, 78
TERMID= option, 78
TERMINFO= option, 78
BOOLRULE procedure, SCORE statement, 80
OUTMATCH= option, 80
RULETERMS= option, 80
BOOLRULE procedure, TERMINFO statement, 80
ID= option, 81
LABEL= option, 81

C= option
PROC SVMACHINE statement, 343
CANDIDATETERMS= option
OUTPUT statement, 79
CASESENSITIVE option
DISPLAY statement (GVARCLUS), 210
DISPLAY statement (MWPCA), 242
DISPLAY statement (RPCA), 290
CELLWGT= option
PARSE statement, 368
CENTER option
PROC MWPCA statement, 241
PROC RPCA statement, 287
CHAID option
GROW statement (FOREST), 148
CHISQUARE option
GROW statement (FOREST), 148
CODE statement
ASTORE procedure, 18, 32
BNET procedure, 53
BOOLRULE procedure, 18
FACTMAC procedure, 18, 107
FOREST procedure, 18, 147
GRADBOOST procedure, 18, 183
NNET procedure, 18, 260
RPCA procedure, 289
SVDD procedure, 307
SVMACHINE procedure, 18, 347
TEXTMINE procedure, 18
TMSCORE procedure, 18
COL= option
SVD statement, 373
COLSTATISTICS= option
PROC RPCA statement, 287
COMB= option
HIDDEN statement, 262
TARGET statement, 267
COMMENT option
CODE statement (ASTORE), 19
CODE statement (BOOLRULE), 19
CODE statement (FACTMAC), 19
CODE statement (FOREST), 19
CODE statement (GRADBOOST), 19
CODE statement (NNET), 19
CODE statement (SVMACHINE), 19
CODE statement (TEXTMINE), 19
CODE statement (TMSCORE), 19
COMMFREQ=option
OPTIMIZATION statement, 263
CONF= option
PROC MBANALYSIS statement, 224
CONFIG= option
TMSCORE statement, 415
COPYVAR= option
SCORE statement (ASTORE), 34
COPYVARS= option
OUTPUT statement, 55, 109, 150, 185, 265, 349
CROSSVALIDATION statement
FOREST procedure, 147
GRADBOOST procedure, 184
NNET procedure, 261
CTOL= option
SOLVER statement, 310
CUSTOMER statement
FISM procedure, 126
MBANALYSIS procedure, 225

DATA= option
HIERARCHY statement, 226
PROC ASTORE statement, 33
PROC BNET statement, 49

PROC BOOLRULE statement, 76
PROC FACTMAC statement, 103
PROC FASTKNN statement, 118
PROC FISM statement, 126
PROC FOREST statement, 140
PROC GRADBOOST statement, 176
PROC GVARCLUS statement, 207
PROC MBANALYSIS statement, 224
PROC MWPCA statement, 241
PROC NNET statement, 253
PROC RPCA statement, 287
PROC SVDD statement, 306
PROC SVMACHINE statement, 343
PROC TEXTMINE statement, 366
PROC TMSCORE statement, 414
DECOMP= option
PROC RPCA statement, 288
DESCENDING option
TARGET statement, 350
DISPLAY statement
GVARCLUS procedure, 209
MWPCA procedure, 242
RPCA procedure, 290
DISPLAYOUT statement
GVARCLUS procedure, 210
MWPCA procedure, 243
RPCA procedure, 291
DOC= option
PROC BOOLRULE statement, 76
PROC TMSCORE statement, 414
DOC_ID statement
TEXTMINE procedure, 367
TMSCORE procedure, 416
DOCID= option
PROC BOOLRULE statement, 77
DOCINFO statement
BOOLRULE procedure, 78
DOCINFO= option
PROC BOOLRULE statement, 77
DOWNLOAD statement
ASTORE procedure, 32
DROPOUTHIDDEN= option
TRAIN statement, 268
DROPOUTINPUT= option
TRAIN statement, 268

EARLYSTOP option
PROC GRADBOOST statement, 176
ENTITIES= option
PARSE statement, 368
ENTROPY option
GROW statement (FOREST), 148
ENTRY= option
SVD statement, 373
EPCODE= option
DESCRIBE statement, 32
ERROR= option
TARGET statement, 267
EVALHISTORY= option
AUTOTUNE statement, 14
EVENTS= option
DOCINFO statement, 78
EXACT option
PROC GVARCLUS statement, 208
EXACTWEIGHT option
SVD statement, 374
EXCLUDE option
DISPLAY statement (GVARCLUS), 210
DISPLAY statement (MWPCA), 243
DISPLAY statement (RPCA), 291
EXCLUDEALL option
DISPLAY statement (GVARCLUS), 210
DISPLAY statement (MWPCA), 243
DISPLAY statement (RPCA), 291

FACTMAC procedure, AUTOTUNE statement, 105
FACTMAC procedure, CODE statement, 107
COMMENT option, 19
FILE= option, 19
FORMATWIDTH= option, 19
INDENTSIZE= option, 19
LABELID= option, 19
LINESIZE= option, 19
NOTRIM option, 19
OUT= option, 19
PCATALL option, 19
FACTMAC procedure, ID statement, 108
FACTMAC procedure, INPUT statement, 108
FACTMAC procedure, OUTPUT statement, 108
COPYVARS= option, 109
LEVEL= option, 109
OUT= option, 108
FACTMAC procedure, PROC FACTMAC statement, 103
DATA= option, 103
FILE= option, 107
LEARNSTEP= option, 104
LEVEL= option, 108
MAXITER= option, 104
NFACTORS= option, 104
NONNEGATIVE option, 104
NOPRINT, 104
NTHREADS= option, 104
OUTMODEL, 104
SEED= option, 104
FACTMAC procedure, SAVESTATE statement, 109
RSTORE= option, 109
FACTMAC procedure, syntax, 103

FACTMAC procedure, TARGET statement, 109
FASTKNN procedure, ID statement, 119
FASTKNN procedure, INPUT statement, 119
FASTKNN procedure, OUTPUT statement, 119
    OUT= option, 119
FASTKNN procedure, PROC FASTKNN statement, 118
    DATA= option, 118
    K= option, 118
    NOPRINT option, 118
    NTHREADS= option, 118
    OUTDIST= option, 119
    QUERY= option, 119
    THRESHDIST= option, 119
FASTKNN procedure, syntax, 117
FILE= option
    CODE statement, 261, 290, 307, 347
    CODE statement (ASTORE), 19
    CODE statement (BOOLRULE), 19
    CODE statement (FACTMAC), 19
    CODE statement (FOREST), 19
    CODE statement (GRADBOOST), 19
    CODE statement (NNET), 19
    CODE statement (SVMACHINE), 19
    CODE statement (TEXTMINE), 19
    CODE statement (TMSCORE), 19
    PROC FACTMAC statement, 107
FISM procedure, CUSTOMER statement, 126
FISM procedure, OUTPUT statement, 126
    OUT= option, 127
    OUTFREQ= option, 127
FISM procedure, PROC FISM statement, 125
    DATA= option, 126
    ITEMS= option, 126
    PCTSUP= option, 126
    SUPPORT= option, 126
FISM procedure, syntax, 125
FISM procedure, TARGET statement, 127
FIXEDMU option
    RPCA statement, 245
    PROC RPCA statement, 288
FOREST procedure, AUTOTUNE statement, 143
    TUNINGPARAMETERS= option, 144
FOREST procedure, CODE statement, 147
    COMMENT option, 19
    FILE= option, 19
    FORMATWIDTH= option, 19
    INDENTSIZE= option, 19
    LABELID= option, 19
    LINESIZE= option, 19
    NOTRIM option, 19
    OUT= option, 19
    PCATALL option, 19
FOREST procedure, CROSSVALIDATION statement, 147
    KFOLD= option, 147
FOREST procedure, GROW statement, 148
    CHAID option, 148
    CHISQUARE option, 148
    ENTROPY option, 148
    FTEST option, 148
    GINI option, 148
    IGR option, 148
    RSS option, 148
FOREST procedure, ID statement, 149
FOREST procedure, INPUT statement, 149
    LEVEL= option, 149
FOREST procedure, OUTPUT statement, 149
    COPYVARS= option, 150
    OUT= option, 149
    ROLE option, 150
FOREST procedure, PARTITION statement
    FRACTION option, 20, 150
    ROLEVAR= option, 20, 150
FOREST procedure, PROC FOREST statement, 139
    ASSIGNMISSING= option, 139
    BINMETHOD = option, 140
    DATA= option, 140
    INBAGFRACTION= option, 141
    INMODEL= option, 141
    LOH= option, 141
    MAXBRANCH= option, 141
    MAXDEPTH= option, 141
    MINLEAFSIZE= option, 141
    MINUSEINSEARCH= option, 141
    NOPRINT option, 142
    NTREES= option, 142
    NUMBIN= option, 142
    OUTMODEL= option, 142
    PRINTTARGET option, 142
    RBAIMP option, 142
    SEED= option, 142
    VARS_TO_TRY= option, 142
    VII= option, 143
    VOTE= option, 143
FOREST procedure, SAVESTATE statement, 151
    RSTORE= option, 151
FOREST procedure, syntax, 138
FOREST procedure, TARGET statement, 151
    LEVEL= option, 151
FORMATWIDTH= option
    CODE statement (ASTORE), 19
    CODE statement (BOOLRULE), 19
    CODE statement (FACTMAC), 19
    CODE statement (FOREST), 19
    CODE statement (GRADBOOST), 19
    CODE statement (NNET), 19

CODE statement (SVMACHINE), 19
CODE statement (TEXTMINE), 19
CODE statement (TMSCORE), 19
FRACTION option
PARTITION statement, 266
PARTITION statement (BNET), 55
PARTITION statement (FOREST), 20, 150
PARTITION statement (GRADBOOST), 20, 186
PARTITION statement (SVMACHINE), 349
FRACTION= option
AUTOTUNE statement, 15
FREQ statement
BNET procedure, 53
GVARCLUS procedure, 211
FTEST option
GROW statement (FOREST), 148

GINI option
GROW statement (FOREST), 148
GLIM option
ARCHITECTURE statement, 255
GNEG= option
PROC BOOLRULE statement, 77
GPOS= option
PROC BOOLRULE statement, 77
GRADBOOST procedure, AUTOTUNE statement, 179
TUNINGPARAMETERS= option, 105, 180
GRADBOOST procedure, CODE statement, 183
COMMENT option, 19
FILE= option, 19
FORMATWIDTH= option, 19
INDENTSIZE= option, 19
LABELID= option, 19
LINESIZE= option, 19
NOTRIM option, 19
OUT= option, 19
PCATALL option, 19
GRADBOOST procedure, CROSSVALIDATION statement, 184
KFOLD= option, 184
GRADBOOST procedure, ID statement, 184
GRADBOOST procedure, INPUT statement, 185
LEVEL= option, 185
GRADBOOST procedure, OUTPUT statement, 185
COPYVARS= option, 185
OUT= option, 185
ROLE option, 185
GRADBOOST procedure, PARTITION statement
FRACTION option, 20, 186
ROLEVAR= option, 20, 186
GRADBOOST procedure, PROC GRADBOOST statement, 174
ADDTREES option, 175
ASSIGNMISSING= option, 175
BINMETHOD = option, 175
DATA= option, 176
EARLYSTOP option, 176
INMODEL= option, 176
LASSO= option, 177
LEARNINGRATE= option, 177
MAXBRANCH= option, 177
MAXDEPTH= option, 177
MINLEAFSIZE= option, 177
MINUSEINSEARCH= option, 177
NOPRINT option, 177
NTREES= option, 177
NUMBIN= option, 178
OUTMODEL option, 178
PRINTTARGET option, 178
RBAIMP option, 178
RIDGE= option, 178
SAMPLINGRATE option, 178
SEED= option, 178
VARS_TO_TRY= option, 178
VII= option, 179
GRADBOOST procedure, SAVESTATE statement, 186
RSTORE= option, 187
GRADBOOST procedure, syntax, 173
GRADBOOST procedure, TARGET statement, 187
LEVEL= option, 187
GROUP= option
SELECT statement, 372
GROW statement
FOREST procedure, 148
GVARCLUS procedure, DISPLAY statement
CASESENSITIVE option, 210
EXCLUDE option, 210
EXCLUDEALL option, 210
TRACE option, 210
GVARCLUS procedure, DISPLAYOUT statement
INCLUDEALL option, 211
NOREPLACE option, 211
REPEATED option, 211
GVARCLUS procedure, INPUT statement, 211
LEVEL= option, 211
GVARCLUS procedure, PROC GVARCLUS statement, 206
DATA= option, 207
EXACT option, 208
MAXITER= option, 208
MAXMEMBER= option, 208
MAXSTEP= option, 208
MINCLUSTER= option, 208
OUTCP= option, 208
OUTEDGE= option, 208
OUTTREE= option, 209

OUTVERT= option, 209
RHO= option, 209
STOP= option, 209
TOL= option, 209
TOLERANCE= option, 209
GVARCLUS procedure, syntax, 206

HIDDEN statement
NNET procedure, 261
HIERARCHY statement
MBANALYSIS procedure, 225

ID statement
BNET procedure, 53
FACTMAC procedure, 108
FASTKNN procedure, 119
FOREST procedure, 149
GRADBOOST procedure, 184
MWPCA procedure, 243
RPCA procedure, 291
SVDD procedure, 307
SVMACHINE procedure, 347
ID= option
DOCINFO statement, 78
TERMINFO statement, 81
IGNORE option
SELECT statement, 372
IGR option
GROW statement (FOREST), 148
IN_TERMS= option
SVD statement, 374
INBAGFRACTION= option
PROC FOREST statement, 141
INCLUDEALL option
DISPLAYOUT statement (GVARCLUS), 211
DISPLAYOUT statement (MWPCA), 243
DISPLAYOUT statement (RPCA), 291
INDENTSIZE= option
CODE statement (ASTORE), 19
CODE statement (BOOLRULE), 19
CODE statement (FACTMAC), 19
CODE statement (FOREST), 19
CODE statement (GRADBOOST), 19
CODE statement (NNET), 19
CODE statement (SVMACHINE), 19
CODE statement (TEXTMINE), 19
CODE statement (TMSCORE), 19
INMODEL= option
PROC FOREST statement, 141
PROC GRADBOOST statement, 176
PROC NNET statement, 254
INPUT statement
BNET procedure, 54
FACTMAC procedure, 108
FASTKNN procedure, 119
FOREST procedure, 149
GRADBOOST procedure, 185
GVARCLUS procedure, 211
MWPCA procedure, 244
NNET procedure, 262
RPCA procedure, 292
SVDD procedure, 308
SVMACHINE procedure, 347
ITEMS= option
PROC FISM statement, 126
PROC MBANALYSIS statement, 225
ITERATIONREPORT option
PROC SVMACHINE statement, 343

K= option
PROC FASTKNN statement, 118
SVD statement, 374
KEEP option
SELECT statement, 372
KEEPVARS, KEEPVARIABLES
SVD statement, 374
KERNEL statement
SVDD procedure, 308
SVMACHINE procedure, 348
KFOLD= option
AUTOTUNE statement, 15
CROSSVALIDATION statement, 147, 184, 261

LABEL= option
TERMINFO statement, 81
LABELID= option
CODE statement (ASTORE), 19
CODE statement (BOOLRULE), 19
CODE statement (FACTMAC), 19
CODE statement (FOREST), 19
CODE statement (GRADBOOST), 19
CODE statement (NNET), 19
CODE statement (SVMACHINE), 19
CODE statement (TEXTMINE), 19
CODE statement (TMSCORE), 19
LABELS option
SELECT statement, 372
LAMBDA= option
RPCA statement, 245
PROC RPCA statement, 288
LAMBDAWEIGHT= option
RPCA statement, 245
PROC RPCA statement, 288
LANGUAGE= option
PROC TEXTMINE statement, 366
LASSO= option
PROC GRADBOOST statement, 177
LEARNINGRATE= option

PROC GRADBOOST statement, 177
LEARNINGRATE=option
OPTIMIZATION statement, 263
LEARNSTEP= option
PROC FACTMAC statement, 104
LEVEL= option
INPUT statement, 149, 185, 211, 262, 308, 347
OUTPUT statement, 109
PROC FACTMAC statement, 108
TARGET statement, 151, 187, 267
LIFT= option
PROC MBANALYSIS statement, 225
LINEAR option
PROC SVMACHINE statement, 348
LINESIZE= option
CODE statement (ASTORE), 19
CODE statement (BOOLRULE), 19
CODE statement (FACTMAC), 19
CODE statement (FOREST), 19
CODE statement (GRADBOOST), 19
CODE statement (NNET), 19
CODE statement (SVMACHINE), 19
CODE statement (TEXTMINE), 19
CODE statement (TMSCORE), 19
LOH= option
PROC FOREST statement, 141

MAX_K= option
SVD statement, 374
MAXBRANCH= option
PROC FOREST statement, 141
PROC GRADBOOST statement, 177
MAXBSKTSZ= option
PROC MBANALYSIS statement, 225
MAXCANDIDATES= option
PROC BOOLRULE statement, 77
MAXCANDS= option
PROC BOOLRULE statement, 77
MAXDEPTH= option
PROC FOREST statement, 141
PROC GRADBOOST statement, 177
MAXEVALS= option
AUTOTUNE statement, 15
MAXITER= option
AUTOTUNE statement, 16
OPTIMIZATION statement, 264
PROC FACTMAC statement, 104
PROC GVARCLUS statement, 208
PROC RPCA statement, 288
PROC SVMACHINE statement, 344
RPCA statement, 245
SOLVER statement, 309
MAXMEMBER= option
PROC GVARCLUS statement, 208
MAXRANK= option
SVD statement, 246, 293
MAXSV= option
SOLVER statement, 309
MAXTIME= option
AUTOTUNE statement, 16
OPTIMIZATION statement, 264
MAXTRIESIN= option
PROC BOOLRULE statement, 77
MAXTRIESOUT= option
PROC BOOLRULE statement, 77
MBANALYSIS procedure, CUSTOMER statement, 225
MBANALYSIS procedure, HIERARCHY statement, 225
DATA= option, 226
MBANALYSIS procedure, OUTPUT statement, 226
OUT= option, 226
OUTFREQ= option, 226
OUTRULE= option, 226
MBANALYSIS procedure, PROC MBANALYSIS statement, 224
CONF= option, 224
DATA= option, 224
ITEMS= option, 225
LIFT= option, 225
MAXBSKTSZ= option, 225
MINBSKTSZ= option, 225
NORM= option, 225
PCTSUP= option, 224
SUP_LIFT= option, 225
SUPPORT= option, 224
MBANALYSIS procedure, SAVESTATE statement, 226
RSTORE= option, 227
MBANALYSIS procedure, syntax, 223
MBANALYSIS procedure, TARGET statement, 227
METHOD= option
PROC RPCA statement, 288
RPCA statement, 245
SVD statement, 246, 293
MINBSKTSZ= option
PROC MBANALYSIS statement, 225
MINCLUSTER= option
PROC GVARCLUS statement, 208
MINIBATCHSIZE=option
OPTIMIZATION statement, 263
MINLEAFSIZE= option
PROC FOREST statement, 141
PROC GRADBOOST statement, 177
MINSUPPORTS= option
PROC BOOLRULE statement, 77
MINUSEINSEARCH= option
PROC FOREST statement, 141

PROC GRADBOOST statement, 177
MISSING= option
PROC NNET statement, 254
MLP DIRECT option
ARCHITECTURE statement, 255
MLP option
ARCHITECTURE statement, 255
MNEG= option
PROC BOOLRULE statement, 78
MOMENTUM=option
OPTIMIZATION statement, 264
MPOS= option
PROC BOOLRULE statement, 78
MU= option
RPCA statement, 246
PROC RPCA statement, 288
MULTITERM= option
PARSE statement, 368
MWPCA procedure, DISPLAY statement
CASESENSITIVE option, 242
EXCLUDE option, 243
EXCLUDEALL option, 243
TRACE option, 243
MWPCA procedure, DISPLAYOUT statement
INCLUDEALL option, 243
NOREPLACE option, 243
REPEATED option, 243
MWPCA procedure, ID statement, 243
MWPCA procedure, INPUT statement, 244
MWPCA procedure, OUTPUT statement, 244
NPC= option, 244
OUT= option, 244
STANDARDPC= option, 245
MWPCA procedure, PROC MWPCA statement, 241
CENTER, 241
DATA= option, 241
NTHREADS= option, 242
ROBUST= option, 242
STEPSIZE= option, 241
WINDOWSIZE= option, 241
MWPCA procedure, RPCA statement, 245
FIXEDMU option, 245
LAMBDA= option, 245
LAMBDAWEIGHT= option, 245
MAXITER= option, 245
METHOD= option, 245
MU= option, 246
TOLERANCE= option, 246
MWPCA procedure, SVD statement, 246
MAXRANK= option, 246
MWPCA procedure, syntax, 240

NBINS option
PROC SVDD statement, 306
NEWVARNAMES
TEXTMINE statement, 366
NFACTORS= option
PROC FACTMAC statement, 104
NMATCH = option
SOLVER statement, 310
NNET procedure, ARCHITECTURE statement, 255
GLIM option, 255
MLP DIRECT option, 255
MLP option, 255
NNET procedure, AUTOTUNE statement, 256
TUNINGPARAMETERS= option, 256
NNET procedure, CODE statement, 260
COMMENT option, 19
FILE= option, 19, 261
FORMATWIDTH= option, 19
INDENTSIZE= option, 19
LABELID= option, 19
LINESIZE= option, 19
NOCOMPPGM option, 261
NOTRIM option, 19
OUT= option, 19
PCATALL option, 19
NNET procedure, CROSSVALIDATION statement, 261
KFOLD= option, 261
NNET procedure, HIDDEN statement, 261
ACT= option, 261
COMB= option, 262
NNET procedure, INPUT statement, 262
LEVEL= option, 262
NNET procedure, OPTIMIZATION statement, 262
ALGORITHM= option, 263
ANNEALINGRATE= option, 263
COMMFREQ= option, 263
LEARNINGRATE= option, 263
MAXITER= option, 264
MAXTIME= option, 264
MINIBATCHSIZE= option, 263
MOMENTUM= option, 264
REGL1=option, 264
REGL2= option, 264
SEED= option, 264
USELOCKING option, 264
NNET procedure, OUTPUT statement, 264
COPYVARS= option, 265
OUT= option, 265
ROLE= option, 265
NNET procedure, PARTITION statement, 265
FRACTION option, 266
ROLEVAR= option, 265
NNET procedure, PROC NNET statement, 253
DATA= option, 253
INMODEL= option, 254

MISSING= option, 254
NTHREADS= option, 254
PRINTTARGET, 254
STANDARDIZE= option, 254
TARGETSTANDARDIZE= option, 255
NNET procedure, syntax, 253
NNET procedure, TARGET statement, 266
ACT= option, 266
COMB= option, 267
ERROR= option, 267
LEVEL= option, 267
NNET procedure, TRAIN statement, 267
DROPOUTHIDDEN=*ratio*, 268
DROPOUTINPUT= option, 268
NUMTRIES= option, 268
OUTMODEL= option, 268
RESUME option, 268
STAGNATION= option, 268
VALIDATION= option, 269
VALIDGOAL= option, 269
WSEED= option, 269
NNET procedure, WEIGHT statement, 269
NOCOMPPGM option
CODE statement, 261
NOCUTOFFS option
SVD statement, 374
NONG option
PARSE statement, 368
NONNEGATIVE option
PROC FACTMAC statement, 104
NONOUNGROUPS option
PARSE statement, 368
NOPRINT
PROC FACTMAC statement, 104
NOPRINT option
PROC FASTKNN statement, 118
PROC FOREST statement, 142
PROC GRADBOOST statement, 177
PROC SVDD statement, 306
PROC SVMACHINE statement, 344
NOREPLACE option
DISPLAYOUT statement (GVARCLUS), 211
DISPLAYOUT statement (MWPCA), 243
DISPLAYOUT statement (RPCA), 291
NORM= option
PROC MBANALYSIS statement, 225
NOSCALE option
PROC SVMACHINE statement, 344
NOSTEMMING option
PARSE statement, 369
NOTAGGING option
PARSE statement, 369
NOTRIM option
CODE statement (ASTORE), 19
CODE statement (BOOLRULE), 19
CODE statement (FACTMAC), 19
CODE statement (FOREST), 19
CODE statement (GRADBOOST), 19
CODE statement (NNET), 19
CODE statement (SVMACHINE), 19
CODE statement (TEXTMINE), 19
CODE statement (TMSCORE), 19
NPC= option
OUTPUT statement, 244
NSAMP= option
SOLVER statement, 310
NSUBSESSIONWORKERS= option
AUTOTUNE statement, 16
NTHREADS= option
PROC FACTMAC statement, 104
PROC FASTKNN statement, 118
PROC MWPCA statement, 242
PROC NNET statement, 254
PROC RPCA statement, 289
PROC SVDD statement, 306
PROC SVMACHINE statement, 344
PROC TEXTMINE statement, 366
NTREES= option
PROC FOREST statement, 142
PROC GRADBOOST statement, 177
NUMBIN= option
PROC FOREST statement, 142
PROC GRADBOOST statement, 178
NUMLABELS= option
SVD statement, 374
NUMTRIES= option
TRAIN statement, 268

OBJECTIVE= option
AUTOTUNE statement, 16
OPTIMIZATION statement
NNET procedure, 262
OUT= option
CODE statement (ASTORE), 19
CODE statement (BOOLRULE), 19
CODE statement (FACTMAC), 19
CODE statement (FOREST), 19
CODE statement (GRADBOOST), 19
CODE statement (NNET), 19
CODE statement (SVMACHINE), 19
CODE statement (TEXTMINE), 19
CODE statement (TMSCORE), 19
OUTPUT statement, 127, 226
OUTPUT statement (BNET), 54
OUTPUT statement (FACTMAC), 108
OUTPUT statement (FASTKNN), 119
OUTPUT statement (FOREST), 149
OUTPUT statement (GRADBOOST), 185

OUTPUT statement (MWPCA), 244
OUTPUT statement (NNET), 265
OUTPUT statement (SVMACHINE), 348
SCORE statement (ASTORE), 33
OUTCHILD= option
PARSE statement, 369
OUTCONFIG= option
PARSE statement, 369
OUTCP= option
PROC GVARCLUS statement, 208
OUTDECOMP statement
RPCA procedure, 292
OUTDIST= option
PROC FASTKNN statement, 119
OUTDOCPRO= option
SVD statement, 374
OUTEDGE= option
PROC GVARCLUS statement, 208
OUTERROR= option
PROC RPCA statement, 289
OUTFREQ= option
OUTPUT statement, 127, 226
OUTLIER_FRACTION= option
PROC SVDD statement, 306
OUTLOWRANK= option
PROC RPCA statement, 289
OUTMATCH= option
SCORE statement, 80
OUTMODEL
PROC FACTMAC statement, 104
OUTMODEL option
PROC GRADBOOST statement, 178
OUTMODEL= option
PROC FOREST statement, 142
TRAIN statement, 268
OUTPARENT= option
PARSE statement, 369
PROC TMSCORE statement, 415
OUTPOS= option
PARSE statement, 369
OUTPUT statement
BNET procedure, 54
BOOLRULE procedure, 79
FACTMAC procedure, 108
FASTKNN procedure, 119
FISM procedure, 126
FOREST procedure, 149
GRADBOOST procedure, 185
MBANALYSIS procedure, 226
MWPCA procedure, 244
NNET procedure, 264
SVMACHINE procedure, 348
OUTRULE= option
OUTPUT statement, 226
OUTSPARSE= option
PROC RPCA statement, 289
OUTSV= option
PROC SVDD statement, 307
OUTTERMS= option
PARSE statement, 370
OUTTOPICS= option
SVD statement, 375
OUTTREE= option
PROC GVARCLUS statement, 209
OUTVERT= option
PROC GVARCLUS statement, 209

PARSE statement
TEXTMINE procedure, 367
PARTITION statement
ASTORE procedure, 19
BNET procedure, 55
BOOLRULE procedure, 19
FACTMAC procedure, 19
FOREST procedure, 19, 20, 150
GRADBOOST procedure, 19, 20, 186
NNET procedure, 19, 265
SVMACHINE procedure, 19, 349
TEXTMINE procedure, 19
TMSCORE procedure, 19
PCATALL option
CODE statement (ASTORE), 19
CODE statement (BOOLRULE), 19
CODE statement (FACTMAC), 19
CODE statement (FOREST), 19
CODE statement (GRADBOOST), 19
CODE statement (NNET), 19
CODE statement (SVMACHINE), 19
CODE statement (TEXTMINE), 19
CODE statement (TMSCORE), 19
PCLOADINGS= option
OUTDECOMP statement, 292
PCSCORES= option
OUTDECOMP statement, 293
PCTSUP= option
PROC FISM statement, 126
PROC MBANALYSIS statement, 224
POLYNOMIAL option
PROC SVMACHINE statement, 348
POPSIZE= option
AUTOTUNE statement, 16, 17
POWER= option
SVD statement, 294
PREFIX= option
PROC RPCA statement, 289
PRINTTARGET
PROC NNET statement, 254
PRINTTARGET option

PROC BNET statement, 52
PROC FOREST statement, 142
PROC GRADBOOST statement, 178
PROC SVMACHINE statement, 344
PROC ASTORE statement
ASTORE procedure, 31
PROC BNET statement
ALPHA= option, 49
BESTMODEL option, 49
BNET procedure, 48
INDEPTEST= option, 50
MAXPARENTS= option, 50
MIALPHA= option, 50
MISSINGINT= option, 50
MISSINGNOM= option, 50
NTHREADS= option, 51
NUMBIN= option, 51
OUTNETWORK= option, 51
PARENTING= option, 51
PRESCREENING= option, 51
STRUCTURE= option, 52
VARSELECT= option, 52
PROC BOOLRULE statement
BOOLRULE procedure, 75
PROC FACTMAC statement, *see* FACTMAC procedure
PROC FASTKNN statement, *see* FASTKNN procedure
PROC FOREST statement, *see* FOREST procedure
PROC GRADBOOST statement, *see* GRADBOOST procedure
PROC GVARCLUS statement, *see* GVARCLUS procedure
PROC MWPCA statement, *see* MWPCA procedure
PROC NNET statement, *see* NNET procedure
PROC RPCA statement, *see* RPCA procedure
PROC SVDD statement, *see* SVDD procedure
PROC SVMACHINE statement, *see* SVMACHINE procedure
PROC TEXTMINE statement
TEXTMINE procedure, 365
PROC TMSCORE statement
TMSCORE procedure, 414

QUERY= option
PROC FASTKNN statement, 119

RBAIMP option
PROC FOREST statement, 142
PROC GRADBOOST statement, 178
RBF option
KERNEL statement, 308
REDUCEF= option
PARSE statement, 370
REGL1=option
OPTIMIZATION statement, 264
REGL2= option
OPTIMIZATION statement, 264
REPEATED option
DISPLAYOUT statement (GVARCLUS), 211
DISPLAYOUT statement (MWPCA), 243
DISPLAYOUT statement (RPCA), 291
RES= option
SVD statement, 375
RESOLUTION= option
SVD statement, 375
RESUME option
TRAIN statement, 268
RHO= option
PROC GVARCLUS statement, 209
RIDGE= option
PROC GRADBOOST statement, 178
ROBUST= option
PROC MWPCA statement, 242
ROLE option
OUTPUT statement, 150, 185
ROLE= option
OUTPUT statement, 55, 265, 349
ROLEVAR= option
PARTITION statement, 265
PARTITION statement (BNET), 55
PARTITION statement (FOREST), 20, 150
PARTITION statement (GRADBOOST), 20, 186
PARTITION statement (SVMACHINE), 349
ROTATION= option
SVD statement, 375
ROW= option
SVD statement, 376
RPCA procedure, CODE statement, 289
FILE= option, 290
RPCA procedure, DISPLAY statement
CASESENSITIVE option, 290
EXCLUDE option, 291
EXCLUDEALL option, 291
TRACE option, 291
RPCA procedure, DISPLAYOUT statement
INCLUDEALL option, 291
NOREPLACE option, 291
REPEATED option, 291
RPCA procedure, ID statement, 291
RPCA procedure, INPUT statement, 292
RPCA procedure, OUTDECOMP statement, 292
PCLOADINGS= option, 292
PCSCORES= option, 293
SVDDIAG= option, 292
SVDLEFT= option, 292
SVDRIGHT= option, 292
RPCA procedure, PROC RPCA statement, 286

CENTER option, 287
COLSTATISTICS= option, 287
DATA= option, 287
DECOMP= option, 288
FIXEDMU option, 288
LAMBDA= option, 288
LAMBDAWEIGHT= option, 288
MAXITER= option, 288
METHOD= option, 288
MU= option, 288
NTHREADS= option, 289
OUTERROR= option, 289
OUTLOWRANK= option, 289
OUTSPARSE= option, 289
PREFIX= option, 289
SCALE option, 289
TOLERANCE= option, 289
RPCA procedure, SAVESTATE statement, 293
RSTORE=, 293
RPCA procedure, SVD statement, 293
METHOD= option, 293, 294
RPCA procedure, syntax, 286
RPCA statement
MWPCA procedure, 245
RSS option
GROW statement (FOREST), 148
RSTORE= option
DOWNLOAD statement, 32
PROC SVDD statement, 309
SAVESTATE statement, 56, 109, 151, 187, 227, 293, 350, 371
UPLOAD statement, 34
RSTORE=option
DESCRIBE statement, 32
RTOL= option
SOLVER statement, 310
RULES= option
OUTPUT statement, 79
RULETERMS= option
OUTPUT statement, 80
SCORE statement, 80

SAMPLESIZE= option
AUTOTUNE statement, 17
SAMPLINGRATE option
PROC GRADBOOST statement, 178
SAVESTATE statement
BNET procedure, 55
FACTMAC procedure, 109
FOREST procedure, 151
GRADBOOST procedure, 186
MBANALYSIS procedure, 226
RPCA procedure, 293
SVDD procedure, 309
SVMACHINE procedure, 350, 371
SCALE option
PROC RPCA statement, 289
PROC SVMACHINE statement, 344
SCORE option
PROC SVDD statement, 307
SCORE statement
ASTORE procedure, 33
BOOLRULE procedure, 80
SEARCHMETHOD= option
AUTOTUNE statement, 17
SEED= option
PROC FACTMAC statement, 104
PROC FOREST statement, 142
PROC GRADBOOST statement, 178
SOLVER statement, 310
SVD statement, 294
SEED=option
OPTIMIZATION statement, 264
SELECT statement
TEXTMINE procedure, 371
SETOPTION statement
ASTORE procedure, 34
SHOWDROPPEDTERMS= option
PARSE statement, 370
SOLVER statement
SVDD procedure, 309
STAGNATION= option
TRAIN statement, 268
STANDARDIZE= option
PROC NNET statement, 254
STANDARDPC= option
OUTPUT statement, 245
START= option
PARSE statement, 370
STD option
PROC SVDD statement, 307
STEPSIZE= option
PROC MWPCA statement, 241
STOCHS option
PROC SVDD statement, 309
STOL= option
SOLVER statement, 309
STOP= option
PARSE statement, 370
PROC GVARCLUS statement, 209
STORE= option
DESCRIBE statement, 32
DOWNLOAD statement, 33
UPLOAD statement, 34
SUP_LIFT= option
PROC MBANALYSIS statement, 225
SUPPORT= option
PROC FISM statement, 126

PROC MBANALYSIS statement, 224
SVD statement
MWPCA procedure, 246
RPCA procedure, 293
TEXTMINE procedure, 372
SVDD procedure, CODE statement, 307
FILE= option, 307
SVDD procedure, ID statement, 307
SVDD procedure, INPUT statement, 308
LEVEL= option, 308
SVDD procedure, KERNEL Statement
RBF option, 308
SVDD procedure, KERNEL statement, 308
SVDD procedure, PROC SVDD statement, 306
DATA= option, 306
NBINS, 306
NOPRINT, 306
NTHREADS= option, 306
OUTLIER_FRACTION= option, 306
OUTSV=, 307
RSTORE=, 309
SCORE, 307
STD option, 307
UCC option, 307
SVDD procedure, PROC SVDD statement, SOLVER statement
CTOL= option, 310
MAXITER= option, 309
MAXSV= option, 309
NMATCH= option, 310
NSAMP= option, 310
RTOL= option, 310
SEED option, 310
STOL= option, 309
SVDD procedure, SAVESTATE statement, 309
SVDD procedure, SOLVER statement, 309
ACTSET option, 309
STOCHS option, 309
SVDD procedure, syntax, 305
SVDD procedure, WEIGHT statement, 310
SVDDIAG= option
OUTDECOMP statement, 292
SVDDOCPRO= option
PROC TMSCORE statement, 415
SVDLEFT= option
OUTDECOMP statement, 292
SVDRIGHT= option
OUTDECOMP statement, 292
SVDS= option
SVD statement, 376
SVDU= option
PROC TMSCORE statement, 415
SVD statement, 376
SVDV= option
SVD statement, 376
SVMACHINE procedure, AUTOTUNE statement, 344
TUNINGPARAMETERS= option, 345
SVMACHINE procedure, CODE statement, 347
COMMENT option, 19
FILE= option, 19, 347
FORMATWIDTH= option, 19
INDENTSIZE= option, 19
LABELID= option, 19
LINESIZE= option, 19
NOTRIM option, 19
OUT= option, 19
PCATALL option, 19
SVMACHINE procedure, ID statement, 347
SVMACHINE procedure, INPUT statement, 347
LEVEL= option, 347
SVMACHINE procedure, KERNEL statement, 348
SVMACHINE procedure, OUTPUT statement, 348
COPYVARS= option, 349
OUT= option, 348
ROLE= option, 349
SVMACHINE procedure, PARTITION statement
FRACTION option, 349
ROLEVAR= option, 349
SVMACHINE procedure, PROC SVMACHINE statement, 343
C= option, 343
DATA= option, 343
ITERATIONREPORT, 343
LINEAR option, 348
MAXITER= option, 344
NOPRINT, 344
NOSCALE, 344
NTHREADS= option, 344
POLYNOMIAL option, 348
PRINTTARGET option, 344
SCALE option, 344
TOLERANCE option, 344
USEMISS option, 344
SVMACHINE procedure, SAVESTATE statement, 350, 371
RSTORE= option, 350, 371
SVMACHINE procedure, syntax, 343
SVMACHINE procedure, TARGET statement, 350
ASCENDING option, 350
DESCENDING option, 350
SYNONYM= option
PARSE statement, 370
syntax
BOOLRULE procedure, 75
TEXTMINE procedure, 365
TMSCORE procedure, 414

TARGET statement

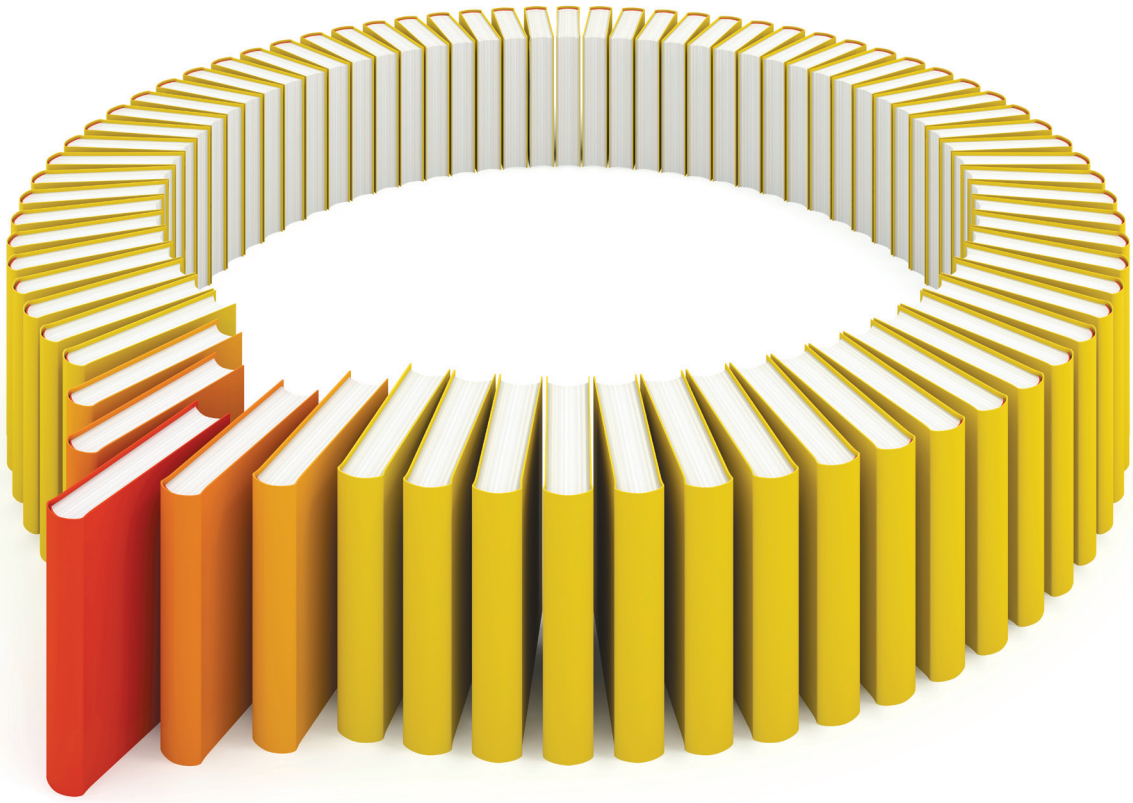
BNET procedure, 56
FACTMAC procedure, 109
FISM procedure, 127
FOREST procedure, 151
GRADBOOST procedure, 187
MBANALYSIS procedure, 227
NNET procedure, 266
SVMACHINE procedure, 350
TEXTMINE procedure, 376
TARGET= option
DOCINFO statement, 79
TARGETEVENT= option
AUTOTUNE statement, 18
TARGETSTANDARDIZE= option
PROC NNET statement, 255
TARGETTYPE= option
DOCINFO statement, 79
TERMID= option
PROC BOOLRULE statement, 78
TERMINFO statement
BOOLRULE procedure, 80
TERMINFO= option
PROC BOOLRULE statement, 78
TERMS= option
PROC TMSCORE statement, 415
TERMWGT= option
PARSE statement, 371
TEXTMINE procedure, 365
PARSE statement, 367
PROC TEXTMINE statement, 365
SELECT statement, 371
SVD statement, 372
syntax, 365
TEXTMINE procedure, CODE statement
COMMENT option, 19
FILE= option, 19
FORMATWIDTH= option, 19
INDENTSIZE= option, 19
LABELID= option, 19
LINESIZE= option, 19
NOTRIM option, 19
OUT= option, 19
PCATALL option, 19
TEXTMINE procedure, DOC_ID statement, 367
TEXTMINE procedure, PARSE statement, 367
CELLWGT= option, 368
ENTITIES= option, 368
MULTITERM= option, 368
NONG option, 368
NONOUNGROUPS option, 368
NOSTEMMING option, 369
NOTAGGING option, 369
OUTCHILD= option, 369
OUTCONFIG= option, 369
OUTPARENT= option, 369
OUTPOS= option, 369
OUTTERMS= option, 370
REDUCEF= option, 370
SHOWDROPPEDTERMS= option, 370
START= option, 370
STOP= option, 370
SYNONYM= option, 370
TERMWGT= option, 371
TEXTMINE procedure, PROC TEXTMINE statement, 365
DATA= option, 366
LANGUAGE= option, 366
NEWVARNAMES, 366
NTHREADS= option, 366
TEXTMINE procedure, SELECT statement, 371
GROUP= option, 372
IGNORE option, 372
KEEP option, 372
LABELS option, 372
TEXTMINE procedure, SVD statement, 372
COL= option, 373
ENTRY= option, 373
EXACTWEIGHT option, 374
IN_TERMS= option, 374
K= option, 374
KEEPVARS, KEEPVARIABLES, 374
MAX_K= option, 374
NOCUTOFFS option, 374
NUMLABELS= option, 374
OUTDOCPRO= option, 374
OUTTOPICS= option, 375
RES= option, 375
RESOLUTION= option, 375
ROTATION= option, 375
ROW= option, 376
SVDS= option, 376
SVDU= option, 376
SVDV= option, 376
TOL= option, 376
TEXTMINE procedure, TARGET statement, 376
TEXTMINE procedure, VAR statement, 376
TEXTMINE procedure, VARIABLES statement, 376
THRESHDIST= option
PROC FASTKNN statement, 119
TMSCORE procedure, 414
PROC TMSCORE statement, 414
syntax, 414
TMSCORE procedure, CODE statement
COMMENT option, 19
FILE= option, 19
FORMATWIDTH= option, 19
INDENTSIZE= option, 19
LABELID= option, 19

LINESIZE= option, 19
NOTRIM option, 19
OUT= option, 19
PCATALL option, 19
TMSCORE procedure, DOC_ID statement, 416
TMSCORE procedure, PROC TMSCORE statement, 414
DATA= option, 414
DOC= option, 414
OUTPARENT= option, 415
SVDDOCPRO= option, 415
SVDU= option, 415
TERMS= option, 415
TMSCORE procedure, TMSCORE statement
CONFIG= option, 415
TMSCORE procedure, VAR statement, 416
TMSCORE procedure, VARIABLES statement, 416
TOL= option
PROC GVARCLUS statement, 209
SVD statement, 376
TOLERANCE option
PROC SVMACHINE statement, 344
TOLERANCE= option
PROC GVARCLUS statement, 209
PROC RPCA statement, 289
RPCA statement, 246
TRACE option
DISPLAY statement (GVARCLUS), 210
DISPLAY statement (MWPCA), 243
DISPLAY statement (RPCA), 291
TRAIN statement
NNET procedure, 267
TUNINGPARAMETERS= option
AUTOTUNE statement, 18, 105, 144, 180, 256, 345

UCC option
PROC SVDD statement, 307
UPLOAD statement
ASTORE procedure, 34
USELOCKING option
OPTIMIZATION statement, 264
USEMISS option
PROC SVMACHINE statement, 344
USEPARAMETERS= option
AUTOTUNE statement, 18

VALIDATION= option
TRAIN statement, 269
VALIDGOAL= option
TRAIN statement, 269
VAR statement
TEXTMINE procedure, 376
TMSCORE procedure, 416
VARIABLES statement
TEXTMINE procedure, 376
TMSCORE procedure, 416
VARS_TO_TRY= option
PROC FOREST statement, 142
PROC GRADBOOST statement, 178
VII= option
PROC FOREST statement, 143
PROC GRADBOOST statement, 179
VOTE= option
PROC FOREST statement, 143

WEIGHT statement
FOREST procedure, 151
GRADBOOST procedure, 187
NNET procedure, 269
SVDD procedure, 310
WINDOWSIZE= option
PROC MWPCA statement, 241
WINSUMMARY= option
OUTPUT statement, 245
WSEED= option
TRAIN statement, 269



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