IBM SPSS Modeler 15 Scripting and Automation Guide

Note: Before using this information and the product it supports, read the general information under Notices on p. 297.

This edition applies to IBM SPSS Modeler 15 and to all subsequent releases and modifications until otherwise indicated in new editions.

Preface

IBM® SPSS® Modeler is the IBM Corp. enterprise-strength data mining workbench. SPSS Modeler helps organizations to improve customer and citizen relationships through an in-depth understanding of data. Organizations use the insight gained from SPSS Modeler to retain profitable customers, identify cross-selling opportunities, attract new customers, detect fraud, reduce risk, and improve government service delivery.

SPSS Modeler's visual interface invites users to apply their specific business expertise, which leads to more powerful predictive models and shortens time-to-solution. SPSS Modeler offers many modeling techniques, such as prediction, classification, segmentation, and association detection algorithms. Once models are created, IBM® SPSS® Modeler Solution Publisher enables their delivery enterprise-wide to decision makers or to a database.

About IBM Business Analytics

IBM Business Analytics software delivers complete, consistent and accurate information that decision-makers trust to improve business performance. A comprehensive portfolio of business intelligence, predictive analytics, financial performance and strategy management, and analytic applications provides clear, immediate and actionable insights into current performance and the ability to predict future outcomes. Combined with rich industry solutions, proven practices and professional services, organizations of every size can drive the highest productivity, confidently automate decisions and deliver better results.

As part of this portfolio, IBM SPSS Predictive Analytics software helps organizations predict future events and proactively act upon that insight to drive better business outcomes. Commercial, government and academic customers worldwide rely on IBM SPSS technology as a competitive advantage in attracting, retaining and growing customers, while reducing fraud and mitigating risk. By incorporating IBM SPSS software into their daily operations, organizations become predictive enterprises – able to direct and automate decisions to meet business goals and achieve measurable competitive advantage. For further information or to reach a representative visit *http://www.ibm.com/spss*.

Technical support

Technical support is available to maintenance customers. Customers may contact Technical Support for assistance in using IBM Corp. products or for installation help for one of the supported hardware environments. To reach Technical Support, see the IBM Corp. web site at *http://www.ibm.com/support*. Be prepared to identify yourself, your organization, and your support agreement when requesting assistance.

Contents

4 Scripting Commands 30

10 Stream Properties 110

11 Project Properties 113

12 Source Node Properties 114

13 Record Operations Node Properties 132

14 Field Operations Node Properties 141

15 Graph Node Properties 167

16 Modeling Node Properties 181

17 Model Nugget Node Properties 234

18 Database Modeling Node Properties 243

19 Output Node Properties 269

20 Export Node Properties 282

Chapter

1

About IBM SPSS Modeler

IBM® SPSS® Modeler is a set of data mining tools that enable you to quickly develop predictive models using business expertise and deploy them into business operations to improve decision making. Designed around the industry-standard CRISP-DM model, SPSS Modeler supports the entire data mining process, from data to better business results.

SPSS Modeler offers a variety of modeling methods taken from machine learning, artificial intelligence, and statistics. The methods available on the Modeling palette allow you to derive new information from your data and to develop predictive models. Each method has certain strengths and is best suited for particular types of problems.

SPSS Modeler can be purchased as a standalone product, or used as a client in combination with SPSS Modeler Server. A number of additional options are also available, as summarized in the following sections. For more information, see *http://www.ibm.com/software/analytics/spss/products/modeler/*.

IBM SPSS Modeler Products

The IBM® SPSS® Modeler family of products and associated software comprises the following.

- IBM SPSS Modeler
- IBM SPSS Modeler Server
- IBM SPSS Modeler Administration Console
- IBM SPSS Modeler Batch
- IBM SPSS Modeler Solution Publisher
- IBM SPSS Modeler Server adapters for IBM SPSS Collaboration and Deployment Services

IBM SPSS Modeler

SPSS Modeler is a functionally complete version of the product that you install and run on your personal computer. You can run SPSS Modeler in local mode as a standalone product, or use it in distributed mode along with IBM® SPSS® Modeler Server for improved performance on large data sets.

With SPSS Modeler, you can build accurate predictive models quickly and intuitively, without programming. Using the unique visual interface, you can easily visualize the data mining process. With the support of the advanced analytics embedded in the product, you can discover previously hidden patterns and trends in your data. You can model outcomes and understand the factors that influence them, enabling you to take advantage of business opportunities and mitigate risks.

SPSS Modeler is available in two editions: SPSS Modeler Professional and SPSS Modeler Premium. For more information, see the topic IBM SPSS Modeler Editions on p. 3.

IBM SPSS Modeler Server

SPSS Modeler uses a client/server architecture to distribute requests for resource-intensive operations to powerful server software, resulting in faster performance on larger data sets.

SPSS Modeler Server is a separately-licensed product that runs continually in distributed analysis mode on a server host in conjunction with one or more IBM® SPSS® Modeler installations. In this way, SPSS Modeler Server provides superior performance on large data sets because memory-intensive operations can be done on the server without downloading data to the client computer. IBM® SPSS® Modeler Server also provides support for SQL optimization and in-database modeling capabilities, delivering further benefits in performance and automation.

IBM SPSS Modeler Administration Console

The Modeler Administration Console is a graphical application for managing many of the SPSS Modeler Server configuration options, which are also configurable by means of an options file. The application provides a console user interface to monitor and configure your SPSS Modeler Server installations, and is available free-of-charge to current SPSS Modeler Server customers. The application can be installed only on Windows computers; however, it can administer a server installed on any supported platform.

IBM SPSS Modeler Batch

While data mining is usually an interactive process, it is also possible to run SPSS Modeler from a command line, without the need for the graphical user interface. For example, you might have long-running or repetitive tasks that you want to perform with no user intervention. SPSS Modeler Batch is a special version of the product that provides support for the complete analytical capabilities of SPSS Modeler without access to the regular user interface. An SPSS Modeler Server license is required to use SPSS Modeler Batch.

IBM SPSS Modeler Solution Publisher

SPSS Modeler Solution Publisher is a tool that enables you to create a packaged version of an SPSS Modeler stream that can be run by an external runtime engine or embedded in an external application. In this way, you can publish and deploy complete SPSS Modeler streams for use in environments that do not have SPSS Modeler installed. SPSS Modeler Solution Publisher is distributed as part of the IBM SPSS Collaboration and Deployment Services - Scoring service, for which a separate license is required. With this license, you receive SPSS Modeler Solution Publisher Runtime, which enables you to execute the published streams.

IBM SPSS Modeler Server Adapters for IBM SPSS Collaboration and Deployment Services

A number of adapters for IBM® SPSS® Collaboration and Deployment Services are available that enable SPSS Modeler and SPSS Modeler Server to interact with an IBM SPSS Collaboration and Deployment Services repository. In this way, an SPSS Modeler stream deployed to the repository can be shared by multiple users, or accessed from the thin-client application IBM SPSS Modeler Advantage. You install the adapter on the system that hosts the repository.

IBM SPSS Modeler Editions

SPSS Modeler is available in the following editions.

SPSS Modeler Professional

SPSS Modeler Professional provides all the tools you need to work with most types of structured data, such as behaviors and interactions tracked in CRM systems, demographics, purchasing behavior and sales data.

SPSS Modeler Premium

SPSS Modeler Premium is a separately-licensed product that extends SPSS Modeler Professional to work with specialized data such as that used for entity analytics or social networking, and with unstructured text data. SPSS Modeler Premium comprises the following components.

IBM® SPSS® Modeler Entity Analytics adds a completely new dimension to IBM® SPSS® Modeler predictive analytics. Whereas predictive analytics attempts to predict future behavior from past data, entity analytics focuses on improving the coherence and consistency of current data by resolving identity conflicts within the records themselves. An identity can be that of an individual, an organization, an object, or any other entity for which ambiguity might exist. Identity resolution can be vital in a number of fields, including customer relationship management, fraud detection, anti-money laundering, and national and international security.

IBM SPSS Modeler Social Network Analysis transforms information about relationships into fields that characterize the social behavior of individuals and groups. Using data describing the relationships underlying social networks, IBM® SPSS® Modeler Social Network Analysis identifies social leaders who influence the behavior of others in the network. In addition, you can determine which people are most affected by other network participants. By combining these results with other measures, you can create comprehensive profiles of individuals on which to base your predictive models. Models that include this social information will perform better than models that do not.

IBM® SPSS® Modeler Text Analytics uses advanced linguistic technologies and Natural Language Processing (NLP) to rapidly process a large variety of unstructured text data, extract and organize the key concepts, and group these concepts into categories. Extracted concepts and categories can be combined with existing structured data, such as demographics, and applied to modeling using the full suite of SPSS Modeler data mining tools to yield better and more focused decisions.

IBM SPSS Modeler Documentation

Documentation in online help format is available from the Help menu of SPSS Modeler. This includes documentation for SPSS Modeler, SPSS Modeler Server, and SPSS Modeler Solution Publisher, as well as the Applications Guide and other supporting materials.

Complete documentation for each product (including installation instructions) is available in PDF format under the *\Documentation* folder on each product DVD. Installation documents can also be downloaded from the web at *http://www-01.ibm.com/support/docview.wss?uid=swg27023172*.

Documentation in both formats is also available from the SPSS Modeler Information Center at *http://publib.boulder.ibm.com/infocenter/spssmodl/v15r0m0/*.

SPSS Modeler Professional Documentation

The SPSS Modeler Professional documentation suite (excluding installation instructions) is as follows.

- **IBM SPSS Modeler User's Guide.** General introduction to using SPSS Modeler, including how to build data streams, handle missing values, build CLEM expressions, work with projects and reports, and package streams for deployment to IBM SPSS Collaboration and Deployment Services, Predictive Applications, or IBM SPSS Modeler Advantage.
- **IBM SPSS Modeler Source, Process, and Output Nodes.** Descriptions of all the nodes used to read, process, and output data in different formats. Effectively this means all nodes other than modeling nodes.
- **IBM SPSS Modeler Modeling Nodes.** Descriptions of all the nodes used to create data mining models. IBM® SPSS® Modeler offers a variety of modeling methods taken from machine learning, artificial intelligence, and statistics.
- **IBM SPSS Modeler Algorithms Guide.** Descriptions of the mathematical foundations of the modeling methods used in SPSS Modeler. This guide is available in PDF format only.
- **IBM SPSS Modeler Applications Guide.** The examples in this guide provide brief, targeted introductions to specific modeling methods and techniques. An online version of this guide is also available from the Help menu. For more information, see the topic Application Examples on p. 5.
- **IBM SPSS Modeler Scripting and Automation.** Information on automating the system through scripting, including the properties that can be used to manipulate nodes and streams.
- **IBM SPSS Modeler Deployment Guide.** Information on running SPSS Modeler streams and scenarios as steps in processing jobs under IBM® SPSS® Collaboration and Deployment Services Deployment Manager.
- **IBM SPSS Modeler CLEF Developer's Guide.** CLEF provides the ability to integrate third-party programs such as data processing routines or modeling algorithms as nodes in SPSS Modeler.
- **IBM SPSS Modeler In-Database Mining Guide.** Information on how to use the power of your database to improve performance and extend the range of analytical capabilities through third-party algorithms.
- **IBM SPSS Modeler Server Administration and Performance Guide.** Information on how to configure and administer IBM® SPSS® Modeler Server.

- **IBM SPSS Modeler Administration Console User Guide.** Information on installing and using the console user interface for monitoring and configuring SPSS Modeler Server. The console is implemented as a plug-in to the Deployment Manager application.
- **IBM SPSS Modeler Solution Publisher Guide.** SPSS Modeler Solution Publisher is an add-on component that enables organizations to publish streams for use outside of the standard SPSS Modeler environment.
- **IBM SPSS Modeler CRISP-DM Guide.** Step-by-step guide to using the CRISP-DM methodology for data mining with SPSS Modeler.
- **IBM SPSS Modeler Batch User's Guide.** Complete guide to using IBM SPSS Modeler in batch mode, including details of batch mode execution and command-line arguments. This guide is available in PDF format only.

SPSS Modeler Premium Documentation

The SPSS Modeler Premium documentation suite (excluding installation instructions) is as follows.

- **IBM SPSS Modeler Entity Analytics User Guide.** Information on using entity analytics with SPSS Modeler, covering repository installation and configuration, entity analytics nodes, and administrative tasks.
- **IBM SPSS Modeler Social Network Analysis User Guide.** A guide to performing social network analysis with SPSS Modeler, including group analysis and diffusion analysis.
- **SPSS Modeler Text Analytics User's Guide.** Information on using text analytics with SPSS Modeler, covering the text mining nodes, interactive workbench, templates, and other resources.
- **IBM SPSS Modeler Text Analytics Administration Console User Guide.** Information on installing and using the console user interface for monitoring and configuring IBM® SPSS® Modeler Server for use with SPSS Modeler Text Analytics . The console is implemented as a plug-in to the Deployment Manager application.

Application Examples

While the data mining tools in SPSS Modeler can help solve a wide variety of business and organizational problems, the application examples provide brief, targeted introductions to specific modeling methods and techniques. The data sets used here are much smaller than the enormous data stores managed by some data miners, but the concepts and methods involved should be scalable to real-world applications.

You can access the examples by clicking Application Examples on the Help menu in SPSS Modeler. The data files and sample streams are installed in the *Demos* folder under the product installation directory. For more information, see the topic Demos Folder on p. 6.

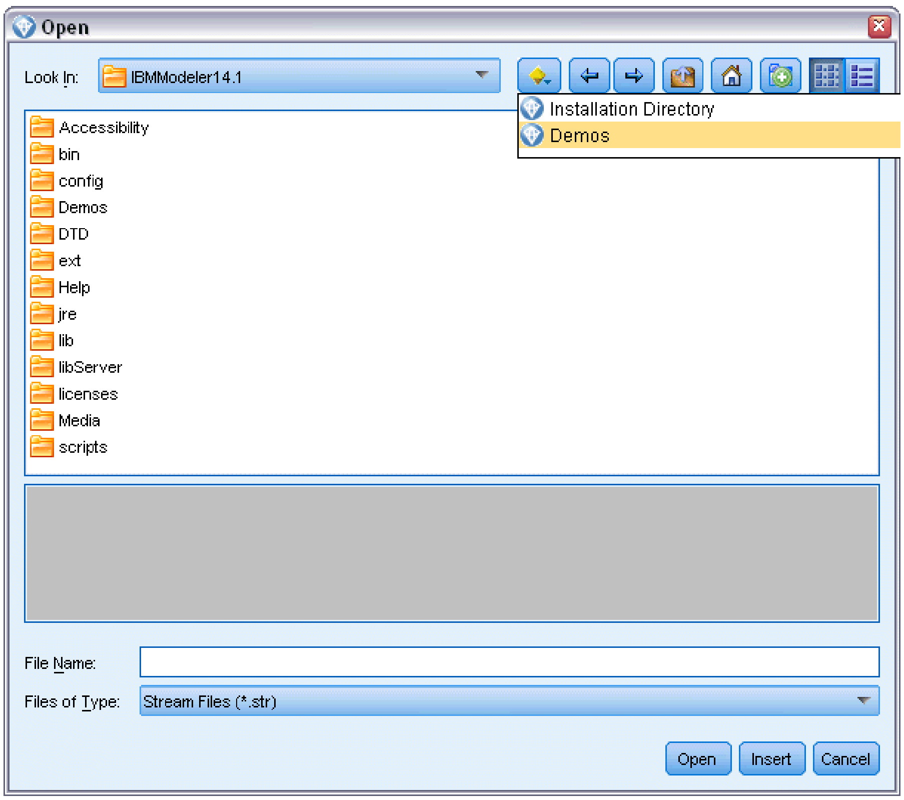
Database modeling examples. See the examples in the *IBM SPSS Modeler In-Database Mining Guide*.

Scripting examples. See the examples in the *IBM SPSS Modeler Scripting and Automation Guide*.

Demos Folder

The data files and sample streams used with the application examples are installed in the *Demos* folder under the product installation directory. This folder can also be accessed from the IBM SPSS Modeler 15 program group on the Windows Start menu, or by clicking *Demos* on the list of recent directories in the File Open dialog box.

Figure 1-1
Selecting the Demos folder from the list of recently-used directories



Part I:
Scripting and the Scripting Language

Chapter

2

Scripting Overview

Scripting in IBM® SPSS® Modeler is a powerful tool for automating processes in the user interface. Scripts can perform the same types of actions that you perform with a mouse or a keyboard, and you can use them to automate tasks that would be highly repetitive or time consuming to perform manually.

You can use scripts to:

- Impose a specific order for node executions in a stream.
- Set properties for a node as well as perform derivations using a subset of CLEM (Control Language for Expression Manipulation).
- Specify an automatic sequence of actions that normally involves user interaction—for example, you can build a model and then test it.
- Set up complex processes that require substantial user interaction—for example, cross-validation procedures that require repeated model generation and testing.
- Set up processes that manipulate streams—for example, you can take a model training stream, run it, and produce the corresponding model-testing stream automatically.

This chapter provides high-level descriptions and examples of stream-level scripts, standalone scripts, and scripts within SuperNodes in the SPSS Modeler interface. More information on scripting language, syntax, and commands is provided in the chapters that follow.

Note: You cannot import and run scripts created in IBM® SPSS® Statistics within SPSS Modeler.

Types of Scripts

IBM® SPSS® Modeler uses three types of scripts:

- **Stream scripts** are stored as a stream property and are therefore saved and loaded with a specific stream. For example, you can write a stream script that automates the process of training and applying a model nugget. You can also specify that whenever a particular stream is executed, the script should be run instead of the stream's canvas content.
- **Standalone scripts** are not associated with any particular stream and are saved in external text files. You might use a standalone script, for example, to manipulate multiple streams together.
- **SuperNode scripts** are stored as a SuperNode stream property. SuperNode scripts are only available in terminal SuperNodes. You might use a SuperNode script to control the execution sequence of the SuperNode contents. For nonterminal (source or process) SuperNodes, you can define properties for the SuperNode or the nodes it contains in your stream script directly.

Stream Scripts

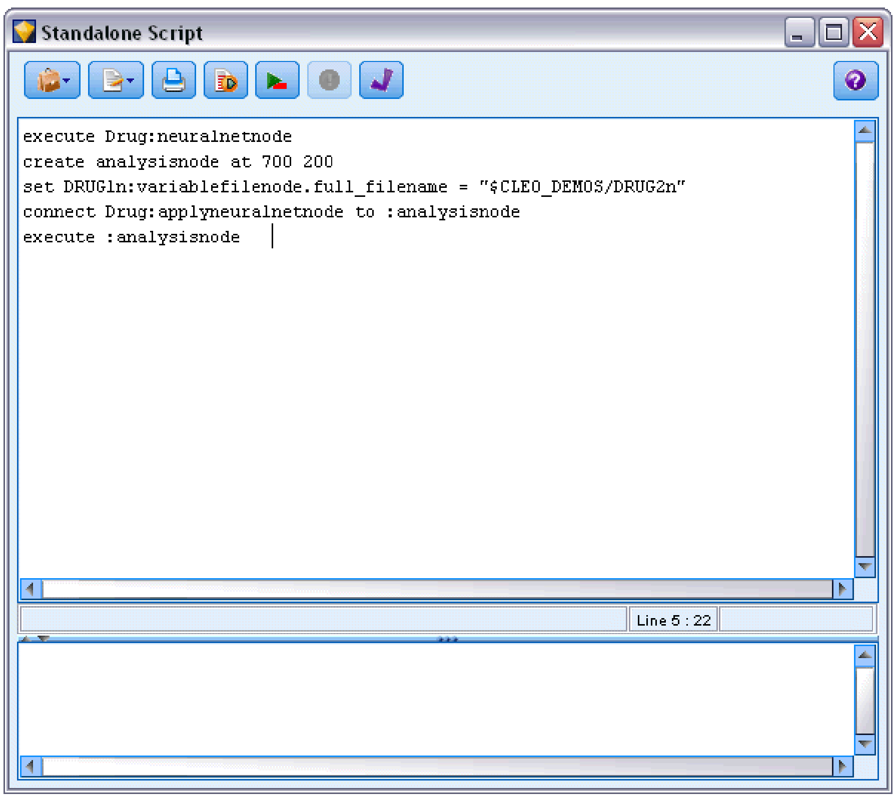
Scripts can be used to customize operations within a particular stream, and they are saved with that stream. Stream scripts can be used to specify a particular execution order for the terminal nodes within a stream. You use the stream script dialog box to edit the script that is saved with the current stream.

To access the stream script tab in the Stream Properties dialog box:

▶ From the Tools menu, choose:
Stream Properties > Script...

▶ Click the Script tab to work with scripts for the current stream.

Figure 2-1
Stream Script dialog box



The toolbar icons at the top of this dialog box let you perform the following operations:

- Import the contents of a preexisting standalone script into the window.
- Save a script as a text file.
- Print a script.
- Append default script.
- Execute the entire current script.

- Execute selected lines from a script.
- Check the syntax of the script and, if any errors are found, display them for review in the lower panel of the dialog box.

Additionally, you can specify whether this script should or should not be run when the stream is executed. You can select Run this script to run the script each time the stream is executed, respecting the execution order of the script. This setting provides automation at the stream level for quicker model building. However, the default setting is to ignore this script during stream execution. Even if you select the option Ignore this script, you can always run the script directly from this dialog box.

Stream Script Example: Training a Neural Net

A stream can be used to train a neural network model when executed. Normally, to test the model, you might run the modeling node to add the model to the stream, make the appropriate connections, and execute an Analysis node.

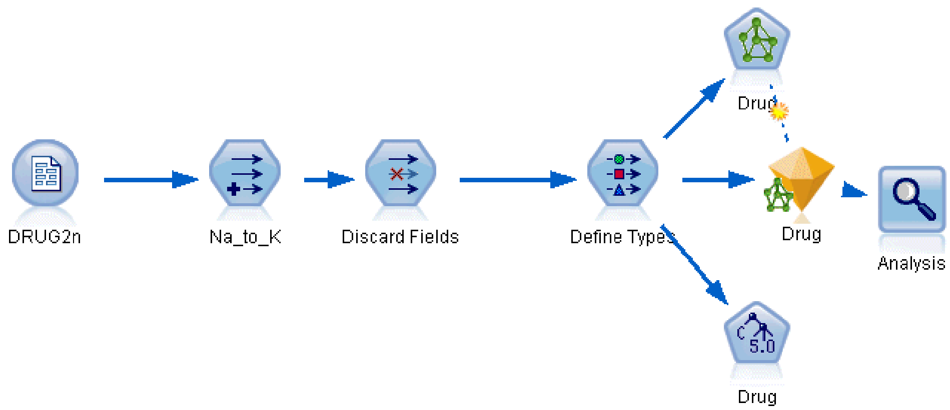
Using an IBM® SPSS® Modeler script, you can automate the process of testing the model nugget after you have created it. For example, the following stream script to test the demo stream *druglearn.str* (available in the */Demos/streams/* folder under your SPSS Modeler installation) could be run from the Stream Properties dialog (Tools > Stream Properties > Script):

```
execute Drug:neuralnetworknode
create analysisnode at 700 200
set DRUG1n:variablefilenode.full_filename = "$CLEO_DEMOS/DRUG2n"
connect :applyneuralnetworknode to :analysisnode
execute :analysisnode
```

The following bullets describe each line in this script example.

- The first line executes the Neural Net node called Drug already found in the demo stream so as to create a model nugget and place it on the stream canvas, connected to the Type node already in the stream.
- In line 2, the script creates an Analysis node and places it at the canvas position 700 x 200.
- In line 3, the original data source used in the stream is switched to a test dataset called DRUG2n.
- In line 4, the Neural Net model nugget is connected to the Analysis node. Note that no names are used to denote the Neural Net model nugget or the Analysis node since no other similar nodes exist in the stream.
- Finally, the Analysis node is executed to produce the Analysis report.

Figure 2-2
Resulting stream



This script was designed to work with an existing stream, since it assumes that a Neural Net node named *Drug* already exists. However, it is also possible to use a script to build and run a stream from scratch, starting with a blank canvas. To learn more about scripting language in general, see Scripting Language Overview on p. 19. To learn more about scripting commands specifically, see Scripting Commands on p. 30.

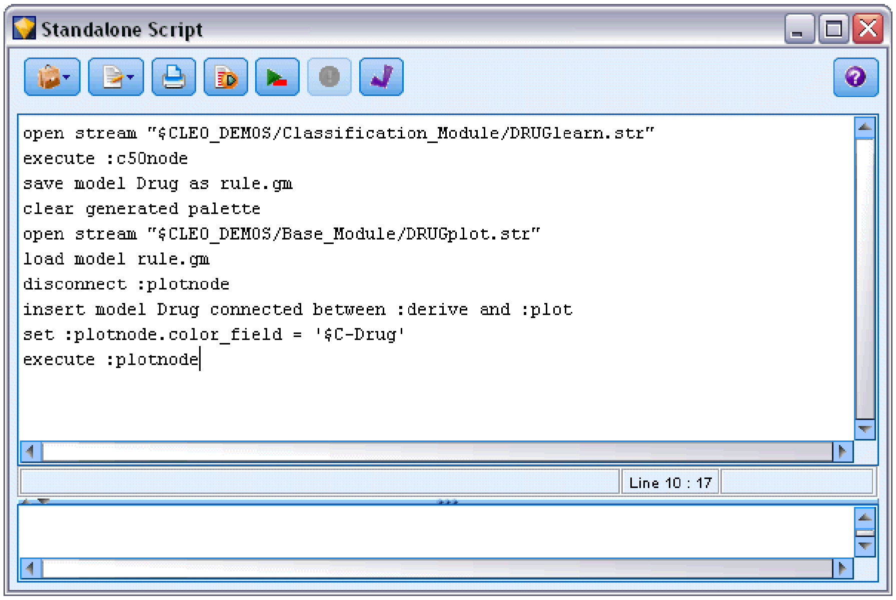
Standalone Scripts

The Standalone Script dialog box is used to create or edit a script that is saved as a text file. It displays the name of the file and provides facilities for loading, saving, importing, and executing scripts.

To access the standalone script dialog box:

▶ From the main menu, choose:
Tools > Standalone Script

Figure 2-3
Standalone Script dialog box



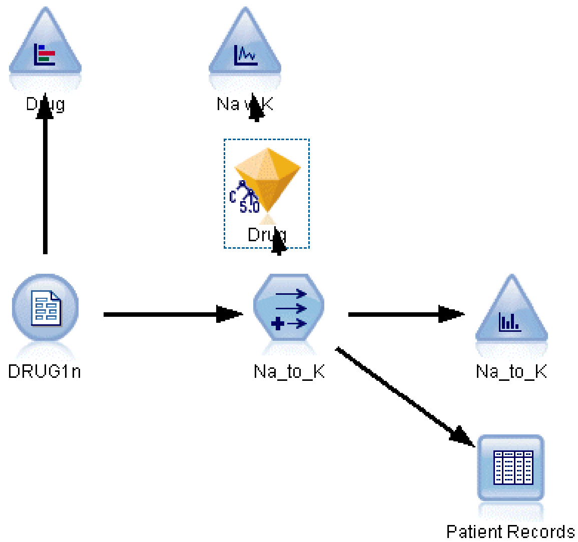
The same toolbar and script syntax-checking options are available for standalone scripts as for stream scripts. For more information, see the topic Stream Scripts on p. 9.

Standalone Script Example: Saving and Loading a Model

Standalone scripts are useful for stream manipulation. Suppose that you have two streams—one that creates a model and another that uses graphs to explore the generated rule set from the first stream with existing data fields. A standalone script for this scenario might look something like this:

```
open stream "$CLEO_DEMOS/streams/druglearn.str"
execute :c50node
save model Drug as rule.gm
clear generated palette
open stream "$CLEO_DEMOS/streams/drugplot.str"
load model rule.gm
disconnect :plotnode
insert model Drug connected between :derive and :plot
set :plotnode.color_field = '$C-Drug'
execute :plotnode
```

Figure 2-4
Resulting stream



Note: To learn more about scripting language in general, see Scripting Language Overview on p. 19. To learn more about scripting commands specifically, see Scripting Commands on p. 30.

Standalone Script Example: Generating a Feature Selection Model

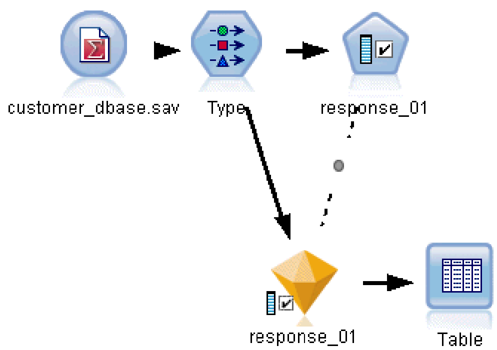
Starting with a blank canvas, this example builds a stream that generates a Feature Selection model, applies the model, and creates a table that lists the 15 most important fields relative to the specified target.

```
create stream 'featureselection'
create statisticsimportnode
position :statisticsimportnode at 50 50
set :statisticsimportnode.full_filename = "$CLEO_DEMOS/customer_dbase.sav"

create typenode
position :typenode at 150 50
set :typenode.direction.'response_01' = Target
connect :statisticsimportnode to :typenode

create featureselectionnode
position :featureselectionnode at 250 50
set :featureselectionnode.screen_missing_values=true
set :featureselectionnode.max_missing_values=80
set :featureselectionnode.criteria = Likelihood
set :featureselectionnode.important_label = "Check Me Out!"
set :featureselectionnode.selection_mode = TopN
set :featureselectionnode.top_n = 15
connect :typenode to :featureselectionnode
execute :featureselectionnode
```

```
create tablenode
position :tablenode at 250 250
connect response_01:applyfeatureselectionnode to :tablenode
execute :tablenode
```

The script creates a source node to read in the data, uses a Type node to set the role (direction) for the *response_01* field to `Target`, and then creates and executes a Feature Selection node. The script also connects the nodes and positions each on the stream canvas to produce a readable layout. The resulting model nugget is then connected to a Table node, which lists the 15 most important fields as determined by the `selection_mode` and `top_n` properties. For more information, see the topic featureselectionnode Properties in Chapter 16 on p. 203.

Figure 2-5
Resulting stream



SuperNode Scripts

You can create and save scripts within any terminal SuperNodes using IBM® SPSS® Modeler's scripting language. These scripts are only available for terminal SuperNodes and are often used when creating template streams or to impose a special execution order for the SuperNode contents. SuperNode scripts also enable you to have more than one script running within a stream.

For example, let's say you needed to specify the order of execution for a complex stream, and your SuperNode contains several nodes including a SetGlobals node, which needs to be executed before deriving a new field used in a Plot node. In this case, you can create a SuperNode script that executes the SetGlobals node first. Values calculated by this node, such as the average or standard deviation, can then be used when the Plot node is executed.

Within a SuperNode script, you can specify node properties in the same manner as other scripts. Alternatively, you can change and define the properties for any SuperNode or its encapsulated nodes directly from a stream script. For more information, see the topic SuperNode Properties in Chapter 22 on p. 295. This method works for source and process SuperNodes as well as terminal SuperNodes.

Note: Since only terminal SuperNodes can execute their own scripts, the Scripts tab of the SuperNode dialog box is available only for terminal SuperNodes.

To open the SuperNode script dialog box from the main canvas:

▶ Select a terminal SuperNode on the stream canvas and, from the SuperNode menu, choose:
SuperNode Script...

To open the SuperNode script dialog box from the zoomed-in SuperNode canvas:

▶ Right-click on the SuperNode canvas, and from the context menu, choose:
SuperNode Script...

SuperNode Script Example

The following SuperNode script declares the order in which the terminal nodes inside the SuperNode should be executed. This order ensures that the Set Globals node is executed first so that the values calculated by this node can then be used when another node is executed.

```
execute 'Set Globals'
execute 'gains'
execute 'profit'
execute 'age v. $CC-pep'
execute 'Table'
```

Executing and Interrupting Scripts

A number of ways of executing scripts are available. For example, on the stream script or standalone script dialog, the "Run this script" button executes the complete script:

Figure 2-6
Run This Script button

The "Run selected lines" button executes a single line, or a block of adjacent lines, that you have selected in the script:

Figure 2-7
Run Selected Lines button

You can execute a script using any of the following methods:

- Click the "Run this script" or "Run selected lines" button within a stream script or standalone script dialog box.
- Run a stream where Run this script is set as the default execution method.
- Use the `-execute` flag on startup in interactive mode. For more information, see the topic Using Command Line Arguments in Chapter 7 on p. 68.

Note: A SuperNode script is executed when the SuperNode is executed as long as you have selected Run this script within the SuperNode script dialog box.

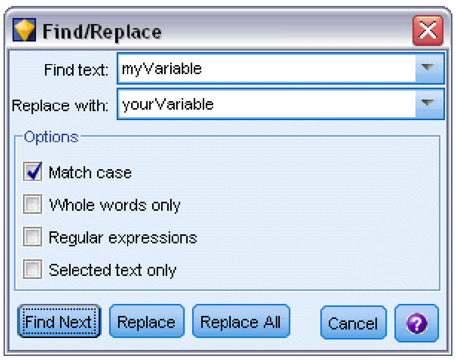
Interrupting Script Execution

Within the stream script dialog box, the red stop button is activated during script execution. Using this button, you can abandon the execution of the script and any current stream.

Find and Replace

The Find/Replace dialog box is available in places where you edit script or expression text, including the script editor, CLEM expression builder, or when defining a template in the Report node. When editing text in any of these areas, press Ctrl-F to access the dialog box, making sure cursor has focus in a text area. If working in a Filler node, for example, you can access the dialog box from any of the text areas on the Settings tab, or from the text field in the Expression Builder.

Figure 2-8
Find/Replace dialog box



- With the cursor in a text area, press Ctrl+F to access the Find/Replace dialog box.
- Enter the text you want to search for, or choose from the drop-down list of recently searched items.
- Enter the replacement text, if any.
- Click Find Next to start the search.
- Click Replace to replace the current selection, or Replace All to update all or selected instances.
- The dialog box closes after each operation. Press F3 from any text area to repeat the last find operation, or press Ctrl+F to access the dialog box again.

Search Options

Match case. Specifies whether the find operation is case-sensitive; for example, whether *myvar* matches *myVar*. Replacement text is always inserted exactly as entered, regardless of this setting.

Whole words only. Specifies whether the find operation matches text embedded within words. If selected, for example, a search on *spider* will not match *spiderman* or *spider-man*.

Regular expressions. Specifies whether regular expression syntax is used (see next section). When selected, the Whole words only option is disabled and its value is ignored.

Selected text only. Controls the scope of the search when using the Replace All option.

Regular Expression Syntax

Regular expressions allow you to search on special characters such as tabs or newline characters, classes or ranges of characters such as *a* through *d*, any digit or non-digit, and boundaries such as the beginning or end of a line. The following types of expressions are supported.

Character Matches

Characters	Matches
x	The character x
\\	The backslash character
\0n	The character with octal value 0n (0 <= n <= 7)
\0nn	The character with octal value 0nn (0 <= n <= 7)
\0mnn	The character with octal value 0mnn (0 <= m <= 3, 0 <= n <= 7)
\xhh	The character with hexadecimal value 0xhh
\uhhhh	The character with hexadecimal value 0xhhhh
\t	The tab character ('\u0009')
\n	The newline (line feed) character ('\u000A')
\r	The carriage-return character ('\u000D')
\f	The form-feed character ('\u000C')
\a	The alert (bell) character ('\u0007')
\e	The escape character ('\u001B')
\cx	The control character corresponding to x

Matching Character Classes

Character classes	Matches
[abc]	a, b, or c (simple class)
[^abc]	Any character except a, b, or c (subtraction)
[a-zA-Z]	a through z or A through Z, inclusive (range)
[a-d[m-p]]	a through d, or m through p (union). Alternatively this could be specified as [a-dm-p]
[a-z&&[def]]	a through z, and d, e, or f (intersection)
[a-z&&[^bc]]	a through z, except for b and c (subtraction). Alternatively this could be specified as [ad-z]
[a-z&&[^m-p]]	a through z, and not m through p (subtraction). Alternatively this could be specified as [a-lq-z]

Predefined Character Classes

Predefined character classes	Matches
.	Any character (may or may not match line terminators)
\d	Any digit: [0-9]
\D	A non-digit: [^0-9]
\s	A white space character: [\t\n\x0B\f\r]
\S	A non-white space character: [^\s]

Predefined character classes	Matches
\w	A word character: [a-zA-Z_0-9]
\W	A non-word character: [^\w]

Boundary Matches

Boundary matchers	Matches
^	The beginning of a line
$	The end of a line
\b	A word boundary
\B	A non-word boundary
\A	The beginning of the input
\Z	The end of the input but for the final terminator, if any
\z	The end of the input

Chapter

3

Scripting Language

Scripting Language Overview

The IBM® SPSS® Modeler scripting language consists of:

- A format for referencing nodes, streams, projects, output, and other SPSS Modeler objects.
- A set of scripting statements or commands that can be used to manipulate these objects.
- A scripting expression language for setting the values of variables, parameters, and other objects.
- Support for comments, continuations, and blocks of literal text.

This section describes the basic syntax for using the scripting language. Information about specific properties and commands is provided in the sections that follow.

Scripting Syntax

To improve clarity during parsing, the following rules should be followed when working with scripts in IBM® SPSS® Modeler:

- Variable names, such as `income` or `referrerID`, must be unquoted.
- Variable names, such as `^mystream`, are preceded with a caret (^) symbol when referencing an existing variable whose value has already been set. The caret is not used when declaring or setting the value of the variable. For more information, see the topic Referencing Nodes on p. 20.
- References to session, stream, and SuperNode parameters, such as `'$P-Maxvalue'`, should be single-quoted.
- If double quotes are used, an expression is treated as a string literal—for example, `"Web graph of BP and Drug"`. This can lead to unexpected results if single and double quotes are not used with care—for example, `"$P-Maxvalue"` will be a string rather than a reference to the value stored in a parameter.
- Filenames, such as `"druglearn.str"`, should be double-quoted.
- Node names, such as `databasenode` or `Na_to_K`, can be unquoted or single-quoted. *Note*: Names must be quoted if they include spaces or special characters. You cannot, however, use a node name in a script if the name starts with a number, such as `'2a_referrerID'`.
- Flag properties should be read or set by using values of `true` and `false` (written in lowercase as shown). Variations including `Off`, `OFF`, `off`, `No`, `NO`, `no`, `n`, `N`, `f`, `F`, `False`, `FALSE`, or `0` are also recognized when setting values but may cause errors when reading property values in some cases. All other values are regarded as `true`. Using lowercase `true` and `false` consistently will avoid any confusion.

- Literal strings or blocks that include line breaks, spaces, or single or double quotes within the block can be wrapped in triple quotes. For more information, see the topic Blocks of Literal Text on p. 28.
- CLEM expressions, such as "Age >= 55", should be double-quoted—for example:

  `set :derivenode.flag_expr = "Age >= 55"`
- If you use quotation marks within a CLEM expression, make sure that each quotation mark is preceded by a backslash (\)—for example:

  `set :node.parameter = "BP = \"HIGH\""`

While not strictly required in all instances, these guidelines are recommended for improved clarity. The script checker available in all scripting dialog boxes flags ambiguous syntax.

Referencing Nodes

There are a number of ways to reference nodes in scripts:

- You can specify nodes by name—for example, DRUG1n. You can qualify the name by type—for example, Drug:neuralnetworknode refers to a Neural Net node named Drug and not to any other kind of node.
- You can specify nodes by type only—for example, :neuralnetworknode refers to all Neural Net nodes. Any valid node type can be used—for example, samplenode, neuralnetworknode, and kmeansnode. The node suffix is optional and can be omitted, but including it is recommended because it makes identifying errors in scripts easier.
- You can reference each node by its unique ID as displayed on the Annotations tab for each node. Use an “@” symbol followed by the ID; for example, @id5E5GJK23L.custom_name = "My Node".

Generated models. The same rules apply to generated model nodes. You can use the name of the node as displayed on the generated models palette in the managers window, or you can reference generated model nodes by type. Note that the names used to reference generated models in the manager are distinct from those used for models that have been added to a stream for purposes of scoring (the latter use an “apply” prefix). For more information, see the topic Model Nugget Names in Chapter 4 on p. 42.

Referencing Nodes Using Variables

You can supply node names and types as the values of local script variables by using the caret (^) syntax. For example, where a node name is required, ^n means the node whose name is stored in the variable n, and Drug:^t means the node named Drug whose type is stored in the variable t.

Node references can be stored in local script variables (declared using a var statement) but not in stream, session, or SuperNode parameters. To guarantee unambiguous references to nodes, assign a unique node ID to a variable as you create the node.

```
var x
set x = create typenode
set ^x.custom_name = "My Node"
```

- The first line creates a variable named x.
- The second line creates a new Type node and stores a reference to the node in x. Note that x stores a reference to the node itself, not the node name.
- The third line sets the value of the custom_name property for the node to "My Node". The caret is used to indicate that x is the name of a variable rather than a node. (Without the caret, the system would look for a node named x. The caret is not needed when declaring or setting the variable because the object of a var command, for example, can only be a variable. But in the third line, x could logically be a node name rather than a variable, so the caret is needed to distinguish the two.)

A common pitfall is to try to store a reference to a node in a variable without first declaring it.

```
set x = create typenode
set ^x.custom_name = "My Node"
```

In this case, the SET command attempts to create x as a stream, session, or SuperNode parameter, rather than as a variable, and returns an error because a reference to a node cannot be stored in a parameter.

Referencing Nodes by ID

You can also store a unique node ID in a variable. For example:

```
var n
set n = "id5E5GJK23L"
set @^n.custom_name = "My Node"
```

Looping through nodes in a stream. You can also use the stream.nodes property to return a list of all nodes in a stream, and then iterate through that list to access individual nodes. For more information, see the topic Stream Report in Chapter 6 on p. 65.

Examples

```
NAME:TYPE
```

NAME is the name of a node, and TYPE is its type. At a minimum, you must include either NAME or TYPE. You can omit one, but you cannot omit both. For example, the following command creates a new Derive node between an existing Variable File node named drug1n and an existing Plot node (new nodes do not use the colon):

```
create derivenode connected between drug1n and :plotnode
```

You can also precede either NAME or TYPE by a caret (^) symbol to indicate the presence of a parameter—for example:

```
Drug:^t
```

This reference means a node named Drug, where t is a parameter that specifies the type of node. For example, if the value of ^t is c50node, the above reference can be translated as:

```
Drug:c50node
```

Similarly, a parameter can be used for the node name. For example, the following can both be used in a context where a node name is required:

```
^n:derivenode
^n
```

Retrieving Objects

The `get` command returns a reference to a stream, node, or output object, making it possible to manipulate these objects using scripts. For example:

```
var mynode
set mynode = get node flag1:derivenode
position ^mynode at 400 400

var mytable = get output :tableoutput
export output ^mytable as c:/mytable.htm format html

set stream = get stream 'Stream1'
set ^stream.execute_method = "Script"
```

Setting the Current Object

The following special variables can be used to reference current objects:

- `node`
- `stream`
- `output`
- `project`

With the exception of `project`, they can also be reset in order to change the current context. Unlike other scripting variables, they don't need to be declared first with the `var` command because they are predefined.

```
set node = create typenode
rename ^node as "mytypenode"

set output = get output :statisticsoutput
export output ^output as c:/myoutput.htm format html
```

Because these special variables match the names of the objects they reference, the distinction between variable and object may be obscured in some cases, resulting in subtle distinctions in usage. For more information, see the topic set Command in Chapter 4 on p. 32.

Comments

Assigning a value of the wrong type to a special variable (such as setting a node object to the variable `stream`) causes a run-time error.

In cases where the special variable can be used, any variable can also be used. For example, saving the current stream can be carried out with:

```
save stream as 'C:/My Streams/Churn.str'
```

It is also valid to say:

```
save my_stream as 'C:/My Streams/Churn.str'
```

where my_stream has previously been assigned a stream value.

Opening Streams and Other Objects

In a stand-alone script, you can open a stream by specifying the filename and location of the file—for example:

```
open stream "c:/demos/druglearn.str"
```

Other types of objects can be opened using the load command—for example:

```
load node c:/mynode.nod
```

```
load model c:/mymodel.gm
```

Open stream versus load stream. The load stream command adds the specified stream to the canvas without clearing the nodes from the current stream. This command was used more extensively in earlier releases and has largely been superseded by the ability to open, manage, and copy nodes between multiple streams.

Working with Multiple Streams

Aside from the commands used to access streams from the file system or from the IBM® SPSS® Collaboration and Deployment Services Repository (open, load, and retrieve), most scripting commands automatically apply to the current stream. In stand-alone scripts, however, you may want to open and manipulate multiple streams from the same script. You can do this by setting a reference to any open stream, or by using the with... endwith command to temporarily reassign the current stream.

For example, to close a stream other than the current one, the get stream command can be used to reference the desired stream:

```
set stream = get stream "druglearn"
close stream
```

This script reassigns the special variable stream to the stream druglearn (essentially making it the current stream) and then closes the stream.

Alternatively, the current stream can be temporarily reassigned using the with stream statement—for example:

```
with stream 'druglearn'
  create typenode
  execute_script
endwith
```

The statements above execute the create action and execute the stream's script with the specified stream set as the current stream. The original current stream is restored once each statement has been executed. Conditional statements and loop constructs can also be included—for example:

```
with stream 'druglearn'
  create tablenode at 500 400
  create selectnode connected between :typenode and :tablenode
  for I from 1 to 5
    set :selectnode.condition = 'Age > ' >< (I * 10)
    execute :selectnode
  endfor
endwith
```

The statements above will set the current stream to STREAM for all expressions within the loop and restore the original value when the loop has completed.

Local Script Variables

Local script variables are declared with the var command and are set for the current script only. Variables are distinct from parameters, which can be set for a session, stream, or SuperNode and can contain only strings or numbers.

```
var my_node
set my_node = create distributionnode
rename ^my_node as "Distribution of Flag"
```

When referring to existing variables, be sure to use the caret (^) symbol preceding the parameter name. For example, given the above script:

- The first line declares the variable.
- The second line sets its value.
- The third line renames the node referenced by the variable (not the variable itself). The caret indicates that ^my_node is the name of a variable rather than the literal name of the node. (Without the caret, the rename command would look for a node named my_node. The caret is not needed in the first or second line because the object of a var command can only be a variable. The caret is used only when referencing a variable that has already been set, in which case its removal would result in an ambiguous reference.)
- When resolving variable references, the local variable list is searched before the list of session, stream, or SuperNode parameters. For example, if a variable x existed as a local variable and as a session parameter, using the syntax '$P-X' in a scripting statement would ensure that the session parameter is used rather than the local variable.

Note: In practical terms, if you set a variable without first declaring it using a var command, a stream, session, or SuperNode parameter is created, depending on the context of the current script. For example, the following code creates a local script variable named z and sets its value to [1 2 3]:

```
var z
set z = [1 2 3]
```

If the var command is omitted (and assuming a variable or node of that name doesn't already exist), then z is created as a parameter rather than a variable.

Stream, Session, and SuperNode Parameters

Parameters can be defined for use in CLEM expressions and in scripting. They are, in effect, user-defined variables that are saved and persisted with the current stream, session, or SuperNode and can be accessed from the user interface as well as through scripting. If you save a stream, for example, any parameters set for that stream are also saved. (This distinguishes them from local script variables, which can be used only in the script in which they are declared.) Parameters are often used in scripting as part of a CLEM expression in which the parameter value is specified in the script.

The scope of a parameter depends on where it is set:

- Stream parameters can be set in a stream script or in the stream properties dialog box, and they are available to all nodes in the stream. They are displayed on the Parameters list in the Expression Builder.
- Session parameters can be set in a stand-alone script or in the session parameters dialog box. They are available to all streams used in the current session (all streams listed on the Streams tab in the managers pane).

Parameters can also be set for SuperNodes, in which case they are visible only to nodes encapsulated within that SuperNode.

Setting Parameters in Scripts

You can set parameters in scripts using the set command and the following syntax:

```
set foodtype = pizza
```

If there are no nodes or variables named foodtype declared in the current script, this command creates a parameter named foodtype, with a default value of pizza.

User interface. Alternatively, parameters can be set or viewed through the user interface by choosing Stream Properties or Set Session Parameters from the Tools menu. These dialog boxes also allow you to specify additional options, such as storage type, that are not available through scripting.

Command line. You can also set parameters from the command line, in which case they are created as session parameters.

Referring to Parameters in Scripts

You can refer to previously created parameters by encapsulating them in single quotes, prefaced with the string $P—for example, '$P-minvalue'. You can also refer simply to the parameter name, such as minvalue. The value for a parameter is always a string or a number. For example, you can refer to the foodtype parameter and set a new value using the following syntax:

```
set foodtype = pasta
```

You can also refer to parameters within the context of a CLEM expression used in a script. The following script is an example. It sets the properties for a Select node to include records in which the value for Age is greater than that specified by the stream parameter named cutoff. The parameter is used in a CLEM expression with the proper syntax for CLEM—'$P-cutoff':

```
set :selectnode {
mode = "Include"
condition = "Age >= '$P-cutoff'"
}
```

The script above uses the default value for the stream parameter named cutoff. You can specify a new parameter value by adding the following syntax above the Select node specifications:

```
set cutoff = 50
```

The resulting script selects all records in which the value of Age is greater than 50.

Controlling Script Execution

Script execution normally processes one statement after another. However, you can override this execution order by using a conditional if statement and several varieties of for loops—for example:

```
if s.maxsize > 10000 then
s.maxsize = 10000
connect s to :derive
endif
```

The for loop has a variety of forms—for example:

```
for PARAMETER in LIST
  STATEMENTS
endfor
```

The script above executes STATEMENTS once for each value in LIST assigned to PARAMETER, using the order of the list. The list has no surrounding brackets, and its contents are constants. A number of other forms are also available. For more information, see the topic General Scripting Commands in Chapter 4 on p. 30.

Operators in Scripts

In addition to the usual CLEM operators, you can manipulate local scripting variables (declared using a var command) using the "+" and "–" operators. The + operator adds an element to the list, and the – operator removes an item. Following is an example:

```
var z       # create a new local variable
set z = [1 2 3]    # set it to the list containing 1, 2, and 3
set z = z + 4    # add an element; z now equals [1 2 3 4]
```

These operators cannot be used with stream, SuperNode, or session parameters (defined in scripts using the set command) or outside of scripts in general CLEM expressions (such as a formula in a Derive node).

CLEM Expressions in Scripts

You can use CLEM expressions, functions, and operators within IBM® SPSS® Modeler scripts; however, your scripting expression cannot contain calls to any @ functions, date/time functions, and bitwise operations. Additionally, the following rules apply to CLEM expressions in scripting:

- Parameters must be specified in single quotes and with the $P- prefix.
- CLEM expressions must be enclosed in quotes. If the CLEM expression itself contains quoted strings or quoted field names, the embedded quotes must be preceded by a backslash (\). For more information, see the topic Scripting Syntax on p. 19.

You can use global values, such as GLOBAL_MEAN(Age), in scripting; however, you cannot use the @GLOBAL function itself within the scripting environment.

Examples of CLEM expressions used in scripting are:

```
set :balancenode.directives = [{1.3 "Age > 60"}]

set :fillernode.condition = "(Age > 60) and (BP = \"High\")"

set :derivenode.formula_expr = "substring(5, 1, Drug)"

set Flag:derivenode.flag_expr = "Drug = X"

set :selectnode.condition = "Age >= '$P-cutoff'"

set :derivenode.formula_expr = "Age - GLOBAL_MEAN(Age)"
```

Inserting Comments and Continuations

The following characters are used in scripting to denote comments and continuations:

Character	Usage	Example
#	The hash sign is a comment. The rest of the line is ignored.	#This is a single-line comment.
\	A line ending with a backslash indicates that the statement continues onto the next line.	See example below.

Character	Usage	Example
/*	The sequence /* indicates the beginning of a comment. Everything is ignored until a */ end comment marker is found.	See example below.
"""	Literal strings or blocks that include line breaks, spaces, or single or double quotes within the block can be wrapped in triple quotes. For more information, see the topic Blocks of Literal Text on p. 28.	

Examples

```
/* This is a
multi line
comment
*/


#following is a multi-line statement
set :fixedfilenode.fields = [{"Age" 1 3}\
 {"Sex" 5 7} {"BP" 9 10} {"Cholesterol" 12 22}\
 {"Na" 24 25} {"K" 27 27} {"Drug" 29 32}]
```

Blocks of Literal Text

Literal text blocks that include spaces, tabs, and line breaks can be included in scripts by setting them off in triple quotes. Any text within the quoted block is preserved as literal text, including spaces, line breaks, and embedded single and double quotes. No line continuation or escape characters are needed.

For example, you can use this technique to embed a set of tree-growing directives in a script, as follows:

```
 set :cartnode.tree_directives = """
   Create Root_Node
   Grow Node Index 0 Children 1 2 SplitOn ("DRUG",
      Group ( "drugA", "drugB", "drugC" )
      Group ( "drugY", "drugX" ))
   End Tree
"""
```

This is also useful for paths and annotations—for example:

```
set :node.annotation = """This node was built to help identify which of the following indicators
   Dairy
   Fish
   Vegetable
   Meat
   Pastries
   Confectionary
is showing unusual sales behaviour"""
```

IBM® SPSS® Modeler ignores a line break following the opening literal marker. For example, the following is equivalent to the preceding example:

```
set :node.annotation = """
This node was built to help identify which of the following indicators
Etc...
"""
```

Chapter

4

Scripting Commands

This section summarizes the commands that can be used in IBM® SPSS® Modeler scripts, organized by object type. For more information on the scripting language, see Chapter 3. For more information about node, stream, project, and SuperNode properties, see Chapter 9 through Chapter 22.

General Scripting Commands

Unless otherwise indicated, the following commands are available in all standalone, stream, and SuperNode scripts.

execute_all

```
execute_all
```

Executes all terminal nodes in the current stream.

```
open stream "c:/demos/druglearn.str"
execute_all
```

execute_script

```
execute_script
```

Standalone scripts only. Executes the stream script associated with the current stream. (Restricted to standalone scripts since it would otherwise result in the stream script calling itself.)

```
open stream "c:/demos/mysample.str"
execute_script
```

exit

```
exit CODE
```

Exits the current script. The exit code can be used to evaluate the script or condition of a stream or node—for example:

```
create tablenode
create variablefilenode
connect :variablefilenode to :tablenode

set :variablefilenode.full_filename = "$CLEO_DEMOS/DRUG1n"
execute 'Table'

set param  = value :tablenode.output at 1 1
```

```
if ^param = 23 then
  create derivenode
  else exit 2
endif
```

for...endfor

The for...endfor command loops through a set of statements based on a condition. The command can take a number of forms, all of which follow the same general structure.

```
for PARAMETER in LIST
  STATEMENTS
endfor
```

for PARAMETER in LIST. Executes STATEMENTS once for each value in LIST assigned to PARAMETER, using the order of the list. For example, the Filter.include property could be set to true for multiple fields as follows:

```
for f in Age Sex
  set Filter.include.^f=true
endfor
```

for PARAMETER from N to M. Executes STATEMENTS once for each integer between N and M, inclusive—for example:

```
for I from 1 to 5
  set :selectnode.condition = 'Age > ' >< (I * 10)
  execute :selectnode
endfor
```

for PARAMETER in_fields_to NODE. Executes STATEMENTS once for each field on the upstream side of NODE. For example, the following sets the include property to true for all fields including those previously set to false:

```
for f in_fields_to Filter
  set Filter.include.^f = "true"
endfor
```

Note: In cases where a node can have multiple input fields with the same name — such as a Merge or Append — this method returns the list of downstream fields rather than upstream, in order to avoid any conflicts that might otherwise result.

for PARAMETER in_fields_at NODE. Execute STATEMENTS once for each field coming out of (or downstream from) the specified NODE. Thus if the node is a Filter, then only fields that are passed through are included, and the node should not be a terminal node as no fields would be returned. For example, in contrast to the above, the following script would have no effect because the loop would only execute for those fields already set to true:

```
for f in_fields_at Filter
  set Filter.include.^f = "true"
endfor
```

for PARAMETER in_models. Executes STATEMENTS once for each model nugget in the Models palette. For example, the following script inserts each model from the palette into the current stream. (The xpos variable is used to avoid stacking the nodes on top of one another on the stream canvas.)

```
var xpos
set xpos = 100
for m in_models
 set xpos = xpos + 100
 insert model ^m at ^xpos 100
endfor
```

for PARAMETER in_streams. *Standalone scripts only.* Executes STATEMENTS once for each loaded stream (as listed in the Streams palette). If PARAMETER is the special variable stream, the current stream is set for STATEMENTS in the loop. The original value of stream is restored when the loop terminates.

if...then...else...

```
if EXPR then
   STATEMENTS 1
  else
   STATEMENTS 2
endif
```

Executes STATEMENTS 1 if the specified expression is true and STATEMENTS 2 if the expression is false. The else clause is optional.

```
if :samplenode.use_max_size = true then
   set x = "yes"
  else
   set x = "no"
endif
```

set Command

```
set VARIABLE = EXPRESSION
set PARAMETER = EXPRESSION
set PROPERTY = EXPRESSION
```

Sets the value of a local script variable, special variable, parameter, or property.

Setting Variables

To set the value of a local script variable, first declare the variable using the var command—for example:

```
var xpos
var ypos
set xpos = 100
set ypos = 100
```

The value of the variable can be a CLEM expression valid in scripting, a script command that returns a value (such as load, create, or get), or a literal value.

```
set xpos = ^xpos + 50

var x
set x = create typenode


var s
set s = get stream 'Druglearn'
```

Setting Special Variables to Reference Objects

The special variables node, stream, output, and project are used to reference the "current" object in each context. With the exception of project, they can also be reset in order to change the current context. Unlike other scripting variables, they don't need to be declared first with the var command since they are predefined.

```
set node = create typenode
rename ^node as "mytypenode"

set output = get output :statisticsoutput
export output ^output as c:/myoutput.htm format html
```

While useful, these variables exhibit some subtle distinctions in usage, as demonstrated by the following sample:

```
set stream = get stream 'Stream7'
set ^stream.execute_method = "Script"
save stream as c:/sample7.str
close stream
```

- The first line resets the current stream, or more literally sets the value of the special variable stream. (In other words, stream is a variable rather than part of the command.)
- The second line uses this variable to set a property for the current stream (see below for more on properties). The caret is used to indicate that ^stream is the name of a variable rather than the name of an object such as a node. (Without the caret, the set command would look for a node named *stream*.)
- The last two lines save and close the current stream. As before, stream is a variable, but in this case no caret is used because the save and close commands as used in this example can only apply to a stream. (The caret is generally only used in cases where its removal would result in an ambiguous reference.)

Referencing the current project. The special variable project can be used to reference the current project (see example of setting project properties below). The value of project cannot be reset because only one project can be open (and thus current) at any one time.

Setting Parameters

Stream, session, and SuperNode parameters can be set in the same manner as variables but without using the var command.

```
set p = 1
set minvalue = 21
```

Note: In practical terms, if the object of a set command does not match the name of a declared variable, a special variable, or an existing object such as a node, then a parameter is created. For more information, see the topic Stream, Session, and SuperNode Parameters in Chapter 3 on p. 25.

Setting Node, Stream, and Project Properties

Properties for nodes, streams, and projects can also be set—for example:

```
set :variablefilenode.full_filename = "$CLEO_DEMOS/DRUG1n"

set ^stream.execute_method = "Script"

load project "C:/myproject.cpj"
set ^project.structure = Phase
```

For a complete list of the properties available for nodes, streams, and projects, see *Properties Reference* on p. 106.

Setting multiple properties. You can assign multiple expressions to properties for nodes or other objects in a single operation. This method is used when multiple changes need to be made to a node before the data model is determined. The format used to set multiple properties is:

```
set NODE {
  NODEPROPERTY1 = EXPRESSION1
  NODEPROPERTY2 = EXPRESSION2
}
```

For example:

```
set :samplenode {
 max_size = 200
 mode = "Include"
 sample_type = "First"
}

set ^project {
 summary = "Initial modeling work on the latest data"
 ordering = NameAddedType
}
```

Setting flag values (true and false). When reading or writing flag-type properties, the values true and false should be in lower case—for example:

```
set :variablefilenode.read_field_names = true
```

Note: Variations, including Off, OFF, off, No, NO, no, n, N, f, F, false, False, FALSE, or 0, are also recognized when setting values but may cause errors when reading property values in some cases. All other values are regarded as true. Using lowercase true and false consistently will avoid any confusion.

Example: Setting Node Properties

There are many node-specific properties (sometimes called slot parameters) used to set options found in the user-interface dialog boxes for each node. For example, to create a stream and specify options for each node, you could use a script similar to the one shown here. For more information about node, stream, project, and SuperNode properties, see Chapter 9 through Chapter 22.

```
create varfilenode at 100 100
set :varfilenode {
full_filename = "demos/drug1n"
read_field_names = true
}
create tablenode at 400 100
create samplenode connected between :varfilenode and :tablenode
set :samplenode {
max_size = 200
mode = "Include"
sample_type = "First"
}
create plotnode at 300 300
create derivenode connected between drug1n and :plotnode
set :derivenode {
new_name = "Ratio of Na to K"
formula_expr = "'Na' / 'K'"
}
set :plotnode {
x_field = 'Ratio of Na to K'
y_field = 'Age'
color_field = 'BP'
}
```

var Command

```
var VARNAME
```

Declares a local script variable.

```
var my_node
set my_node = create distributionnode
rename ^my_node as "Distribution of Flag"
```

Variables are distinct from parameters, which can be set for a session, stream, or SuperNode and can contain only strings or numbers. In practical terms, if you set a variable without first declaring it using a VAR command, a stream, session, or SuperNode parameter is created, depending on the context of the current script. For more information, see the topic Local Script Variables in Chapter 3 on p. 24.

Node Objects

The following scripting commands are available for node objects.

create NODE

```
create NODE
create NODE at X Y
create NODE between NODE1 and NODE2
create NODE connected between NODE1 and NODE2
```

Creates a node of the specified type—for example:

```
create statisticsimportnode
```

Optionally, position and connection options can also be specified:

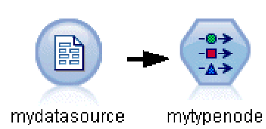
```
create featureselectionnode at 400 100

create typenode between :statisticsimportnode and :featureselectionnode

create selectnode connected between :typenode and :featureselectionnode
```

You can also create a node using variables to avoid ambiguity. For instance, in the example below, a Type node is created and the reference variable *x* is set to contain a reference to that Type node. You can then use the variable *x* to return the object referenced by *x* (in this case, the Type node) and perform additional operations, such as renaming, positioning, or connecting the new node.

```
var x
set x = create typenode
rename ^x as "mytypenode"
position ^x at 200 200
var y
set y = create varfilenode
rename ^y as "mydatasource"
position ^y at 100 200
connect ^y to ^x
```

The example above creates two nodes, renames each, positions them, and finally connects them on the stream canvas.

Figure 4-1
Nodes created using variables



Alternatively, the special (predefined) variable `node` can be used in a similar manner to the `x` and `y` variables in the above example. In this case, the variable need not be declared using the `var` command (since it is predefined), and the resulting script may be a bit easier to read.

```
set node = create typenode
rename ^node as "mytypenode"
position ^node at 200 200
set node = create varfilenode
rename ^node as "mydatasource"
```

```
position ^node at 100 200
connect mydatasource to mytypenode
```

Note: Special variables, such as `node`, can be reused to reference multiple nodes. Simply use the `set` command to reset the object referenced by the variable. For more information, see the topic Setting the Current Object in Chapter 3 on p. 22.

Duplicating nodes. You can also use the `duplicate` command to duplicate an existing node. For more information, see the topic duplicate NODE on p. 38.

connect NODE

```
connect NODE1 to NODE2
connect NODE1 between NODE2 and NODE3
```

Connects `NODE1` to other nodes as specified.

```
connect :statisticsimportnode to :typenode
```

```
connect :selectnode between :typenode and :featureselectionnode
```

delete NODE

```
delete NODE
```

Deletes the specified node from the current stream.

```
delete :statisticsimportnode
```

```
delete DRUG1N:variablefilenode
```

disable NODE

```
disable NODE
```

Disables the specified node from the current stream, with the result that the node is ignored during execution of the stream. This saves you from having to remove or bypass the node and means you can leave it connected to the remaining nodes. You can still edit the node settings; however, any changes will not take effect until you enable the node again.

```
disable :statisticsimportnode
```

```
disable DRUG1N:variablefilenode
```

disconnect NODE

```
disconnect NODE
disconnect NODE1 from NODE2
disconnect NODE1 between NODE2 and NODE3
```

Disconnects the specified node from all other nodes (the default) or from specific nodes as indicated.

```
disconnect :typenode
```

```
disconnect :typenode from :selectnode
```

duplicate NODE

```
duplicate NODE as NEWNAME
```

Creates a new node as a duplicate of the specified node. Optionally, the position can also be specified in absolute or relative terms.

```
duplicate :derivenode as flag1 at 100 400
```

```
duplicate flag1 as flag2 connected between flag1 and flag3
```

enable NODE

```
enable NODE
```

Enables a previously disabled node in the current stream, with the result that the node is included during execution of the stream. If you have edited the node settings whilst it was disabled, the changes will now take effect.

```
enable :statisticsimportnode
```

```
enable DRUG1N:variablefilenode
```

execute NODE

```
execute NODE
```

Executes the specified node—for example:

```
execute :neuralnetworknode
```

If the node is not a terminal node, execution is equivalent to the Run From Here pop-up menu option.

To execute all terminal nodes in the current stream:

```
execute_all
```

Standalone scripts only. To execute the stream script associated with the current stream:

```
execute_script
```

Note: Scripts associated with different streams can be executed by setting the stream as the current stream or by using the with command. For more information, see the topic Working with Multiple Streams in Chapter 3 on p. 23.

export NODE as FILE

```
export node NODE in DIRECTORY format FORMAT
export node NODE as FILE format FORMAT
```

PMML export. To export a generated model in PMML format:

```
export Drug as c:/mymodel.txt format pmml
```

SQL export. To export a generated model in SQL format:

```
export Drug in c:/mymodels format sql
```

```
export Drug as c:/mymodel.txt format sql
```

Node details. To export node details in HTML or text format:

```
export Drug as c:\mymodel.htm format html
```

```
export Drug as c:\mymodel.txt format text
```

Node summary. To export the node summary in HTML or text format:

```
export Drug summary in c:/mymodels format html
```

```
export Drug summary as c:/mymodel.txt format text
```

```
export 'assocapriori' as 'C:/temp/assoc_apriori' format html
```

flush NODE

```
flush NODE
```

Flushes the cache on the specified node or on all nodes in the stream. If the cache is not enabled or is not full for a given node, this operation does nothing.

```
flush :mergenode
```

To flush all nodes in the current stream:

```
flush_all
```

get node NODE

```
get node NODE
```

Gets a reference to an existing node. This can be a useful way to ensure non-ambiguous references to nodes.

```
var mynode
set mynode = get node flag1:derivenode
position ^mynode at 400 400
```

load node FILENAME

```
load node FILENAME
```

Loads a saved node into the current stream.

```
load node c:/mynode.nod
```

position NODE

```
position NODE at X Y
position NODE between NODE1 and NODE2
position NODE connected between NODE1 and NODE2
```

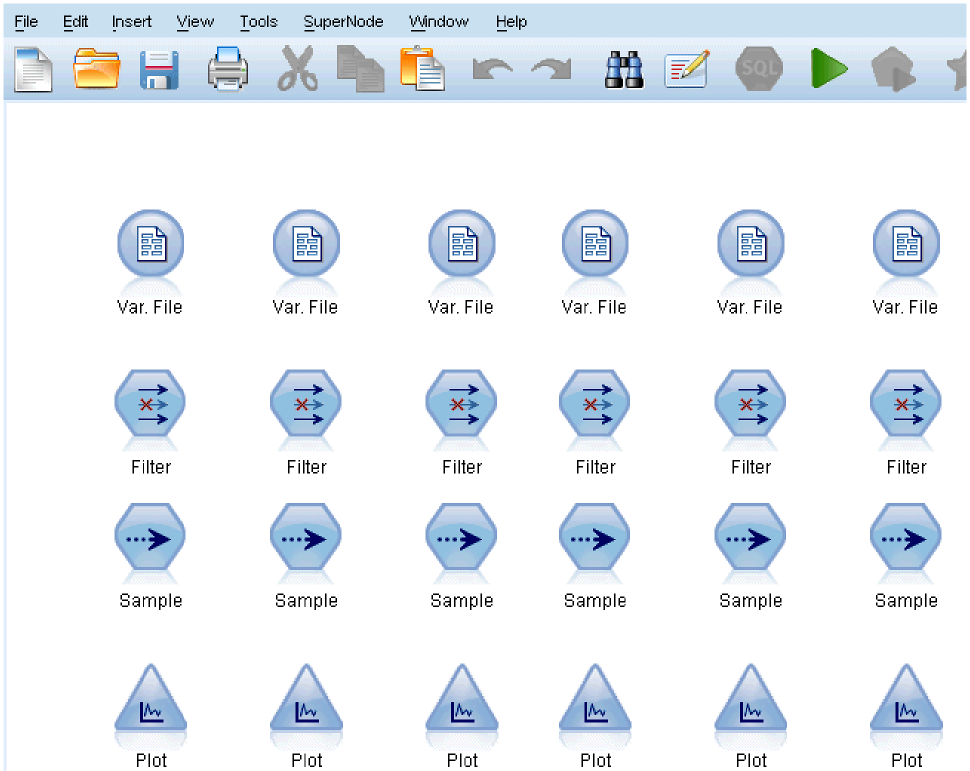
Positions a node in the stream canvas in absolute or relative terms. Optionally, connection options can also be specified:

```
position DRUG1n:variablefilenode at 100 100

position Drug:net between DRUG2n and analysis

position :typenode connected between :variablefilenode and :tablenode
```

Positioning Coordinates

Positioning nodes on the stream canvas uses an invisible *x-y* grid. You can use the image below as a reference for the *x-y* grid coordinates.

Figure 4-2
Nodes created at the position specified with x-y coordinates



rename NODE as NEWNAME

```
rename NODE as NEWNAME
```

Renames the specified node.

```
rename :derivenode as 'Flag1'
```

```
rename :varfilenode as 'testdata'
```

retrieve node REPOSITORY_PATH

```
retrieve node REPOSITORY_PATH {label LABEL | version VERSION}
```

Retrieves the specified node from the IBM® SPSS® Collaboration and Deployment Services Repository. For more information, see the topic Accessing Objects in the IBM SPSS Collaboration and Deployment Services Repository in Chapter 5 on p. 58.

```
retrieve node "/samples/drugtypenode"
```

save node NODE as FILENAME

```
save node NODE as FILENAME
```

Saves the specified node.

```
save node :statisticsimportnode as c:/mynode.nod
```

store node NODE as REPOSITORY_PATH

```
store node NODE as REPOSITORY_PATH {label LABEL}
```

Stores a node in the IBM® SPSS® Collaboration and Deployment Services Repository. For more information, see the topic Accessing Objects in the IBM SPSS Collaboration and Deployment Services Repository in Chapter 5 on p. 58.

```
store node DRUG1n as "/samples/drug1ntypenode"
```

```
store node :typenode as "/samples/drugtypenode"
```

Model Objects

The following scripting commands are available for model objects.

Model Nugget Names

Model nuggets (also known as generated models) can be referenced by type, just like node and output objects. The following tables list the model object reference names.

Note these names are used specifically to reference model nuggets in the Models palette (in the upper right corner of the IBM® SPSS® Modeler window). To reference model nodes that have been added to a stream for purposes of scoring, a different set of names prefixed with apply... are used. For more information, see the topic Model Nugget Node Properties in Chapter 17 on p. 234.

For example, the following script adds a model nugget to the current stream, connects it to a Type node, and creates and executes a Table node. Note the different name used to insert the model from the palette as distinct from the name used to reference the "apply" model node once added to the stream (:featureselection versus :applyfeatureselectionnode).

```
insert model :featureselection at 150 250
connect Type to :applyfeatureselectionnode
create tablenode at 250 250
connect :applyfeatureselectionnode to :tablenode
execute :tablenode
```

Note: This is an example only. Under normal circumstances, referencing models by both name *and* type is recommended to avoid confusion (for example, response_01:featureselection).

Model Nugget Names (Modeling Palette)

Model name	Model
anomalydetection	Anomaly
apriori	Apriori
autoclassifier	Auto Classifier
autocluster	Auto Cluster

Model name	Model
autonumeric	Auto Numeric
bayesnet	Bayesian network
c50	C5.0
carma	Carma
cart	C&R Tree
chaid	CHAID
coxreg	Cox regression
decisionlist	Decision List
discriminant	Discriminant
factor	PCA/Factor
featureselection	Feature Selection
genlin	Generalized linear regression
glmm	GLMM
kmeans	K-Means
knn	*k*-nearest neighbor
kohonen	Kohonen
linear	Linear
logreg	Logistic regression
neuralnetwork	Neural Net
quest	QUEST
regression	Linear regression
sequence	Sequence
slrm	Self-learning response model
statisticsmodel	IBM® SPSS® Statistics model
svm	Support vector machine
timeseries	Time Series
twostep	TwoStep

Model Nugget Names (Database Modeling Palette)

Model name	Model
db2imcluster	IBM ISW Clustering
db2imlog	IBM ISW Logistic Regression
db2imnb	IBM ISW Naive Bayes
db2imreg	IBM ISW Regression
db2imtree	IBM ISW Decision Tree
msassoc	MS Association Rules
msbayes	MS Naive Bayes
mscluster	MS Clustering
mslogistic	MS Logistic Regression
msneuralnetwork	MS Neural Network

Model name	Model
msregression	MS Linear Regression
mssequencecluster	MS Sequence Clustering
mstimeseries	MS Time Series
mstree	MS Decision Tree
netezzabayes	Netezza Bayes Net
netezzadectree	Netezza Decision Tree
netezzadivcluster	Netezza Divisive Clustering
netezzaglm	Netezza Generalized Linear
netezzakmeans	Netezza K-Means
netezzaknn	Netezza KNN
netezzalineregression	Netezza Linear Regression
netezzanaivebayes	Netezza Naive Bayes
netezzapca	Netezza PCA
netezzaregtree	Netezza Regression Tree
netezzatimeseries	Netezza Time Series
oraabn	Oracle Adaptive Bayes
oraai	Oracle AI
oradecisiontree	Oracle Decision Tree
oraglm	Oracle GLM
orakmeans	Oracle *k*-Means
oranb	Oracle Naive Bayes
oranmf	Oracle NMF
oraocluster	Oracle O-Cluster
orasvm	Oracle SVM

Avoiding Duplicate Model Names

When using scripts to manipulate generated models, be aware that allowing duplicate model names can result in ambiguous references. To avoid this, it is a good idea to require unique names for generated models when scripting.

To set options for duplicate model names:

▶ From the menus choose:
Tools > User Options

▶ Click the Notifications tab.

▶ Select Replace previous model to restrict duplicate naming for generated models.

The behavior of script execution can vary between SPSS Modeler and IBM SPSS Collaboration and Deployment Services when there are ambiguous model references. The SPSS Modeler client includes the option “Replace previous model”, which automatically replaces models that have the same name (for example, where a script iterates through a loop to produce a different model each time). However, this option is not available when the same script is run in IBM SPSS Collaboration and Deployment Services. You can avoid this situation either by renaming the model generated in each iteration to avoid ambiguous references to models, or by clearing the current model (for example, adding a `clear generated palette` statement) before the end of the loop.

delete model MODEL

```
delete model MODEL
```

Deletes a specified model (or clears all models) from the model nuggets palette.

```
delete model Drug
```

```
delete model Drug:c50
```

To delete the last model inserted by the current script:

```
delete last model
```

For this last statement to function, the `insert model` statement must have been executed at least once within the current script execution.

To clear all model nuggets from the Models palette:

```
clear generated palette
```

export model MODEL as FILE

```
export model MODEL in DIRECTORY format FORMAT
export model MODEL as FILE format FORMAT
```

PMML export. To export the generated model in PMML format:

```
export model Drug in c:/mymodels format pmml
```

```
export model Drug as c:/mymodel.xml format pmml
```

SQL export. To export a generated model in SQL format:

```
export Drug in c:/mymodels format sql
```

```
export Drug as c:/mymodel.txt format sql
```

Note: SQL export is only available for certain model types.

Model details. To export model details (as displayed on the Model tab when browsing the model nugget) in HTML or text format:

```
export model Drug as c:\mymodel.htm format html
```

```
export model Drug as c:\mymodel.txt format text
```

Note: These formats are unavailable for models that do not have a Model tab.

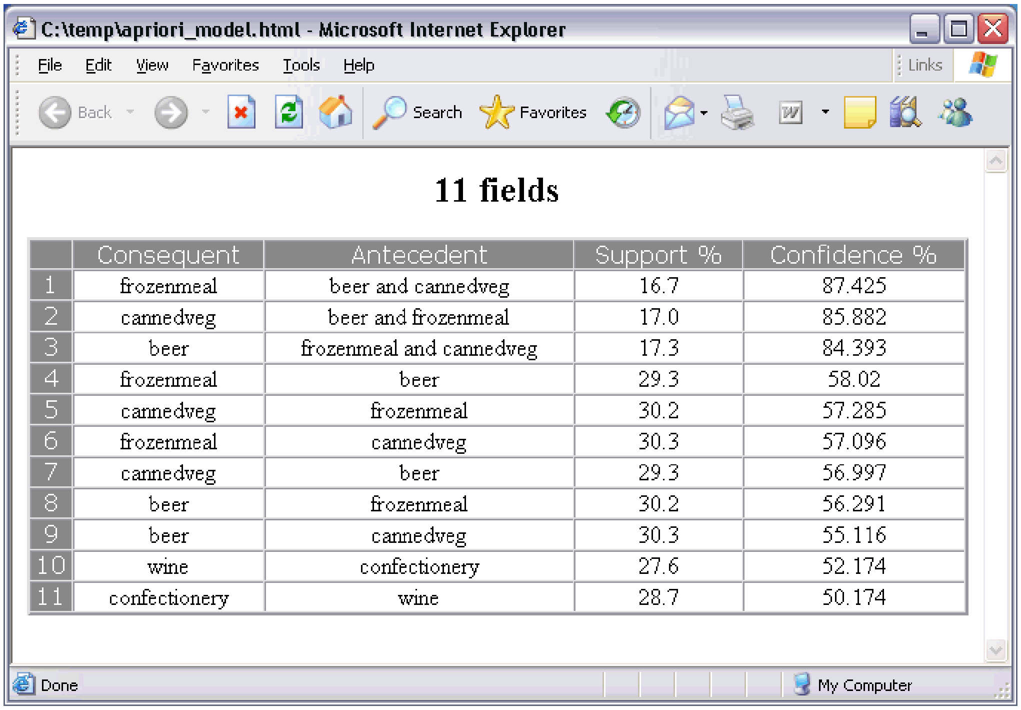
Model summary. To export the model summary (Summary tab when browsing the model nugget) in HTML or text format:

```
export model Drug summary in c:/mymodels format html
```

```
export model Drug summary as c:/mymodel.txt format text
```

```
export model 'assocapriori' as 'C:/temp/assoc_apriori' format html
```

Figure 4-3
Association model tab exported as HTML



	Consequent	Antecedent	Support %	Confidence %
1	frozenmeal	beer and cannedveg	16.7	87.425
2	cannedveg	beer and frozenmeal	17.0	85.882
3	beer	frozenmeal and cannedveg	17.3	84.393
4	frozenmeal	beer	29.3	58.02
5	cannedveg	frozenmeal	30.2	57.285
6	frozenmeal	cannedveg	30.3	57.096
7	cannedveg	beer	29.3	56.997
8	beer	frozenmeal	30.2	56.291
9	beer	cannedveg	30.3	55.116
10	wine	confectionery	27.6	52.174
11	confectionery	wine	28.7	50.174

import model MODEL

```
import model MODEL
```

PMML import. To import a generated model in PMML format:

```
import model 'C:\mymodel.xml'
```

Doing so creates a model nugget in the Models palette.

insert model MODEL

```
insert model MODEL
insert model MODEL at X Y
insert model MODEL between NODE1 and NODE2
insert model MODEL connected between NODE1 and NODE2
```

Adds the model to the current stream. Optionally, position and connection options can also be specified.

```
insert model Kohonen between :typenode and :analysisnode
```

```
insert model Drug:neuralnetwork connected between 'Define Types' and 'Analysis'
```

load model FILENAME

```
load model FILENAME
```

Loads a saved model into the Models palette.

```
load model c:/mymodel.gm
```

retrieve model REPOSITORY_PATH

```
retrieve model REPOSITORY_PATH {label LABEL | version VERSION}
```

Retrieves a saved model from the IBM® SPSS® Collaboration and Deployment Services Repository. For more information, see the topic Accessing Objects in the IBM SPSS Collaboration and Deployment Services Repository in Chapter 5 on p. 58.

```
retrieve model "/my folder/Kohonen.gm"
```

save model MODEL as FILENAME

```
save model MODEL as FILENAME
```

Saves the specified model as a generated model file.

```
save model Drug as c:/mymodel.gm
```

store model MODEL as REPOSITORY_PATH

```
store model MODEL as REPOSITORY_PATH {label LABEL}
```

Stores the specified model in the IBM® SPSS® Collaboration and Deployment Services Repository. For more information, see the topic Accessing Objects in the IBM SPSS Collaboration and Deployment Services Repository in Chapter 5 on p. 58.

```
store model Kohonen as "/my folder/Kohonen.gm"
```

The extension (**.gm*) is optional but must be used consistently when storing and retrieving the model. For example, if stored simply as “Kohenen,” the model would then need to be retrieved by the same name. (To put it another way, the extension, if used, is simply part of the model name.)

Stream Objects

The following scripting commands are available for stream objects.

create stream DEFAULT_FILENAME

```
create stream DEFAULT_FILENAME
```

Standalone scripts only. Creates a new stream in memory with the specified name. The stream is not automatically saved.

```
create stream 'Druglearn'
```

close STREAM

```
close STREAM
```

Standalone scripts only. Closes the specified stream.

To close the current stream, type the command using all lowercase characters, as follows:

```
close stream
```

Standalone Scripts

If working with multiple streams, be aware that `stream` (in lower case as shown) is actually a special variable used to reference the current stream. To close a different stream, the value of this variable can be reset:

```
set stream = get stream 'Stream5'
close stream
```

Alternatively, any declared variable that references a stream can be specified—for example:

```
var s
set s = get stream 'Stream2'
save s as c:/stream2.str
close s
```

Finally, the current stream can be temporarily reassigned using the `with stream` command:

```
with stream 'Stream1'
 close stream
endwith
```

clear stream

```
clear stream
```

Removes all nodes from the current stream.

get stream STREAM

```
get stream STREAM
```

Standalone scripts only. Used to get a reference to the specified stream, which can be assigned to a local variable (or the special variable `stream`). The specified stream must already be open.

```
var s
set s = get stream 'Druglearn'
close s
```

load stream FILENAME

```
load stream FILENAME
```

Standalone scripts only. Adds the specified stream to the canvas without clearing the nodes from the current stream.

```
load stream "c:/demos/druglearn.str"
```

Open stream versus load stream. The `load stream` command adds the specified stream to the canvas without clearing the nodes from the current stream. This command was used more extensively in earlier releases of IBM® SPSS® Modeler and has largely been superseded in newer releases by the ability to open, manage, and copy nodes between multiple streams.

open stream FILENAME

```
open stream FILENAME
```

Standalone scripts only. Opens the specified stream.

```
open stream "c:/demos/druglearn.str"
```

retrieve stream REPOSITORY_PATH

```
retrieve stream REPOSITORY_PATH {label LABEL | version VERSION}
retrieve stream URI [(#m.marker | #l.label)]
```

Retrieves the specified stream from the IBM® SPSS® Collaboration and Deployment Services Repository. For more information, see the topic Accessing Objects in the IBM SPSS Collaboration and Deployment Services Repository in Chapter 5 on p. 58.

```
retrieve stream "/myfolder/druglearn.str"

retrieve stream "spsscr:///models/drug%20model.gm#m.0:2005-10-12%2014:15:41.281"
```

save STREAM as FILENAME

```
save STREAM
save STREAM as FILENAME
```

To save changes to the current stream (assuming it has been saved previously), type the command using all lowercase characters, as follows:

```
save stream
```

To save a stream for the first time under a new filename:

```
create stream nifty
create featureselectionnode
save stream as c:/nifty.str
```

Standalone Scripts

If working with multiple streams in a standalone script, be aware that stream (when in lower case as above) is actually a special variable used to reference the current stream. To save a different stream, the value of this variable can be reset:

```
set stream = get stream 'Stream5'
save stream
```

Alternatively, any declared variable that references a stream can be specified—for example:

```
var s
set s = get stream 'Stream2'
save s as c:/stream2.str
close s
```

Finally, the current stream can be temporarily reassigned using the with stream command:

```
with stream 'Stream1'
 save stream
endwith
```

For more information, see the topic Working with Multiple Streams in Chapter 3 on p. 23.

store stream as REPOSITORY_PATH

```
store stream as REPOSITORY_PATH {label LABEL}
store stream as URI [#l.label]

store stream as "/folder_1/folder_2/mystream.str"
```

Stores the current stream in the IBM® SPSS® Collaboration and Deployment Services Repository. For more information, see the topic Accessing Objects in the IBM SPSS Collaboration and Deployment Services Repository in Chapter 5 on p. 58.

```
store stream as "/folder_1/folder_2/druglearn.str"
store stream as "spsscr:///folder_1/folder_2/mystream.str"
```

Standalone Scripts

If working with multiple streams in a standalone script, be aware that stream (when in lower case as above) is actually a special variable used to reference the current stream. To store a different stream, the value of this variable can be reset:

```
set stream = get stream 'Stream5'
store stream as "/folder_1/mystream.str"
```

Alternatively, any declared variable that references a stream can be specified, or the current stream can be temporarily reassigned using the with stream command:

```
with stream 'Stream6'
 store stream as "/folder_1/mystream.str"
endwith
```

with stream STREAM

```
with stream STREAM
  STATEMENTS
endwith
```

Standalone scripts only. Executes STATEMENTS with the specified STREAM set as the current stream. The original current stream is restored once the statements have been executed.

```
with stream 'druglearn'
  create typenode
  execute_script
endwith
```

Project Objects

The following scripting commands are available for project objects.

The extension (**.cpj*) is optional but must be used consistently when storing and retrieving a given project.

execute_project

```
execute_project
```

Generates the current project's default report.

load project FILENAME

```
load project FILENAME
```

Opens the specified project.

```
load project "C:/clemdata/DrugData.cpj"
set ^project.summary="Initial modeling work on the latest data."
set ^project.ordering=NameAddedType
execute_project
```

retrieve project REPOSITORY_PATH

```
retrieve project REPOSITORY_PATH {label LABEL | version VERSION}
```

Retrieves a project from the IBM® SPSS® Collaboration and Deployment Services Repository. For more information, see the topic Accessing Objects in the IBM SPSS Collaboration and Deployment Services Repository in Chapter 5 on p. 58.

```
retrieve project "/CRISPDM/DrugExample.cpj"
```

save project as FILENAME

```
save project
save project as FILENAME
```

Saves the current project.

store project as REPOSITORY_PATH

```
store project as REPOSITORY_PATH {label LABEL}
```

Stores the current project in the IBM® SPSS® Collaboration and Deployment Services Repository. For more information, see the topic Accessing Objects in the IBM SPSS Collaboration and Deployment Services Repository in Chapter 5 on p. 58.

```
store project as "/CRISPDM/DrugExample.cpj"
```

State Objects

A saved state can be loaded using the `load state` command.

load state FILENAME

```
load state FILENAME
```

Loads the specified state.

```
load state "c:/data/myproject.cst"
```

Result Objects

Results can be accessed using the `value` command.

value RESULT

```
value RESULT at ROW COLUMN
```

Terminal nodes include a read-only parameter called `output` that allows access to the most recently generated object. For nodes that produce tabular output in rows and columns, this makes it possible to access the value for a specified cell—for example:

```
execute :tablenode
set last_row = :tablenode.output.row_count
```

```
set last_column = :tablenode.output.column_count
set last_value = value :tablenode.output at ^last_row ^last_column
var myresults
set myresults = open create 'C:/myresults.txt'
write myresults 'The value in the last cell is ' >< ^last_value
```

Row and column are offset from 1. If the output object does not exist, an error is returned.

Result Object Properties

The following properties are common to result objects (such as Table and Matrix results) that include data in rows and columns:

Property	Description
row_count	Returns the number of rows in the data.
column_count	Returns the number of columns in the data.

File Objects

The following scripting commands are available for file objects.

close FILE

```
close FILE
```

The statement above closes the specified file.

open FILE

```
open create FILENAME
open append FILENAME
```

The statements above open the specified file.

- **create.** Creates the file if it doesn't exist or overwrites if it does.
- **append.** Appends to an existing file. Generates an error if the file doesn't exist.

This returns the file handle for the opened file.

```
var file
set file = open create 'C:/script.out'
for I from 1 to 3
  write file 'Stream ' >< I
endfor
close file
```

write FILE

```
write FILE TEXT_EXPRESSION
writeln FILE TEXT_EXPRESSION
```

The expressions above write the text expression to the file. The first statement writes the text as is, while the second also writes a new line after the expression has been written. It generates an error if FILE is not an open file object.

```
var file
set file = open create 'C:/hello.txt'
writeln file 'Hello'
writeln file 'World'
write file 'Would you like to play a game?'
close file
```

Output Objects

The following scripting commands are available for output objects.

Output Type Names

The following table lists all output object types and the nodes that create them. For a complete list of the export formats available for each type of output object, see the properties description for the node that creates the output type, available in Chapter 15, *Graph Node Properties*, and Chapter 19, *Output Node Properties*.

Output object type	**Node**
analysisoutput	Analysis
collectionoutput	Collection
dataauditoutput	Data Audit
distributionoutput	Distribution
evaluationoutput	Evaluation
histogramoutput	Histogram
matrixoutput	Matrix
meansoutput	Means
multiplotoutput	Multiplot
plotoutput	Plot
qualityoutput	Quality
reportdocumentoutput	This object type is not from a node; it’s the output created by a project report
reportoutput	Report
statisticsprocedureoutput	Statistics Output
statisticsoutput	Statistics
tableoutput	Table
timeplotoutput	Time Plot
weboutput	Web

delete output OUTPUT

```
delete output OUTPUT
```

Deletes the specified output from the manager palette. For example:

```
delete output :statisticsoutput
```

To delete all output items from the manager palette:

```
clear outputs
```

export output OUTPUT

```
export output OUTPUT as FILE format FORMAT
```

Exports output in the specified format. Note the available formats depend on the output type but should mirror those available on the Export menu when browsing the specified output.

```
export output :statisticsoutput as "C:/output/statistics.html" format html
export output :matrixoutput as "C:/output/matrix.csv" format delimited
export output :tableoutput as "C:/output/table.tab" format transposed formatted
```

get output OUTPUT

```
get output OUTPUT
```

Gets a reference to the specified output. For example, a loop could be used to get a series of output objects and export each in turn.

```
execute_all
for item in statisticsoutput matrixoutput tableoutput
 var theoutput
 set theoutput = get output :^item
 set filename = 'c:/'><^item ><'.htm'
 export output ^theoutput as ^filename format html
endfor
```

load output FILENAME

```
load output FILENAME
```

Loads the specified output.

```
load output 'c:/matrix.cou'
```

retrieve output REPOSITORY_PATH

```
retrieve output REPOSITORY_PATH {label LABEL | version VERSION}
```

Retrieves the specified output from the IBM® SPSS® Collaboration and Deployment Services Repository. For more information, see the topic Accessing Objects in the IBM SPSS Collaboration and Deployment Services Repository in Chapter 5 on p. 58.

```
retrieve output "/results/mytable"
```

save output OUTPUT as FILENAME

```
save output as FILENAME
```

Saves the specified output.

```
save output :matrixoutput as 'c:/matrix.cou'
```

store output OUTPUT as REPOSITORY_PATH

```
store output OUTPUT as REPOSITORY_PATH {label LABEL}
```

Stores the specified output in the IBM® SPSS® Collaboration and Deployment Services Repository. For more information, see the topic Accessing Objects in the IBM SPSS Collaboration and Deployment Services Repository in Chapter 5 on p. 58.

```
store output "Data Audit of [6 fields]" as "/my folder/My Audit"
```

```
store output :tableoutput as "/results/mytable"
```

Chapter

5

Scripting Tips

This section provides an overview of tips and techniques for using scripts, including modifying stream execution, using an encoded password in a script, and accessing objects in the IBM® SPSS® Collaboration and Deployment Services Repository.

Modifying Stream Execution

When a stream is run, its terminal nodes are executed in an order optimized for the default situation. In some cases, you may prefer a different execution order. To modify the execution order of a stream, complete the following steps from the Script tab of the stream properties dialog box:

▶ Begin with an empty script.

▶ Click the Append default script button on the toolbar to add the default stream script.

▶ Change the order of statements in the default stream script to the order in which you want statements to be executed.

Looping through Nodes

You can use a `for` loop in combination with the `^stream.nodes` property to loop through all of the nodes in a stream. For example, the following script loops through all nodes and changes field names in any Filter nodes to upper case.

This script can be used in any stream that has a Filter node, even if no fields are actually filtered. Simply add a Filter node that passes all fields in order to change field names to upper case across the board.

```
var my_node
var loop_me
var var_name

for my_node in ^stream.nodes
  if ^my_node.node_type = filternode then
    for loop_me in_fields_to ^my_node:filternode
      set var_name = lowertoupper(^my_node:filternode.new_name.^loop_me)
      set ^my_node:filternode.new_name.^loop_me = ^var_name
    endfor
  else
  endif
endfor
```

The script loops through all nodes in the current stream, as returned by the `^stream.nodes` property, and checks whether each node is a Filter. If so, the script loops through each field in the node and uses the `lowertoupper()` function to change the name to upper case.

Tip: To change field names to lower case, use the uppertolower() function instead.

Accessing Objects in the IBM SPSS Collaboration and Deployment Services Repository

Note: A separate license is required to access an IBM® SPSS® Collaboration and Deployment Services repository. For more information, see *http://www.ibm.com/software/analytics/spss/products/deployment/cds/*

If you have licensed the IBM® SPSS® Collaboration and Deployment Services Repository, you can store, retrieve, lock and unlock objects from the repository using script commands. The repository allows you to manage the life cycle of data mining models and related predictive objects in the context of enterprise applications, tools, and solutions.

Connecting to the IBM SPSS Collaboration and Deployment Services Repository

In order to access the repository, you must first set up a valid connection to it, either through the Tools menu of the IBM® SPSS® Modeler user interface or through the command line. (For more information, see the topic IBM SPSS Collaboration and Deployment Services Repository Connection Arguments in Chapter 7 on p. 72.)

Storing and Retrieving Objects

Within a script, the retrieve and store commands allow you to access various objects, including streams, models, output, nodes, and projects. The syntax is as follows:

```
store object as REPOSITORY_PATH {label LABEL}
store object as URI [#l.label]

retrieve object REPOSITORY_PATH {label LABEL | version VERSION}
retrieve object URI [(#m.marker | #l.label)]
```

The REPOSITORY_PATH gives the location of the object in the repository. The path must be enclosed in quotation marks and use forward slashes as delimiters. It is not case sensitive.

```
store stream as "/folder_1/folder_2/mystream.str"
store model Drug as "/myfolder/drugmodel"
store model Drug as "/myfolder/drugmodel.gm" label "final"
store node DRUG1n as "/samples/drug1ntypenode"
store project as "/CRISPDM/DrugExample.cpj"
store output "Data Audit of [6 fields]" as "/my folder/My Audit"
```

Optionally, an extension such as *.str* or *.gm* can be included in the object name, but this is not required as long as the name is consistent. For example, if a model is stored without an extension, it must be retrieved by the same name:

```
store model "/myfolder/drugmodel"
retrieve model "/myfolder/drugmodel"
```

versus:

```
store model "/myfolder/drugmodel.gm"
retrieve model "/myfolder/drugmodel.gm" version "0:2005-10-12 14:15:41.281"
```

Note that when you are retrieving objects, the most recent version of the object is always returned unless you specify a version or label. When retrieving a node object, the node is automatically inserted into the current stream. When retrieving a stream object, you must use a standalone script. You cannot retrieve a stream object from within a stream script.

Locking and Unlocking Objects

From a script, you can lock an object to prevent other users from updating any of its existing versions or creating new versions. You can also unlock an object that you have locked.

The syntax to lock and unlock an object is:

```
lock REPOSITORY_PATH
lock URI

unlock REPOSITORY_PATH
unlock URI
```

As with storing and retrieving objects, the `REPOSITORY_PATH` gives the location of the object in the repository. The path must be enclosed in quotation marks and use forward slashes as delimiters. It is not case sensitive.

```
lock "/myfolder/Stream1.str”

unlock "/myfolder/Stream1.str"
```

Alternatively, you can use a Uniform Resource Identifier (URI) rather than a repository path to give the location of the object. The URI must include the prefix `spsscr:` and must be fully enclosed in quotation marks. Only forward slashes are allowed as path delimiters, and spaces must be encoded. That is, use `%20` instead of a space in the path. The URI is not case sensitive. Here are some examples:

```
lock "spsscr:///myfolder/Stream1.str"

unlock "spsscr:///myfolder/Stream1.str"
```

Note that object locking applies to all versions of an object - you cannot lock or unlock individual versions.

Generating an Encoded Password

In certain cases, you may need to include a password in a script; for example, you may want to access a password-protected data source. Encoded passwords can be used in:

- Node properties for Database Source and Output nodes

- Command line arguments for logging into the server
- Database connection properties stored in a *.par* file (the parameter file generated from the Publish tab of an export node)

Through the user interface, a tool is available to generate encoded passwords based on the Blowfish algorithm (see *http://www.schneier.com/blowfish.html* for more information). Once encoded, you can copy and store the password to script files and command line arguments. The node property `epassword` used for `databasenode` and `databaseexportnode` stores the encoded password.

▶ To generate an encoded password, from the Tools menu choose:

Encode Password...

Figure 5-1
Encode Password Tool

▶ Specify a password in the Password text box.

▶ Click Encode to generate a random encoding of your password.

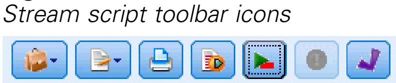
▶ Click the Copy button to copy the encoded password to the Clipboard.

▶ Paste the password to the desired script or parameter.

Script Checking

You can quickly check the syntax of all types of scripts by clicking the red check button on the toolbar of the Standalone Script dialog box.

Figure 5-2
Stream script toolbar icons



Script checking alerts you to any errors in your code and makes recommendations for improvement. To view the line with errors, click on the feedback in the lower half of the dialog box. This highlights the error in red.

Scripting from the Command Line

Scripting enables you to run operations typically performed in the user interface. Simply specify and run a standalone stream on the command line when launching IBM® SPSS® Modeler. For example:

```
client -script scores.txt -execute
```

The `-script` flag loads the specified script, while the `-execute` flag executes all commands in the script file.

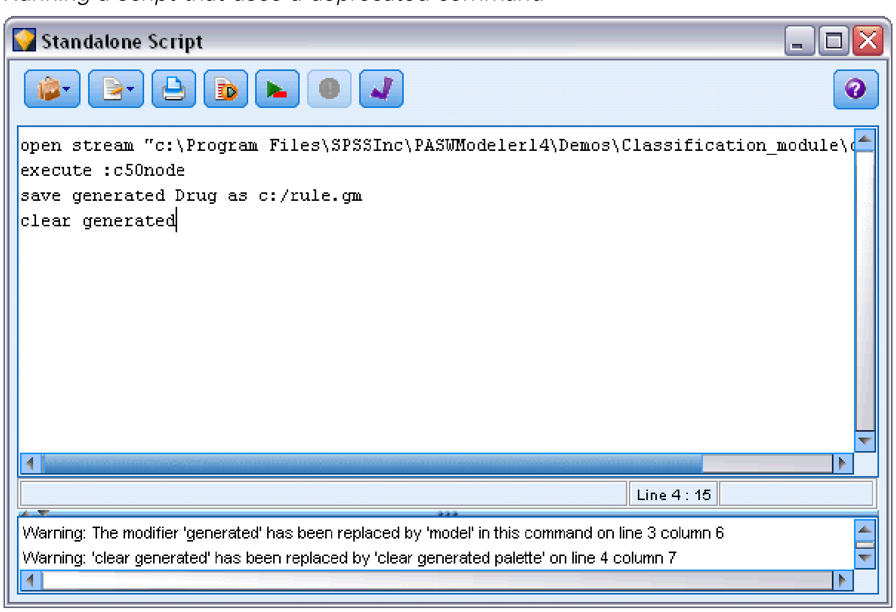
Compatibility with Previous Releases

Scripts created in previous releases of IBM® SPSS® Modeler should generally work unchanged in the current release. However, model nuggets may now be inserted in the stream automatically (this is the default setting), and may either replace or supplement an existing nugget of that type in the stream. Whether this actually happens depends on the settings of the Add model to stream and Replace previous model options (Tools > Options > User Options > Notifications). You may, for example, need to modify a script from a previous release in which nugget replacement is handled by deleting the existing nugget and inserting the new one.

Scripts created in the current release may not work in earlier releases.

If a script created in an older release uses a command that has since been replaced (or deprecated), the old form will still be supported, but a warning message will be displayed. For example, the old `generated` keyword has been replaced by `model`, and `clear generated` has been replaced by `clear generated palette`. Scripts that use the old forms will still run, but a warning will be displayed.

Figure 5-3
Running a script that uses a deprecated command



Chapter

6

Scripting Examples

This section provides a number of examples that demonstrate how scripts can be used in IBM® SPSS® Modeler.

Type Node Report

This script creates an HTML report listing information about the fields in the current stream. The script can be used with any stream that has an instantiated Type node, and it could easily be extended to cover additional properties or nodes.

- Standard HTML tags are used to format the results for display in a standard browser.
- An IBM® SPSS® Modeler Type node is used to access properties for each field. The script could easily be extended to list additional properties exposed through the Type node, such as missing values or the field role. For more information, see the topic typenode Properties in Chapter 14 on p. 163.
- SPSS Modeler scripting commands are used to write the output to a file and to loop through fields in order to access properties of each. For more information, see the topic Scripting Commands in Chapter 4 on p. 30.

Figure 6-1
Type node report sample script

```
# This script creates an HTML file and adds data from the Type node.
var myreport
set myreport = open create "C:/typenodereport.html"

# set up the HTML page
writeln myreport "<html>"
writeln myreport "<header>Type node information from SPSS Modeler</header>"
writeln myreport "<body><br/><br/>"

#create the table and write out the headers
writeln myreport "<table border=\"1\">"
writeln myreport "<tr bgcolor=\"C0C0C0\">"
writeln myreport "<td>Field</td><td>Type</td><td>Values</td>"
writeln myreport "</tr>"

# loop through fields and add a row for each
var current_field
for current_field in_fields_at Type
  writeln myreport "<tr>"
   write myreport "<td>" >< ^current_field >< "</td>"
   write myreport "<td>" >< Type:typenode.type.^current_field >< "</td>"

   # add values for numeric fields
   if Type:typenode.type.^current_field = Range then
```

```
    writeln myreport "<td>" >< Type:typenode.values.^current_field >< "</td>"
  endif

  # add values for flag fields
  if Type:typenode.type.^current_field = Flag then
    writeln myreport "<td>" >< Type:typenode.values.^current_field >< "</td>"
  endif

  # add values for nominal fields
  if Type:typenode.type.^current_field = Set then
   writeln myreport "<td>"
   var current_value
   for current_value in Type:typenode.values.^current_field
     writeln myreport  ^current_value >< "<BR/>"
   endfor
   writeln myreport "</td>"
  endif

 writeln myreport "</tr>"
endfor
writeln myreport "</table>"
writeln myreport "</body>"
writeln myreport "</html>"
close myreport
```

Creating the Output File

The script begins by creating a new HTML file and adds the tags needed to create a table with a heading row listing the column titles *Field*, *Type*, and *Values*. (Each <td> </td> tag pair creates a cell within a table row.) These columns will be populated for each field based on properties from the Type node.

```
# This script creates an HTML file and adds data from the Type node.
var myreport
set myreport = open create "C:/typenodereport.html"

# set up the HTML page
writeln myreport "<html>"
writeln myreport "<header>Type node information from SPSS Modeler</header>"
writeln myreport "<body><br/><br/>"

#create the table and write out the headers
writeln myreport "<table border=\"1\">"
writeln myreport "<tr bgcolor=\"C0C0C0\">"
writeln myreport "<td>Field</td><td>Type</td><td>Values</td>"
writeln myreport "</tr>"
```

Looping through Fields

Next, the script loops through all fields in the Type node and adds a row for each field, listing the field name and type.

```
# loop through fields and add a row for each
var current_field
for current_field in_fields_at Type
  writeln myreport "<tr>"
    write myreport "<td>" >< ^current_field >< "</td>"
    write myreport "<td>" >< Type:typenode.type.^current_field >< "</td>"
```

Values for Continuous and Flag Fields

For continuous (numeric range) fields, the typenode.values property returns the low and high values in the format [0.500517, 0.899774], which is displayed in the table. For flag fields, the true/false values are displayed in a similar format.

```
# add values for numeric fields
    if Type:typenode.type.^current_field = Range then
      writeln myreport "<td>" >< Type:typenode.values.^current_field >< "</td>"
    endif

    # add values for flag fields
    if Type:typenode.type.^current_field = Flag then
      writeln myreport "<td>" >< Type:typenode.values.^current_field >< "</td>"
    endif
```

Values for Nominal Fields

For nominal fields, the typenode.values property returns the complete list of defined values. The script loops through this list for each field to insert each value in turn, with a line break (
 tag) between each.

```
# add values for nominal fields
    if Type:typenode.type.^current_field = Set then
      writeln myreport "<td>"
      var current_value
      for current_value in Type:typenode.values.^current_field
        writeln myreport  ^current_value >< "<BR/>"
      endfor
      writeln myreport "</td>"
    endif
```

Closing the File

Finally, the script closes the row, closes the <table>, <body>, and <html> tags, and closes the output file.

```
writeln myreport "</tr>"
endfor
writeln myreport "</table>"
writeln myreport "</body>"
writeln myreport "</html>"
close myreport
```

Stream Report

This script creates an HTML report listing the name, type, and annotation for each node in the current stream. In addition to the basics of creating an HTML file and accessing node and stream properties, it demonstrates how to create a loop that executes a specific set of statements for each node within a stream. It can be used with any stream.

Figure 6-2
Stream report sample script

```
# Create the HTML page with heading
var myfile
set myfile = open create "c:\stream_report.html"
writeln myfile "<HTML>"
writeln myfile " <BODY>"
writeln myfile " <HEAD>Report for stream " >< ^stream.name >< ".str</HEAD>"
writeln myfile " <p>" >< ^stream.annotation >< "</p>"

#Create the table with header row
writeln myfile "<TABLE border=\"1\" width=\"90%\">"
writeln myfile " <tr bgcolor=\"lightgrey\" colspan=\"3\">"
writeln myfile "   <th>Node Name</th>"
writeln myfile "   <th>Type</th>"
writeln myfile "   <th>Annotation</th>"
writeln myfile " </tr>"

# Loop through nodes and add name, type, and annotation for each
# The ^stream.nodes property returns the list of nodes
var current_node
for current_node in ^stream.nodes
 writeln myfile "<tr>"
 writeln myfile "  <td>"
 writeln myfile     ^current_node.name
 writeln myfile "  </td>"
 writeln myfile "  <td>"
 writeln myfile     ^current_node.node_type
 writeln myfile "  </td>"
 writeln myfile "  <td>"
 writeln myfile     ^current_node.annotation >< " "
 writeln myfile "  </td>"
 writeln myfile "</tr>"
endfor
```

```
writeln myfile "</TABLE>"
writeln myfile "</BODY>"
writeln myfile "</HTML>"
close myfile
```

Creating the Report

The script begins by creating a new HTML file with <BODY> and <HEAD> elements. The ^stream.name property returns the name of the current stream, which is inserted into the heading. The >< operator is used to concatenate strings together.

```
# Create the HTML page with heading
var myfile
set myfile = open create "c:\stream_report.html"
writeln myfile "<HTML>"
writeln myfile " <BODY>"
writeln myfile " <HEAD>Report for stream " >< ^stream.name >< ".str</HEAD>"
writeln myfile " <p>" >< ^stream.annotation >< "</p>"
```

Next, the script creates an HTML table with a heading row listing the column titles *Node Name*, *Type*, and *Annotation*. (Each <td></td> tag pair creates a cell within a table row.)

```
#Create the table with header row
writeln myfile "<TABLE border=\"1\" width=\"90%\">"
writeln myfile " <tr bgcolor=\"lightgrey\" colspan=\"3\">"
writeln myfile "   <th>Node Name</th>"
writeln myfile "   <th>Type</th>"
writeln myfile "   <th>Annotation</th>"
writeln myfile " </tr>"
```

Next, the script loops through all nodes in the current stream. A row is added to the table for each node, listing the name, type, and annotation. An invisible nonbreaking space ( ) is inserted following the annotation to avoid creating an empty cell in cases where no annotation is specified for a given node. (Empty cells may result in unexpected formatting when displaying the table.)

```
# Loop through nodes and add name, type, and annotation for each
# The ^stream.nodes property returns the list of nodes
var current_node
for current_node in ^stream.nodes
 writeln myfile "<tr>"
 writeln myfile "  <td>"
 writeln myfile     ^current_node.name
 writeln myfile "  </td>"
 writeln myfile "  <td>"
 writeln myfile     ^current_node.node_type
 writeln myfile "  </td>"
 writeln myfile "  <td>"
 writeln myfile     ^current_node.annotation >< " "
 writeln myfile "  </td>"
 writeln myfile "</tr>"
endfor
```

Finally, the script adds the HTML tags necessary to close the document and closes the file.

```
writeln myfile "</TABLE>"
writeln myfile "</BODY>"
writeln myfile "</HTML>"
close myfile
```

Chapter

7

Command Line Arguments

Invoking the Software

You can use the command line of your operating system to launch IBM® SPSS® Modeler as follows:

- On a computer where IBM® SPSS® Modeler is installed, open a DOS, or command-prompt, window.
- To launch the SPSS Modeler interface in interactive mode, type the modelerclient command followed by the required arguments; for example:

```
modelerclient -stream report.str -execute
```

The available arguments (flags) allow you to connect to a server, load streams, run scripts, or specify other parameters as needed.

Using Command Line Arguments

You can append command line arguments (also referred to as **flags**) to the initial modelerclient command to alter the invocation of IBM® SPSS® Modeler.

Several types of command line arguments are available, and are described later in this section.

Table 7-1
Types of command line arguments

Argument type	Where described
System arguments	For more information, see the topic System Arguments on p. 69.
Parameter arguments	For more information, see the topic Parameter Arguments on p. 71.
Server connection arguments	For more information, see the topic Server Connection Arguments on p. 71.
IBM® SPSS® Collaboration and Deployment Services Repository connection arguments	For more information, see the topic IBM SPSS Collaboration and Deployment Services Repository Connection Arguments on p. 72.

For example, you can use the -server, -stream and -execute flags to connect to a server and then load and run a stream, as follows:

```
modelerclient -server -hostname myserver -port 80 -username dminer
-password 1234 -stream mystream.str -execute
```

Note that when running against a local client installation, the server connection arguments are not required.

Parameter values that contain spaces can be enclosed in double quotes—for example:

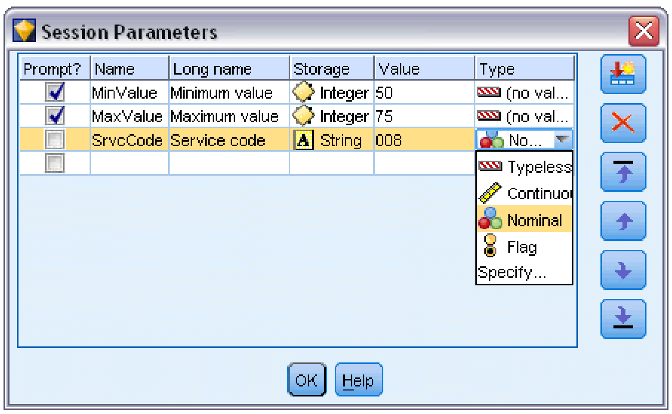
```
modelerclient -stream mystream.str -Pusername="Joe User" -execute
```

You can also execute SPSS Modeler states and scripts in this manner, using the `-state` and `-script` flags, respectively.

Debugging Command Line Arguments

To debug a command line, use the `modelerclient` command to launch SPSS Modeler with the desired arguments. This allows you to verify that commands will execute as expected. You can also confirm the values of any parameters passed from the command line in the Session Parameters dialog box (Tools menu, Set Session Parameters).

Figure 7-1
Setting parameters for the session



System Arguments

The following table describes system arguments available for command line invocation of the user interface:

Argument	Behavior/Description
@ <commandFile>	The @ character followed by a filename specifies a command list. When `modelerclient` encounters an argument beginning with @, it operates on the commands in that file as if they had been on the command line. For more information, see the topic Combining Multiple Arguments on p. 73.
`-directory` <dir>	Sets the default working directory. In local mode, this directory is used for both data and output. Example: `-directory c:/` or `-directory c:\\`
`-server_directory` <dir>	Sets the default server directory for data. The working directory, specified by using the `-directory` flag, is used for output.
`-execute`	After starting, execute any stream, state, or script loaded at startup. If a script is loaded in addition to a stream or state, the script alone will be executed.
`-stream` <stream>	At startup, load the stream specified. Multiple streams can be specified, but the last stream specified will be set as the current stream.
`-script` <script>	At startup, load the standalone script specified. This can be specified in addition to a stream or state as described below, but only one script can be loaded at startup.

Argument	Behavior/Description
-model <model>	At startup, load the generated model (*.gm* format file) specified.
-state <state>	At startup, load the saved state specified.
-project <project>	Load the specified project. Only one project can be loaded at startup.
-output <output>	At startup, load the saved output object (*.cou* format file).
-help	Display a list of command line arguments. When this option is specified, all other arguments are ignored and the Help screen is displayed.
-P <name>=<value>	Used to set a startup parameter. Can also be used to set node properties (slot parameters).

Note: Default directories can also be set in the user interface. To access the options, from the File menu, choose Set Working Directory or Set Server Directory.

Loading Multiple Files

From the command line, you can load multiple streams, states, and outputs at startup by repeating the relevant argument for each object loaded. For example, to load and run two streams called *report.str* and *train.str*, you would use the following command:

```
modelerclient -stream report.str -stream train.str -execute
```

Loading Objects from the IBM SPSS Collaboration and Deployment Services Repository

Because you can load certain objects from a file or from the IBM® SPSS® Collaboration and Deployment Services Repository (if licensed), the filename prefix spsscr: and, optionally, file: (for objects on disk) tells IBM® SPSS® Modeler where to look for the object. The prefix works with the following flags:

- -stream
- -script
- -output
- -model
- -project

You use the prefix to create a URI that specifies the location of the object—for example, -stream "spsscr:///folder_1/scoring_stream.str". The presence of the spsscr: prefix requires that a valid connection to the IBM SPSS Collaboration and Deployment Services Repository has been specified in the same command. So, for example, the full command would look like this:

```
modelerclient -spsscr_hostname myhost -spsscr_port 8080
-spsscr_username myusername -spsscr_password mypassword
-stream "spsscr:///folder_1/scoring_stream.str" -execute
```

For more details about URIs for objects in the IBM SPSS Collaboration and Deployment Services Repository, see Accessing Objects in the IBM SPSS Collaboration and Deployment Services Repository in Chapter 5 on p. 58. Note that from the command line, you *must* use a URI. The simpler REPOSITORY_PATH is not supported. (It works only within scripts.)

Parameter Arguments

Parameters can be used as flags during command line execution of IBM® SPSS® Modeler. In command line arguments, the `-P` flag is used to denote a parameter of the form `-P` <*name*>=<*value*>.

Parameters can be any of the following:

- **Simple parameters** (or parameters used directly in CLEM expressions).
- **Slot parameters**, also referred to as **node properties**. These parameters are used to modify the settings of nodes in the stream. For more information, see the topic Node Properties Overview in Chapter 9 on p. 109.
- **Command line parameters**, used to alter the invocation of SPSS Modeler.

For example, you can supply data source user names and passwords as a command line flag, as follows:

```
modelerclient -stream response.str -P:databasenode.datasource={"ORA 10gR2", user1, mypsw, true}
```

The format is the same as that of the `datasource` parameter of the `databasenode` node property. For more information, see the topic databasenode Properties in Chapter 12 on p. 118.

Server Connection Arguments

The `-server` flag tells IBM® SPSS® Modeler that it should connect to a public server, and the flags `-hostname`, `-use_ssl`, `-port`, `-username`, `-password`, and `-domain` are used to tell SPSS Modeler how to connect to the public server. If no -server argument is specified, the default or local server is used.

Examples

To connect to a public server:

```
modelerclient -server -hostname myserver -port 80 -username dminer
-password 1234 -stream mystream.str -execute
```

To connect to a server cluster:

```
modelerclient -server -cluster "QA Machines" \
-spsscr_hostname pes_host -spsscr_port 8080 \
-spsscr_username asmith -spsscr_epassword xyz
```

Note that connecting to a server cluster requires the Coordinator of Processes through IBM® SPSS® Collaboration and Deployment Services, so the `-cluster` argument must be used in combination with the repository connection options (`spsscr_*`). For more information, see the topic IBM SPSS Collaboration and Deployment Services Repository Connection Arguments on p. 72.

Argument	Behavior/Description
-server	Runs SPSS Modeler in server mode, connecting to a public server using the flags -hostname, -port, -username, -password, and -domain.
-hostname <name>	The hostname of the server machine. Available in server mode only.
-use_ssl	Specifies that the connection should use SSL (secure socket layer). This flag is optional; the default setting is *not* to use SSL.
-port <number>	The port number of the specified server. Available in server mode only.
-cluster <name>	Specifies a connection to a server cluster rather than a named server; this argument is an alternative to the hostname, port and use_ssl arguments. The name is the cluster name, or a unique URI which identifies the cluster in the IBM® SPSS® Collaboration and Deployment Services Repository. The server cluster is managed by the Coordinator of Processes through IBM SPSS Collaboration and Deployment Services. For more information, see the topic IBM SPSS Collaboration and Deployment Services Repository Connection Arguments on p. 72.
-username <name>	The user name with which to log on to the server. Available in server mode only.
-password <password>	The password with which to log on to the server. Available in server mode only. *Note*: If the -password argument is not used, you will be prompted for a password.
-epassword <encodedpasswordstring>	The encoded password with which to log on to the server. Available in server mode only. *Note*: An encoded password can be generated from the Tools menu of the SPSS Modeler application.
-domain <name>	The domain used to log on to the server. Available in server mode only.
-P <name>=<value>	Used to set a startup parameter. Can also be used to set node properties (slot parameters).

IBM SPSS Collaboration and Deployment Services Repository Connection Arguments

Note: A separate license is required to access an IBM® SPSS® Collaboration and Deployment Services repository. For more information, see *http://www.ibm.com/software/analytics/spss/products/deployment/cds/*

If you want to store or retrieve objects from IBM SPSS Collaboration and Deployment Services via the command line, you must specify a valid connection to the IBM® SPSS® Collaboration and Deployment Services Repository. For example:

```
modelerclient -spsscr_hostname myhost -spsscr_port 8080
-spsscr_username myusername -spsscr_password mypassword
-stream "spsscr:///folder_1/scoring_stream.str" -execute
```

The following table lists the arguments that can be used to set up the connection:

Argument	Behavior/Description
-spsscr_hostname <hostname or IP address>	The hostname or IP address of the server on which the IBM SPSS Collaboration and Deployment Services Repository is installed.
-spsscr_port <number>	The port number on which the IBM SPSS Collaboration and Deployment Services Repositoryaccepts connections (typically, 8080 by default).

Argument	Behavior/Description
-spsscr_use_ssl	Specifies that the connection should use SSL (secure socket layer). This flag is optional; the default setting is *not* to use SSL.
-spsscr_username <name>	The user name with which to log on to the IBM SPSS Collaboration and Deployment Services Repository.
-spsscr_password <password>	The password with which to log on to the IBM SPSS Collaboration and Deployment Services Repository.
-spsscr_epassword <encoded password>	The encoded password with which to log on to the IBM SPSS Collaboration and Deployment Services Repository.
-spsscr_domain <name>	The domain used to log on to the IBM SPSS Collaboration and Deployment Services Repository. This flag is optional—do not use it unless you log on by using LDAP or Active Directory.

Combining Multiple Arguments

Multiple arguments can be combined in a single command file specified at invocation by using the @ symbol followed by the filename. This enables you to shorten the command line invocation and overcome any operating system limitations on command length. For example, the following startup command uses the arguments specified in the file referenced by <commandFileName>.

```
modelerclient @<commandFileName>
```

Enclose the filename and path to the command file in quotation marks if spaces are required, as follows:

```
modelerclient @ "C:\Program Files\IBM\SPSS\Modeler\nn\scripts\my_command_file.txt"
```

The command file can contain all arguments previously specified individually at startup, with one argument per line. For example:

```
-stream report.str
-Porder.full_filename=APR_orders.dat
-Preport.filename=APR_report.txt
-execute
```

When writing and referencing command files, be sure to follow these constraints:

- Use only one command per line.
- Do not embed an @CommandFile argument within a command file.

Chapter

8

CLEM Language Reference

CLEM Reference Overview

This section describes the Control Language for Expression Manipulation (CLEM), which is a powerful tool used to analyze and manipulate the data used in IBM® SPSS® Modeler streams. You can use CLEM within nodes to perform tasks ranging from evaluating conditions or deriving values to inserting data into reports.

A subset of the CLEM language can also be used when you are scripting in the user interface. This allows you to perform many of the same data manipulations in an automated fashion. For more information, see the topic CLEM Expressions in Scripts in Chapter 3 on p. 27.

CLEM expressions consist of values, field names, operators, and functions. Using the correct syntax, you can create a wide variety of powerful data operations.

CLEM Datatypes

CLEM datatypes can be made up of any of the following:

- Integers
- Reals
- Characters
- Strings
- Lists
- Fields
- Date/Time

Rules for Quoting

Although IBM® SPSS® Modeler is flexible when you are determining the fields, values, parameters, and strings used in a CLEM expression, the following general rules provide a list of "good practices" to use in creating expressions:

- Strings—Always use double quotes when writing strings, such as `"Type 2"`. Single quotes can be used instead but at the risk of confusion with quoted fields.
- Fields—Use single quotes only where necessary to enclose spaces or other special characters, such as `'Order Number'`. Fields that are quoted but undefined in the data set will be misread as strings.
- Parameters—Always use single quotes when using parameters, such as `'$P-threshold'`.
- Characters—Always use single backquotes (`), such as ``stripchar(`d`, "drugA")``.

Additionally, these rules are covered in more detail in the following topics.

Integers

Integers are represented as a sequence of decimal digits. Optionally, you can place a minus sign (–) before the integer to denote a negative number—for example, 1234, 999, –77.

The CLEM language handles integers of arbitrary precision. The maximum integer size depends on your platform. If the values are too large to be displayed in an integer field, changing the field type to Real usually restores the value.

Reals

Real refers to a floating-point number. Reals are represented by one or more digits followed by a decimal point followed by one or more digits. CLEM reals are held in double precision.

Optionally, you can place a minus sign (–) before the real to denote a negative number—for example, 1.234, 0.999, –77.001. Use the form <*number*> e <*exponent*> to express a real number in exponential notation—for example, 1234.0e5, 1.7e–2. When the IBM® SPSS® Modeler application reads number strings from files and converts them automatically to numbers, numbers with no leading digit before the decimal point or with no digit after the point are accepted—for example, 999. or .11. However, these forms are illegal in CLEM expressions.

Note: When referencing real numbers in CLEM expressions, a period must be used as the decimal separator, regardless of any settings for the current stream or locale. For example, specify

```
Na > 0.6
```

rather than

```
Na > 0,6
```

This applies even if a comma is selected as the decimal symbol in the stream properties dialog box and is consistent with the general guideline that code syntax should be independent of any specific locale or convention.

Characters

Characters (usually shown as CHAR) are typically used within a CLEM expression to perform tests on strings. For example, you can use the function isuppercode to determine whether the first character of a string is upper case. The following CLEM expression uses a character to indicate that the test should be performed on the first character of the string:

```
isuppercode(subscrs(1, "MyString"))
```

To express the code (in contrast to the location) of a particular character in a CLEM expression, use single backquotes of the form `<*character*>`—for example, `A`, `Z`.

Note: There is no CHAR storage type for a field, so if a field is derived or filled with an expression that results in a CHAR, then that result will be converted to a string.

Strings

Generally, you should enclose strings in double quotation marks. Examples of strings are "c35product2" and "referrerID". To indicate special characters in a string, use a backslash—for example, "\$65443". (To indicate a backslash character, use a double backslash, \\.) You can use single quotes around a string, but the result is indistinguishable from a quoted field ('referrerID'). For more information, see the topic String Functions on p. 88.

Lists

A list is an ordered sequence of elements, which may be of mixed type. Lists are enclosed in square brackets ([]). Examples of lists are [1 2 4 16] and ["abc" "def"]. Lists are not used as the value of IBM® SPSS® Modeler fields. They are used to provide arguments to functions, such as member and oneof.

Fields

Names in CLEM expressions that are not names of functions are assumed to be field names. You can write these simply as Power, val27, state_flag, and so on, but if the name begins with a digit or includes non-alphabetic characters, such as spaces (with the exception of the underscore), place the name within single quotation marks—for example, 'Power Increase', '2nd answer', '#101', '$P-NextField'.

Note: Fields that are quoted but undefined in the data set will be misread as strings.

Dates

Date calculations are based on a "baseline" date, which is specified in the stream properties dialog box. The default baseline date is 1 January 1900.

The CLEM language supports the following date formats.

Format	Examples
DDMMYY	150163
MMDDYY	011563
YYMMDD	630115
YYYYMMDD	19630115
YYYYDDD	Four-digit year followed by a three-digit number representing the day of the year—for example, 2000032 represents the 32nd day of 2000, or 1 February 2000.
DAY	Day of the week in the current locale—for example, Monday, Tuesday, ..., in English.
MONTH	Month in the current locale—for example, January, February, ….
DD/MM/YY	15/01/63
DD/MM/YYYY	15/01/1963

Format	Examples
MM/DD/YY	01/15/63
MM/DD/YYYY	01/15/1963
DD-MM-YY	15-01-63
DD-MM-YYYY	15-01-1963
MM-DD-YY	01-15-63
MM-DD-YYYY	01-15-1963
DD.MM.YY	15.01.63
DD.MM.YYYY	15.01.1963
MM.DD.YY	01.15.63
MM.DD.YYYY	01.15.1963
DD-MON-YY	15-JAN-63, 15-jan-63, 15-Jan-63
DD/MON/YY	15/JAN/63, 15/jan/63, 15/Jan/63
DD.MON.YY	15.JAN.63, 15.jan.63, 15.Jan.63
DD-MON-YYYY	15-JAN-1963, 15-jan-1963, 15-Jan-1963
DD/MON/YYYY	15/JAN/1963, 15/jan/1963, 15/Jan/1963
DD.MON.YYYY	15.JAN.1963, 15.jan.1963, 15.Jan.1963
MON YYYY	Jan 2004
q Q YYYY	Date represented as a digit (1–4) representing the quarter followed by the letter *Q* and a four-digit year—for example, 25 December 2004 would be represented as 4 Q 2004.
ww WK YYYY	Two-digit number representing the week of the year followed by the letters *WK* and then a four-digit year. The week of the year is calculated assuming that the first day of the week is Monday and there is at least one day in the first week.

Time

The CLEM language supports the following time formats.

Format	Examples
HHMMSS	120112, 010101, 221212
HHMM	1223, 0745, 2207
MMSS	5558, 0100
HH:MM:SS	12:01:12, 01:01:01, 22:12:12
HH:MM	12:23, 07:45, 22:07
MM:SS	55:58, 01:00
(H)H:(M)M:(S)S	12:1:12, 1:1:1, 22:12:12
(H)H:(M)M	12:23, 7:45, 22:7
(M)M:(S)S	55:58, 1:0
HH.MM.SS	12.01.12, 01.01.01, 22.12.12
HH.MM	12.23, 07.45, 22.07
MM.SS	55.58, 01.00

Format	Examples
(H)H.(M)M.(S)S	12.1.12, 1.1.1, 22.12.12
(H)H.(M)M	12.23, 7.45, 22.7
(M)M.(S)S	55.58, 1.0

CLEM Operators

The following operators are available.

Operation	Comments	Precedence (see next section)
or	Used between two CLEM expressions. Returns a value of true if either is true or if both are true.	10
and	Used between two CLEM expressions. Returns a value of true if both are true.	9
=	Used between any two comparable items. Returns true if ITEM1 is equal to ITEM2.	7
==	Identical to =.	7
/=	Used between any two comparable items. Returns true if ITEM1 is *not* equal to ITEM2.	7
/==	Identical to /=.	7
>	Used between any two comparable items. Returns true if ITEM1 is strictly greater than ITEM2.	6
>=	Used between any two comparable items. Returns true if ITEM1 is greater than or equal to ITEM2.	6
<	Used between any two comparable items. Returns true if ITEM1 is strictly less than ITEM2	6
<=	Used between any two comparable items. Returns true if ITEM1 is less than or equal to ITEM2.	6
&&=_0	Used between two integers. Equivalent to the Boolean expression INT1 && INT2 = 0.	6
&&/=_0	Used between two integers. Equivalent to the Boolean expression INT1 && INT2 /= 0.	6
+	Adds two numbers: NUM1 + NUM2.	5
><	Concatenates two strings; for example, STRING1 >< STRING2.	5
-	Subtracts one number from another: NUM1 - NUM2. Can also be used in front of a number: - NUM.	5
*	Used to multiply two numbers: NUM1 * NUM2.	4

<table>
<tr><th>Operation</th><th>Comments</th><th>Precedence (see next section)</th></tr>
<tr><td>&&</td><td>Used between two integers. The result is the bitwise ‘and’ of the integers INT1 and INT2.</td><td>4</td></tr>
<tr><td>&&~~</td><td>Used between two integers. The result is the bitwise ‘and’ of INT1 and the bitwise complement of INT2.</td><td>4</td></tr>
<tr><td>||</td><td>Used between two integers. The result is the bitwise ‘inclusive or’ of INT1 and INT2.</td><td>4</td></tr>
<tr><td>~~</td><td>Used in front of an integer. Produces the bitwise complement of INT.</td><td>4</td></tr>
<tr><td>||/&</td><td>Used between two integers. The result is the bitwise ‘exclusive or’ of INT1 and INT2.</td><td>4</td></tr>
<tr><td>INT1 << N</td><td>Used between two integers. Produces the bit pattern of INT shifted left by N positions.</td><td>4</td></tr>
<tr><td>INT1 >> N</td><td>Used between two integers. Produces the bit pattern of INT shifted right by N positions.</td><td>4</td></tr>
<tr><td>/</td><td>Used to divide one number by another: NUM1 / NUM2.</td><td>4</td></tr>
<tr><td>**</td><td>Used between two numbers: BASE ** POWER. Returns BASE raised to the power POWER.</td><td>3</td></tr>
<tr><td>rem</td><td>Used between two integers: INT1 rem INT2. Returns the remainder, INT1 - (INT1 div INT2) * INT2.</td><td>2</td></tr>
<tr><td>div</td><td>Used between two integers: INT1 div INT2. Performs integer division.</td><td>2</td></tr>
</table>

Operator Precedence

Precedences determine the parsing of complex expressions, especially unbracketed expressions with more than one infix operator. For example,

```
3 + 4 * 5
```

parses as `3 + (4 * 5)` rather than `(3 + 4) * 5` because the relative precedences dictate that * is to be parsed before +. Every operator in the CLEM language has a precedence value associated with it; the lower this value, the more important it is on the parsing list, meaning that it will be processed sooner than other operators with higher precedence values.

Functions Reference

The following CLEM functions are available for working with data in IBM® SPSS® Modeler. You can enter these functions as code in a variety of dialog boxes, such as Derive and Set To Flag nodes, or you can use the Expression Builder to create valid CLEM expressions without memorizing function lists or field names.

Function Type	Description
Information	Used to gain insight into field values. For example, the function `is_string` returns true for all records whose type is a string.
Conversion	Used to construct new fields or convert storage type. For example, the function `to_timestamp` converts the selected field to a timestamp.
Comparison	Used to compare field values to each other or to a specified string. For example, <= is used to compare whether the values of two fields are lesser or equal.
Logical	Used to perform logical operations, such as `if`, `then`, `else` operations.
Numeric	Used to perform numeric calculations, such as the natural log of field values.
Trigonometric	Used to perform trigonometric calculations, such as the arccosine of a specified angle.
Probability	Return probabilities based on various distributions, such as probability that a value from Student's *t* distribution will be less than a specific value.
Bitwise	Used to manipulate integers as bit patterns.
Random	Used to randomly select items or generate numbers.
String	Used to perform a wide variety of operations on strings, such as `stripchar`, which allows you to remove a specified character.
SoundEx	Used to find strings when the precise spelling is not known; based on phonetic assumptions about how certain letters are pronounced.
Date and time	Used to perform a variety of operations on date, time, and timestamp fields.
Sequence	Used to gain insight into the record sequence of a data set or perform operations based on that sequence.
Global	Used to access global values created by a Set Globals node. For example, `@MEAN` is used to refer to the mean average of all values for a field across the entire data set.
Blanks and null	Used to access, flag, and frequently fill user-specified blanks or system-missing values. For example, `@BLANK(FIELD)` is used to raise a true flag for records where blanks are present.
Special fields	Used to denote the specific fields under examination. For example, `@FIELD` is used when deriving multiple fields.

Conventions in Function Descriptions

The following conventions are used throughout this guide when referring to items in a function.

Convention	Description
BOOL	A Boolean, or flag, such as true or false.
NUM, *NUM1*, *NUM2*	Any number.
REAL, *REAL1*, *REAL2*	Any real number, such as `1.234` or `–77.01`.

Convention	Description
INT, *INT1*, *INT2*	Any integer, such as 1 or –77.
CHAR	A character code, such as `A`.
STRING	A string, such as "referrerID".
LIST	A list of items, such as ["abc" "def"].
ITEM	A field, such as Customer or extract_concept.
DATE	A date field, such as start_date, where values are in a format such as DD-MON-YYYY.
TIME	A time field, such as power_flux, where values are in a format such as HHMMSS.

Functions in this guide are listed with the function in one column, the result type (integer, string, and so on) in another, and a description (where available) in a third column. For example, the following is the description of the rem function.

Function	Result	Description
INT1 rem INT2	*Number*	Returns the remainder of *INT1* divided by *INT2*. For example, INT1 – (INT1 div INT2) * INT2.

Details on usage conventions, such as how to list items or specify characters in a function, are described elsewhere. For more information, see the topic CLEM Datatypes on p. 74.

Information Functions

Information functions are used to gain insight into the values of a particular field. They are typically used to derive flag fields. For example, you can use the @BLANK function to create a flag field indicating records whose values are blank for the selected field. Similarly, you can check the storage type for a field using any of the storage type functions, such as is_string.

Function	Result	Description
@BLANK(FIELD)	*Boolean*	Returns true for all records whose values are blank according to the blank-handling rules set in an upstream Type node or source node (Types tab). Note that this function cannot be called from a script. For more information, see the topic CLEM Expressions in Scripts in Chapter 3 on p. 27.
@NULL(ITEM)	*Boolean*	Returns true for all records whose values are undefined. Undefined values are system null values, displayed in IBM® SPSS® Modeler as $null$. Note that this function cannot be called from a script. For more information, see the topic CLEM Expressions in Scripts in Chapter 3 on p. 27.
is_date(ITEM)	*Boolean*	Returns true for all records whose type is a date.
is_datetime(ITEM)	*Boolean*	Returns true for all records whose type is a date, time, or timestamp.
is_integer(ITEM)	*Boolean*	Returns true for all records whose type is an integer.
is_number(ITEM)	*Boolean*	Returns true for all records whose type is a number.
is_real(ITEM)	*Boolean*	Returns true for all records whose type is a real.
is_string(ITEM)	*Boolean*	Returns true for all records whose type is a string.

Function	Result	Description
is_time(ITEM)	*Boolean*	Returns true for all records whose type is a time.
is_timestamp(ITEM)	*Boolean*	Returns true for all records whose type is a timestamp.

Conversion Functions

Conversion functions allow you to construct new fields and convert the storage type of existing files. For example, you can form new strings by joining strings together or by taking strings apart. To join two strings, use the operator ><. For example, if the field Site has the value "BRAMLEY", then "xx" >< Site returns "xxBRAMLEY". The result of >< is always a string, even if the arguments are not strings. Thus, if field V1 is 3 and field V2 is 5, then V1 >< V2 returns "35" (a string, not a number).

Conversion functions (and any other functions that require a specific type of input, such as a date or time value) depend on the current formats specified in the Stream Options dialog box. For example, if you want to convert a string field with values *Jan 2003*, *Feb 2003*, and so on, select the matching date format MON YYYY as the default date format for the stream.

Function	Result	Description
ITEM1 >< ITEM2	*String*	Concatenates values for two fields and returns the resulting string as *ITEM1ITEM2*.
to_integer(ITEM)	*Integer*	Converts the storage of the specified field to an integer.
to_real(ITEM)	*Real*	Converts the storage of the specified field to a real.
to_number(ITEM)	*Number*	Converts the storage of the specified field to a number.
to_string(ITEM)	*String*	Converts the storage of the specified field to a string.
to_time(ITEM)	*Time*	Converts the storage of the specified field to a time.
to_date(ITEM)	*Date*	Converts the storage of the specified field to a date.
to_timestamp(ITEM)	*Timestamp*	Converts the storage of the specified field to a timestamp.
to_datetime(ITEM)	*Datetime*	Converts the storage of the specified field to a date, time, or timestamp value.
datetime_date(ITEM)	*Date*	Returns the date value for a *number*, *string*, or *timestamp*. Note this is the only function that allows you to convert a number (in seconds) back to a date. If ITEM is a string, creates a date by parsing a string in the current date format. The date format specified in the stream properties dialog box must be correct for this function to be successful. If ITEM is a number, it is interpreted as a number of seconds since the base date (or epoch). Fractions of a day are truncated. If ITEM is a timestamp, the date part of the timestamp is returned. If ITEM is a date, it is returned unchanged.

Comparison Functions

Comparison functions are used to compare field values to each other or to a specified string. For example, you can check strings for equality using =. An example of string equality verification is: Class = "class 1".

For purposes of numeric comparison, *greater* means closer to positive infinity, and *lesser* means closer to negative infinity. That is, all negative numbers are less than any positive number.

Function	Result	Description
count_equal(ITEM1, LIST)	*Integer*	Returns the number of values from a list of fields that are equal to *ITEM1* or null if *ITEM1* is null.
count_greater_than(ITEM1, LIST)	*Integer*	Returns the number of values from a list of fields that are greater than *ITEM1* or null if *ITEM1* is null.
count_less_than(ITEM1, LIST)	*Integer*	Returns the number of values from a list of fields that are less than *ITEM1* or null if *ITEM1* is null.
count_not_equal(ITEM1, LIST)	*Integer*	Returns the number of values from a list of fields that are not equal to *ITEM1* or null if *ITEM1* is null.
count_nulls(LIST)	*Integer*	Returns the number of null values from a list of fields.
count_non_nulls(LIST)	*Integer*	Returns the number of non-null values from a list of fields.
date_before(DATE1, DATE2)	*Boolean*	Used to check the ordering of date values. Returns a true value if *DATE1* is before *DATE2*.
first_index(ITEM, LIST)	*Integer*	Returns the index of the first field containing ITEM from a LIST of fields or 0 if the value is not found. Supported for string, integer, and real types only.
first_non_null(LIST)	*Any*	Returns the first non-null value in the supplied list of fields. All storage types supported.
first_non_null_index(LIST)	*Integer*	Returns the index of the first field in the specified LIST containing a non-null value or 0 if all values are null. All storage types are supported.
ITEM1 = ITEM2	*Boolean*	Returns true for records where *ITEM1* is equal to *ITEM2*.
ITEM1 /= ITEM2	*Boolean*	Returns true if the two strings are not identical or 0 if they are identical.
ITEM1 < ITEM2	*Boolean*	Returns true for records where *ITEM1* is less than *ITEM2*.
ITEM1 <= ITEM2	*Boolean*	Returns true for records where *ITEM1* is less than or equal to *ITEM2*.
ITEM1 > ITEM2	*Boolean*	Returns true for records where *ITEM1* is greater than *ITEM2*.
ITEM1 >= ITEM2	*Boolean*	Returns true for records where *ITEM1* is greater than or equal to *ITEM2*.
last_index(ITEM, LIST)	*Integer*	Returns the index of the last field containing ITEM from a LIST of fields or 0 if the value is not found. Supported for string, integer, and real types only.
last_non_null(LIST)	*Any*	Returns the last non-null value in the supplied list of fields. All storage types supported.
last_non_null_index(LIST)	*Integer*	Returns the index of the last field in the specified LIST containing a non-null value or 0 if all values are null. All storage types are supported.
max(ITEM1, ITEM2)	*Any*	Returns the greater of the two items—*ITEM1* or *ITEM2*.
max_index(LIST)	*Integer*	Returns the index of the field containing the maximum value from a list of numeric fields or 0 if all values are null. For example, if the third field listed contains the maximum, the index value 3 is returned. If multiple fields contain the maximum value, the one listed first (leftmost) is returned.
max_n(LIST)	*Number*	Returns the maximum value from a list of numeric fields or null if all of the field values are null.

Function	Result	Description
member(ITEM, LIST)	*Boolean*	Returns true if *ITEM* is a member of the specified *LIST*. Otherwise, a false value is returned. A list of field names can also be specified.
min(ITEM1, ITEM2)	*Any*	Returns the lesser of the two items—*ITEM1* or *ITEM2*.
min_index(LIST)	*Integer*	Returns the index of the field containing the minimum value from a list of numeric fields or 0 if all values are null. For example, if the third field listed contains the minimum, the index value 3 is returned. If multiple fields contain the minimum value, the one listed first (leftmost) is returned.
min_n(LIST)	*Number*	Returns the minimum value from a list of numeric fields or null if all of the field values are null.
time_before(TIME1, TIME2)	*Boolean*	Used to check the ordering of time values. Returns a true value if *TIME1* is before *TIME2*.
value_at(INT, LIST)		Returns the value of each listed field at offset INT or NULL if the offset is outside the range of valid values (that is, less than 1 or greater than the number of listed fields). All storage types supported.

Logical Functions

CLEM expressions can be used to perform logical operations.

Function	Result	Description
COND1 and COND2	*Boolean*	This operation is a logical conjunction and returns a true value if both *COND1* and *COND2* are true. If *COND1* is false, then *COND2* is not evaluated; this makes it possible to have conjunctions where *COND1* first tests that an operation in *COND2* is legal. For example, length(Label) >=6 and Label(6) = 'x'.
COND1 or COND2	*Boolean*	This operation is a logical (inclusive) disjunction and returns a true value if either *COND1* or *COND2* is true or if both are true. If *COND1* is true, *COND2* is not evaluated.
not(COND)	*Boolean*	This operation is a logical negation and returns a true value if *COND* is false. Otherwise, this operation returns a value of 0.
if COND then EXPR1 else EXPR2 endif	*Any*	This operation is a conditional evaluation. If *COND* is true, this operation returns the result of *EXPR1*. Otherwise, the result of evaluating *EXPR2* is returned.
if COND1 then EXPR1 elseif COND2 then EXPR2 else EXPR_N endif	*Any*	This operation is a multibranch conditional evaluation. If *COND1* is true, this operation returns the result of *EXPR1*. Otherwise, if *COND2* is true, this operation returns the result of evaluating *EXPR2*. Otherwise, the result of evaluating *EXPR_N* is returned.

Numeric Functions

CLEM contains a number of commonly used numeric functions.

Function	Result	Description
–NUM	*Number*	Used to negate *NUM*. Returns the corresponding number with the opposite sign.
NUM1 + NUM2	*Number*	Returns the sum of *NUM1* and *NUM2*.
code –NUM2	*Number*	Returns the value of *NUM2* subtracted from *NUM1*.
NUM1 * NUM2	*Number*	Returns the value of *NUM1* multiplied by *NUM2*.
NUM1 / NUM2	*Number*	Returns the value of *NUM1* divided by *NUM2*.
INT1 div INT2	*Number*	Used to perform integer division. Returns the value of *INT1* divided by *INT2*.
INT1 rem INT2	*Number*	Returns the remainder of *INT1* divided by *INT2*. For example, INT1 – (INT1 div INT2) * INT2.
INT1 mod INT2	*Number*	This function has been deprecated. Use the rem function instead.
BASE ** POWER	*Number*	Returns *BASE* raised to the power *POWER*, where either may be any number (except that *BASE* must not be zero if *POWER* is zero of any type other than integer 0). If *POWER* is an integer, the computation is performed by successively multiplying powers of *BASE*. Thus, if *BASE* is an integer, the result will be an integer. If *POWER* is integer 0, the result is always a 1 of the same type as *BASE*. Otherwise, if *POWER* is not an integer, the result is computed as exp(POWER * log(BASE)).
abs(NUM)	*Number*	Returns the absolute value of *NUM*, which is always a number of the same type.
exp(NUM)	*Real*	Returns *e* raised to the power *NUM*, where *e* is the base of natural logarithms.
fracof(NUM)	*Real*	Returns the fractional part of *NUM*, defined as NUM–intof(NUM).
intof(NUM)	*Integer*	Truncates its argument to an integer. It returns the integer of the same sign as *NUM* and with the largest magnitude such that abs(INT) <= abs(NUM).
log(NUM)	*Real*	Returns the natural (base *e*) logarithm of *NUM*, which must not be a zero of any kind.
log10(NUM)	*Real*	Returns the base 10 logarithm of *NUM*, which must not be a zero of any kind. This function is defined as log(NUM) / log(10).
negate(NUM)	*Number*	Used to negate *NUM*. Returns the corresponding number with the opposite sign.
round(NUM)	*Integer*	Used to round *NUM* to an integer by taking intof(NUM+0.5) if *NUM* is positive or intof(NUM–0.5) if *NUM* is negative.
sign(NUM)	*Number*	Used to determine the sign of *NUM*. This operation returns –1, 0, or 1 if *NUM* is an integer. If *NUM* is a real, it returns –1.0, 0.0, or 1.0, depending on whether *NUM* is negative, zero, or positive.
sqrt(NUM)	*Real*	Returns the square root of *NUM*. *NUM* must be positive.
sum_n(LIST)	*Number*	Returns the sum of values from a list of numeric fields or null if all of the field values are null.

Function	Result	Description
mean_n(LIST)	*Number*	Returns the mean value from a list of numeric fields or null if all of the field values are null.
sdev_n(LIST)	*Number*	Returns the standard deviation from a list of numeric fields or null if all of the field values are null.

Trigonometric Functions

All of the functions in this section either take an angle as an argument or return one as a result. In both cases, the units of the angle (radians or degrees) are controlled by the setting of the relevant stream option.

Function	Result	Description
arccos(NUM)	*Real*	Computes the arccosine of the specified angle.
arccosh(NUM)	*Real*	Computes the hyperbolic arccosine of the specified angle.
arcsin(NUM)	*Real*	Computes the arcsine of the specified angle.
arcsinh(NUM)	*Real*	Computes the hyperbolic arcsine of the specified angle.
arctan(NUM)	*Real*	Computes the arctangent of the specified angle.
arctan2(NUM_Y, NUM_X)	*Real*	Computes the arctangent of NUM_Y / NUM_X and uses the signs of the two numbers to derive quadrant information. The result is a real in the range - pi < ANGLE <= pi (radians) – 180 < ANGLE <= 180 (degrees)
arctanh(NUM)	*Real*	Computes the hyperbolic arctangent of the specified angle.
cos(NUM)	*Real*	Computes the cosine of the specified angle.
cosh(NUM)	*Real*	Computes the hyperbolic cosine of the specified angle.
pi	*Real*	This constant is the best real approximation to pi.
sin(NUM)	*Real*	Computes the sine of the specified angle.
sinh(NUM)	*Real*	Computes the hyperbolic sine of the specified angle.
tan(NUM)	*Real*	Computes the tangent of the specified angle.
tanh(NUM)	*Real*	Computes the hyperbolic tangent of the specified angle.

Probability Functions

Probability functions return probabilities based on various distributions, such as the probability that a value from Student's *t* distribution will be less than a specific value.

Function	Result	Description
cdf_chisq(NUM, DF)	*Real*	Returns the probability that a value from the chi-square distribution with the specified degrees of freedom will be less than the specified number.
cdf_f(NUM, DF1, DF2)	*Real*	Returns the probability that a value from the *F* distribution, with degrees of freedom *DF1* and *DF2*, will be less than the specified number.

Function	**Result**	**Description**
cdf_normal(NUM, MEAN, STDDEV)	*Real*	Returns the probability that a value from the normal distribution with the specified mean and standard deviation will be less than the specified number.
cdf_t(NUM, DF)	*Real*	Returns the probability that a value from Student's *t* distribution with the specified degrees of freedom will be less than the specified number.

Bitwise Integer Operations

These functions enable integers to be manipulated as bit patterns representing two's-complement values, where bit position N has weight 2**N. Bits are numbered from 0 upward. These operations act as though the sign bit of an integer is extended indefinitely to the left. Thus, everywhere above its most significant bit, a positive integer has 0 bits and a negative integer has 1 bit.

Note: Bitwise functions cannot be called from scripts. For more information, see the topic CLEM Expressions in Scripts in Chapter 3 on p. 27.

Function	**Result**	**Description**
~~ INT1	*Integer*	Produces the bitwise complement of the integer *INT1*. That is, there is a 1 in the result for each bit position for which *INT1* has 0. It is always true that ~~ INT = –(INT + 1). Note that this function cannot be called from a script. For more information, see the topic CLEM Expressions in Scripts in Chapter 3 on p. 27.
INT1 || INT2	*Integer*	The result of this operation is the bitwise "inclusive or" of *INT1* and *INT2*. That is, there is a 1 in the result for each bit position for which there is a 1 in either *INT1* or *INT2* or both.
INT1 ||/& INT2	*Integer*	The result of this operation is the bitwise "exclusive or" of *INT1* and *INT2*. That is, there is a 1 in the result for each bit position for which there is a 1 in either *INT1* or *INT2* but not in both.
INT1 && INT2	*Integer*	Produces the bitwise "and" of the integers *INT1* and *INT2*. That is, there is a 1 in the result for each bit position for which there is a 1 in both *INT1* and *INT2*.
INT1 &&~~ INT2	*Integer*	Produces the bitwise "and" of *INT1* and the bitwise complement of *INT2*. That is, there is a 1 in the result for each bit position for which there is a 1 in *INT1* and a 0 in *INT2*. This is the same as INT1&& (~~INT2) and is useful for clearing bits of *INT1* set in *INT2*.
INT << N	*Integer*	Produces the bit pattern of *INT1* shifted left by *N* positions. A negative value for *N* produces a right shift.
INT >> N	*Integer*	Produces the bit pattern of *INT1* shifted right by *N* positions. A negative value for *N* produces a left shift.
INT1 &&=_0 INT2	*Boolean*	Equivalent to the Boolean expression INT1 && INT2 /== 0 but is more efficient.
INT1 &&/=_0 INT2	*Boolean*	Equivalent to the Boolean expression INT1 && INT2 == 0 but is more efficient.

Function	Result	Description
integer_bitcount(INT)	*Integer*	Counts the number of 1 or 0 bits in the two's-complement representation of *INT*. If *INT* is non-negative, *N* is the number of 1 bits. If *INT* is negative, it is the number of 0 bits. Owing to the sign extension, there are an infinite number of 0 bits in a non-negative integer or 1 bits in a negative integer. It is always the case that integer_bitcount(INT) = integer_bitcount(-(INT+1)).
integer_leastbit(INT)	*Integer*	Returns the bit position *N* of the least-significant bit set in the integer *INT*. *N* is the highest power of 2 by which *INT* divides exactly.
integer_length(INT)	*Integer*	Returns the length in bits of *INT* as a two's-complement integer. That is, *N* is the smallest integer such that INT < (1 << N) if INT >= 0 INT >= (–1 << N) if INT < 0. If *INT* is non-negative, then the representation of *INT* as an unsigned integer requires a field of at least *N* bits. Alternatively, a minimum of *N+1* bits is required to represent *INT* as a signed integer, regardless of its sign.
testbit(INT, N)	*Boolean*	Tests the bit at position *N* in the integer *INT* and returns the state of bit *N* as a Boolean value, which is true for 1 and false for 0.

Random Functions

The following functions are used to randomly select items or randomly generate numbers.

Function	Result	Description
oneof(LIST)	*Any*	Returns a randomly chosen element of *LIST*. List items should be entered as [ITEM1,ITEM2,...,ITEM_N]. Note that a list of field names can also be specified.
random(NUM)	*Number*	Returns a uniformly distributed random number of the same type (*INT* or *REAL*), starting from 1 to *NUM*. If you use an integer, then only integers are returned. If you use a real (decimal) number, then real numbers are returned (decimal precision determined by the stream options). The largest random number returned by the function could equal *NUM*.
random0(NUM)	*Number*	This has the same properties as random(NUM), but starting from 0. The largest random number returned by the function will never equal *X*.

String Functions

In CLEM, you can perform the following operations with strings:

- Compare strings
- Create strings
- Access characters

In CLEM, a string is any sequence of characters between matching double quotation marks ("string quotes"). Characters (CHAR) can be any single alphanumeric character. They are declared in CLEM expressions using single backquotes in the form of `*<character>*`, such as `z`, `A`, or `2`. Characters that are out-of-bounds or negative indices to a string will result in undefined behavior.

Note. Comparisons between strings that do and do not use SQL pushback may generate different results where trailing spaces exist.

Function	**Result**	**Description**
allbutfirst(N, STRING)	*String*	Returns a string, which is *STRING* with the first *N* characters removed.
allbutlast(N, STRING)	*String*	Returns a string, which is *STRING* with the last characters removed.
alphabefore(STRING1, STRING2)	*Boolean*	Used to check the alphabetical ordering of strings. Returns true if *STRING1* precedes *STRING2*.
endstring(LENGTH, STRING)	*String*	Extracts the last *N* characters from the specified string. If the string length is less than or equal to the specified length, then it is unchanged.
hasendstring(STRING, SUBSTRING)	*Integer*	This function is the same as isendstring(SUBSTRING, STRING).
hasmidstring(STRING, SUBSTRING)	*Integer*	This function is the same as ismidstring(SUBSTRING, STRING) (embedded substring).
hasstartstring(STRING, SUBSTRING)	*Integer*	This function is the same as isstartstring(SUBSTRING, STRING).
hassubstring(STRING, N, SUBSTRING)	*Integer*	This function is the same as issubstring(SUBSTRING, N, STRING), where *N* defaults to 1.
count_substring(STRING, SUBSTRING)	*Integer*	Returns the number of times the specified substring occurs within the string. For example, count_substring("foooo.txt", "oo") returns 3.
hassubstring(STRING, SUBSTRING)	*Integer*	This function is the same as issubstring(SUBSTRING, 1, STRING), where *N* defaults to 1.
isalphacode(CHAR)	*Boolean*	Returns a value of true if *CHAR* is a character in the specified string (often a field name) whose character code is a letter. Otherwise, this function returns a value of 0. For example, isalphacode(produce_num(1)).
isendstring(SUBSTRING, STRING)	*Integer*	If the string *STRING* ends with the substring *SUBSTRING*, then this function returns the integer subscript of *SUBSTRING* in *STRING*. Otherwise, this function returns a value of 0.
islowercode(CHAR)	*Boolean*	Returns a value of true if *CHAR* is a lowercase letter character for the specified string (often a field name). Otherwise, this function returns a value of 0. For example, both islowercode(``) and islowercode(country_name(2)) are valid expressions.

Function	Result	Description
ismidstring(SUBSTRING, STRING)	*Integer*	If *SUBSTRING* is a substring of *STRING* but does not start on the first character of *STRING* or end on the last, then this function returns the subscript at which the substring starts. Otherwise, this function returns a value of 0.
isnumbercode(CHAR)	*Boolean*	Returns a value of true if *CHAR* for the specified string (often a field name) is a character whose character code is a digit. Otherwise, this function returns a value of 0. For example, isnumbercode(product_id(2)).
isstartstring(SUBSTRING, STRING)	*Integer*	If the string *STRING* starts with the substring *SUBSTRING*, then this function returns the subscript 1. Otherwise, this function returns a value of 0.
issubstring(SUBSTRING, N, STRING)	*Integer*	Searches the string *STRING*, starting from its *N*th character, for a substring equal to the string *SUBSTRING*. If found, this function returns the integer subscript at which the matching substring begins. Otherwise, this function returns a value of 0. If *N* is not given, this function defaults to 1.
issubstring(SUBSTRING, STRING)	*Integer*	Searches the string *STRING*, starting from its *N*th character, for a substring equal to the string *SUBSTRING*. If found, this function returns the integer subscript at which the matching substring begins. Otherwise, this function returns a value of 0. If *N* is not given, this function defaults to 1.
issubstring_count(SUBSTRING, N, STRING):	*Integer*	Returns the index of the *N*th occurrence of *SUBSTRING* within the specified *STRING*. If there are fewer than *N* occurrences of *SUBSTRING*, 0 is returned.
issubstring_lim(SUBSTRING, N, STARTLIM, ENDLIM, STRING)	*Integer*	This function is the same as issubstring, but the match is constrained to start on or before the subscript *STARTLIM* and to end on or before the subscript *ENDLIM*. The *STARTLIM* or *ENDLIM* constraints may be disabled by supplying a value of false for either argument—for example, issubstring_lim(SUBSTRING, N, false, false, STRING) is the same as issubstring.
isuppercode(CHAR)	*Boolean*	Returns a value of true if *CHAR* is an uppercase letter character. Otherwise, this function returns a value of 0. For example, both isuppercode(``) and isuppercode(country_name(2)) are valid expressions.
last(CHAR)	*String*	Returns the last character *CHAR* of *STRING* (which must be at least one character long).
length(STRING)	*Integer*	Returns the length of the string *STRING*—that is, the number of characters in it.

Function	Result	Description
locchar(CHAR, N, STRING)	*Integer*	Used to identify the location of characters in symbolic fields. The function searches the string *STRING* for the character *CHAR*, starting the search at the *N*th character of *STRING*. This function returns a value indicating the location (starting at *N*) where the character is found. If the character is not found, this function returns a value of 0. If the function has an invalid offset *(N)* (for example, an offset that is beyond the length of the string), this function returns $null$. For example, locchar(\`n\`, 2, web_page) searches the field called *web_page* for the \`n\` character beginning at the second character in the field value. *Note*: Be sure to use single backquotes to encapsulate the specified character.
locchar_back(CHAR, N, STRING)	*Integer*	Similar to locchar, except that the search is performed backward starting from the *N*th character. For example, locchar_back(\`n\`, 9, web_page) searches the field *web_page* starting from the ninth character and moving backward toward the start of the string. If the function has an invalid offset (for example, an offset that is beyond the length of the string), this function returns $null$. Ideally, you should use locchar_back in conjunction with the function length(<field>) to dynamically use the length of the current value of the field. For example, locchar_back(\`n\`, (length(web_page)), web_page).
lowertoupper(CHAR) lowertoupper (STRING)	*CHAR* or *String*	Input can be either a string or character, which is used in this function to return a new item of the same type, with any lowercase characters converted to their uppercase equivalents. For example, lowertoupper(\`a\`), lowertoupper("My string"), and lowertoupper(field_name(2)) are all valid expressions.
matches	*Boolean*	Returns true if a string matches a specified pattern. The pattern must be a string literal; it cannot be a field name containing a pattern. A question mark (?) can be included in the pattern to match exactly one character; an asterisk (*) matches zero or more characters. To match a literal question mark or asterisk (rather than using these as wildcards), a backslash (\) can be used as an escape character.
replace(SUBSTRING, NEWSUBSTRING, STRING)	*String*	Within the specified *STRING*, replace all instances of *SUBSTRING* with *NEWSUBSTRING*.
replicate(COUNT, STRING)	*String*	Returns a string that consists of the original string copied the specified number of times.

Function	Result	Description
stripchar(CHAR,STRING)	*String*	Enables you to remove specified characters from a string or field. You can use this function, for example, to remove extra symbols, such as currency notations, from data to achieve a simple number or name. For example, using the syntax stripchar(\`$\`, 'Cost') returns a new field with the dollar sign removed from all values. *Note*: Be sure to use single backquotes to encapsulate the specified character.
skipchar(CHAR, N, STRING)	*Integer*	Searches the string *STRING* for any character other than *CHAR*, starting at the *N*th character. This function returns an integer substring indicating the point at which one is found or 0 if every character from the *N*th onward is a *CHAR*. If the function has an invalid offset (for example, an offset that is beyond the length of the string), this function returns $null$. locchar is often used in conjunction with the skipchar functions to determine the value of *N* (the point at which to start searching the string). For example, skipchar(\`s\`, (locchar(\`s\`, 1, "MyString")), "MyString").
skipchar_back(CHAR, N, STRING)	*Integer*	Similar to skipchar, except that the search is performed **backward**, starting from the *N*th character.
startstring(LENGTH, STRING)	*String*	Extracts the first *N* characters from the specified string. If the string length is less than or equal to the specified length, then it is unchanged.
strmember(CHAR, STRING)	*Integer*	Equivalent to locchar(CHAR, 1, STRING). It returns an integer substring indicating the point at which *CHAR* first occurs, or 0. If the function has an invalid offset (for example, an offset that is beyond the length of the string), this function returns $null$.
subscrs(N, STRING)	*CHAR*	Returns the *N*th character *CHAR* of the input string *STRING*. This function can also be written in a shorthand form as STRING(N). For example, lowertoupper("name"(1)) is a valid expression.
substring(N, LEN, STRING)	*String*	Returns a string *SUBSTRING*, which consists of the *LEN* characters of the string *STRING*, starting from the character at subscript *N*.
substring_between(N1, N2, STRING)	*String*	Returns the substring of *STRING*, which begins at subscript *N1* and ends at subscript *N2*.
trim(STRING)	*String*	Removes leading and trailing white space characters from the specified string.
trim_start(STRING)	*String*	Removes leading white space characters from the specified string.
trimend(STRING)	*String*	Removes trailing white space characters from the specified string.

Function	Result	Description
unicode_char(NUM)	*CHAR*	Returns the character with Unicode value *NUM*.
unicode_value(CHAR)	*NUM*	Returns the Unicode value of *CHAR*
uppertolower(CHAR) uppertolower (STRING)	*CHAR* or *String*	Input can be either a string or character and is used in this function to return a new item of the same type with any uppercase characters converted to their lowercase equivalents. *Note*: Remember to specify strings with double quotes and characters with single backquotes. Simple field names should be specified without quotes.

SoundEx Functions

SoundEx is a method used to find strings when the sound is known but the precise spelling is not. Developed in 1918, it searches out words with similar sounds based on phonetic assumptions about how certain letters are pronounced. It can be used to search names in a database, for example, where spellings and pronunciations for similar names may vary. The basic SoundEx algorithm is documented in a number of sources and, despite known limitations (for example, leading letter combinations such as ph and f will not match even though they sound the same), is supported in some form by most databases.

Function	Result	Description
soundex(STRING)	*Integer*	Returns the four-character SoundEx code for the specified *STRING*.
soundex_difference(STRING1, STRING2)	*Integer*	Returns an integer between 0 and 4 that indicates the number of characters that are the same in the SoundEx encoding for the two strings, where 0 indicates no similarity and 4 indicates strong similarity or identical strings.

Date and Time Functions

CLEM includes a family of functions for handling fields with datetime storage of string variables representing dates and times. The formats of date and time used are specific to each stream and are specified in the stream properties dialog box. The date and time functions parse date and time strings according to the currently selected format.

When you specify a year in a date that uses only two digits (that is, the century is not specified), IBM® SPSS® Modeler uses the default century that is specified in the stream properties dialog box.

Note: Date and time functions cannot be called from scripts. For more information, see the topic CLEM Expressions in Scripts in Chapter 3 on p. 27.

Function	Result	Description
@TODAY	*String*	If you select Rollover days/mins in the stream properties dialog box, this function returns the current date as a string in the current date format. If you use a two-digit date format and do not select Rollover days/mins, this function returns $null$ on the current server. Note that this function cannot be called from a script. For more information, see the topic CLEM Expressions in Scripts in Chapter 3 on p. 27.
to_time(ITEM)	*Time*	Converts the storage of the specified field to a time.
to_date(ITEM)	*Date*	Converts the storage of the specified field to a date.
to_timestamp(ITEM)	*Timestamp*	Converts the storage of the specified field to a timestamp.
to_datetime(ITEM)	*Datetime*	Converts the storage of the specified field to a date, time, or timestamp value.
datetime_date(ITEM)	*Date*	Returns the date value for a *number*, *string*, or *timestamp*. Note this is the only function that allows you to convert a number (in seconds) back to a date. If ITEM is a string, creates a date by parsing a string in the current date format. The date format specified in the stream properties dialog box must be correct for this function to be successful. If ITEM is a number, it is interpreted as a number of seconds since the base date (or epoch). Fractions of a day are truncated. If ITEM is timestamp, the date part of the timestamp is returned. If ITEM is a date, it is returned unchanged.
date_before(DATE1, DATE2)	*Boolean*	Returns a value of true if *DATE1* represents a date or timestamp before that represented by *DATE2*. Otherwise, this function returns a value of 0.
date_days_difference(DATE1, DATE2)	*Integer*	Returns the time in days from the date or timestamp represented by *DATE1* to that represented by *DATE2*, as an integer. If *DATE2* is before *DATE1*, this function returns a negative number.
date_in_days(DATE)	*Integer*	Returns the time in days from the baseline date to the date or timestamp represented by *DATE*, as an integer. If *DATE* is before the baseline date, this function returns a negative number. You must include a valid date for the calculation to work appropriately. For example, you should not specify 29 February 2001 as the date. Because 2001 is a not a leap year, this date does not exist.
date_in_months(DATE)	*Real*	Returns the time in months from the baseline date to the date or timestamp represented by *DATE*, as a real number. This is an approximate figure based on a month of 30.4375 days. If *DATE* is before the baseline date, this function returns a negative number. You must include a valid date for the calculation to work appropriately. For example, you should not specify 29 February 2001 as the date. Because 2001 is a not a leap year, this date does not exist.

Function	Result	Description
date_in_weeks(DATE)	*Real*	Returns the time in weeks from the baseline date to the date or timestamp represented by *DATE*, as a real number. This is based on a week of 7.0 days. If *DATE* is before the baseline date, this function returns a negative number. You must include a valid date for the calculation to work appropriately. For example, you should not specify 29 February 2001 as the date. Because 2001 is a not a leap year, this date does not exist.
date_in_years(DATE)	*Real*	Returns the time in years from the baseline date to the date or timestamp represented by *DATE*, as a real number. This is an approximate figure based on a year of 365.25 days. If *DATE* is before the baseline date, this function returns a negative number. You must include a valid date for the calculation to work appropriately. For example, you should not specify 29 February 2001 as the date. Because 2001 is a not a leap year, this date does not exist.
date_months_difference (DATE1, DATE2)	*Real*	Returns the time in months from the date or timestamp represented by *DATE1* to that represented by *DATE2*, as a real number. This is an approximate figure based on a month of 30.4375 days. If *DATE2* is before *DATE1*, this function returns a negative number.
datetime_date(YEAR, MONTH, DAY)	*Date*	Creates a date value for the given *YEAR*, *MONTH*, and *DAY*. The arguments must be integers.
datetime_day(DATE)	*Integer*	Returns the day of the month from a given *DATE* or timestamp. The result is an integer in the range 1 to 31.
datetime_day_name(DAY)	*String*	Returns the full name of the given *DAY*. The argument must be an integer in the range 1 (Sunday) to 7 (Saturday).
datetime_hour(TIME)	*Integer*	Returns the hour from a *TIME* or timestamp. The result is an integer in the range 0 to 23.
datetime_in_seconds(TIME)	*Real*	Returns the seconds portion stored in *TIME*.
datetime_in_seconds(DATE), datetime_in_seconds(DATE-TIME)	*Real*	Returns the accumulated number, converted into seconds, from the difference between the current *DATE* or *DATETIME* and the baseline date (1900-01-01).
datetime_minute(TIME)	*Integer*	Returns the minute from a *TIME* or timestamp. The result is an integer in the range 0 to 59.
datetime_month(DATE)	*Integer*	Returns the month from a *DATE* or timestamp. The result is an integer in the range 1 to 12.
datetime_month_name (MONTH)	*String*	Returns the full name of the given *MONTH*. The argument must be an integer in the range 1 to 12.
datetime_now	*Timestamp*	Returns the current time as a timestamp.
datetime_second(TIME)	*Integer*	Returns the second from a *TIME* or timestamp. The result is an integer in the range 0 to 59.
datetime_day_short_ name(DAY)	*String*	Returns the abbreviated name of the given *DAY*. The argument must be an integer in the range 1 (Sunday) to 7 (Saturday).
datetime_month_short_ name(MONTH)	*String*	Returns the abbreviated name of the given *MONTH*. The argument must be an integer in the range 1 to 12.
datetime_time(HOUR, MINUTE, SECOND)	*Time*	Returns the time value for the specified *HOUR*, *MINUTE*, and *SECOND*. The arguments must be integers.

Function	Result	Description
datetime_time(ITEM)	*Time*	Returns the time value of the given *ITEM*.
datetime_timestamp(YEAR, MONTH, DAY, HOUR, MINUTE, SECOND)	*Timestamp*	Returns the timestamp value for the given *YEAR*, *MONTH*, *DAY*, *HOUR*, *MINUTE*, and *SECOND*.
datetime_timestamp(DATE, TIME)	*Timestamp*	Returns the timestamp value for the given *DATE* and *TIME*.
datetime_timestamp (NUMBER)	*Timestamp*	Returns the timestamp value of the given number of seconds.
datetime_weekday(DATE)	*Integer*	Returns the day of the week from the given *DATE* or timestamp.
datetime_year(DATE)	*Integer*	Returns the year from a *DATE* or timestamp. The result is an integer such as 2002.
date_weeks_difference (DATE1, DATE2)	*Real*	Returns the time in weeks from the date or timestamp represented by *DATE1* to that represented by *DATE2*, as a real number. This is based on a week of 7.0 days. If *DATE2* is before *DATE1*, this function returns a negative number.
date_years_difference (DATE1, DATE2)	*Real*	Returns the time in years from the date or timestamp represented by *DATE1* to that represented by *DATE2*, as a real number. This is an approximate figure based on a year of 365.25 days. If *DATE2* is before *DATE1*, this function returns a negative number.
time_before(TIME1, TIME2)	*Boolean*	Returns a value of true if *TIME1* represents a time or timestamp before that represented by *TIME2*. Otherwise, this function returns a value of 0.
time_hours_difference (TIME1, TIME2)	*Real*	Returns the time difference in hours between the times or timestamps represented by *TIME1* and *TIME2*, as a real number. If you select Rollover days/mins in the stream properties dialog box, a higher value of *TIME1* is taken to refer to the previous day. If you do not select the rollover option, a higher value of *TIME1* causes the returned value to be negative.
time_in_hours(TIME)	*Real*	Returns the time in hours represented by *TIME*, as a real number. For example, under time format HHMM, the expression time_in_hours('0130') evaluates to 1.5. *TIME* can represent a time or a timestamp.
time_in_mins(TIME)	*Real*	Returns the time in minutes represented by *TIME*, as a real number. *TIME* can represent a time or a timestamp.
time_in_secs(TIME)	*Integer*	Returns the time in seconds represented by *TIME*, as an integer. *TIME* can represent a time or a timestamp.

Function	Result	Description
time_mins_difference(TIME1, TIME2)	*Real*	Returns the time difference in minutes between the times or timestamps represented by *TIME1* and *TIME2*, as a real number. If you select Rollover days/mins in the stream properties dialog box, a higher value of *TIME1* is taken to refer to the previous day (or the previous hour, if only minutes and seconds are specified in the current format). If you do not select the rollover option, a higher value of *TIME1* will cause the returned value to be negative.
time_secs_difference(TIME1, TIME2)	*Integer*	Returns the time difference in seconds between the times or timestamps represented by *TIME1* and *TIME2*, as an integer. If you select Rollover days/mins in the stream properties dialog box, a higher value of *TIME1* is taken to refer to the previous day (or the previous hour, if only minutes and seconds are specified in the current format). If you do not select the rollover option, a higher value of *TIME1* causes the returned value to be negative.

Converting Date and Time Values

Note that conversion functions (and any other functions that require a specific type of input, such as a date or time value) depend on the current formats specified in the Stream Options dialog box. For example, if you have a field named *DATE* that is stored as a string with values *Jan 2003*, *Feb 2003*, and so on, you could convert it to date storage as follows:

```
to_date(DATE)
```

For this conversion to work, select the matching date format MON YYYY as the default date format for the stream.

For an example that converts string values to dates using a Filler node, see the stream *broadband_create_models.str*, installed in the *Demos* folder under the *streams* subfolder.

Dates stored as numbers. Note that *DATE* in the previous example is the name of a field, while to_date is a CLEM function. If you have dates stored as numbers, you can convert them using the datetime_date function, where the number is interpreted as a number of seconds since the base date (or epoch).

```
datetime_date(DATE)
```

By converting a date to a number of seconds (and back), you can perform calculations such as computing the current date plus or minus a fixed number of days, for example:

```
datetime_date((date_in_days(DATE)-7)*60*60*24)
```

Sequence Functions

For some operations, the sequence of events is important. The application allows you to work with the following record sequences:

- Sequences and time series

- Sequence functions
- Record indexing
- Averaging, summing, and comparing values
- Monitoring change—differentiation
- @SINCE
- Offset values
- Additional sequence facilities

For many applications, each record passing through a stream can be considered as an individual case, independent of all others. In such situations, the order of records is usually unimportant.

For some classes of problems, however, the record sequence is very important. These are typically time series situations, in which the sequence of records represents an ordered sequence of events or occurrences. Each record represents a snapshot at a particular instant in time; much of the richest information, however, might be contained not in instantaneous values but in the way in which such values are changing and behaving over time.

Of course, the relevant parameter may be something other than time. For example, the records could represent analyses performed at distances along a line, but the same principles would apply.

Sequence and special functions are immediately recognizable by the following characteristics:

- They are all prefixed by @.
- Their names are given in upper case.

Sequence functions can refer to the record currently being processed by a node, the records that have already passed through a node, and even, in one case, records that have yet to pass through a node. Sequence functions can be mixed freely with other components of CLEM expressions, although some have restrictions on what can be used as their arguments.

Examples

You may find it useful to know how long it has been since a certain event occurred or a condition was true. Use the function @SINCE to do this—for example:

```
@SINCE(Income > Outgoings)
```

This function returns the offset of the last record where this condition was true—that is, the number of records before this one in which the condition was true. If the condition has never been true, @SINCE returns @INDEX + 1.

Sometimes you may want to refer to a value of the current record in the expression used by @SINCE. You can do this using the function @THIS, which specifies that a field name always applies to the current record. To find the offset of the last record that had a Concentration field value more than twice that of the current record, you could use:

```
@SINCE(Concentration > 2 * @THIS(Concentration))
```

In some cases the condition given to @SINCE is true of the current record by definition—for example:

```
@SINCE(ID == @THIS(ID))
```

For this reason, @SINCE does not evaluate its condition for the current record. Use a similar function, @SINCE0, if you want to evaluate the condition for the current record as well as previous ones; if the condition is true in the current record, @SINCE0 returns 0.

Note: @ functions cannot be called from scripts. For more information, see the topic CLEM Expressions in Scripts in Chapter 3 on p. 27.

Function	Result	Description
MEAN(FIELD)	*Real*	Returns the mean average of values for the specified *FIELD* or *FIELDS*.
@MEAN(FIELD, EXPR)	*Real*	Returns the mean average of values for *FIELD* over the last *EXPR* records received by the current node, including the current record. *FIELD* must be the name of a numeric field. *EXPR* may be any expression evaluating to an integer greater than 0. If *EXPR* is omitted or if it exceeds the number of records received so far, the average over all of the records received so far is returned. Note that this function cannot be called from a script. For more information, see the topic CLEM Expressions in Scripts in Chapter 3 on p. 27.
@MEAN(FIELD, EXPR, INT)	*Real*	Returns the mean average of values for *FIELD* over the last *EXPR* records received by the current node, including the current record. *FIELD* must be the name of a numeric field. *EXPR* may be any expression evaluating to an integer greater than 0. If *EXPR* is omitted or if it exceeds the number of records received so far, the average over all of the records received so far is returned. *INT* specifies the maximum number of values to look back. This is far more efficient than using just two arguments.
@DIFF1(FIELD)	*Real*	Returns the first differential of *FIELD1*. The single-argument form thus simply returns the difference between the current value and the previous value of the field. Returns 0 if the relevant previous records do not exist.
@DIFF1(FIELD1, FIELD2)	*Real*	The two-argument form gives the first differential of *FIELD1* with respect to *FIELD2*. Returns 0 if the relevant previous records do not exist.
@DIFF2(FIELD)	*Real*	Returns the second differential of *FIELD1*. The single-argument form thus simply returns the difference between the current value and the previous value of the field. Returns 0 if the relevant previous records do not exist
@DIFF2(FIELD1, FIELD2)	*Real*	The two-argument form gives the first differential of *FIELD1* with respect to *FIELD2*. Returns 0 if the relevant previous records do not exist.
@INDEX	*Integer*	Returns the index of the current record. Indices are allocated to records as they arrive at the current node. The first record is given index 1, and the index is incremented by 1 for each subsequent record.

Function	Result	Description
@LAST_NON_BLANK(FIELD)	*Any*	Returns the last value for *FIELD* that was not blank, as defined in an upstream source or Type node. If there are no nonblank values for *FIELD* in the records read so far, $null$ is returned. Note that blank values, also called user-missing values, can be defined separately for each field.
@MAX(FIELD)	*Number*	Returns the maximum value for the specified *FIELD*.
@MAX(FIELD, EXPR)	*Number*	Returns the maximum value for *FIELD* over the last *EXPR* records received so far, including the current record. *FIELD* must be the name of a numeric field. *EXPR* may be any expression evaluating to an integer greater than 0.
@MAX(FIELD, EXPR, INT)	*Number*	Returns the maximum value for *FIELD* over the last *EXPR* records received so far, including the current record. *FIELD* must be the name of a numeric field. *EXPR* may be any expression evaluating to an integer greater than 0. If *EXPR* is omitted, or if it exceeds the number of records received so far, the maximum value over all of the records received so far is returned. *INT* specifies the maximum number of values to look back. This is far more efficient than using just two arguments.
@MIN(FIELD)	*Number*	Returns the minimum value for the specified *FIELD*.
@MIN(FIELD, EXPR)	*Number*	Returns the minimum value for *FIELD* over the last *EXPR* records received so far, including the current record. *FIELD* must be the name of a numeric field. *EXPR* may be any expression evaluating to an integer greater than 0.
@MIN(FIELD, EXPR, INT)	*Number*	Returns the minimum value for *FIELD* over the last *EXPR* records received so far, including the current record. *FIELD* must be the name of a numeric field. *EXPR* may be any expression evaluating to an integer greater than 0. If *EXPR* is omitted, or if it exceeds the number of records received so far, the minimum value over all of the records received so far is returned. *INT* specifies the maximum number of values to look back. This is far more efficient than using just two arguments.
@OFFSET(FIELD, EXPR)	*Any*	Returns the value of *FIELD* in the record offset from the current record by the value of *EXPR*. A positive offset refers to a record that has already passed, while a negative one specifies a "lookahead" to a record that has yet to arrive. For example, @OFFSET(Status, 1) returns the value of the Status field in the previous record, while @OFFSET(Status, –4) "looks ahead" four records in the sequence (that is, to records that have not yet passed through this node) to obtain the value. *Note that a negative (look ahead) offset must be specified as a constant.* For positive offsets only, *EXPR* may also be an arbitrary CLEM expression, which is evaluated for the current record to give the offset. In this case, the three-argument version of this function should improve performance (see next function). If the expression returns anything other than a non-negative integer, this causes an error—that is, it is not legal to have calculated lookahead offsets.

Function	Result	Description
		Note: A self-referential @OFFSET function cannot use literal lookahead. For example, in a Filler node, you cannot replace the value of field1 using an expression such as @OFFSET(field1,-2).
@OFFSET(FIELD, EXPR, INT)	*Any*	Performs the same operation as the @OFFSET function with the addition of a third argument, *INT*, which specifies the maximum number of values to look back. In cases where the offset is computed from an expression, this third argument should improve performance. For example, in an expression such as@OFFSET(Foo, Month, 12), the system knows to keep only the last twelve values of Foo; otherwise, it has to store every value just in case. In cases where the offset value is a constant—including negative "lookahead" offsets, which must be constant—the third argument is pointless and the two-argument version of this function should be used. See also the note about self-referential functions in the two-argument version described earlier.
@SDEV(FIELD)	*Real*	Returns the standard deviation of values for the specified *FIELD* or *FIELDS*.
@SDEV(FIELD, EXPR)	*Real*	Returns the standard deviation of values for *FIELD* over the last *EXPR* records received by the current node, including the current record. *FIELD* must be the name of a numeric field. *EXPR* may be any expression evaluating to an integer greater than 0. If *EXPR* is omitted, or if it exceeds the number of records received so far, the standard deviation over all of the records received so far is returned.
@SDEV(FIELD, EXPR, INT)	*Real*	Returns the standard deviation of values for *FIELD* over the last *EXPR* records received by the current node, including the current record. *FIELD* must be the name of a numeric field. *EXPR* may be any expression evaluating to an integer greater than 0. If *EXPR* is omitted, or if it exceeds the number of records received so far, the standard deviation over all of the records received so far is returned. *INT* specifies the maximum number of values to look back. This is far more efficient than using just two arguments.
@SINCE(EXPR)	*Any*	Returns the number of records that have passed since *EXPR*, an arbitrary CLEM expression, was true.
@SINCE(EXPR, INT)	*Any*	Adding the second argument, *INT*, specifies the maximum number of records to look back. If *EXPR* has never been true, *INT* is @INDEX+1.
@SINCE0(EXPR)	*Any*	Considers the current record, while @SINCE does not; @SINCE0 returns 0 if *EXPR* is true for the current record.
@SINCE0(EXPR, INT)	*Any*	Adding the second argument, *INT* specifies the maximum number of records to look back.
@SUM(FIELD)	*Number*	Returns the sum of values for the specified *FIELD* or *FIELDS*.

Function	Result	Description
@SUM(FIELD, EXPR)	*Number*	Returns the sum of values for *FIELD* over the last *EXPR* records received by the current node, including the current record. *FIELD* must be the name of a numeric field. *EXPR* may be any expression evaluating to an integer greater than 0. If *EXPR* is omitted, or if it exceeds the number of records received so far, the sum over all of the records received so far is returned.
@SUM(FIELD, EXPR, INT)	*Number*	Returns the sum of values for *FIELD* over the last *EXPR* records received by the current node, including the current record. *FIELD* must be the name of a numeric field. *EXPR* may be any expression evaluating to an integer greater than 0. If *EXPR* is omitted, or if it exceeds the number of records received so far, the sum over all of the records received so far is returned. *INT* specifies the maximum number of values to look back. This is far more efficient than using just two arguments.
@THIS(FIELD)	*Any*	Returns the value of the field named *FIELD* in the current record. Used only in @SINCE expressions.

Global Functions

The functions @MEAN,@SUM, @MIN, @MAX, and @SDEV work on, at most, all of the records read up to and including the current one. In some cases, however, it is useful to be able to work out how values in the current record compare with values seen in the entire data set. Using a Set Globals node to generate values across the entire data set, you can access these values in a CLEM expression using the global functions.

For example,

```
@GLOBAL_MAX(Age)
```

returns the highest value of Age in the data set, while the expression

```
(Value - @GLOBAL_MEAN(Value)) / @GLOBAL_SDEV(Value)
```

expresses the difference between this record's Value and the global mean as a number of standard deviations. You can use global values only after they have been calculated by a Set Globals node. All current global values can be canceled by clicking the Clear Global Values button on the Globals tab in the stream properties dialog box.

Note: @ functions cannot be called from scripts. For more information, see the topic CLEM Expressions in Scripts in Chapter 3 on p. 27.

Function	Result	Description
@GLOBAL_MAX(FIELD)	*Number*	Returns the maximum value for *FIELD* over the whole data set, as previously generated by a Set Globals node. *FIELD* must be the name of a numeric field. If the corresponding global value has not been set, an error occurs. Note that this function cannot be called from a script. For more information, see the topic CLEM Expressions in Scripts in Chapter 3 on p. 27.

Function	Result	Description
@GLOBAL_MIN(FIELD)	*Number*	Returns the minimum value for *FIELD* over the whole data set, as previously generated by a Set Globals node. *FIELD* must be the name of a numeric field. If the corresponding global value has not been set, an error occurs.
@GLOBAL_SDEV(FIELD)	*Number*	Returns the standard deviation of values for *FIELD* over the whole data set, as previously generated by a Set Globals node. *FIELD* must be the name of a numeric field. If the corresponding global value has not been set, an error occurs.
@GLOBAL_MEAN(FIELD)	*Number*	Returns the mean average of values for *FIELD* over the whole data set, as previously generated by a Set Globals node. *FIELD* must be the name of a numeric field. If the corresponding global value has not been set, an error occurs.
@GLOBAL_SUM(FIELD)	*Number*	Returns the sum of values for *FIELD* over the whole data set, as previously generated by a Set Globals node. *FIELD* must be the name of a numeric field. If the corresponding global value has not been set, an error occurs.

Functions Handling Blanks and Null Values

Using CLEM, you can specify that certain values in a field are to be regarded as "blanks," or missing values. The following functions work with blanks.

Note: @ functions cannot be called from scripts. For more information, see the topic CLEM Expressions in Scripts in Chapter 3 on p. 27.

Function	Result	Description
@BLANK(FIELD)	*Boolean*	Returns true for all records whose values are blank according to the blank-handling rules set in an upstream Type node or source node (Types tab). Note that this function cannot be called from a script. For more information, see the topic CLEM Expressions in Scripts in Chapter 3 on p. 27.
@LAST_NON_BLANK(FIELD)	*Any*	Returns the last value for *FIELD* that was not blank, as defined in an upstream source or Type node. If there are no nonblank values for *FIELD* in the records read so far, $null$ is returned. Note that blank values, also called user-missing values, can be defined separately for each field.
@NULL(FIELD)	*Boolean*	Returns true if the value of *FIELD* is the system-missing $null$. Returns false for all other values, including user-defined blanks. If you want to check for both, use @BLANK(FIELD)and @NULL(FIELD).
undef	*Any*	Used generally in CLEM to enter a $null$ value—for example, to fill blank values with nulls in the Filler node.

Blank fields may be "filled in" with the Filler node. In both Filler and Derive nodes (multiple mode only), the special CLEM function @FIELD refers to the current field(s) being examined.

Special Fields

Special functions are used to denote the specific fields under examination, or to generate a list of fields as input. For example, when deriving multiple fields at once, you should use @FIELD to denote "perform this derive action on the selected fields." Using the expression log(@FIELD) derives a new log field for each selected field.

Note: @ functions cannot be called from scripts. For more information, see the topic CLEM Expressions in Scripts in Chapter 3 on p. 27.

Function	Result	Description
@FIELD	*Any*	Performs an action on all fields specified in the expression context. Note that this function cannot be called from a script. For more information, see the topic CLEM Expressions in Scripts in Chapter 3 on p. 27.
@TARGET	*Any*	When a CLEM expression is used in a user-defined analysis function, @TARGET represents the target field or "correct value" for the target/predicted pair being analyzed. This function is commonly used in an Analysis node.
@PREDICTED	*Any*	When a CLEM expression is used in a user-defined analysis function,@PREDICTED represents the predicted value for the target/predicted pair being analyzed. This function is commonly used in an Analysis node.
@PARTITION_FIELD	*Any*	Substitutes the name of the current partition field.
@TRAINING_PARTITION	*Any*	Returns the value of the current training partition. For example, to select training records using a Select node, use the CLEM expression: @PARTITION_FIELD = @TRAINING_PARTITION This ensures that the Select node will always work regardless of which values are used to represent each partition in the data.
@TESTING_PARTITION	*Any*	Returns the value of the current testing partition.
@VALIDATION_PARTITION	*Any*	Returns the value of the current validation partition.
@FIELDS_BETWEEN(start, end)	*Any*	Returns the list of field names between the specified start and end fields (inclusive) based on the natural (that is, insert) order of the fields in the data.
@FIELDS_MATCHING(pattern)	*Any*	Returns a list a field names matching a specified pattern. A question mark (?) can be included in the pattern to match exactly one character; an asterisk (*) matches zero or more characters. To match a literal question mark or asterisk (rather than using these as wildcards), a backslash (\) can be used as an escape character.
@MULTI_RESPONSE_SET	*Any*	Returns the list of fields in the named multiple response set.

Part II:
Properties Reference

Chapter
9
Properties Reference

Properties Reference Overview

You can specify a number of different properties for nodes, streams, SuperNodes, and projects. Some properties are common to all nodes, such as name, annotation, and ToolTip, while others are specific to certain types of nodes. Other properties refer to high-level stream operations, such as caching or SuperNode behavior. Properties can be accessed through the standard user interface (for example, when you open a dialog box to edit options for a node) and can also be used in a number of other ways.

- Properties can be modified through scripts, as described in this section. For more information, see Syntax for Properties below.
- Node properties can be used in SuperNode parameters.
- Node properties can also be used as part of a command line option (using the -P flag) when starting IBM® SPSS® Modeler.

In the context of scripting within SPSS Modeler, node and stream properties are often called **slot parameters**. In this guide, they are referred to as node or stream properties.

For more information on the scripting language, see Chapter 3.

Syntax for Properties

Properties must use the following syntax structure:

```
NAME:TYPE.PROPERTY
```

where NAME is the name of a node, and TYPE is its type (for example, multiplotnode or derivenode). You can omit either NAME or TYPE, but you must include at least one of them. PROPERTY is the name of the node or stream parameter that your expression refers to. For example, the following syntax is used to filter the *Age* field from downstream data:

```
set mynode:filternode.include.Age = false
```

To use a custom value for any of the parameters (NAME, TYPE, or PROPERTY), first set the value in a statement, such as set derive.new_name = mynewfield. From that point on, you can use the value, mynewfield, as the parameter by preceding it with the ^ symbol. For example, you can set the type for the Derive node named above by using the following syntax:

```
set ^mynewfield.result_type = "Conditional"
```

All nodes used in IBM® SPSS® Modeler can be specified in the TYPE parameter of the syntax NAME:TYPE.PROPERTY.

Structured Properties

There are two ways in which scripting uses structured properties for increased clarity when parsing:

- To give structure to the names of properties for complex nodes, such as Type, Filter, or Balance nodes.
- To provide a format for specifying multiple properties at once.

Structuring for Complex Interfaces

The scripts for nodes with tables and other complex interfaces (for example, the Type, Filter, and Balance nodes) must follow a particular structure in order to parse correctly. These structured properties need a name that is more complex than the name for a single identifier. For example, within a Filter node, each available field (on its upstream side) is switched on or off. In order to refer to this information, the Filter node stores one item of information per field (whether each field is true or false), and these multiple items are accessed and updated by a single property called **field**. This property may have (or be given) the value true or false. Suppose that a Filter node named mynode has (on its upstream side) a field called *Age*. To switch this to off, set the property mynode.include.Age to the value false, as follows:

```
set mynode.include.Age = false
```

Structuring to Set Multiple Properties

For many nodes, you can assign more than one node or stream property at a time. This is referred to as the **multiset command** or **set block**. For more information, see the topic set Command in Chapter 4 on p. 32.

In some cases, a structured property can be quite complex. The backslash (\) character can be used as a line continuation character to help you line up the arguments for clarity. An example is as follows:

```
mynode:sortnode.keys = [{ 'K' Descending} \
                  { 'Age' Ascending}\
                  { 'Na' Descending }]
```

Another advantage that structured properties have is their ability to set several properties on a node before the node is stable. By default, a multiset sets all properties in the block before taking any action based on an individual property setting. For example, when defining a Fixed File node, using two steps to set field properties would result in errors because the node is not consistent until both settings are valid. Defining properties as a multiset circumvents this problem by setting both properties before updating the data model.

Abbreviations

Standard abbreviations are used throughout the syntax for node properties. Learning the abbreviations is helpful in constructing scripts.

Abbreviation	**Meaning**
abs	Absolute value
len	Length
min	Minimum
max	Maximum
correl	Correlation
covar	Covariance
num	Number or numeric
pct	Percent or percentage
transp	Transparency
xval	Cross-validation
var	Variance or variable (in source nodes)

Node and Stream Property Examples

Node and stream properties can be used in a variety of ways with IBM® SPSS® Modeler. They are most commonly used as part of a script, either a **standalone script**, used to automate multiple streams or operations, or a **stream script**, used to automate processes within a single stream. You can also specify node parameters by using the node properties within the SuperNode. At the most basic level, properties can also be used as a command line option for starting SPSS Modeler. Using the `-p` argument as part of command line invocation, you can use a stream property to change a setting in the stream.

s.max_size	Refers to the property max_size of the node named s.
s:samplenode.max_size	Refers to the property max_size of the node named s, which must be a Sample node.
:samplenode.max_size	Refers to the property max_size of the Sample node in the current stream (there must be only one Sample node).
s:sample.max_size	Refers to the property max_size of the node named s, which must be a Sample node.
t.direction.Age	Refers to the role of the field *Age* in the Type node t.
:.max_size	*** NOT LEGAL *** You must specify either the node name or the node type.

The example `s:sample.max_size` illustrates that you do not need to spell out node types in full.

The example `t.direction.Age` illustrates that some slot names can themselves be structured—in cases where the attributes of a node are more complex than simply individual slots with individual values. Such slots are called **structured** or **complex** properties.

Node Properties Overview

Each type of node has its own set of legal properties, and each property has a type. This type may be a general type—number, flag, or string—in which case settings for the property are coerced to the correct type. An error is raised if they cannot be coerced. Alternatively, the property reference may specify the range of legal values, such as Discard, PairAndDiscard, and IncludeAsText, in which case an error is raised if any other value is used. Flag properties should be read or set by using values of true and false. (Variations including Off, OFF, off, No, NO, no, n, N, f, F, false, False, FALSE, or 0 are also recognized when setting values but may cause errors when reading property values in some cases. All other values are regarded as true. Using true and false consistently will avoid any confusion.) In this guide's reference tables, the structured properties are indicated as such in the *Property description* column, and their usage formats are given.

Common Node Properties

A number of properties are common to all nodes (including SuperNodes) in IBM® SPSS® Modeler.

Property name	Data type	Property description
use_custom_name	*flag*	
name	*string*	Read-only property that reads the name (either auto or custom) for a node on the canvas.
custom_name	*string*	Specifies a custom name for the node.
tooltip	*string*	
annotation	*string*	
keywords	*string*	Structured slot that specifies a list of keywords associated with the object (for example, ["Keyword1" "Keyword2"]).
cache_enabled	*flag*	
node_type	source_supernode process_supernode terminal_supernode all node names as specified for scripting	Read-only property used to refer to a node by type. For example, instead of referring to a node only by name, such as real_income, you can also specify the type, such as userinputnode or filternode.

SuperNode-specific properties are discussed separately, as with all other nodes. For more information, see the topic SuperNode Properties in Chapter 22 on p. 295.

Chapter

10

Stream Properties

A variety of stream properties can be controlled by scripting. To reference stream properties, you must use a special stream variable, denoted with a ^ preceding the stream:

```
set ^stream.execute_method = Script
```

Example

The nodes property is used to refer to the nodes in the current stream. The following stream script provides an example:

```
var listofnodes
var thenode
set listofnodes = ^stream.nodes

set ^stream.annotation = ^stream.annotation >< "\n\nThis stream is called \"" >< ^stream.name >
< "\" and contains/ the following nodes\n"

for thenode in listofnodes
set ^stream.annotation = ^stream.annotation >< "\n" >< ^thenode.node_type
endfor
```

The above example uses the nodes property to create a list of all nodes in the stream and write that list in the stream annotations. The annotation produced looks like this:

```
This stream is called "druglearn" and contains the following nodes

derivenode
neuralnetworknode
variablefilenode
typenode
c50node
filternode
```

Stream properties are described in the following table.

Property name	Data type	Property description
execute_method	Normal Script	

Property name	Data type	Property description
date_format	"DDMMYY" "MMDDYY" "YYMMDD" "YYYYMMDD" "YYYYDDD" DAY MONTH "DD-MM-YY" "DD-MM-YYYY" "MM-DD-YY" "MM-DD-YYYY" "DD-MON-YY" "DD-MON-YYYY" "YYYY-MM-DD" "DD.MM.YY" "DD.MM.YYYY" "MM.DD.YY" "MM.DD.YYYY" "DD.MON.YY" "DD.MON.YYYY" "DD/MM/YY" "DD/MM/YYYY" "MM/DD/YY" "MM/DD/YYYY" "DD/MON/YY" "DD/MON/YYYY" MON YYYY q Q YYYY ww WK YYYY	
date_baseline	*number*	
date_2digit_baseline	*number*	
time_format	"HHMMSS" "HHMM" "MMSS" "HH:MM:SS" "HH:MM" "MM:SS" "(H)H:(M)M:(S)S" "(H)H:(M)M" "(M)M:(S)S" "HH.MM.SS" "HH.MM" "MM.SS" "(H)H.(M)M.(S)S" "(H)H.(M)M" "(M)M.(S)S"	
time_rollover	*flag*	
import_datetime_as_string	*flag*	
decimal_places	*number*	
decimal_symbol	Default Period Comma	
angles_in_radians	*flag*	
use_max_set_size	*flag*	
max_set_size	*number*	

Property name	Data type	Property description
ruleset_evaluation	Voting FirstHit	
refresh_source_nodes	*flag*	Use to refresh source nodes automatically upon stream execution.
script	*string*	
annotation	*string*	Example: set ^stream.annotation = "something interesting"
name	*string*	Example: set x = ^stream.name *Note*: This property is read-only. If you want to change the name of a stream, you should save it with a different name.
parameters		Use this property to update stream parameters from within a stand-alone script. Example: set ^stream.parameters.height = 23
nodes		See detailed information below.
encoding	SystemDefault "UTF-8"	
stream_rewriting	*boolean*	
stream_rewriting_maximise_sql	*boolean*	
stream_rewriting_optimise_clem_execution	*boolean*	
stream_rewriting_optimise_syntax_execution	*boolean*	
enable_parallelism	*boolean*	
sql_generation	*boolean*	
database_caching	*boolean*	
sql_logging	*boolean*	
sql_generation_logging	*boolean*	
sql_log_native	*boolean*	
sql_log_prettyprint	*boolean*	
record_count_suppress_input	*boolean*	
record_count_feedback_interval	*integer*	

Chapter

11

Project Properties

A number of properties are available for scripting with projects.

Example

```
load project "C:/clemdata/DrugData.cpj"
set ^project.summary="Initial modeling work on the latest drug data."
set ^project.ordering=NameAddedType
execute_project
```

Property name	Data type	Property description
summary	*string*	The project summary—typically an abbreviated version of the annotation.
title	*string*	The report title.
author	*string*	The report author.
structure	Phase Class	Determines how the project is organized—by data mining phase or by object type (class).
include_mode	IncludedItems ExcludedItems AllItems	Determines which items to include in the project report.
select_mode	AllItems RecentItems OldItems	Determines (by age) which items to include in the report.
recent_item_limit	*integer*	Used when select_mode is RecentItems.
old_item_limit	*integer*	Used when select_mode is OldItems.
ordering	TypeNameAdded TypeAddedName NameAddedType AddedNameType	Determines the order in which items are listed in the report.

Chapter

12

Source Node Properties

Source Node Common Properties

Properties that are common to all source nodes are listed below, with information on specific nodes in the topics that follow.

Example

```
create variablefilenode
set :variablefilenode.full_filename = "$CLEO_DEMOS/DRUG4n"
set :variablefilenode.use_custom_values.Age = True
set :variablefilenode.direction.Age = Input
set :variablefilenode.type.Age = Range
#storage is read only
set :variablefilenode.check.Age = None
set :variablefilenode.values.Age = [1 100]
```

Property name	Data type	Property description
direction	Input Target Both None Partition Split Frequency RecordID	Keyed property for field roles. Usage format: NODE.direction.FIELDNAME *Note*: The values In and Out are now deprecated. Support for them may be withdrawn in a future release.
type	Range Flag Set Typeless Discrete Ordered Set Default	Type of field. Setting this property to *Default* will clear any values property setting, and if value_mode is set to *Specify*, it will be reset to *Read*. If value_mode is already set to *Pass* or *Read*, it will be unaffected by the type setting. Usage format: NODE.type.FIELDNAME
storage	Unknown String Integer Real Time Date Timestamp	Read-only keyed property for field storage type. Usage format: NODE.storage.FIELDNAME
check	None Nullify Coerce Discard Warn Abort	Keyed property for field type and range checking. Usage format: NODE.check.FIELDNAME

Property name	Data type	Property description
values	[*value value*]	For a continuous (range) field, the first value is the minimum, and the last value is the maximum. For nominal (set) fields, specify all values. For flag fields, the first value represents *false*, and the last value represents *true*. Setting this property automatically sets the value_mode property to *Specify*. Usage format: NODE.values.FIELDNAME
value_mode	Read Pass Read+ Current Specify	Determines how values are set for a field on the next data pass. Usage format: NODE.value_mode.FIELDNAME Note that you cannot set this property to *Specify* directly; to use specific values, set the values property.
default_value_mode	Read Pass	Specifies the default method for setting values for all fields. Usage format: NODE.default_value_mode Example: set mynode.default_value_mode = Pass This setting can be overridden for specific fields by using the value_mode property.
extend_values	*flag*	Applies when value_mode is set to *Read*. Set to *T* to add newly read values to any existing values for the field. Set to *F* to discard existing values in favor of the newly read values. Usage format: NODE.extend_values.FIELDNAME
value_labels	*string*	Used to specify a value label. Example: set :varfilenode.value_labels.Age = [{3 three}{5 five}] Note that values must be specified first.
enable_missing	*flag*	When set to *T*, activates tracking of missing values for the field. Usage format: NODE.enable_missing.FIELDNAME
missing_values	[*value value ...*]	Specifies data values that denote missing data. Usage format: NODE.missing_values.FIELDNAME
range_missing	*flag*	When this property is set to *T*, specifies whether a missing-value (blank) range is defined for a field. Usage format: NODE.range_missing.FIELDNAME
missing_lower	*string*	When range_missing is true, specifies the lower bound of the missing-value range. Usage format: NODE.missing_lower.FIELDNAME
missing_upper	*string*	When range_missing is true, specifies the upper bound of the missing-value range. Usage format: NODE.missing_upper.FIELDNAME

Property name	Data type	Property description
null_missing	*flag*	When this property is set to *T*, nulls (undefined values that are displayed as $null$ in the software) are considered missing values. Usage format: NODE.null_missing.FIELDNAME
whitespace_missing	*flag*	When this property is set to *T*, values containing only white space (spaces, tabs, and new lines) are considered missing values. Usage format: NODE.whitespace_missing.FIELDNAME
description	*string*	Used to specify a field label or description.
default_include	*flag*	Keyed property to specify whether the default behavior is to pass or filter fields: NODE.default_include Example: set mynode:filternode.default_include = false
include	*flag*	Keyed property used to determine whether individual fields are included or filtered: NODE.include.FIELDNAME. Example: set mynode:filternode.include.Age = true
new_name	*string*	Example: set mynode:filternode.new_name.'Age' = "years"

cognosimportnode Properties

The IBM Cognos BI source node imports data from Cognos BI databases.

Example

```
create cognosimportnode
set :cognosimportnode.cognos_connection = {'http://mycogsrv1:9300/p2pd/servlet/dispatch', true, "", "", ""}
set :cognosimportnode.cognos_package_name = '/Public Folders/GOSALES'
set :cognosimportnode.cognos_items = {"[GreatOutdoors].[BRANCH].[BRANCH_CODE]",
"[GreatOutdoors].[BRANCH].[COUNTRY_CODE]"}
```

cognosimportnode properties	Data type	Property description
mode	Data Report	Specifies whether to import Cognos BI data (default) or reports.

cognosimportnode properties	Data type	Property description
cognos_connection	*{"field","field", ... ,"field"}*	A list property containing the connection details for the Cognos server. The format is: {"*Cognos_server_URL*", *login_mode*, "*namespace*", "*username*", "*password*"} where: *Cognos_server_URL* is the URL of the Cognos server containing the source *login_mode* indicates whether anonymous login is used, and is either true or false; if set to true, the following fields should be set to "" *namespace* specifies the security authentication provider used to log on to the server *username* and *password* are those used to log on to the Cognos server
cognos_package_name	*string*	The path and name of the Cognos package from which you are importing data objects, for example: /Public Folders/GOSALES Note: Only forward slashes are valid.
cognos_items	*{"field","field", ... ,"field"}*	The name of one or more data objects to be imported. The format of *field* is [*namespace*].[*query_subject*].[*query_item*] Example: set :cognosimport.cognos_items = {"[Inventory (query)].[Inventory].[Opening inventory]", "[Inventory (query)].[Inventory].[Quantity shipped]", "[Inventory (query)].[Inventory].[Additions]", "[Inventory (query)].[Inventory].[Unit cost]", "[Inventory (query)].[Inventory].[Closing inventory]", "[Inventory (query)].[Inventory].[Average unit cost]"}
cognos_filters	*field*	The name of one or more filters to apply before importing data. Example: set :cognosimport.cognos_filters = {"[Inventory].[Filter].[MyFilter]"}
cognos_data_parameters	*list*	Values for prompt parameters for data. Name-and-value pairs are enclosed in braces, and multiple pairs are separated by commas and the whole string enclosed in square brackets. Format: [{"*param1*", "*value*"},…,{"*paramN*", "*value*"}] Example: set :cognosimport.cognos_data_parameters = [{"SexValue","F"},{"a","1"},{"b","1"}]
cognos_report_location	*field*	The Cognos path of a folder or package from which to import reports, for example: /Public Folders/GOSALES Note: Only forward slashes are valid.
cognos_report_name	*field*	The path and name within the report location of a report to import, for example: set :cognosimport.cognos_report_name = /Jimmy/Package/Drug4nPackage/3columns

cognosimportnode properties	Data type	Property description
cognos_report_parameters	*list*	Values for report parameters. Name-and-value pairs are enclosed in braces, and multiple pairs are separated by commas and the whole string enclosed in square brackets. Format: [{"*param1*", "*value*"},…,{"*paramN*", "*value*"}] Example: set :cognosimport.cognos_report_parameters = [{"SexValue","F"},{"a","1"},{"b","1"}]

databasenode Properties

The Database node can be used to import data from a variety of other packages using ODBC (Open Database Connectivity), including Microsoft SQL Server, DB2, Oracle, and others.

Example

```
create databasenode
set :databasenode.mode = Table
set :databasenode.query = "SELECT * FROM drug4n"
set :databasenode.datasource = "Drug4n_db"
set :databasenode.username = "spss"
set :databasenode.password = "spss"
var test_e
set test_e = :databasenode.epassword
set :databasenode.tablename = ".Drug4n"
```

databasenode properties	Data type	Property description
mode	Table Query	Specify *Table* to connect to a database table by using dialog box controls, or specify *Query* to query the selected database by using SQL.
datasource	*string*	Database name (see also note below).
username	*string*	Database connection details (see also note below).
password	*string*	
epassword	*string*	Specifies an encoded password as an alternative to hard-coding a password in a script. For more information, see the topic Generating an Encoded Password in Chapter 5 on p. 59. This property is read-only during execution.
tablename	*string*	Name of the table you want to access.

databasenode properties	Data type	Property description
strip_spaces	None Left Right Both	Options for discarding leading and trailing spaces in strings.
use_quotes	AsNeeded Always Never	Specify whether table and column names are enclosed in quotation marks when queries are sent to the database (for example, if they contain spaces or punctuation).
query	*string*	Specifies the SQL code for the query you want to submit.

Note: If the database name (in the datasource property) contains spaces, then instead of individual properties for datasource, username and password, use a single datasource property in the following format:

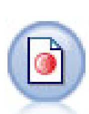
databasenode properties	Data type	Property description
datasource	*string*	Format: {database_name,username,password[,true | false]} The last parameter is for use with encrypted passwords. If this is set to true, the password will be decrypted before use.

Example

```
create databasenode
set :databasenode.mode = Table
set :databasenode.query = "SELECT * FROM drug4n"
set :databasenode.datasource = {"ORA 10gR2", user1, mypsw, true}
var test_e
set test_e = :databasenode.epassword
set :databasenode.tablename = ".Drug4n"
```

Use this format also if you are changing the data source; however, if you just want to change the username or password, you can use the username or password properties.

datacollectionimportnode Properties



The IBM® SPSS® Data Collection Data Import node imports survey data based on the Data Collection Data Model used by IBM Corp. market research products. The Data Collection Data Library must be installed to use this node.

Example

```
create datacollectionimportnode
set :datacollectionimportnode.metadata_name="mrQvDsc"
set :datacollectionimportnode.metadata_file="C:/Program Files/IBM/SPSS/DataCollection/DDL/Data/
Quanvert/Museum/museum.pkd"
```

```
set :datacollectionimportnode.casedata_name="mrQvDsc"
set :datacollectionimportnode.casedata_source_type=File
set :datacollectionimportnode.casedata_file="C:/Program Files/IBM/SPSS/DataCollection/DDL/Data/
Quanvert/Museum/museum.pkd"
set :datacollectionimportnode.import_system_variables = Common
set :datacollectionimportnode.import_multi_response = MultipleFlags
```

datacollectionimportnode properties	**Data type**	**Property description**
metadata_name	*string*	The name of the MDSC. The special value DimensionsMDD indicates that the standard Data Collection metadata document should be used. Other possible values include: mrADODsc mrI2dDsc mrLogDsc mrQdiDrsDsc mrQvDsc mrSampleReportingMDSC mrSavDsc mrSCDsc mrScriptMDSC The special value none indicates that there is no MDSC.
metadata_file	*string*	Name of the file where the metadata is stored.
casedata_name	*string*	The name of the CDSC. Possible values include: mrADODsc mrI2dDsc mrLogDsc mrPunchDSC mrQdiDrsDsc mrQvDsc mrRdbDsc2 mrSavDsc mrScDSC mrXmlDsc The special value none indicates that there is no CDSC.
casedata_source_type	Unknown File Folder UDL DSN	Indicates the source type of the CDSC.
casedata_file	*string*	When casedata_source_type is *File*, specifies the file containing the case data.
casedata_folder	*string*	When casedata_source_type is *Folder*, specifies the folder containing the case data.
casedata_udl_string	*string*	When casedata_source_type is *UDL*, specifies the OLD-DB connection string for the data source containing the case data.

datacollectionimportnode properties	Data type	Property description
casedata_dsn_string	*string*	When casedata_source_type is *DSN*, specifies the ODBC connection string for the data source.
casedata_project	*string*	When reading case data from a Data Collection database, you can enter the name of the project. For all other case data types, this setting should be left blank.
version_import_mode	All Latest Specify	Defines how versions should be handled.
specific_version	*string*	When version_import_mode is *Specify*, defines the version of the case data to be imported.
use_language	*string*	Defines whether labels of a specific language should be used.
language	*string*	If use_language is true, defines the language code to use on import. The language code should be one of those available in the case data.
use_context	*string*	Defines whether a specific context should be imported. Contexts are used to vary the description associated with responses.
context	*string*	If use_context is true, defines the context to import. The context should be one of those available in the case data.
use_label_type	*string*	Defines whether a specific type of label should be imported.
label_type	*string*	If use_label_type is true, defines the label type to import. The label type should be one of those available in the case data.
user_id	*string*	For databases requiring an explicit login, you can provide a user ID and password to access the data source.
password	*string*	
import_system_variables	Common None All	Specifies which system variables are imported.
import_codes_variables	*flag*	
import_sourcefile_variables	*flag*	
import_multi_response	MultipleFlags Single	

excelimportnode Properties

The Excel Import node imports data from any version of Microsoft Excel. An ODBC data source is not required.

Example

```
#To use a named range:
create excelimportnode
set :excelimportnode.excel_file_type = Excel2007
set :excelimportnode.full_filename = "C:/drug.xls"
set :excelimportnode.use_named_range = True
set :excelimportnode.named_range = "DRUG"
set :excelimportnode.read_field_names = True

#To use an explicit range:
create excelimportnode
set :excelimportnode.excel_file_type = Excel2007
set :excelimportnode.full_filename = "C:/drug.xls"
set :excelimportnode.worksheet_mode = Name
set :excelimportnode.worksheet_name = "Drug"
set :excelimportnode.explicit_range_start = A1
set :excelimportnode.explicit_range_end = F300
```

excelimportnode properties	**Data type**	**Property description**
excel_file_type	Excel2003 Excel2007	
full_filename	*string*	The complete filename, including path.
use_named_range	*Boolean*	Whether to use a named range. If true, the named_range property is used to specify the range to read, and other worksheet and data range settings are ignored.
named_range	*string*	
worksheet_mode	Index Name	Specifies whether the worksheet is defined by index or name.
worksheet_index	*integer*	Index of the worksheet to be read, beginning with 0 for the first worksheet, 1 for the second, and so on.
worksheet_name	*string*	Name of the worksheet to be read.
data_range_mode	FirstNonBlank ExplicitRange	Specifies how the range should be determined.
blank_rows	StopReading ReturnBlankRows	When data_range_mode is *FirstNonBlank*, specifies how blank rows should be treated.
explicit_range_start	*string*	When data_range_mode is *ExplicitRange*, specifies the starting point of the range to read.
explicit_range_end	*string*	
read_field_names	*Boolean*	Specifies whether the first row in the specified range should be used as field (column) names.

evimportnode Properties

The Enterprise View node creates a connection to an IBM SPSS Collaboration and Deployment Services Repository, enabling you to read Enterprise View data into a stream and to package a model in a scenario that can be accessed from the repository by other users.

Example

```
create evimportnode
set :evimportnode.connection = ['Training data','/Application views/Marketing','LATEST','Analytic',
    '/Data Providers/Marketing']
set :evimportnode.tablename = "cust1"
```

evimportnode properties	**Data type**	**Property description**
connection	*list*	Structured property—list of parameters making up an Enterprise View connection. Usage format: evimportnode.connection = [description,app_view_path, app_view_version_label,environment,DPD_path]
tablename	*string*	The name of a table in the Application View.

fixedfilenode Properties

The Fixed File node imports data from fixed-field text files—that is, files whose fields are not delimited but start at the same position and are of a fixed length. Machine-generated or legacy data are frequently stored in fixed-field format.

Example

```
create fixedfilenode
set :fixedfilenode.full_filename = "$CLEO_DEMOS/DRUG4n"
set :fixedfilenode.record_len = 32
set :fixedfilenode.skip_header = 1
set :fixedfilenode.fields = [{'Age' 1 3} {'Sex' 5 7} {'BP' 9 10} {'Cholesterol' 12 22} {'Na' 24 25} {'K' 27 27} {'Drug' 29 32}]
set :fixedfilenode.decimal_symbol = Period
set :fixedfilenode.lines_to_scan = 30
```

fixedfilenode properties	**Data type**	**Property description**
record_len	*number*	Specifies the number of characters in each record.
line_oriented	*flag*	Skips the new-line character at the end of each record.
decimal_symbol	Default Comma Period	The type of decimal separator used in your data source. Example: set :fixedfilenode.decimal_symbol = Period

fixedfilenode properties	Data type	Property description
skip_header	*number*	Specifies the number of lines to ignore at the beginning of the first record. Useful for ignoring column headers.
auto_recognize_datetime	*flag*	Specifies whether dates or times are automatically identified in the source data.
lines_to_scan	*number*	Example: set :fixedfilenode.lines_to_scan = 50.
fields	*list*	Structured property. Usage format: fixedfilenode.fields = [{field start length} {field start length}]
full_filename	*string*	Full name of file to read, including directory.
strip_spaces	None Left Right Both	Discards leading and trailing spaces in strings on import.
invalid_char_mode	Discard Replace	Removes invalid characters (null, 0, or any character non-existent in current encoding) from the data input or replaces invalid characters with the specified one-character symbol.
invalid_char_replacement	*string*	
use_custom_values	*flag*	Keyed slot in the form: set :varfilenode.use_custom_values.Age = true
custom_storage	Unknown String Integer Real Time Date Timestamp	Keyed slot in the form: set :varfilenode.custom_storage.'Age' = "Real"
custom_date_format	"DDMMYY" "MMDDYY" "YYMMDD" "YYYYMMDD" "YYYYDDD" DAY MONTH "DD-MM-YY" "DD-MM-YYYY" "MM-DD-YY" "MM-DD-YYYY" "DD-MON-YY" "DD-MON-YYYY" "YYYY-MM-DD" "DD.MM.YY" "DD.MM.YYYY" "MM.DD.YY" "MM.DD.YYYY" "DD.MON.YY" "DD.MON.YYYY" "DD/MM/YY" "DD/MM/YYYY" "MM/DD/YY"	This property is applicable only if a custom storage has been specified. Example: set:varfilenode.custom Keyed slot in the form: set :varfilenode.custom_date_ format.'LaunchDate' = "DDMMYY"

fixedfilenode properties	Data type	Property description
	"MM/DD/YYYY" "DD/MON/YY" "DD/MON/YYYY" MON YYYY q Q YYYY ww WK YYYY	
custom_time_format	"HHMMSS" "HHMM" "MMSS" "HH:MM:SS" "HH:MM" "MM:SS" "(H)H:(M)M:(S)S" "(H)H:(M)M" "(M)M:(S)S" "HH.MM.SS" "HH.MM" "MM.SS" "(H)H.(M)M.(S)S" "(H)H.(M)M" "(M)M.(S)S"	This property is applicable only if a custom storage has been specified. Keyed slot in the form: set :varfilenode.custom_time_format.'Initialize' = "HHMM"
custom_decimal_symbol	*field*	Applicable only if a custom storage has been specified. Keyed slot in the form: set :varfilenode.custom_decimal_symbol.'Revenue' = "Comma"
encoding	StreamDefault SystemDefault "UTF-8"	Specifies the text-encoding method.

sasimportnode Properties

The SAS Import node imports SAS data into IBM® SPSS® Modeler.

Example

```
create sasimportnode
set :sasimportnode.format = Windows
set :sasimportnode.full_filename = "C:/data/retail.sas7bdat"
set :sasimportnode.member_name = "Test"
set :sasimportnode.read_formats = False
set :sasimportnode.full_format_filename = "Test"
```

```
set :sasimportnode.import_names = True
```

sasimportnode properties	Data type	Property description
format	Windows UNIX Transport SAS7 SAS8 SAS9	Format of the file to be imported.
full_filename	*string*	The complete filename that you enter, including its path.
member_name	*string*	Specify the member to import from the specified SAS transport file.
read_formats	*flag*	Reads data formats (such as variable labels) from the specified format file.
full_format_filename	*string*	
import_names	NamesAndLabels LabelsasNames	Specifies the method for mapping variable names and labels on import.

statisticsimportnode Properties

The IBM® SPSS® Statistics File node reads data from the *.sav* file format used by SPSS Statistics, as well as cache files saved in IBM® SPSS® Modeler, which also use the same format.

The properties for this node are described under statisticsimportnode Properties on p. 292.

userinputnode Properties

The User Input node provides an easy way to create synthetic data—either from scratch or by altering existing data. This is useful, for example, when you want to create a test dataset for modeling.

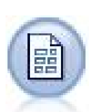
Example

```
create userinputnode
set :userinputnode.data.test1 = "2, 4, 8"
set :userinputnode.names = [test1 test2]
set :userinputnode.custom_storage.test1 = Integer
```

set :userinputnode.data_mode = "Ordered"

userinputnode properties	Data type	Property description
data		Keyed property of the form: set :userinputnode.data.Age = "1 2 3 4" Alternatively, the string can specify low, high, and step size values separated by commas. Example: set :userinputnode.data.Age = "10, 70, 5" The data for each field can be of different lengths but must be consistent with the field's storage. Setting values for a field that isn't present creates that field. Additionally, setting the values for a field to an empty string (" ") removes the specified field.
names		Structured slot that sets or returns a list of field names generated by the node. Example: ['Field1' 'Field2']
custom_storage	Unknown String Integer Real Time Date Timestamp	Keyed slot that sets or returns the storage for a field. Example: set :userinputnode.custom_storage.'Age' = "Real"
data_mode	Combined Ordered	If Combined is specified, records are generated for each combination of set values and min/max values. The number of records generated is equal to the product of the number of values in each field. If Ordered is specified, one value is taken from each column for each record in order to generate a row of data. The number of records generated is equal to the largest number values associated with a field. Any fields with fewer data values will be padded with null values.
values		*This property has been deprecated in favor of* userinputnode.data *and should no longer be used.*

variablefilenode Properties



The Variable File node reads data from free-field text files—that is, files whose records contain a constant number of fields but a varied number of characters. This node is also useful for files with fixed-length header text and certain types of annotations.

Example

```
create variablefilenode
set :variablefilenode.full_filename = "$CLEO_DEMOS/DRUG4n"
set :variablefilenode.read_field_names = True
```

```
set :variablefilenode.delimit_other = True
set :variablefilenode.other = ','
set :variablefilenode.quotes_1 = Discard
set :variablefilenode.decimal_symbol = Comma
set :variablefilenode.invalid_char_mode = "Replace"
set :variablefilenode.invalid_char_replacement = "|"
set :variablefilenode.use_custom_values.Age = True
set :variablefilenode.direction.Age = Input
set :variablefilenode.type.Age = Range
set :variablefilenode.values.Age = [1 100]
```

variablefilenode properties	Data type	Property description
skip_header	*number*	Specifies the number of characters to ignore at the beginning of the first record. Usage format: variablefilenode:skip_header = 3
num_fields_auto	*flag*	Determines the number of fields in each record automatically. Records must be terminated with a new-line character. Usage format: variablefilenode:num_fields_auto
num_fields	*number*	Manually specifies the number of fields in each record.
delimit_space	*flag*	Specifies the character used to delimit field boundaries in the file.
delimit_tab	*flag*	
delimit_new_line	*flag*	
delimit_non_printing	*flag*	
delimit_comma	*flag*	In cases where the comma is both the field delimiter and the decimal separator for streams, set delimit_other to *true*, and specify a comma as the delimiter by using the other property.
delimit_other	*flag*	Allows you to specify a custom delimiter using the other property.
other	*string*	Specifies the delimiter used when delimit_other is *true*.
decimal_symbol	Default Comma Period	Specifies the decimal separator used in the data source.
multi_blank	*flag*	Treats multiple adjacent blank delimiter characters as a single delimiter.
read_field_names	*flag*	Treats the first row in the data file as labels for the column.
strip_spaces	None Left Right Both	Discards leading and trailing spaces in strings on import.
invalid_char_mode	Discard Replace	Removes invalid characters (null, 0, or any character non-existent in current encoding) from the data input or replaces invalid characters with the specified one-character symbol.

variablefilenode properties	**Data type**	**Property description**
invalid_char_replacement	*string*	
lines_to_scan	*number*	Specifies how many lines to scan for specified data types.
auto_recognize_datetime	*flag*	Specifies whether dates or times are automatically identified in the source data.
quotes_1	Discard PairAndDiscard IncludeAsText	Specifies how single quotation marks are treated upon import.
quotes_2	Discard PairAndDiscard IncludeAsText	Specifies how double quotation marks are treated upon import.
full_filename	*string*	Full name of file to be read, including directory.
use_custom_values	*flag*	Keyed slot in the form: set :varfilenode.use_custom_values.Age = true
custom_storage	Unknown String Integer Real Time Date Timestamp	Keyed slot in the form: set :varfilenode.custom_storage.'Age' = "Real"
custom_date_format	"DDMMYY" "MMDDYY" "YYMMDD" "YYYYMMDD" "YYYYDDD" DAY MONTH "DD-MM-YY" "DD-MM-YYYY" "MM-DD-YY" "MM-DD-YYYY" "DD-MON-YY" "DD-MON-YYYY" "YYYY-MM-DD" "DD.MM.YY" "DD.MM.YYYY" "MM.DD.YY" "MM.DD.YYYY" "DD.MON.YY" "DD.MON.YYYY" "DD/MM/YY" "DD/MM/YYYY" "MM/DD/YY" "MM/DD/YYYY" "DD/MON/YY" "DD/MON/YYYY" MON YYYY q Q YYYY ww WK YYYY	Applicable only if a custom storage has been specified. Example: set:varfilenode.custom Keyed slot in the form: set :varfilenode.custom_date_format. 'LaunchDate' = "DDMMYY"

variablefilenode properties	Data type	Property description
custom_time_format	"HHMMSS" "HHMM" "MMSS" "HH:MM:SS" "HH:MM" "MM:SS" "(H)H:(M)M:(S)S" "(H)H:(M)M" "(M)M:(S)S" "HH.MM.SS" "HH.MM" "MM.SS" "(H)H.(M)M.(S)S" "(H)H.(M)M" "(M)M.(S)S"	Applicable only if a custom storage has been specified. Keyed slot in the form: set :varfilenode.custom_time_format. 'Initialize' = "HHMM"
custom_decimal_symbol	*field*	Applicable only if a custom storage has been specified. Keyed slot in the form: set :varfilenode.custom_decimal_ symbol.'Revenue' = "Comma"
encoding	StreamDefault SystemDefault "UTF-8"	Specifies the text-encoding method.

xmlimportnode Properties

The XML source node imports data in XML format into the stream. You can import a single file, or all files in a directory. You can optionally specify a schema file from which to read the XML structure.

Example

```
create xmlimportnode
set :xmlimportnode.full_filename = "c:\import\ebooks.xml"
set :xmlimportnode.records = "/author/name"
```

xmlimportnode properties	Data type	Property description
read	single directory	Reads a single data file (default), or all XML files in a directory.
recurse	*flag*	Specifies whether to additionally read XML files from all the subdirectories of the specified directory.
full_filename	*string*	(required) Full path and file name of XML file to import (if read = single).
directory_name	*string*	(required) Full path and name of directory from which to import XML files (if read = directory).
full_schema_filename	*string*	Full path and file name of XSD or DTD file from which to read the XML structure. If you omit this parameter, structure is read from the XML source file.

xmlimportnode properties	Data type	Property description
records	*string*	XPath expression (e.g. /author/name) to define the record boundary. Each time this element is encountered in the source file, a new record is created.
mode	read specify	Read all data (default), or specify which items to read.
fields		List of items (elements and attributes) to import. Each item in the list is an XPath expression.

Chapter

13

Record Operations Node Properties

appendnode Properties

The Append node concatenates sets of records. It is useful for combining datasets with similar structures but different data.

Example

```
create appendnode
set :appendnode.match_by = Name
set :appendnode.match_case = True
set :appendnode.include_fields_from = All
set :appendnode.create_tag_field = True
set :appendnode.tag_field_name = "Append_Flag"
```

appendnode properties	Data type	Property description
match_by	Position Name	You can append datasets based on the position of fields in the main data source or the name of fields in the input datasets.
match_case	*flag*	Enables case sensitivity when matching field names.
include_fields_from	Main All	
create_tag_field	*flag*	
tag_field_name	*string*	

aggregatenode Properties

The Aggregate node replaces a sequence of input records with summarized, aggregated output records.

Example

```
create aggregatenode
connect :databasenode to :aggregatenode
set :aggregatenode.contiguous = True
set :aggregatenode.keys = ['Drug']
set :aggregatenode.aggregates.Age = [Sum Mean]
set :aggregatenode.inc_record_count = True
set :aggregatenode.count_field = "index"
set :aggregatenode.extension = "Aggregated_"
```

```
set :aggregatenode.add_as = Prefix
```

aggregatenode properties	Data type	Property description
keys	*[field field ... field]*	Lists fields that can be used as keys for aggregation. For example, if Sex and Region are your key fields, each unique combination of M and F with regions N and S (four unique combinations) will have an aggregated record.
contiguous	*flag*	Select this option if you know that all records with the same key values are grouped together in the input (for example, if the input is sorted on the key fields). Doing so can improve performance.
aggregates		Structured property listing the numeric fields whose values will be aggregated, as well as the selected modes of aggregation. Example: set :aggregatenode. aggregates.Age = [Sum Mean Min Max SDev Median Count Variance Firstquartile Thirdquartile], where the desired aggregation methods are included in the list.
extension	*string*	Specify a prefix or suffix for duplicate aggregated fields (sample below).
add_as	Suffix Prefix	
inc_record_count	*flag*	Creates an extra field that specifies how many input records were aggregated to form each aggregate record.
count_field	*string*	Specifies the name of the record count field.

balancenode Properties

The Balance node corrects imbalances in a dataset, so it conforms to a specified condition. The balancing directive adjusts the proportion of records where a condition is true by the factor specified.

Example

```
create balancenode
set :balancenode.training_data_only = true
set :balancenode.directives = \
 [{1.3 "Age > 60"}{1.5 "Na > 0.5"}]
```

balancenode properties	Data type	Property description
directives		Structured property to balance proportion of field values based on number specified (see example below).
training_data_only	*flag*	Specifies that only training data should be balanced. If no partition field is present in the stream, then this option is ignored.

Example

```
create balancenode
set :balancenode.directives = \
  [{1.3 "Age > 60"}{1.5 "Na > 0.5"}]
```

This node property uses the format:

[{ *number string* } \ { *number string*} \ ... { *number string* }].

Note: If strings (using double quotation marks) are embedded in the expression, they need to be preceded by the escape character " \ ". The " \ " character is also the line continuation character, allowing you to line up the arguments for clarity.

distinctnode Properties

The Distinct node removes duplicate records, either by passing the first distinct record to the data stream or by discarding the first record and passing any duplicates to the data stream instead.

Example

```
create distinctnode
set :distinctnode.mode = Include
set :distinctnode.fields = ['Age' 'Sex']
set :distinctnode.keys_pre_sorted = True
```

distinctnode properties	Data type	Property description
mode	Include Discard	You can include the first distinct record in the data stream, or discard the first distinct record and pass any duplicate records to the data stream instead.
fields	[*field field field*]	Lists fields used to determine whether records are identical.
low_distinct_key_count	*flag*	Specifies that you have only a small number of records and/or a small number of unique values of the key field(s).
keys_pre_sorted	*flag*	Specifies that all records with the same key values are grouped together in the input.

mergenode Properties

The Merge node takes multiple input records and creates a single output record containing some or all of the input fields. It is useful for merging data from different sources, such as internal customer data and purchased demographic data.

Example

```
create mergenode
connect customerdata to :mergenode
connect salesdata to :mergenode
set :mergenode.method = Keys
set :mergenode.key_fields = ['id']
set :mergenode.common_keys = true
set :mergenode.join = PartialOuter
set :mergenode.outer_join_tag.2 = true
set :mergenode.outer_join_tag.4 = true
set :mergenode.single_large_input = true
set :mergenode.single_large_input_tag = '2'
set :mergenode.use_existing_sort_keys = true
set :mergenode.existing_sort_keys = [{'id' Ascending}]
```

mergenode properties	Data type	Property description
method	Order Keys Condition	Specify whether records are merged in the order they are listed in the data files, if one or more key fields will be used to merge records with the same value in the key fields, or if records will be merged if a specified condition is satisfied.
condition	*string*	If method is set to Condition, specifies the condition for including or discarding records.
key_fields	[*field field field*]	
common_keys	*flag*	
join	Inner FullOuter PartialOuter Anti	An example is as follows: set :merge.join = FullOuter
outer_join_tag.n	*flag*	In this property, *n* is the tag name as displayed in the Select Dataset dialog box. Note that multiple tag names may be specified, as any number of datasets could contribute incomplete records.
single_large_input	*flag*	Specifies whether optimization for having one input relatively large compared to the other inputs will be used.
single_large_input_tag	*string*	Specifies the tag name as displayed in the Select Large Dataset dialog box. Note that the usage of this property differs slightly from the outer_join_tag property (flag versus string) because only one input dataset can be specified.
use_existing_sort_keys	*flag*	Specifies whether the inputs are already sorted by one or more key fields.
existing_sort_keys	[{*string* Ascending} \ {*string* Descending}]	Specifies the fields that are already sorted and the direction in which they are sorted.

rfmaggregatenode Properties

The Recency, Frequency, Monetary (RFM) Aggregate node enables you to take customers' historical transactional data, strip away any unused data, and combine all of their remaining transaction data into a single row that lists when they last dealt with you, how many transactions they have made, and the total monetary value of those transactions.

Example

```
create rfmaggregatenode
connect :fillernode to :rfmaggregatenode
set :rfmaggregatenode.relative_to = Fixed
set :rfmaggregatenode.reference_date = "2007-10-12"
set :rfmaggregatenode.id_field = "CardID"
set :rfmaggregatenode.date_field = "Date"
set :rfmaggregatenode.value_field = "Amount"
set :rfmaggregatenode.only_recent_transactions = True
set :rfmaggregatenode.transaction_date_after = "2000-10-01"
```

rfmaggregatenode properties	**Data type**	**Property description**
relative_to	Fixed Today	Specify the date from which the recency of transactions will be calculated.
reference_date	*date*	Only available if Fixed is chosen in relative_to.
contiguous	*flag*	If your data are presorted so that all records with the same ID appear together in the data stream, selecting this option speeds up processing.
id_field	*field*	Specify the field to be used to identify the customer and their transactions.
date_field	*field*	Specify the date field to be used to calculate recency against.
value_field	*field*	Specify the field to be used to calculate the monetary value.
extension	*string*	Specify a prefix or suffix for duplicate aggregated fields.
add_as	Suffix Prefix	Specify if the extension should be added as a suffix or a prefix.
discard_low_value_records	*flag*	Enable use of the discard_records_below setting.
discard_records_below	*number*	Specify a minimum value below which any transaction details are not used when calculating the RFM totals. The units of value relate to the value field selected.
only_recent_transactions	*flag*	Enable use of either the specify_transaction_date or transaction_within_last settings.
specify_transaction_date	*flag*	

rfmaggregatenode properties	**Data type**	**Property description**
transaction_date_after	*date*	Only available if specify_transaction_date is selected. Specify the transaction date after which records will be included in your analysis.
transaction_within_last	*number*	Only available if transaction_within_last is selected. Specify the number and type of periods (days, weeks, months, or years) back from the Calculate Recency relative to date after which records will be included in your analysis.
transaction_scale	Days Weeks Months Years	Only available if transaction_within_last is selected. Specify the number and type of periods (days, weeks, months, or years) back from the Calculate Recency relative to date after which records will be included in your analysis.
save_r2	*flag*	Displays the date of the second most recent transaction for each customer.
save_r3	*flag*	Only available if save_r2 is selected. Displays the date of the third most recent transaction for each customer.

samplenode Properties

The Sample node selects a subset of records. A variety of sample types are supported, including stratified, clustered, and nonrandom (structured) samples. Sampling can be useful to improve performance, and to select groups of related records or transactions for analysis.

Example

```
/* Create two Sample nodes to extract
  different samples from the same data */

create variablefilenode
set :variablefilenode.full_filename = "$CLEO_DEMOS/DRUG1n"

set node = create samplenode at 300 100
rename ^node as 'First 500'
connect :variablefilenode to 'First 500'
set 'First 500':samplenode.method = Simple
set 'First 500':samplenode.mode = Include
set 'First 500':samplenode.sample_type = First
set 'First 500':samplenode.first_n = 500

set node = create samplenode at 300 200
rename ^node as 'Custom Strata'
connect :variablefilenode to 'Custom Strata'
set 'Custom Strata':samplenode.method = Complex
set 'Custom Strata':samplenode.stratify_by = ['Sex' 'Cholesterol']
```

```
set 'Custom Strata':samplenode.sample_units = Proportions
set 'Custom Strata':samplenode.sample_size_proportions = Custom
set 'Custom Strata':samplenode.sizes_proportions= \
    [{"M" "High" "Default"}{"M" "Normal" "Default"} \
    {"F" "High" "0.3"}{"F" "Normal" "0.3"}]
```

samplenode properties	Data type	Property description
method	Simple Complex	
mode	Include Discard	Include or discard records that meet the specified condition.
sample_type	First OneInN RandomPct	Specifies the sampling method. An example is as follows: set :samplenode.sample_type = First set :samplenode.first_n = 100
first_n	*integer*	Records up to the specified cutoff point will be included or discarded.
one_in_n	*number*	Include or discard every *n*th record.
rand_pct	*number*	Specify the percentage of records to include or discard.
use_max_size	*flag*	Enable use of the maximum_size setting.
maximum_size	*integer*	Specify the largest sample to be included or discarded from the data stream. This option is redundant and therefore disabled when First and Include are specified.
set_random_seed	*flag*	Enables use of the random seed setting.
random_seed	*integer*	Specify the value used as a random seed.
complex_sample_type	Random Systematic	
sample_units	Proportions Counts	
sample_size_proportions	Fixed Custom Variable	
sample_size_counts	Fixed Custom Variable	
fixed_proportions	*number*	
fixed_counts	*integer*	
variable_proportions	*field*	
variable_counts	*field*	
use_min_stratum_size	*flag*	
minimum_stratum_size	*integer*	This option only applies when a Complex sample is taken with Sample units=Proportions.
use_max_stratum_size	*flag*	
maximum_stratum_size	*integer*	This option only applies when a Complex sample is taken with Sample units=Proportions.
clusters	*field*	
stratify_by	*[field1 ... fieldN]*	

samplenode properties	Data type	Property description
specify_input_weight	*flag*	
input_weight	*field*	
new_output_weight	*string*	
sizes_proportions	[{string*string value*}{string*string value*}...]	If sample_units=proportions and sample_size_proportions=Custom, specifies a value for each possible combination of values of stratification fields.
default_proportion	*number*	
sizes_counts	[{string*string value*}{string*string value*}...]	Specifies a value for each possible combination of values of stratification fields. Usage is similar to sizes_proportions but specifying an integer rather than a proportion.
default_count	*number*	

selectnode Properties

The Select node selects or discards a subset of records from the data stream based on a specific condition. For example, you might select the records that pertain to a particular sales region.

Example

```
create selectnode
set :selectnode.mode = Include
set :selectnode.condition = "Age < 18"
```

selectnode properties	Data type	Property description
mode	Include Discard	Specifies whether to include or discard selected records.
condition	*string*	Condition for including or discarding records.

sortnode Properties

The Sort node sorts records into ascending or descending order based on the values of one or more fields.

Example

```
create sortnode
set :sortnode.keys = [{'Age' Ascending}{'Sex' Descending}]
set :sortnode.default_ascending = False
set :sortnode.use_existing_keys = True
```

```
set :sortnode.existing_keys = [{'Age' Ascending}]
```

sortnode properties	Data type	Property description
keys	[{*string* Ascending} \ {*string* Descending}]	Specifies the fields you want to sort against (example below). If no direction is specified, the default is used.
default_ascending	*flag*	Specifies the default sort order.
use_existing_keys	*flag*	Specifies whether sorting is optimized by using the previous sort order for fields that are already sorted.
existing_keys		Specifies the fields that are already sorted and the direction in which they are sorted. Uses the same format as the keys property.

Chapter
14

Field Operations Node Properties

anonymizenode Properties

The Anonymize node transforms the way field names and values are represented downstream, thus disguising the original data. This can be useful if you want to allow other users to build models using sensitive data, such as customer names or other details.

Example

```
create anonymizenode
set: anonymizenode.enable_anonymize = age
set: anonymizenode.use_prefix = true
set: anonymizenode.prefix = "myprefix"
set: anonymizenode.transformation = Random
set: anonymizenode.set_random_seed = true
set: anonymizenode.random_seed = "123"
```

anonymizenode properties	Data type	Property description
enable_anonymize	*flag*	When set to T, activates anonymization of field values (equivalent to selecting Yes for that field in the Anonymize Values column).
use_prefix	*flag*	When set to T, a custom prefix will be used if one has been specified. Applies to fields that will be anonymized by the Hash method and is equivalent to choosing the Custom radio button in the Replace Values dialog box for that field.
prefix	*string*	Equivalent to typing a prefix into the text box in the Replace Values dialog box. The default prefix is the default value if nothing else has been specified.
transformation	Random Fixed	Determines whether the transformation parameters for a field anonymized by the Transform method will be random or fixed.
set_random_seed	*flag*	When set to T, the specified seed value will be used (if transformation is also set to Random).
random_seed	*integer*	When set_random_seed is set to T, this is the seed for the random number.
scale	*number*	When transformation is set to Fixed, this value is used for "scale by." The maximum scale value is normally 10 but may be reduced to avoid overflow.
translate	*number*	When transformation is set to Fixed, this value is used for "translate." The maximum translate value is normally 1000 but may be reduced to avoid overflow.

autodataprepnode Properties

The Automated Data Preparation (ADP) node can analyze your data and identify fixes, screen out fields that are problematic or not likely to be useful, derive new attributes when appropriate, and improve performance through intelligent screening and sampling techniques. You can use the node in fully automated fashion, allowing the node to choose and apply fixes, or you can preview the changes before they are made and accept, reject, or amend them as desired.

Example

```
create autodataprepnode
set: autodataprepnode.objective = Balanced
set: autodataprepnode.excluded_fields = Filter
set: autodataprepnode.prepare_dates_and_times = true
set: autodataprepnode.compute_time_until_date = true
set: autodataprepnode.reference_date = Today
set: autodataprepnode.units_for_date_durations = Automatic
```

autodataprepnode properties	Data type	Property description
objective	Balanced Speed Accuracy Custom	
custom_fields	*flag*	If true, allows you to specify target, input, and other fields for the current node. If false, the current settings from an upstream Type node are used.
target	*field*	Specifies a single target field.
inputs	[*field1 ... fieldN*]	Input or predictor fields used by the model.
use_frequency	*flag*	
frequency_field	*field*	
use_weight	*flag*	
weight_field	*field*	
excluded_fields	Filter None	
if_fields_do_not_match	StopExecution ClearAnalysis	
prepare_dates_and_times	*flag*	Control access to all the date and time fields
compute_time_until_date	*flag*	
reference_date	Today Fixed	
fixed_date	*date*	
units_for_date_durations	Automatic Fixed	
fixed_date_units	Years Months Days	

autodataprepnode properties	**Data type**	**Property description**
compute_time_until_time	*flag*	
reference_time	CurrentTime Fixed	
fixed_time	*time*	
units_for_time_durations	Automatic Fixed	
fixed_date_units	Hours Minutes Seconds	
extract_year_from_date	*flag*	
extract_month_from_date	*flag*	
extract_day_from_date	*flag*	
extract_hour_from_time	*flag*	
extract_minute_from_time	*flag*	
extract_second_from_time	*flag*	
exclude_low_quality_inputs	*flag*	
exclude_too_many_missing	*flag*	
maximum_percentage_missing	*number*	
exclude_too_many_categories	*flag*	
maximum_number_categories	*number*	
exclude_if_large_category	*flag*	
maximum_percentage_category	*number*	
prepare_inputs_and_target	*flag*	
adjust_type_inputs	*flag*	
adjust_type_target	*flag*	
reorder_nominal_inputs	*flag*	
reorder_nominal_target	*flag*	
replace_outliers_inputs	*flag*	
replace_outliers_target	*flag*	
replace_missing_continuous_inputs	*flag*	
replace_missing_continuous_target	*flag*	
replace_missing_nominal_inputs	*flag*	
replace_missing_nominal_target	*flag*	
replace_missing_ordinal_inputs	*flag*	
replace_missing_ordinal_target	*flag*	
maximum_values_for_ordinal	*number*	
minimum_values_for_continuous	*number*	
outlier_cutoff_value	*number*	
outlier_method	Replace Delete	
rescale_continuous_inputs	*flag*	
rescaling_method	MinMax ZScore	

autodataprepnode properties	Data type	Property description
min_max_minimum	*number*	
min_max_maximum	*number*	
z_score_final_mean	*number*	
z_score_final_sd	*number*	
rescale_continuous_target	*flag*	
target_final_mean	*number*	
target_final_sd	*number*	
transform_select_input_fields	*flag*	
maximize_association_with_target	*flag*	
p_value_for_merging	*number*	
merge_ordinal_features	*flag*	
merge_nominal_features	*flag*	
minimum_cases_in_category	*number*	
bin_continuous_fields	*flag*	
p_value_for_binning	*number*	
perform_feature_selection	*flag*	
p_value_for_selection	*number*	
perform_feature_construction	*flag*	
transformed_target_name_extension	*string*	
transformed_inputs_name_extension	*string*	
constructed_features_root_name	*string*	
years_duration_ name_extension	*string*	
months_duration_ name_extension	*string*	
days_duration_ name_extension	*string*	
hours_duration_ name_extension	*string*	
minutes_duration_ name_extension	*string*	
seconds_duration_ name_extension	*string*	
year_cyclical_name_extension	*string*	
month_cyclical_name_extension	*string*	
day_cyclical_name_extension	*string*	
hour_cyclical_name_extension	*string*	
minute_cyclical_name_extension	*string*	
second_cyclical_name_extension	*string*	

binningnode Properties

The Binning node automatically creates new nominal (set) fields based on the values of one or more existing continuous (numeric range) fields. For example, you can transform a continuous income field into a new categorical field containing groups of income as deviations from the mean. Once you have created bins for the new field, you can generate a Derive node based on the cut points.

Example

```
create binningnode
set :binningnode.fields = [Na K]
set :binningnode.method = Rank
set :binningnode.fixed_width_name_extension = "_binned"
set :binningnode.fixed_width_add_as = Suffix
set :binningnode.fixed_bin_method = Count
set :binningnode.fixed_bin_count = 10
set :binningnode.fixed_bin_width = 3.5
set :binningnode.tile10 = true
```

binningnode properties	Data type	Property description
fields	*[field1 field2 ... fieldn]*	Continuous (numeric range) fields pending transformation. You can bin multiple fields simultaneously.
method	FixedWidth EqualCount Rank SDev Optimal	Method used for determining cut points for new field bins (categories).
rcalculate_bins	Always IfNecessary	Specifies whether the bins are recalculated and the data placed in the relevant bin every time the node is executed, or that data is added only to existing bins and any new bins that have been added.
fixed_width_name_extension	*string*	The default extension is *_BIN*.
fixed_width_add_as	Suffix Prefix	Specifies whether the extension is added to the end (suffix) of the field name or to the start (prefix). The default extension is *income_BIN*.
fixed_bin_method	Width Count	
fixed_bin_count	*integer*	Specifies an integer used to determine the number of fixed-width bins (categories) for the new field(s).
fixed_bin_width	*real*	Value (integer or real) for calculating width of the bin.
equal_count_name_ extension	*string*	The default extension is *_TILE*.
equal_count_add_as	Suffix Prefix	Specifies an extension, either suffix or prefix, used for the field name generated by using standard p-tiles. The default extension is *_TILE* plus *N*, where *N* is the tile number.

binningnode properties	Data type	Property description
tile4	*flag*	Generates four quantile bins, each containing 25% of cases.
tile5	*flag*	Generates five quintile bins.
tile10	*flag*	Generates 10 decile bins.
tile20	*flag*	Generates 20 vingtile bins.
tile100	*flag*	Generates 100 percentile bins.
use_custom_tile	*flag*	
custom_tile_name_extension	*string*	The default extension is *_TILEN*.
custom_tile_add_as	Suffix Prefix	
custom_tile	*integer*	
equal_count_method	RecordCount ValueSum	The RecordCount method seeks to assign an equal number of records to each bin, while ValueSum assigns records so that the sum of the values in each bin is equal.
tied_values_method	Next Current Random	Specifies which bin tied value data is to be put in.
rank_order	Ascending Descending	This property includes Ascending (lowest value is marked 1) or Descending (highest value is marked 1).
rank_add_as	Suffix Prefix	This option applies to rank, fractional rank, and percentage rank.
rank	*flag*	
rank_name_extension	*string*	The default extension is *_RANK*.
rank_fractional	*flag*	Ranks cases where the value of the new field equals rank divided by the sum of the weights of the nonmissing cases. Fractional ranks fall in the range of 0–1.
rank_fractional_name_extension	*string*	The default extension is *_F_RANK*.
rank_pct	*flag*	Each rank is divided by the number of records with valid values and multiplied by 100. Percentage fractional ranks fall in the range of 1–100.
rank_pct_name_extension	*string*	The default extension is *_P_RANK*.
sdev_name_extension	*string*	
sdev_add_as	Suffix Prefix	
sdev_count	One Two Three	
optimal_name_extension	*string*	The default extension is *_OPTIMAL*.
optimal_add_as	Suffix Prefix	
optimal_supervisor_field	*field*	Field chosen as the supervisory field to which the fields selected for binning are related.

binningnode properties	Data type	Property description
optimal_merge_bins	*flag*	Specifies that any bins with small case counts will be added to a larger, neighboring bin.
optimal_small_bin_threshold	*integer*	
optimal_pre_bin	*flag*	Indicates that prebinning of dataset is to take place.
optimal_max_bins	*integer*	Specifies an upper limit to avoid creating an inordinately large number of bins.
optimal_lower_end_point	Inclusive Exclusive	
optimal_first_bin	Unbounded Bounded	
optimal_last_bin	Unbounded Bounded	

derivenode Properties

The Derive node modifies data values or creates new fields from one or more existing fields. It creates fields of type formula, flag, nominal, state, count, and conditional.

Example

```
# Create and configure a Flag Derive field node
create derivenode
rename derive:derivenode as "Flag"
set Flag:derivenode.new_name = "DrugX_Flag"
set Flag:derivenode.result_type = Flag
set Flag:derivenode.flag_true = 1
set Flag:derivenode.flag_false = 0
set Flag:derivenode.flag_expr = "Drug = X"
# Create and configure a Conditional Derive field node
create derivenode
rename derive:derivenode as "Conditional"
set Conditional:derivenode.result_type = Conditional
set Conditional:derivenode.cond_if_cond = "@OFFSET(\'Age\', 1) = \'Age\'"
set Conditional:derivenode.cond_then_expr = "(@OFFSET(\'Age\', 1) = \'Age\') >< @INDEX"
set Conditional:derivenode.cond_else_expr = "\'Age\'"
```

derivenode properties	Data type	Property description
new_name	*string*	Name of new field.
mode	Single Multiple	Specifies single or multiple fields.
fields	[*field field field*]	Used in Multiple mode only to select multiple fields.
name_extension	*string*	Specifies the extension for the new field name(s).

derivenode properties	Data type	Property description
add_as	Suffix Prefix	Adds the extension as a prefix (at the beginning) or as a suffix (at the end) of the field name.
result_type	Formula Flag Set State Count Conditional	The six types of new fields that you can create.
formula_expr	*string*	Expression for calculating a new field value in a Derive node.
flag_expr	*string*	
flag_true	*string*	
flag_false	*string*	
set_default	*string*	
set_value_cond	*string*	Structured to supply the condition associated with a given value. Usage format: set :derivenode. set_value_cond. Retired = 'age > 65'
state_on_val	*string*	Specifies the value for the new field when the On condition is met.
state_off_val	*string*	Specifies the value for the new field when the Off condition is met.
state_on_expression	*string*	
state_off_expression	*string*	
state_initial	On Off	Assigns each record of the new field an initial value of On or Off. This value can change as each condition is met.
count_initial_val	*string*	
count_inc_condition	*string*	
count_inc_expression	*string*	
count_reset_condition	*string*	
cond_if_cond	*string*	
cond_then_expr	*string*	
cond_else_expr	*string*	

ensemblenode Properties

The Ensemble node combines two or more model nuggets to obtain more accurate predictions than can be gained from any one model.

Example

```
# Create and configure an Ensemble node
# Use this node with the models in demos\streams\pm_binaryclassifier.str
```

```
create ensemblenode
set :ensemblenode.ensemble_target_field = response
set :ensemblenode.filter_individual_model_output = false
set :ensemblenode.flag_ensemble_method = ConfidenceWeightedVoting
set :ensemblenode.flag_voting_tie_selection = HighestConfidence
```

ensemblenode properties	**Data type**	**Property description**
ensemble_target_field	*field*	Specifies the target field for all models used in the ensemble.
filter_individual_model_output	*flag*	Specifies whether scoring results from individual models should be suppressed.
flag_ensemble_method	Voting ConfidenceWeightedVoting RawPropensityWeightedVoting AdjustedPropensityWeightedVoting HighestConfidence AverageRawPropensity AverageAdjustedPropensity	Specifies the method used to determine the ensemble score. This setting applies only if the selected target is a flag field.
set_ensemble_method	Voting ConfidenceWeightedVoting HighestConfidence	Specifies the method used to determine the ensemble score. This setting applies only if the selected target is a nominal field.
flag_voting_tie_selection	Random HighestConfidence RawPropensity AdjustedPropensity	If a voting method is selected, specifies how ties are resolved. This setting applies only if the selected target is a flag field.
set_voting_tie_selection	Random HighestConfidence	If a voting method is selected, specifies how ties are resolved. This setting applies only if the selected target is a nominal field.
calculate_standard_error	*flag*	If the target field is continuous, a standard error calculation is run by default to calculate the difference between the measured or estimated values and the true values; and to show how close those estimates matched.

fillernode Properties

The Filler node replaces field values and changes storage. You can choose to replace values based on a CLEM condition, such as @BLANK(@FIELD). Alternatively, you can choose to replace all blanks or null values with a specific value. A Filler node is often used together with a Type node to replace missing values.

Example

```
create fillernode
set :fillernode.fields = ['Age']
set :fillernode.replace_mode = Always
```

```
set :fillernode.condition = "(\'Age\' > 60) and (\'Sex\' = \'M\')"
set :fillernode.replace_with = "\'old man\'"
```

fillernode properties	Data type	Property description
fields	[*field field field*]	Fields from the dataset whose values will be examined and replaced.
replace_mode	Always Conditional Blank Null BlankAndNull	You can replace all values, blank values, or null values, or replace based on a specified condition.
condition	*string*	
replace_with	*string*	

filternode Properties

The Filter node filters (discards) fields, renames fields, and maps fields from one source node to another.

Example

```
create filternode
set :filternode.default_include = True
set :filternode.new_name.'Drug' = 'Chemical'
set :filternode.include.'Drug' = off
```

Using the default_include property. Note that setting the value of the default_include property does not automatically include or exclude all fields; it simply determines the default for the current selection. This is functionally equivalent to clicking the Include fields by default button in the Filter node dialog box. For example, suppose you run the following script:

```
set Filter.default_include=False
# Include only fields in the list
for f in Age Sex
  set Filter.include.^f=True
endfor
```

This will cause the node to pass the fields *Age* and *Sex* and discard all others. Now suppose you run the same script again but name two different fields:

```
set Filter.default_include=False
# Include only fields in the list
for f in BP Na
  set Filter.include.^f=True
endfor
```

This will add two more fields to the filter so that a total of four fields are passed (*Age*, *Sex*, *BP*, *Na*). In other words, resetting the value of default_include to False doesn't automatically reset all fields.

Alternatively, if you now change default_include to True, either using a script or in the Filter node dialog box, this would flip the behavior so the four fields listed above would be discarded rather than included. When in doubt, experimenting with the controls in the Filter node dialog box may be helpful in understanding this interaction.

filternode properties	Data type	Property description
default_include	*flag*	Keyed property to specify whether the default behavior is to pass or filter fields: NODE.include.FIELDNAME An example is as follows: set mynode:filternode.default_include = false Note that setting this property does not automatically include or exclude all fields; it simply determines whether selected fields are included or excluded by default. See example below for additional comments.
include	*flag*	Keyed property for field inclusion and removal. Usage format: NODE.include.FIELDNAME An example is as follows: set mynode: filternode.include.Age = false
new_name	*string*	An example is as follows: set mynode: filternode.new_name. Age = "age"

historynode Properties

The History node creates new fields containing data from fields in previous records. History nodes are most often used for sequential data, such as time series data. Before using a History node, you may want to sort the data using a Sort node.

Example

```
create historynode
set :historynode.fields = ['Drug']
set :historynode.offset = 1
set :historynode.span = 3
set :historynode.unavailable = Discard
set :historynode.fill_with = "undef"
```

historynode properties	Data type	Property description
fields	[*field field field*]	Fields for which you want a history.
offset	*number*	Specifies the latest record (prior to the current record) from which you want to extract historical field values.
span	*number*	Specifies the number of prior records from which you want to extract values.

historynode properties	Data type	Property description
unavailable	Discard Leave Fill	For handling records that have no history values, usually referring to the first several records (at the top of the dataset) for which there are no previous records to use as a history.
fill_with	String Number	Specifies a value or string to be used for records where no history value is available.

partitionnode Properties

The Partition node generates a partition field, which splits the data into separate subsets for the training, testing, and validation stages of model building.

Example

```
create partitionnode
set :partitionnode.create_validation = True
set :partitionnode.training_size = 33
set :partitionnode.testing_size = 33
set :partitionnode.validation_size = 33
set :partitionnode.set_random_seed = True
set :partitionnode.random_seed = "123"
set :partitionnode.value_mode = System
```

partitionnode properties	Data type	Property description
new_name	*string*	Name of the partition field generated by the node.
create_validation	*flag*	Specifies whether a validation partition should be created.
training_size	*integer*	Percentage of records (0–100) to be allocated to the training partition.
testing_size	*integer*	Percentage of records (0–100) to be allocated to the testing partition.
validation_size	*integer*	Percentage of records (0–100) to be allocated to the validation partition. Ignored if a validation partition is not created.
training_label	*string*	Label for the training partition.
testing_label	*string*	Label for the testing partition.
validation_label	*string*	Label for the validation partition. Ignored if a validation partition is not created.
value_mode	System SystemAndLabel Label	Specifies the values used to represent each partition in the data. For example, the training sample can be represented by the system integer 1, the label Training, or a combination of the two, 1_Training.
set_random_seed	*Boolean*	Specifies whether a user-specified random seed should be used.

partitionnode properties	Data type	Property description
random_seed	*integer*	A user-specified random seed value. For this value to be used, set_random_seed must be set to True.
enable_sql_generation	*Boolean*	Specifies whether to use SQL pushback to assign records to partitions.
unique_field		Specifies the input field used to ensure that records are assigned to partitions in a random but repeatable way. For this value to be used, enable_sql_generation must be set to True.

reclassifynode Properties

The Reclassify node transforms one set of categorical values to another. Reclassification is useful for collapsing categories or regrouping data for analysis.

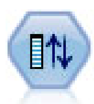
Example

```
create reclassifynode
set :reclassifynode.mode = Multiple
set :reclassifynode.replace_field = true
set :reclassifynode.field = "Drug"
set :reclassifynode.new_name = "Chemical"
set :reclassifynode.fields = [Drug, BP]
set :reclassifynode.name_extension = "reclassified"
set :reclassifynode.add_as = Prefix
set :reclassifynode.reclassify.'drugA' = 'Yes'
set :reclassifynode.use_default = True
set :reclassifynode.default = "BrandX"
set :reclassifynode.pick_list = [BrandX, Placebo, Generic]
```

reclassifynode properties	Data type	Property description
mode	Single Multiple	Single reclassifies the categories for one field. Multiple activates options enabling the transformation of more than one field at a time.
replace_field	*flag*	
field	*string*	Used only in Single mode.
new_name	*string*	Used only in Single mode.
fields	*[field1 field2 ... fieldn]*	Used only in Multiple mode.
name_extension	*string*	Used only in Multiple mode.
add_as	Suffix Prefix	Used only in Multiple mode.

reclassifynode properties	Data type	Property description
reclassify	*string*	Structured property for field values. Usage format: NODE.reclassify. OLD_VALUE An example is as follows: set :reclassifynode.reclassify.'drugB' = 'Yes'
use_default	*flag*	Use the default value.
default	*string*	Specify a default value.
pick_list	*[string string … string]*	Allows a user to import a list of known new values to populate the drop-down list in the table. An example is as follows: set :reclassify.pick_list = [fruit dairy cereals]

reordernode Properties



The Field Reorder node defines the natural order used to display fields downstream. This order affects the display of fields in a variety of places, such as tables, lists, and the Field Chooser. This operation is useful when working with wide datasets to make fields of interest more visible.

Example

```
create reordernode
set :reordernode.mode = Custom
set :reordernode.sort_by = Storage
set :reordernode.ascending = "false"
set :reordernode.start_fields = [Age Cholesterol]
set :reordernode.end_fields = [Drug]
```

reordernode properties	Data type	Property description
mode	Custom Auto	You can sort values automatically or specify a custom order.
sort_by	Name Type Storage	
ascending	*flag*	
start_fields	*[field1 field2 … fieldn]*	New fields are inserted after these fields.
end_fields	*[field1 field2 … fieldn]*	New fields are inserted before these fields.

restructurenode Properties

The Restructure node converts a nominal or flag field into a group of fields that can be populated with the values of yet another field. For example, given a field named *payment type*, with values of *credit*, *cash*, and *debit*, three new fields would be created (*credit*, *cash*, *debit*), each of which might contain the value of the actual payment made.

Example

```
create restructurenode
connect :typenode to :restructurenode
set :restructurenode.fields_from.Drug = ["drugA" "drugX"]
set :restructurenode.include_field_name = "True"
set :restructurenode.value_mode = "OtherFields"
set :restructurenode.value_fields = ["Age" "BP"]
```

restructurenode properties	Data type	Property description
fields_from	[*category category category*] all	For example, set :restructurenode.fields_from.Drug = [drugA drugB] creates fields called Drug_drugA and Drug_drugB. To use all categories of the specified field: set :restructurenode.fields_from.Drug = all
include_field_name	*flag*	Indicates whether to use the field name in the restructured field name.
value_mode	OtherFields Flags	Indicates the mode for specifying the values for the restructured fields. With OtherFields, you must specify which fields to use (see below). With Flags, the values are numeric flags.
value_fields	[*field field field*]	Required if value_mode is OtherFields. Specifies which fields to use as value fields.

rfmanalysisnode Properties

The Recency, Frequency, Monetary (RFM) Analysis node enables you to determine quantitatively which customers are likely to be the best ones by examining how recently they last purchased from you (recency), how often they purchased (frequency), and how much they spent over all transactions (monetary).

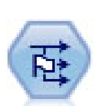
Example

```
create rfmanalysisnode
connect :rfmaggregatenode to :rfmanalysisnode
set :rfmanalysisnode.recency = Recency
set :rfmanalysisnode.frequency = Frequency
set :rfmanalysisnode.monetary = Monetary
set :rfmanalysisnode.tied_values_method = Next
set :rfmanalysisnode.recalculate_bins = IfNecessary
```

set :rfmanalysisnode.recency_thresholds = [1, 500, 800, 1500, 2000, 2500]

rfmanalysisnode properties	Data type	Property description
recency	*field*	Specify the recency field. This may be a date, timestamp, or simple number.
frequency	*field*	Specify the frequency field.
monetary	*field*	Specify the monetary field.
recency_bins	*integer*	Specify the number of recency bins to be generated.
recency_weight	*number*	Specify the weighting to be applied to recency data. The default is 100.
frequency_bins	*integer*	Specify the number of frequency bins to be generated.
frequency_weight	*number*	Specify the weighting to be applied to frequency data. The default is 10.
monetary_bins	*integer*	Specify the number of monetary bins to be generated.
monetary_weight	*number*	Specify the weighting to be applied to monetary data. The default is 1.
tied_values_method	Next Current	Specify which bin tied value data is to be put in.
recalculate_bins	Always IfNecessary	
add_outliers	*flag*	Available only if recalculate_bins is set to IfNecessary. If set, records that lie below the lower bin will be added to the lower bin, and records above the highest bin will be added to the highest bin.
binned_field	Recency Frequency Monetary	
recency_thresholds	*value value*	Available only if recalculate_bins is set to Always. Specify the upper and lower thresholds for the recency bins. The upper threshold of one bin is used as the lower threshold of the next—for example, [10 30 60] would define two bins, the first bin with upper and lower thresholds of 10 and 30, with the second bin thresholds of 30 and 60.
frequency_thresholds	*value value*	Available only if recalculate_bins is set to Always.
monetary_thresholds	*value value*	Available only if recalculate_bins is set to Always.

settoflagnode Properties



The Set to Flag node derives multiple flag fields based on the categorical values defined for one or more nominal fields.

Example

```
create settoflagnode
connect :typenode to :settoflag
set :settoflagnode.fields_from.Drug = ["drugA" "drugX"]
set :settoflagnode.true_value = "1"
set :settoflagnode.false_value = "0"
set :settoflagnode.use_extension = "True"
set :settoflagnode.extension = "Drug_Flag"
set :settoflagnode.add_as = Suffix
set :settoflagnode.aggregate = True
set :settoflagnode.keys = ['Cholesterol']
```

settoflagnode properties	Data type	Property description
fields_from	[*category category category*] all	For example, set :settoflagnode.fields_from.Drug = [drugA drugB] creates flag fields called Drug_drugA and Drug_drugB. To use all categories of the specified field: set :settoflagnode.fields_from.Drug = all
true_value	*string*	Specifies the true value used by the node when setting a flag. The default is T.
false_value	*string*	Specifies the false value used by the node when setting a flag. The default is F.
use_extension	*flag*	Use an extension as a suffix or prefix to the new flag field.
extension	*string*	
add_as	Suffix Prefix	Specifies whether the extension is added as a suffix or prefix.
aggregate	*flag*	Groups records together based on key fields. All flag fields in a group are enabled if any record is set to true.
keys	[*field field field*]	Key fields.

statisticstransformnode Properties

The Statistics Transform node runs a selection of IBM® SPSS® Statistics syntax commands against data sources in IBM® SPSS® Modeler. This node requires a licensed copy of SPSS Statistics.

The properties for this node are described under statisticstransformnode Properties on p. 292.

timeintervalsnode Properties

The Time Intervals node specifies intervals and creates labels (if needed) for modeling time series data. If values are not evenly spaced, the node can pad or aggregate values as needed to generate a uniform interval between records.

Example

```
create timeintervalsnode
set :timeintervalsnode.interval_type=SecondsPerDay
set :timeintervalsnode.days_per_week=4
set :timeintervalsnode.week_begins_on=Tuesday
set :timeintervalsnode.hours_per_day=10
set :timeintervalsnode.day_begins_hour=7
set :timeintervalsnode.day_begins_minute=5
set :timeintervalsnode.day_begins_second=17
set :timeintervalsnode.mode=Label
set :timeintervalsnode.year_start=2005
set :timeintervalsnode.month_start=January
set :timeintervalsnode.day_start=4
set :timeintervalsnode.pad.AGE=MeanOfRecentPoints
set :timeintervalsnode.agg_mode=Specify
set :timeintervalsnode.agg_set_default=Last
```

timeintervalsnode properties	**Data type**	**Property description**
interval_type	None Periods CyclicPeriods Years Quarters Months DaysPerWeek DaysNonPeriodic HoursPerDay HoursNonPeriodic MinutesPerDay MinutesNonPeriodic SecondsPerDay SecondsNonPeriodic	
mode	Label Create	Specifies whether you want to label records consecutively or build the series based on a specified date, timestamp, or time field.
field	*field*	When building the series from the data, specifies the field that indicates the date or time for each record.
period_start	*integer*	Specifies the starting interval for periods or cyclic periods
cycle_start	*integer*	Starting cycle for cyclic periods.
year_start	*integer*	For interval types where applicable, year in which the first interval falls.
quarter_start	*integer*	For interval types where applicable, quarter in which the first interval falls.

timeintervalsnode properties	Data type	Property description
month_start	January February March April May June July August September October November December	
day_start	*integer*	
hour_start	*integer*	
minute_start	*integer*	
second_start	*integer*	
periods_per_cycle	*integer*	For cyclic periods, number within each cycle.
fiscal_year_begins	January February March April May June July August September October November December	For quarterly intervals, specifies the month when the fiscal year begins.
week_begins_on	Sunday Monday Tuesday Wednesday Thursday Friday Saturday Sunday	For periodic intervals (days per week, hours per day, minutes per day, and seconds per day), specifies the day on which the week begins.
day_begins_hour	*integer*	For periodic intervals (hours per day, minutes per day, seconds per day), specifies the hour when the day begins. Can be used in combination with day_begins_minute and day_begins_second to specify an exact time such as *8:05:01*. See usage example below.
day_begins_minute	*integer*	For periodic intervals (hours per day, minutes per day, seconds per day), specifies the minute when the day begins (for example, the *5* in *8:05*).
day_begins_second	*integer*	For periodic intervals (hours per day, minutes per day, seconds per day), specifies the second when the day begins (for example, the *17* in *8:05:17*).

timeintervalsnode properties	Data type	Property description
days_per_week	*integer*	For periodic intervals (days per week, hours per day, minutes per day, and seconds per day), specifies the number of days per week.
hours_per_day	*integer*	For periodic intervals (hours per day, minutes per day, and seconds per day), specifies the number of hours in the day.
interval_increment	1 2 3 4 5 6 10 15 20 30	For minutes per day and seconds per day, specifies the number of minutes or seconds to increment for each record.
field_name_extension	*string*	
field_name_extension_as_prefix	*flag*	
date_format	"DDMMYY" "MMDDYY" "YYMMDD" "YYYYMMDD" "YYYYDDD" DAY MONTH "DD-MM-YY" "DD-MM-YYYY" "MM-DD-YY" "MM-DD-YYYY" "DD-MON-YY" "DD-MON-YYYY" "YYYY-MM-DD" "DD.MM.YY" "DD.MM.YYYY" "MM.DD.YY" "MM.DD.YYYY" "DD.MON.YY" "DD.MON.YYYY" "DD/MM/YY" "DD/MM/YYYY" "MM/DD/YY" "MM/DD/YYYY" "DD/MON/YY" "DD/MON/YYYY" MON YYYY q Q YYYY ww WK YYYY	

timeintervalsnode properties	Data type	Property description
time_format	"HHMMSS" "HHMM" "MMSS" "HH:MM:SS" "HH:MM" "MM:SS" "(H)H:(M)M:(S)S" "(H)H:(M)M" "(M)M:(S)S" "HH.MM.SS" "HH.MM" "MM.SS" "(H)H.(M)M.(S)S" "(H)H.(M)M" "(M)M.(S)S"	
aggregate	Mean Sum Mode Min Max First Last TrueIfAnyTrue	Specifies the aggregation method for a field (for example, aggregate.AGE=Mean).
pad	Blank MeanOfRecentPoints True False	Specifies the padding method for a field (for example, pad.AGE=MeanOfRecentPoints).
agg_mode	All Specify	Specifies whether to aggregate or pad all fields with default functions as needed or specify the fields and functions to use.
agg_range_default	Mean Sum Mode Min Max	Specifies the default function to use when aggregating continuous fields.
agg_set_default	Mode First Last	Specifies the default function to use when aggregating nominal fields.
agg_flag_default	TrueIfAnyTrue Mode First Last	
pad_range_default	Blank MeanOfRecentPoints	Specifies the default function to use when padding continuous fields.
pad_set_default	Blank MostRecentValue	
pad_flag_default	Blank True False	
max_records_to_create	*integer*	Specifies the maximum number of records to create when padding the series.
estimation_from_beginning	*flag*	
estimation_to_end	*flag*	

timeintervalsnode properties	Data type	Property description
estimation_start_offset	*integer*	
estimation_num_holdouts	*integer*	
create_future_records	*flag*	
num_future_records	*integer*	
create_future_field	*flag*	
future_field_name	*string*	

transposenode Properties

The Transpose node swaps the data in rows and columns so that records become fields and fields become records.

Example

```
create transposenode
set :transposenode.transposed_names=Read
set :transposenode.read_from_field="TimeLabel"
set :transposenode.max_num_fields="1000"
set :transposenode.id_field_name="ID"
```

transposenode properties	Data type	Property description
transposed_names	Prefix Read	New field names can be generated automatically based on a specified prefix, or they can be read from an existing field in the data.
prefix	*string*	
num_new_fields	*integer*	When using a prefix, specifies the maximum number of new fields to create.
read_from_field	*field*	Field from which names are read. This must be an instantiated field or an error will occur when the node is executed.
max_num_fields	*integer*	When reading names from a field, specifies an upper limit to avoid creating an inordinately large number of fields.
transpose_type	Numeric String Custom	By default, only continuous (numeric range) fields are transposed, but you can choose a custom subset of numeric fields or transpose all string fields instead.
transpose_fields	[*field field field*]	Specifies the fields to transpose when the Custom option is used.
id_field_name	*field*	

typenode Properties

The Type node specifies field metadata and properties. For example, you can specify a measurement level (continuous, nominal, ordinal, or flag) for each field, set options for handling missing values and system nulls, set the role of a field for modeling purposes, specify field and value labels, and specify values for a field.

Example

```
create typenode
connect :variablefilenode to :typenode
set :typenode.check.'Cholesterol' = Coerce
set :typenode.direction.'Drug' = Input
set :typenode.type.K = Range
set :typenode.values.Drug = [drugA drugB drugC drugD drugX drugY drugZ]
set :typenode.null_missing.BP = false
set :typenode.whitespace_missing.BP = "false"
set :typenode.description.BP = "Blood Pressure"
set :typenode.value_labels.BP = [{HIGH 'High Blood Pressure'}{NORMAL 'normal blood pressure'}]
set :typenode.display_places.K = 5
set :typenode.export_places.K = 2
set :typenode.grouping_symbol.Drug = None
set :typenode.column_width.Cholesterol = 25
set :typenode.justify.Cholesterol = Right
```

Note that in some cases you may need to fully instantiate the Type node in order for other nodes to work correctly, such as the fields from property of the Set to Flag node. You can simply connect a Table node and execute it to instantiate the fields:

```
create tablenode
connect :typenode to :tablenode
execute :tablenode
delete :tablenode
```

typenode properties	Data type	Property description
direction	Input Target Both None Partition Split Frequency RecordID	Keyed property for field roles. Usage format: NODE.direction.FIELDNAME *Note*: The values In and Out are now deprecated. Support for them may be withdrawn in a future release.
type	Range Flag Set Typeless Discrete Ordered Set Default	Type of field. Setting type to Default will clear any values parameter setting, and if value_mode has the value Specify, it will be reset to Read. If value_mode is set to Pass or Read, setting type will not affect value_mode. Usage format: NODE.type.FIELDNAME

typenode properties	Data type	Property description
storage	Unknown String Integer Real Time Date Timestamp	Read-only keyed property for field storage type. Usage format: NODE.storage.FIELDNAME
check	None Nullify Coerce Discard Warn Abort	Keyed property for field type and range checking. Usage format: NODE.check.FIELDNAME
values	[*value value*]	For continuous fields, the first value is the minimum, and the last value is the maximum. For nominal fields, specify all values. For flag fields, the first value represents *false*, and the last value represents *true*. Setting this property automatically sets the value_mode property to Specify. Usage format: NODE.values.FIELDNAME
value_mode	Read Pass Read+ Current Specify	Determines how values are set. Note that you cannot set this property to Specify directly; to use specific values, set the values property. Usage format: NODE.value_mode.FIELDNAME
extend_values	*flag*	Applies when value_mode is set to Read. Set to T to add newly read values to any existing values for the field. Set to F to discard existing values in favor of the newly read values. Usage format: NODE.extend_values.FIELDNAME
enable_missing	*flag*	When set to T, activates tracking of missing values for the field. Usage format: NODE.enable_missing.FIELDNAME
missing_values	[*value value ...*]	Specifies data values that denote missing data. Usage format: NODE.missing_values.FIELDNAME
range_missing	*flag*	Specifies whether a missing-value (blank) range is defined for a field.
missing_lower	*string*	When range_missing is true, specifies the lower bound of the missing-value range.
missing_upper	*string*	When range_missing is true, specifies the upper bound of the missing-value range.
null_missing	*flag*	When set to T, *nulls* (undefined values that are displayed as $null$ in the software) are considered missing values. Usage format: NODE.null_missing.FIELDNAME

typenode properties	Data type	Property description
whitespace_missing	*flag*	When set to T, values containing only white space (spaces, tabs, and new lines) are considered missing values. Usage format: NODE.whitespace _missing. FIELDNAME
description	*string*	Specifies the description for a field.
value_labels	*[{Value LabelString} { Value LabelString} ...]*	Used to specify labels for value pairs. An example is as follows: set :typenode.value_labels.'Drug'=[{drugA label1} {drugB label2}]
display_places	*integer*	Sets the number of decimal places for the field when displayed (applies only to fields with REAL storage). A value of –1 will use the stream default. Usage format: NODE.display_places.FIELDNAME
export_places	*integer*	Sets the number of decimal places for the field when exported (applies only to fields with REAL storage). A value of –1 will use the stream default. Usage format: NODE.export_places.FIELDNAME
decimal_separator	DEFAULT PERIOD COMMA	Sets the decimal separator for the field (applies only to fields with REAL storage). Usage format: NODE.decimal_separator.FIELDNAME
date_format	"DDMMYY" "MMDDYY" "YYMMDD" "YYYYMMDD" "YYYYDDD" DAY MONTH "DD-MM-YY" "DD-MM-YYYY" "MM-DD-YY" "MM-DD-YYYY" "DD-MON-YY" "DD-MON-YYYY" "YYYY-MM-DD" "DD.MM.YY" "DD.MM.YYYY" "MM.DD.YY" "MM.DD.YYYY" "DD.MON.YY" "DD.MON.YYYY" "DD/MM/YY" "DD/MM/YYYY" "MM/DD/YY" "MM/DD/YYYY" "DD/MON/YY" "DD/MON/YYYY" MON YYYY q Q YYYY ww WK YYYY	Sets the date format for the field (applies only to fields with DATE or TIMESTAMP storage). Usage format: NODE.date_format.FIELDNAME An example is as follows: set :tablenode.date_format.'LaunchDate' = "DDMMYY"

typenode properties	Data type	Property description
time_format	"HHMMSS" "HHMM" "MMSS" "HH:MM:SS" "HH:MM" "MM:SS" "(H)H:(M)M:(S)S" "(H)H:(M)M" "(M)M:(S)S" "HH.MM.SS" "HH.MM" "MM.SS" "(H)H.(M)M.(S)S" "(H)H.(M)M" "(M)M.(S)S"	Sets the time format for the field (applies only to fields with TIME or TIMESTAMP storage). Usage format: NODE.time_format.FIELDNAME An example is as follows: set :tablenode.time_format.'BOF_enter' = "HHMMSS"
number_format	DEFAULT STANDARD SCIENTIFIC CURRENCY	Sets the number display format for the field. Usage format: NODE.number_format.FIELDNAME
standard_places	*integer*	Sets the number of decimal places for the field when displayed in standard format. A value of –1 will use the stream default. Note that the existing display_places slot will also change this but is now deprecated. Usage format: NODE.standard_places.FIELDNAME
scientific_places	*integer*	Sets the number of decimal places for the field when displayed in scientific format. A value of –1 will use the stream default. Usage format: NODE.scientific_places.FIELDNAME
currency_places	*integer*	Sets the number of decimal places for the field when displayed in currency format. A value of –1 will use the stream default. Usage format: NODE.currency_places.FIELDNAME
grouping_symbol	DEFAULT NONE LOCALE PERIOD COMMA SPACE	Sets the grouping symbol for the field. Usage format: NODE.grouping_symbol.FIELDNAME
column_width	*integer*	Sets the column width for the field. A value of –1 will set column width to Auto. Usage format: NODE.column_width.FIELDNAME
justify	AUTO CENTER LEFT RIGHT	Sets the column justification for the field. Usage format: NODE.justify.FIELDNAME

Chapter

15

Graph Node Properties

Graph Node Common Properties

This section describes the properties available for graph nodes, including common properties and properties that are specific to each node type.

Common graph node properties	Data type	Property description
title	*string*	Specifies the title. Example: “This is a title.”
caption	*string*	Specifies the caption. Example: “This is a caption.”
output_mode	Screen File	Specifies whether output from the graph node is displayed or written to a file.
output_format	BMP JPEG PNG HTML output (*.cou*)	Specifies the type of output. The exact type of output allowed for each node varies.
full_filename	*string*	Specifies the target path and filename for output generated from the graph node.
use_graph_size	*flag*	Controls whether the graph is sized explicitly, using the width and height properties below. Affects only graphs that are output to screen. Not available for the Distribution node.
graph_width	*number*	When use_graph_size is True, sets the graph width in pixels.
graph_height	*number*	When use_graph_size is True, sets the graph height in pixels.

Notes

Turning off optional fields. Optional fields, such as an overlay field for plots, can be turned off by setting the property value to " " (empty string), as shown in the following example:

```
set :plotnode.color_field = ""
```

Specifying colors. The colors for titles, captions, backgrounds, and labels can be specified by using the hexadecimal strings starting with the hash (#) symbol. For example, to set the graph background to sky blue, you would use the following statement:

```
set mygraph.graph_background="#87CEEB"
```

Here, the first two digits, 87, specify the red content; the middle two digits, CE, specify the green content; and the last two digits, EB, specify the blue content. Each digit can take a value in the range 0–9 or A–F. Together, these values can specify a red-green-blue, or RGB, color.

Note: When specifying colors in RGB, you can use the Field Chooser in the user interface to determine the correct color code. Simply hover over the color to activate a ToolTip with the desired information.

collectionnode Properties

The Collection node shows the distribution of values for one numeric field relative to the values of another. (It creates graphs that are similar to histograms.) It is useful for illustrating a variable or field whose values change over time. Using 3-D graphing, you can also include a symbolic axis displaying distributions by category.

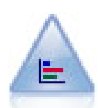
Example

```
create collectionnode
position :collectionnode at ^posX ^posY
# "Plot" tab
set :collectionnode.three_D = True
set :collectionnode.collect_field = 'Drug'
set :collectionnode.over_field = 'Age'
set :collectionnode.by_field = 'BP'
set :collectionnode.operation = Sum
# "Overlay" section
set :collectionnode.color_field = 'Drug'
set :collectionnode.panel_field = 'Sex'
set :collectionnode.animation_field = ''
# "Options" tab
set :collectionnode.range_mode = Automatic
set :collectionnode.range_min = 1
set :collectionnode.range_max = 100
set :collectionnode.bins = ByNumber
set :collectionnode.num_bins = 10
set :collectionnode.bin_width = 5
```

collectionnode properties	Data type	Property description
over_field	*field*	
over_label_auto	*flag*	
over_label	*string*	
collect_field	*field*	
collect_label_auto	*flag*	
collect_label	*string*	
three_D	*flag*	
by_field	*field*	
by_label_auto	*flag*	
by_label	*string*	
operation	Sum Mean Min Max SDev	

collectionnode properties	Data type	Property description
color_field	*string*	
panel_field	*string*	
animation_field	*string*	
range_mode	Automatic UserDefined	
range_min	*number*	
range_max	*number*	
bins	ByNumber ByWidth	
num_bins	*number*	
bin_width	*number*	
use_grid	*flag*	
graph_background	*color*	Standard graph colors are described at the beginning of this section.
page_background	*color*	Standard graph colors are described at the beginning of this section.

distributionnode Properties



The Distribution node shows the occurrence of symbolic (categorical) values, such as mortgage type or gender. Typically, you might use the Distribution node to show imbalances in the data, which you could then rectify using a Balance node before creating a model.

Example

```
create distributionnode
# "Plot" tab
set :distributionnode.plot = Flags
set :distributionnode.x_field = 'Age'
set :distributionnode.color_field = 'Drug'
set :distributionnode.normalize = True
set :distributionnode.sort_mode = ByOccurence
set :distributionnode.use_proportional_scale = True
```

distributionnode properties	Data type	Property description
plot	SelectedFields Flags	
x_field	*field*	
color_field	*field*	Overlay field.
normalize	*flag*	
sort_mode	ByOccurence Alphabetic	
use_proportional_scale	*flag*	

evaluationnode Properties

The Evaluation node helps to evaluate and compare predictive models. The evaluation chart shows how well models predict particular outcomes. It sorts records based on the predicted value and confidence of the prediction. It splits the records into groups of equal size (**quantiles**) and then plots the value of the business criterion for each quantile from highest to lowest. Multiple models are shown as separate lines in the plot.

Example

```
create evaluationnode
position :evaluationnode at ^posX ^posY
# "Plot" tab
set :evaluationnode.chart_type = Gains
set :evaluationnode.cumulative = False
set :evaluationnode.field_detection_method = Name
set :evaluationnode.inc_baseline = True
set :evaluationnode.n_tile = Deciles
set :evaluationnode.style = Point
set :evaluationnode.point_type = Dot
set :evaluationnode.use_fixed_cost = True
set :evaluationnode.cost_value = 5.0
set :evaluationnode.cost_field = 'Na'
set :evaluationnode.use_fixed_revenue = True
set :evaluationnode.revenue_value = 30.0
set :evaluationnode.revenue_field = 'Age'
set :evaluationnode.use_fixed_weight = True
set :evaluationnode.weight_value = 2.0
set :evaluationnode.weight_field = 'K'
```

evaluationnode properties	Data type	Property description
chart_type	Gains Response Lift Profit ROI	
inc_baseline	*flag*	
field_detection_method	Metadata Name	
use_fixed_cost	*flag*	
cost_value	*number*	
cost_field	*string*	
use_fixed_revenue	*flag*	
revenue_value	*number*	
revenue_field	*string*	
use_fixed_weight	*flag*	
weight_value	*number*	
weight_field	*field*	

evaluationnode properties	Data type	Property description
n_tile	Quartiles Quintles Deciles Vingtiles Percentiles 1000-tiles	
cumulative	*flag*	
style	Line Point	
point_type	Rectangle Dot Triangle Hexagon Plus Pentagon Star BowTie HorizontalDash VerticalDash IronCross Factory House Cathedral OnionDome ConcaveTriangle OblateGlobe CatEye FourSidedPillow RoundRectangle Fan	
export_data	*flag*	
data_filename	*string*	
delimiter	*string*	
new_line	*flag*	
inc_field_names	*flag*	
inc_best_line	*flag*	
inc_business_rule	*flag*	
business_rule_condition	*string*	
plot_score_fields	*flag*	
score_fields	*[field1 ... fieldN]*	
target_field	*field*	
use_hit_condition	*flag*	
hit_condition	*string*	
use_score_expression	*flag*	
score_expression	*string*	
caption_auto	*flag*	

graphboardnode Properties

The Graphboard node offers many different types of graphs in one single node. Using this node, you can choose the data fields you want to explore and then select a graph from those available for the selected data. The node automatically filters out any graph types that would not work with the field choices.

Note: If you set a property that is not valid for the graph type (for example, specifying y_field for a histogram), that property is ignored.

Example

```
create graphboardnode
connect DRUG4n to :graphboardnode
set :graphboardnode.graph_type="Line"
set :graphboardnode.x_field = "K"
set :graphboardnode.y_field = "Na"
execute :graphboardnode
```

graphboard properties	**Data type**	**Property description**
graph_type	2DDotplot 3DArea 3DBar 3DDensity 3DHistogram 3DPie 3DScatterplot Area ArrowMap Bar BarCounts BarCountsMap BarMap BinnedScatter Boxplot Bubble ChoroplethMeans ChoroplethMedians ChoroplethSums ChoroplethValues ChoroplethCounts CoordinateMap CoordinateChoroplethMeans CoordinateChoroplethMedians CoordinateChoroplethSums CoordinateChoroplethValues CoordinateChoroplethCounts Dotplot Heatmap HexBinScatter Histogram Line LineChartMap LineOverlayMap Parallel Path	Identifies the graph type.

graphboard properties	Data type	Property description
	Pie PieCountMap PieCounts PieMap PointOverlayMap PolygonOverlayMap Ribbon Scatterplot SPLOM Surface	
x_field	*field*	Specifies a custom label for the *x* axis. Available only for labels.
y_field	*field*	Specifies a custom label for the *y* axis. Available only for labels.
z_field	*field*	Used in some 3-D graphs.
color_field	*field*	Used in heat maps.
size_field	*field*	Used in bubble plots.
categories_field	*field*	
values_field	*field*	
rows_field	*field*	
columns_field	*field*	
fields	*field*	
start_longitude_field	*field*	Used with arrows on a reference map.
end_longitude_field	*field*	
start_latitude_field	*field*	
end_latitude_field	*field*	
data_key_field	*field*	Used in various maps.
panelrow_field	*string*	
panelcol_field	*string*	
animation_field	*string*	
longitude_field	*field*	Used with co-ordinates on maps.
latitude_field	*field*	
map_color_field	*field*	

histogramnode Properties

The Histogram node shows the occurrence of values for numeric fields. It is often used to explore the data before manipulations and model building. Similar to the Distribution node, the Histogram node frequently reveals imbalances in the data.

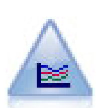
Example

```
create histogramnode
position :histogramnode at ^posX ^posY
# "Plot" tab
set :histogramnode.field = 'Drug'
```

```
set :histogramnode.color_field = 'Drug'
set :histogramnode.panel_field = 'Sex'
set :histogramnode.animation_field = ''
# "Options" tab
set :histogramnode.range_mode = Automatic
set :histogramnode.range_min = 1.0
set :histogramnode.range_max = 100.0
set :histogramnode.num_bins = 10
set :histogramnode.bin_width = 10
set :histogramnode.normalize = True
set :histogramnode.separate_bands = False
```

histogramnode properties	**Data type**	**Property description**
field	*field*	
color_field	*field*	
panel_field	*field*	
animation_field	*field*	
range_mode	Automatic UserDefined	
range_min	*number*	
range_max	*number*	
bins	ByNumber ByWidth	
num_bins	*number*	
bin_width	*number*	
normalize	*flag*	
separate_bands	*flag*	
x_label_auto	*flag*	
x_label	*string*	
y_label_auto	*flag*	
y_label	*string*	
use_grid	*flag*	
graph_background	*color*	Standard graph colors are described at the beginning of this section.
page_background	*color*	Standard graph colors are described at the beginning of this section.
normal_curve	*flag*	Indicates whether the normal distribution curve should be shown on the output.

multiplotnode Properties



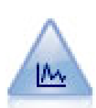
The Multiplot node creates a plot that displays multiple *Y* fields over a single *X* field. The *Y* fields are plotted as colored lines; each is equivalent to a Plot node with Style set to Line and X Mode set to Sort. Multiplots are useful when you want to explore the fluctuation of several variables over time.

Example

```
create multiplotnode
# "Plot" tab
set :multiplotnode.x_field = 'Age'
set :multiplotnode.y_fields = ['Drug' 'BP']
set :multiplotnode.panel_field = 'Sex'
# "Overlay" section
set :multiplotnode.animation_field = ''
set :multiplotnode.tooltip = "test"
set :multiplotnode.normalize = True
set :multiplotnode.use_overlay_expr = False
set :multiplotnode.overlay_expression = "test"
set :multiplotnode.records_limit = 500
set :multiplotnode.if_over_limit = PlotSample
```

multiplotnode properties	Data type	Property description
x_field	*field*	
y_fields	[*field field field*]	
panel_field	*field*	
animation_field	*field*	
normalize	*flag*	
use_overlay_expr	*flag*	
overlay_expression	*string*	
records_limit	*number*	
if_over_limit	PlotBins PlotSample PlotAll	
x_label_auto	*flag*	
x_label	*string*	
y_label_auto	*flag*	
y_label	*string*	
use_grid	*flag*	
graph_background	*color*	Standard graph colors are described at the beginning of this section.
page_background	*color*	Standard graph colors are described at the beginning of this section.

plotnode Properties



The Plot node shows the relationship between numeric fields. You can create a plot by using points (a scatterplot) or lines.

Example

```
create plotnode
# "Plot" tab
```

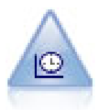
```
set :plotnode.three_D = True
set :plotnode.x_field = 'BP'
set :plotnode.y_field = 'Cholesterol'
set :plotnode.z_field = 'Drug'
# "Overlay" section
set :plotnode.color_field = 'Drug'
set :plotnode.size_field = 'Age'
set :plotnode.shape_field = ''
set :plotnode.panel_field = 'Sex'
set :plotnode.animation_field = 'BP'
set :plotnode.transp_field = ''
set :plotnode.style = Point
# "Output" tab
set :plotnode.output_mode =
set :plotnode.output_format = JPEG
set :plotnode.full_filename = "C:/Temp/Graph_Output/plot_output.jpeg"
```

plotnode properties	Data type	Property description
x_field	*field*	Specifies a custom label for the *x* axis. Available only for labels.
y_field	*field*	Specifies a custom label for the *y* axis. Available only for labels.
three_D	*flag*	Specifies a custom label for the *y* axis. Available only for labels in 3-D graphs.
z_field	*field*	
color_field	*field*	Overlay field.
size_field	*field*	
shape_field	*field*	
panel_field	*field*	Specifies a nominal or flag field for use in making a separate chart for each category. Charts are paneled together in one output window.
animation_field	*field*	Specifies a nominal or flag field for illustrating data value categories by creating a series of charts displayed in sequence using animation.
transp_field	*field*	Specifies a field for illustrating data value categories by using a different level of transparency for each category. Not available for line plots.
overlay_type	None Smoother Function	Specifies whether an overlay function or LOESS smoother is displayed.
overlay_expression	*string*	Specifies the expression used when overlay_type is set to Function.
style	Point Line	

plotnode properties	**Data type**	**Property description**
point_type	Rectangle Dot Triangle Hexagon Plus Pentagon Star BowTie HorizontalDash VerticalDash IronCross Factory House Cathedral OnionDome ConcaveTriangle OblateGlobe CatEye FourSidedPillow RoundRectangle Fan	
x_mode	Sort Overlay AsRead	
x_range_mode	Automatic UserDefined	
x_range_min	*number*	
x_range_max	*number*	
y_range_mode	Automatic UserDefined	
y_range_min	*number*	
y_range_max	*number*	
z_range_mode	Automatic UserDefined	
z_range_min	*number*	
z_range_max	*number*	
jitter	*flag*	
records_limit	*number*	
if_over_limit	PlotBins PlotSample PlotAll	
x_label_auto	*flag*	
x_label	*string*	
y_label_auto	*flag*	
y_label	*string*	
z_label_auto	*flag*	
z_label	*string*	
use_grid	*flag*	
graph_background	*color*	Standard graph colors are described at the beginning of this section.

plotnode properties	Data type	Property description
page_background	*color*	Standard graph colors are described at the beginning of this section.
use_overlay_expr	*flag*	Deprecated in favor of overlay_type.

timeplotnode Properties



The Time Plot node displays one or more sets of time series data. Typically, you would first use a Time Intervals node to create a *TimeLabel* field, which would be used to label the *x* axis.

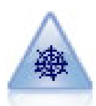
Example

```
create timeplotnode
set :timeplotnode.y_fields = ['sales' 'men' 'women']
set :timeplotnode.panel = True
set :timeplotnode.normalize = True
set :timeplotnode.line = True
set :timeplotnode.smoother = True
set :timeplotnode.use_records_limit = True
set :timeplotnode.records_limit = 2000
# Appearance settings
set :timeplotnode.symbol_size = 2.0
```

timeplotnode properties	Data type	Property description
plot_series	Series Models	
use_custom_x_field	*flag*	
x_field	*field*	
y_fields	[*field field field*]	
panel	*flag*	
normalize	*flag*	
line	*flag*	
points	*flag*	
point_type	Rectangle Dot Triangle Hexagon Plus Pentagon Star BowTie HorizontalDash VerticalDash IronCross Factory House Cathedral OnionDome ConcaveTriangle	

timeplotnode properties	Data type	Property description
	OblateGlobe CatEye FourSidedPillow RoundRectangle Fan	
smoother	*flag*	You can add smoothers to the plot only if you set panel to True.
use_records_limit	*flag*	
records_limit	*integer*	
symbol_size	*number*	Specifies a symbol size. For example, set :webnode.symbol_size = 5.5 creates a symbol size larger than the default.
panel_layout	Horizontal Vertical	

webnode Properties



The Web node illustrates the strength of the relationship between values of two or more symbolic (categorical) fields. The graph uses lines of various widths to indicate connection strength. You might use a Web node, for example, to explore the relationship between the purchase of a set of items at an e-commerce site.

Example

```
create webnode
# "Plot" tab
set :webnode.use_directed_web = True
set :webnode.to_field = 'Drug'
set :webnode.fields = ['BP' 'Cholesterol' 'Sex' 'Drug']
set :webnode.from_fields = ['BP' 'Cholesterol' 'Sex']
set :webnode.true_flags_only = False
set :webnode.line_values = Absolute
set :webnode.strong_links_heavier = True
# "Options" tab
set :webnode.max_num_links = 300
set :webnode.links_above = 10
set :webnode.num_links = ShowAll
set :webnode.discard_links_min = True
set :webnode.links_min_records = 5
set :webnode.discard_links_max = True
set :webnode.weak_below = 10
set :webnode.strong_above = 19
set :webnode.link_size_continuous = True
set :webnode.web_display = Circular
```

webnode properties	Data type	Property description
use_directed_web	*flag*	
fields	[*field field field*]	

webnode properties	**Data type**	**Property description**
to_field	*field*	
from_fields	[*field field field*]	
true_flags_only	*flag*	
line_values	Absolute OverallPct PctLarger PctSmaller	
strong_links_heavier	*flag*	
num_links	ShowMaximum ShowLinksAbove ShowAll	
max_num_links	*number*	
links_above	*number*	
discard_links_min	*flag*	
links_min_records	*number*	
discard_links_max	*flag*	
links_max_records	*number*	
weak_below	*number*	
strong_above	*number*	
link_size_continuous	*flag*	
web_display	Circular Network Directed Grid	
graph_background	*color*	Standard graph colors are described at the beginning of this section.
symbol_size	*number*	Specifies a symbol size. For example, set :webnode.symbol_size = 5.5 creates a symbol size larger than the default.

Chapter

16

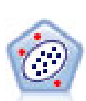
Modeling Node Properties

Common Modeling Node Properties

The following properties are common to some or all modeling nodes. Any exceptions are noted in the documentation for individual modeling nodes as appropriate.

Property	**Values**	**Property description**
custom_fields	*flag*	If true, allows you to specify target, input, and other fields for the current node. If false, the current settings from an upstream Type node are used.
target or targets	*field* or [*field1 ... fieldN*]	Specifies a single target field or multiple target fields depending on the model type.
inputs	[*field1 ... fieldN*]	Input or predictor fields used by the model.
partition	*field*	
use_partitioned_data	*flag*	If a partition field is defined, this option ensures that only data from the training partition is used to build the model.
use_split_data	*flag*	
splits	*[field1 ... fieldN]*	Specifies the field or fields to use for split modeling. Effective only if use_split_data is set to True.
use_frequency	*flag*	Weight and frequency fields are used by specific models as noted for each model type.
frequency_field	*field*	
use_weight	*flag*	
weight_field	*field*	
use_model_name	*flag*	
model_name	*string*	Custom name for new model.
mode	Simple Expert	

anomalydetectionnode Properties



The Anomaly Detection node identifies unusual cases, or outliers, that do not conform to patterns of “normal” data. With this node, it is possible to identify outliers even if they do not fit any previously known patterns and even if you are not exactly sure what you are looking for.

Example

```
create anomalydetectionnode
set :anomalydetectionnode.anomaly_method=PerRecords
set :anomalydetectionnode.percent_records=95
set :anomalydetectionnode.mode=Expert
set :anomalydetectionnode.peer_group_num_auto=true
set :anomalydetectionnode.min_num_peer_groups=3
set :anomalydetectionnode.max_num_peer_groups=10
```

anomalydetectionnode Properties	Values	Property description
inputs	*[field1 ... fieldN]*	Anomaly Detection models screen records based on the specified input fields. They do not use a target field. Weight and frequency fields are also not used. For more information, see the topic Common Modeling Node Properties on p. 181.
mode	Expert Simple	
anomaly_method	IndexLevel PerRecords NumRecords	Specifies the method used to determine the cutoff value for flagging records as anomalous.
index_level	*number*	Specifies the minimum cutoff value for flagging anomalies.
percent_records	*number*	Sets the threshold for flagging records based on the percentage of records in the training data.
num_records	*number*	Sets the threshold for flagging records based on the number of records in the training data.
num_fields	*integer*	The number of fields to report for each anomalous record.
impute_missing_values	*flag*	
adjustment_coeff	*number*	Value used to balance the relative weight given to continuous and categorical fields in calculating the distance.
peer_group_num_auto	*flag*	Automatically calculates the number of peer groups.
min_num_peer_groups	*integer*	Specifies the minimum number of peer groups used when peer_group_num_auto is set to True.
max_num_per_groups	*integer*	Specifies the maximum number of peer groups.
num_peer_groups	*integer*	Specifies the number of peer groups used when peer_group_num_auto is set to False.

anomalydetectionnode Properties	Values	Property description
noise_level	*number*	Determines how outliers are treated during clustering. Specify a value between 0 and 0.5.
noise_ratio	*number*	Specifies the portion of memory allocated for the component that should be used for noise buffering. Specify a value between 0 and 0.5.

apriorinode Properties

The Apriori node extracts a set of rules from the data, pulling out the rules with the highest information content. Apriori offers five different methods of selecting rules and uses a sophisticated indexing scheme to process large data sets efficiently. For large problems, Apriori is generally faster to train; it has no arbitrary limit on the number of rules that can be retained, and it can handle rules with up to 32 preconditions. Apriori requires that input and output fields all be categorical but delivers better performance because it is optimized for this type of data.

Example

```
create apriorinode
# "Fields" tab
set :apriorinode.custom_fields = True
set :apriorinode.use_transactional_data = True
set :apriorinode.id_field = 'Age'
set :apriorinode.contiguous = True
set :apriorinode.content_field = 'Drug'
# These seem to have changed, used to be:
#help set :apriorinode.consequents = ['Age']
#help set :apriorinode.antecedents = ['BP' 'Cholesterol' 'Drug']
# now it seems we have;
#help set :apriorinode.content = ['Age']
set :apriorinode.partition = Test
# "Model" tab
set :apriorinode.use_model_name = False
set :apriorinode.model_name = "Apriori_bp_choles_drug"
set :apriorinode.min_supp = 7.0
set :apriorinode.min_conf = 30.0
set :apriorinode.max_antecedents = 7
set :apriorinode.true_flags = False
set :apriorinode.optimize = Memory
# "Expert" tab
set :apriorinode.mode = Expert
set :apriorinode.evaluation = ConfidenceRatio
```

```
set :apriorinode.lower_bound = 7
```

apriorinode Properties	Values	Property description
consequents	*field*	Apriori models use Consequents and Antecedents in place of the standard target and input fields. Weight and frequency fields are not used. For more information, see the topic Common Modeling Node Properties on p. 181.
antecedents	[*field1 ... fieldN*]	
min_supp	*number*	
min_conf	*number*	
max_antecedents	*number*	
true_flags	*flag*	
optimize	Speed Memory	
use_transactional_data	*flag*	
contiguous	*flag*	
id_field	*string*	
content_field	*string*	
mode	Simple Expert	
evaluation	RuleConfidence DifferenceToPrior ConfidenceRatio InformationDifference NormalizedChiSquare	
lower_bound	*number*	
optimize	Speed Memory	Use to specify whether model building should be optimized for speed or for memory.

autoclassifiernode Properties

The Auto Classifier node creates and compares a number of different models for binary outcomes (yes or no, churn or do not churn, and so on), allowing you to choose the best approach for a given analysis. A number of modeling algorithms are supported, making it possible to select the methods you want to use, the specific options for each, and the criteria for comparing the results. The node generates a set of models based on the specified options and ranks the best candidates according to the criteria you specify.

Example

```
create autoclassifiernode
set :autoclassifiernode.ranking_measure=Accuracy
set :autoclassifiernode.ranking_dataset=Training
set :autoclassifiernode.enable_accuracy_limit=true
set :autoclassifiernode.accuracy_limit=0.9
set :autoclassifiernode.calculate_variable_importance=true
set :autoclassifiernode.use_costs=true
```

set :autoclassifiernode.svm=false

autoclassifiernode Properties	Values	Property description
target	*field*	For flag targets, the Auto Classifier node requires a single target and one or more input fields. Weight and frequency fields can also be specified. For more information, see the topic Common Modeling Node Properties on p. 181.
ranking_measure	Accuracy Area_under_curve Profit Lift Num_variables	
ranking_dataset	Training Test	
number_of_models	*integer*	Number of models to include in the model nugget. Specify an integer between 1 and 100.
calculate_variable_importance	*flag*	
enable_accuracy_limit	*flag*	
accuracy_limit	*integer*	Integer between 0 and 100.
enable_ area_under_curve _limit	*flag*	
area_under_curve_limit	*number*	Real number between 0.0 and 1.0.
enable_profit_limit	*flag*	
profit_limit	*number*	Integer greater than 0.
enable_lift_limit	*flag*	
lift_limit	*number*	Real number greater than 1.0.
enable_number_of_vari-ables_limit	*flag*	
number_of_variables_limit	*number*	Integer greater than 0.
use_fixed_cost	*flag*	
fixed_cost	*number*	Real number greater than 0.0.
variable_cost	*field*	
use_fixed_revenue	*flag*	
fixed_revenue	*number*	Real number greater than 0.0.
variable_revenue	*field*	
use_fixed_weight	*flag*	
fixed_weight	*number*	Real number greater than 0.0
variable_weight	*field*	
lift_percentile	*number*	Integer between 0 and 100.
enable_model_build_time_limit	*flag*	
model_build_time_limit	*number*	Integer set to the number of minutes to limit the time taken to build each individual model.
enable_stop_after_time_limit	*flag*	

autoclassifiernode Properties	**Values**	**Property description**
stop_after_time_limit	*number*	Real number set to the number of hours to limit the overall elapsed time for an auto classifier run.
enable_stop_after_valid_model_produced	*flag*	
use_costs	*flag*	
<algorithm>	*flag*	Enables or disables the use of a specific algorithm, for example: set :autoclassifiernode.chaid=true
<algorithm>.<property>	*string*	Sets a property value for a specific algorithm. For more information, see the topic Setting Algorithm Properties on p. 186.

Setting Algorithm Properties

For the Auto Classifier, Auto Numeric, and Auto Cluster nodes, properties for specific algorithms used by the node can be set using the general form:

```
set :autoclassifiernode.<algorithm>.<property> = <value>

set :autonumericnode.<algorithm>.<property> = <value>

set :autoclusternode.<algorithm>.<property> = <value>
```

For example:

```
set :autoclassifiernode.neuralnetwork.method = MultilayerPerceptron
```

Algorithm names for the Auto Classifier node are cart, chaid, quest, c50, logreg, decisionlist, bayesnet, discriminant, svm and knn.

Algorithm names for the Auto Numeric node are cart, chaid, neuralnetwork, genlin, svm, regression, linear and knn.

Algorithm names for the Auto Cluster node are twostep, k-means, and kohonen.

Property names are standard as documented for each algorithm node.

Algorithm properties that contain periods or other punctuation must be wrapped in single quotes, for example:

```
set :autoclassifiernode.logreg.tolerance = '1.0E-5'
```

Multiple values can also be assigned for property, for example:

```
set :autoclassifiernode.decisionlist.search_direction = [Up Down]
```

To enable or disable the use of a specific algorithm:

```
set :autoclassifiernode.chaid=true
```

Notes:

- Lowercase must be used when setting true and false values (rather than False).
- In cases where certain algorithm options are not available in the Auto Classifier node, or when only a single value can be specified rather than a range of values, the same limits apply with scripting as when accessing the node in the standard manner.

autoclusternode Properties

The Auto Cluster node estimates and compares clustering models, which identify groups of records that have similar characteristics. The node works in the same manner as other automated modeling nodes, allowing you to experiment with multiple combinations of options in a single modeling pass. Models can be compared using basic measures with which to attempt to filter and rank the usefulness of the cluster models, and provide a measure based on the importance of particular fields.

Example

```
create autoclusternode
set :autoclusternode.ranking_measure=Silhouette
set :autoclusternode.ranking_dataset=Training
set :autoclusternode.enable_silhouette_limit=true
set :autoclusternode.silhouette_limit=5
```

autoclusternode Properties	Values	Property description
evaluation	*field*	*Note*: Auto Cluster node only. Identifies the field for which an importance value will be calculated. Alternatively, can be used to identify how well the cluster differentiates the value of this field and, therefore; how well the model will predict this field.
ranking_measure	Silhouette Num_clusters Size_smallest_cluster Size_largest_cluster Smallest_to_largest Importance	
ranking_dataset	Training Test	
summary_limit	*integer*	Number of models to list in the report. Specify an integer between 1 and 100.
enable_silhouette_limit	*flag*	
silhouette_limit	*integer*	Integer between 0 and 100.
enable_number_less_limit	*flag*	
number_less_limit	*number*	Real number between 0.0 and 1.0.
enable_number_greater_limit	*flag*	
number_greater_limit	*number*	Integer greater than 0.
enable_smallest_cluster_limit	*flag*	
smallest_cluster_units	Percentage Counts	

autoclusternode Properties	Values	Property description
smallest_cluster_limit_percentage	*number*	
smallest_cluster_limit_count	*integer*	Integer greater than 0.
enable_largest_cluster_limit	*flag*	
largest_cluster_units	Percentage Counts	
largest_cluster_limit_percentage	*number*	
largest_cluster_limit_count	*integer*	
enable_smallest_largest_limit	*flag*	
smallest_largest_limit	*number*	
enable_importance_limit	*flag*	
importance_limit_condition	Greater_than Less_than	
importance_limit_greater_than	*number*	Integer between 0 and 100.
importance_limit_less_than	*number*	Integer between 0 and 100.
<algorithm>	*flag*	Enables or disables the use of a specific algorithm, for example: set :autoclusternode.kohonen=true
<algorithm>.<property>	*string*	Sets a property value for a specific algorithm. For more information, see the topic Setting Algorithm Properties on p. 186.

autonumericnode Properties

The Auto Numeric node estimates and compares models for continuous numeric range outcomes using a number of different methods. The node works in the same manner as the Auto Classifier node, allowing you to choose the algorithms to use and to experiment with multiple combinations of options in a single modeling pass. Supported algorithms include neural networks, C&R Tree, CHAID, linear regression, generalized linear regression, and support vector machines (SVM). Models can be compared based on correlation, relative error, or number of variables used.

Example

```
create autonumericnode
set :autonumericnode.ranking_measure=Correlation
set :autonumericnode.ranking_dataset=Training
set :autonumericnode.enable_correlation_limit=true
set :autonumericnode.correlation_limit=0.8
set :autonumericnode.calculate_variable_importance=true
set :autonumericnode.neuralnetwork=true
set :autonumericnode.chaid=false
```

autonumericnode Properties	Values	Property description
custom_fields	*flag*	If True, custom field settings will be used instead of type node settings.

autonumericnode Properties	Values	Property description
target	*field*	The Auto Numeric node requires a single target and one or more input fields. Weight and frequency fields can also be specified. For more information, see the topic Common Modeling Node Properties on p. 181.
inputs	*[field1 ... field2]*	
partition	*field*	
use_frequency	*flag*	
frequency_field	*field*	
use_weight	*flag*	
weight_field	*field*	
use_partitioned_data	*flag*	If a partition field is defined, only the training data are used for model building.
ranking_measure	Correlation NumberOfFields	
ranking_dataset	Test Training	
number_of_models	*integer*	Number of models to include in the model nugget. Specify an integer between 1 and 100.
calculate_variable_importance	*flag*	
enable_correlation_limit	*flag*	
correlation_limit	*integer*	
enable_number_of_fields_limit	*flag*	
number_of_fields_limit	*integer*	
enable_relative_error_limit	*flag*	
relative_error_limit	*integer*	
enable_model_build_time_limit	*flag*	
model_build_time_limit	*integer*	
enable_stop_after_time_limit	*flag*	
stop_after_time_limit	*integer*	
stop_if_valid_model	*flag*	
<algorithm>	*flag*	Enables or disables the use of a specific algorithm, for example: set :autonumericnode.chaid=true
<algorithm>.<property>	*string*	Sets a property value for a specific algorithm. For more information, see the topic Setting Algorithm Properties on p. 186.

bayesnetnode Properties

The Bayesian Network node enables you to build a probability model by combining observed and recorded evidence with real-world knowledge to establish the likelihood of occurrences. The node focuses on Tree Augmented Naïve Bayes (TAN) and Markov Blanket networks that are primarily used for classification.

Example

```
create bayesnetnode
set :bayesnetnode.continue_training_existing_model = True
set :bayesnetnode.structure_type = MarkovBlanket
set :bayesnetnode.use_feature_selection = True
# Expert tab
set :bayesnetnode.mode = Expert
set :bayesnetnode.all_probabilities = True
set :bayesnetnode.independence = Pearson
```

bayesnetnode Properties	Values	Property description
inputs	*[field1 ... fieldN]*	Bayesian network models use a single target field, and one or more input fields. Continuous fields are automatically binned. For more information, see the topic Common Modeling Node Properties on p. 181.
continue_training_existing_model	*flag*	
structure_type	TAN MarkovBlanket	Select the structure to be used when building the Bayesian network.
use_feature_selection	*flag*	
parameter_learning_method	Likelihood Bayes	Specifies the method used to estimate the conditional probability tables between nodes where the values of the parents are known.
mode	Expert Simple	
missing_values	*flag*	
all_probabilities	*flag*	
independence	Likelihood Pearson	Specifies the method used to determine whether paired observations on two variables are independent of each other.
significance_level	*number*	Specifies the cutoff value for determining independence.
maximal_conditioning_set	*number*	Sets the maximal number of conditioning variables to be used for independence testing.
inputs_always_selected	*[field1 ... fieldN]*	Specifies which fields from the dataset are always to be used when building the Bayesian network. *Note*: The target field is always selected.
maximum_number_inputs	*number*	Specifies the maximum number of input fields to be used in building the Bayesian network.
calculate_variable_importance	*flag*	
calculate_raw_propensities	*flag*	
calculate_adjusted_propensities	*flag*	
adjusted_propensity_partition	Test Validation	

c50node Properties

The C5.0 node builds either a decision tree or a rule set. The model works by splitting the sample based on the field that provides the maximum information gain at each level. The target field must be categorical. Multiple splits into more than two subgroups are allowed.

Example

```
create c50node
# "Model" tab
set :c50node.use_model_name = False
set :c50node.model_name = "C5_Drug"
set :c50node.use_partitioned_data = True
set :c50node.output_type = DecisionTree
set :c50node.use_xval = True
set :c50node.xval_num_folds = 3
set :c50node.mode = Expert
set :c50node.favor = Generality
set :c50node.min_child_records = 3
# "Costs" tab
set :c50node.use_costs = True
set :c50node.costs = [{"drugA" "drugX" 2}]
```

c50node Properties	Values	Property description
target	*field*	C50 models use a single target field and one or more input fields. A weight field can also be specified. For more information, see the topic Common Modeling Node Properties on p. 181.
output_type	DecisionTree RuleSet	
group_symbolics	*flag*	
use_boost	*flag*	
boost_num_trials	*number*	
use_xval	*flag*	
xval_num_folds	*number*	
mode	Simple Expert	
favor	Accuracy Generality	Favor accuracy or generality.
expected_noise	*number*	
min_child_records	*number*	
pruning_severity	*number*	
use_costs	*flag*	
costs	*structured*	This is a structured property.
use_winnowing	*flag*	
use_global_pruning	*flag*	On (True) by default.
calculate_variable_importance	*flag*	
calculate_raw_propensities	*flag*	

c50node Properties	Values	Property description
calculate_adjusted_propensities	*flag*	
adjusted_propensity_partition	Test Validation	

carmanode Properties

The CARMA model extracts a set of rules from the data without requiring you to specify input or target fields. In contrast to Apriori the CARMA node offers build settings for rule support (support for both antecedent and consequent) rather than just antecedent support. This means that the rules generated can be used for a wider variety of applications—for example, to find a list of products or services (antecedents) whose consequent is the item that you want to promote this holiday season.

Example

```
create carmanode
# "Fields" tab
set :carmanode.custom_fields = True
set :carmanode.use_transactional_data = True
set :carmanode.inputs = ['BP' 'Cholesterol' 'Drug']
set :carmanode.partition = Test
# "Model" tab
set :carmanode.use_model_name = False
set :carmanode.model_name = "age_bp_drug"
set :carmanode.use_partitioned_data = False
set :carmanode.min_supp = 10.0
set :carmanode.min_conf = 30.0
set :carmanode.max_size = 5
# Expert Options
set :carmanode.mode = Expert
#help  set :carmanode.exclude_simple = True
set :carmanode.use_pruning = True
set :carmanode.pruning_value = 300
set :carmanode.vary_support = True
set :carmanode.estimated_transactions = 30
set :carmanode.rules_without_antecedents = True
```

carmanode Properties	Values	Property description
inputs	*[field1 ... fieldn]*	CARMA models use a list of input fields, but no target. Weight and frequency fields are not used. For more information, see the topic Common Modeling Node Properties on p. 181.
id_field	*field*	Field used as the ID field for model building.
contiguous	*flag*	Used to specify whether IDs in the ID field are contiguous.
use_transactional_data	*flag*	
content_field	*field*	

carmanode Properties	Values	Property description
min_supp	*number(percent)*	Relates to rule support rather than antecedent support. The default is 20%.
min_conf	*number(percent)*	The default is 20%.
max_size	*number*	The default is 10.
mode	Simple Expert	The default is Simple.
exclude_multiple	*flag*	Excludes rules with multiple consequents. The default is False.
use_pruning	*flag*	The default is False.
pruning_value	*number*	The default is 500.
vary_support	*flag*	
estimated_transactions	*integer*	
rules_without_antecedents	*flag*	

cartnode Properties

The Classification and Regression (C&R) Tree node generates a decision tree that allows you to predict or classify future observations. The method uses recursive partitioning to split the training records into segments by minimizing the impurity at each step, where a node in the tree is considered “pure” if 100% of cases in the node fall into a specific category of the target field. Target and input fields can be numeric ranges or categorical (nominal, ordinal, or flags); all splits are binary (only two subgroups).

Example

```
create cartnode at 200 100
create variablefilenode at 100 100
connect :variablefilenode to :cartnode
set :variablefilenode.full_filename = "$CLEO_DEMOS/DRUG1n"
# "Fields" tab
set :cartnode.custom_fields = True
set :cartnode.target = 'Drug'
set :cartnode.inputs = ['Age' 'BP' 'Cholesterol']
# "Build Options" tab, 'Objective' panel
set :cartnode.model_output_type = InteractiveBuilder
set :cartnode.use_tree_directives = True
set :cartnode.tree_directives = """Grow Node Index 0 Children 1 2
Grow Node Index 2 Children 3 4"""
# "Build Options" tab, 'Basics' panel
set :cartnode.prune_tree = False
set :cartnode.use_std_err_rule = True
set :cartnode.std_err_multiplier = 3.0
set :cartnode.max_surrogates = 7
# "Build Options" tab, 'Stopping Rules' panel
set :cartnode.use_percentage = True
set :cartnode.min_parent_records_pc = 5
set :cartnode.min_child_records_pc = 3
# "Build Options" tab, 'Advanced' panel
```

```
set :cartnode.min_impurity = 0.0003
set :cartnode.impurity_measure = Twoing
# "Model Options" tab
set :cartnode.use_model_name = False
set :cartnode.model_name = "Cart_Drug"
```

cartnode Properties	Values	Property description
target	*field*	C&R Tree models require a single target and one or more input fields. A frequency field can also be specified. For more information, see the topic Common Modeling Node Properties on p. 181.
continue_training_existing_model	*flag*	
objective	Standard Boosting Bagging psm	psm is used for very large datasets, and requires a Server connection.
model_output_type	Single InteractiveBuilder	
use_tree_directives	*flag*	
tree_directives	*string*	Specify directives for growing the tree. Directives can be wrapped in triple quotes to avoid escaping newlines or quotes. Note that directives may be highly sensitive to minor changes in data or modeling options and may not generalize to other datasets.
use_max_depth	Default Custom	
max_depth	*integer*	Maximum tree depth, from 0 to 1000. Used only if use_max_depth = Custom.
prune_tree	*flag*	Prune tree to avoid overfitting.
use_std_err	*flag*	Use maximum difference in risk (in Standard Errors).
std_err_multiplier	*number*	Maximum difference.
max_surrogates	*number*	Maximum surrogates.
use_percentage	*flag*	
min_parent_records_pc	*number*	
min_child_records_pc	*number*	
min_parent_records_abs	*number*	
min_child_records_abs	*number*	
use_costs	*flag*	
costs	*structured*	Structured property using the form: [{drugA drugB 1.5} {drugA drugC 2.1}] where the arguments within braces ({ }) are actual predicted costs.
priors	Data Equal Custom	

cartnode **Properties**	**Values**	**Property description**
custom_priors	*structured*	Structured property using the form: set :cartnode. custom_priors = [{ drugA 0.3 } { drugB 0.6 }]
adjust_priors	*flag*	
trails	*number*	Number of component models for boosting or bagging.
set_ensemble_method	Voting HighestProbability HighestMeanProbability	Default combining rule for categorical targets.
range_ensemble_method	Mean Median	Default combining rule for continuous targets.
large_boost	*flag*	Apply boosting to very large data sets.
min_impurity	*number*	
impurity_measure	Gini Twoing Ordered	
train_pct	*number*	Overfit prevention set.
set_random_seed	*flag*	Replicate results option.
seed	*number*	
calculate_variable_importance	*flag*	
calculate_raw_propensities	*flag*	
calculate_adjusted_propensities	*flag*	
adjusted_propensity_partition	Test Validation	

chaidnode Properties

The CHAID node generates decision trees using chi-square statistics to identify optimal splits. Unlike the C&R Tree and QUEST nodes, CHAID can generate nonbinary trees, meaning that some splits have more than two branches. Target and input fields can be numeric range (continuous) or categorical. Exhaustive CHAID is a modification of CHAID that does a more thorough job of examining all possible splits but takes longer to compute.

Example

```
create chaidnode at 200 100
create variablefilenode at 100 100
connect :variablefilenode to :chaidnode
set :variablefilenode.full_filename = "$CLEO_DEMOS/DRUG1n"

set :chaidnode.custom_fields = True
set :chaidnode.target = Drug
set :chaidnode.inputs = [Age Na K Cholesterol BP]
set :chaidnode.use_model_name = true
set :chaidnode.model_name = "CHAID"
set :chaidnode.method = Chaid
set :chaidnode.model_output_type = InteractiveBuilder
```

```
set :chaidnode.use_tree_directives = True
set :chaidnode.tree_directives = "Test"
set :chaidnode.split_alpha = 0.03
set :chaidnode.merge_alpha = 0.04
set :chaidnode.chi_square = Pearson
set :chaidnode.use_percentage = false
set :chaidnode.min_parent_records_abs = 40
set :chaidnode.min_child_records_abs = 30
set :chaidnode.epsilon = 0.003
set :chaidnode.max_iterations = 75
set :chaidnode.split_merged_categories = true
set :chaidnode.bonferroni_adjustment = true
```

chaidnode Properties	Values	Property description
target	*field*	CHAID models require a single target and one or more input fields. A frequency field can also be specified. For more information, see the topic Common Modeling Node Properties on p. 181.
continue_training_existing_model	*flag*	
objective	Standard Boosting Bagging psm	psm is used for very large datasets, and requires a Server connection.
model_output_type	Single InteractiveBuilder	
use_tree_directives	*flag*	
tree_directives	*string*	
method	Chaid ExhaustiveChaid	
use_max_depth	Default Custom	
max_depth	*integer*	Maximum tree depth, from 0 to 1000. Used only if use_max_depth = Custom.
use_percentage	*flag*	
min_parent_records_pc	*number*	
min_child_records_pc	*number*	
min_parent_records_abs	*number*	
min_child_records_abs	*number*	
use_costs	*flag*	
costs	*structured*	Structured property using the form: [{drugA drugB 1.5} {drugA drugC 2.1}] where the arguments within braces ({ }) are actual predicted costs.
trails	*number*	Number of component models for boosting or bagging.
set_ensemble_method	Voting HighestProbability HighestMeanProbability	Default combining rule for categorical targets.

chaidnode Properties	Values	Property description
range_ensemble_method	Mean Median	Default combining rule for continuous targets.
large_boost	*flag*	Apply boosting to very large data sets.
split_alpha	*number*	Significance level for splitting.
merge_alpha	*number*	Significance level for merging.
bonferroni_adjustment	*flag*	Adjust significance values using Bonferroni method.
split_merged_categories	*flag*	Allow resplitting of merged categories.
chi_square	Pearson LR	Method used to calculate the chi-square statistic: Pearson or Likelihood Ratio
epsilon	*number*	Minimum change in expected cell frequencies..
max_iterations	*number*	Maximum iterations for convergence.
set_random_seed	*integer*	
seed	*number*	
calculate_variable_importance	*flag*	
calculate_raw_propensities	*flag*	
calculate_adjusted_propensi-ties	*flag*	
adjusted_propensity_partition	Test Validation	
maximum_number_of_models	*integer*	

coxregnode Properties

The Cox regression node enables you to build a survival model for time-to-event data in the presence of censored records. The model produces a survival function that predicts the probability that the event of interest has occurred at a given time (*t*) for given values of the input variables.

Example

```
create coxregnode
set :coxregnode.survival_time = tenure
set :coxregnode.method = BackwardsStepwise
# Expert tab
set :coxregnode.mode = Expert
set :coxregnode.removal_criterion = Conditional
set :coxregnode.survival = True
```

coxregnode Properties	Values	Property description
survival_time	*field*	Cox regression models require a single field containing the survival times.

coxregnode Properties	Values	Property description
target	*field*	Cox regression models require a single target field, and one or more input fields. For more information, see the topic Common Modeling Node Properties on p. 181.
method	Enter Stepwise BackwardsStepwise	
groups	*field*	
model_type	MainEffects Custom	
custom_terms	[*"BP*Sex" "BP*Age"*]	Example: set :coxregnode. custom_terms = ["BP*Sex" "BP" "Age"]
mode	Expert Simple	
max_iterations	*number*	
p_converge	1.0E-4 1.0E-5 1.0E-6 1.0E-7 1.0E-8 0	
p_converge	1.0E-4 1.0E-5 1.0E-6 1.0E-7 1.0E-8 0	
l_converge	1.0E-1 1.0E-2 1.0E-3 1.0E-4 1.0E-5 0	
removal_criterion	LR Wald Conditional	
probability_entry	*number*	
probability_removal	*number*	
output_display	EachStep LastStep	
ci_enable	*flag*	
ci_value	90 95 99	
correlation	*flag*	
display_baseline	*flag*	
survival	*flag*	
hazard	*flag*	
log_minus_log	*flag*	

coxregnode Properties	Values	Property description
one_minus_survival	*flag*	
separate_line	*field*	
value	*number* or *string*	If no value is specified for a field, the default option "Mean" will be used for that field. Usage for a numeric field: coxnode.value = [{"age" "35.8"}] Usage for a categorical field: coxnode.value = [{"color" "pink"}]

decisionlistnode Properties

The Decision List node identifies subgroups, or segments, that show a higher or lower likelihood of a given binary outcome relative to the overall population. For example, you might look for customers who are unlikely to churn or are most likely to respond favorably to a campaign. You can incorporate your business knowledge into the model by adding your own custom segments and previewing alternative models side by side to compare the results. Decision List models consist of a list of rules in which each rule has a condition and an outcome. Rules are applied in order, and the first rule that matches determines the outcome.

Example

```
create decisionlistnode
set :decisionlistnode.search_direction=Down
set :decisionlistnode.target_value=1
set :decisionlistnode.max_rules=4
set :decisionlistnode.min_group_size_pct = 15
```

decisionlistnode Properties	Values	Property description
target	*field*	Decision List models use a single target and one or more input fields. A frequency field can also be specified. For more information, see the topic Common Modeling Node Properties on p. 181.
model_output_type	Model InteractiveBuilder	
search_direction	Up Down	Relates to finding segments; where Up is the equivalent of High Probability, and Down is the equivalent of Low Probability..
target_value	*string*	If not specified, will assume true value for flags.
max_rules	*integer*	The maximum number of segments excluding the remainder.
min_group_size	*integer*	Minimum segment size.
min_group_size_pct	*number*	Minimum segment size as a percentage.

decisionlistnode Properties	Values	Property description
confidence_level	*number*	Minimum threshold that an input field has to improve the likelihood of response (give lift), to make it worth adding to a segment definition.
max_segments_per_rule	*integer*	
mode	Simple Expert	
bin_method	EqualWidth EqualCount	
bin_count	*number*	
max_models_per_cycle	*integer*	Search width for lists.
max_rules_per_cycle	*integer*	Search width for segment rules.
segment_growth	*number*	
include_missing	*flag*	
final_results_only	*flag*	
reuse_fields	*flag*	Allows attributes (input fields which appear in rules) to be re-used.
max_alternatives	*integer*	
calculate_raw_propensities	*flag*	
calculate_adjusted_propensities	*flag*	
adjusted_propensity_partition	Test Validation	

discriminantnode Properties

Discriminant analysis makes more stringent assumptions than logistic regression but can be a valuable alternative or supplement to a logistic regression analysis when those assumptions are met.

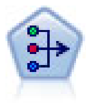
Example

```
create discriminantnode
set :discriminantnode.target = custcat
set :discriminantnode.use_partitioned_data = False
set :discriminantnode.method = Stepwise
```

discriminantnode Properties	Values	Property description
target	*field*	Discriminant models require a single target field and one or more input fields. Weight and frequency fields are not used. For more information, see the topic Common Modeling Node Properties on p. 181.
method	Enter Stepwise	
mode	Simple Expert	

discriminantnode Properties	Values	Property description
prior_probabilities	AllEqual ComputeFromSizes	
covariance_matrix	WithinGroups SeparateGroups	
means	*flag*	Statistics options in the Advanced Output dialog box.
univariate_anovas	*flag*	
box_m	*flag*	
within_group_covariance	*flag*	
within_groups_correlation	*flag*	
separate_groups_covariance	*flag*	
total_covariance	*flag*	
fishers	*flag*	
unstandardized	*flag*	
casewise_results	*flag*	Classification options in the Advanced Output dialog box.
limit_to_first	*number*	Default value is 10.
summary_table	*flag*	
leave_one_classification	*flag*	
combined_groups	*flag*	
separate_groups_covariance	*flag*	Matrices option Separate-groups covariance.
territorial_map	*flag*	
combined_groups	*flag*	Plot option Combined-groups.
separate_groups	*flag*	Plot option Separate-groups.
summary_of_steps	*flag*	
F_pairwise	*flag*	
stepwise_method	WilksLambda UnexplainedVariance MahalanobisDistance SmallestF RaosV	
V_to_enter	*number*	
criteria	UseValue UseProbability	
F_value_entry	*number*	Default value is 3.84.
F_value_removal	*number*	Default value is 2.71.
probability_entry	*number*	Default value is 0.05.
probability_removal	*number*	Default value is 0.10.
calculate_variable_importance	*flag*	
calculate_raw_propensities	*flag*	
calculate_adjusted_propensities	*flag*	
adjusted_propensity_partition	Test Validation	

factornode Properties



The PCA/Factor node provides powerful data-reduction techniques to reduce the complexity of your data. Principal components analysis (PCA) finds linear combinations of the input fields that do the best job of capturing the variance in the entire set of fields, where the components are orthogonal (perpendicular) to each other. Factor analysis attempts to identify underlying factors that explain the pattern of correlations within a set of observed fields. For both approaches, the goal is to find a small number of derived fields that effectively summarizes the information in the original set of fields.

Example

```
create factornode
# "Fields" tab
set :factornode.custom_fields = True
set :factornode.inputs = ['BP' 'Na' 'K']
set :factornode.partition = Test
# "Model" tab
set :factornode.use_model_name = True
set :factornode.model_name = "Factor_Age"
set :factornode.use_partitioned_data = False
set :factornode.method = GLS
# Expert options
set :factornode.mode = Expert
set :factornode.complete_records = true
set :factornode.matrix = Covariance
set :factornode.max_iterations = 30
set :factornode.extract_factors = ByFactors
set :factornode.min_eigenvalue = 3.0
set :factornode.max_factor = 7
set :factornode.sort_values = True
set :factornode.hide_values = True
set :factornode.hide_below = 0.7
# "Rotation" section
set :factornode.rotation = DirectOblimin
set :factornode.delta = 0.3
set :factornode.kappa = 7.0
```

factornode Properties	Values	Property description
inputs	*[field1 ... fieldN]*	PCA/Factor models use a list of input fields, but no target. Weight and frequency fields are not used. For more information, see the topic Common Modeling Node Properties on p. 181.
method	PC ULS GLS ML PAF Alpha Image	

factornode Properties	Values	Property description
mode	Simple Expert	
max_iterations	*number*	
complete_records	*flag*	
matrix	Correlation Covariance	
extract_factors	ByEigenvalues ByFactors	
min_eigenvalue	*number*	
max_factor	*number*	
rotation	None Varimax DirectOblimin Equamax Quartimax Promax	
delta	*number*	If you select DirectOblimin as your rotation data type, you can specify a value for delta. If you do not specify a value, the default value for delta is used.
kappa	*number*	If you select Promax as your rotation data type, you can specify a value for kappa. If you do not specify a value, the default value for kappa is used.
sort_values	*flag*	
hide_values	*flag*	
hide_below	*number*	

featureselectionnode Properties

The Feature Selection node screens input fields for removal based on a set of criteria (such as the percentage of missing values); it then ranks the importance of remaining inputs relative to a specified target. For example, given a data set with hundreds of potential inputs, which are most likely to be useful in modeling patient outcomes?

Example

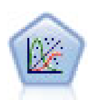
```
create featureselectionnode
set :featureselectionnode.screen_single_category=true
set :featureselectionnode.max_single_category=95
set :featureselectionnode.screen_missing_values=true
set :featureselectionnode.max_missing_values=80
set :featureselectionnode.criteria = Likelihood
set :featureselectionnode.unimportant_below = 0.8
set :featureselectionnode.important_above = 0.9
set :featureselectionnode.important_label = "Check Me Out!"
set :featureselectionnode.selection_mode = TopN
set :featureselectionnode.top_n = 15
```

For a more detailed example that creates and applies a Feature Selection model, see Standalone Script Example: Generating a Feature Selection Model in Chapter 2 on p. 13.

featureselectionnode Properties	Values	Property description
target	*field*	Feature Selection models rank predictors relative to the specified target. Weight and frequency fields are not used. For more information, see the topic Common Modeling Node Properties on p. 181.
screen_single_category	*flag*	If True, screens fields that have too many records falling into the same category relative to the total number of records.
max_single_category	*number*	Specifies the threshold used when screen_single_category is True.
screen_missing_values	*flag*	If True, screens fields with too many missing values, expressed as a percentage of the total number of records.
max_missing_values	*number*	
screen_num_categories	*flag*	If True, screens fields with too many categories relative to the total number of records.
max_num_categories	*number*	
screen_std_dev	*flag*	If True, screens fields with a standard deviation of less than or equal to the specified minimum.
min_std_dev	*number*	
screen_coeff_of_var	*flag*	If True, screens fields with a coefficient of variance less than or equal to the specified minimum.
min_coeff_of_var	*number*	
criteria	Pearson Likelihood CramersV Lambda	When ranking categorical predictors against a categorical target, specifies the measure on which the importance value is based.
unimportant_below	*number*	Specifies the threshold *p* values used to rank variables as important, marginal, or unimportant. Accepts values from 0.0 to 1.0.
important_above	*number*	Accepts values from 0.0 to 1.0.
unimportant_label	*string*	Specifies the label for the unimportant ranking.
marginal_label	*string*	
important_label	*string*	
selection_mode	ImportanceLevel ImportanceValue TopN	
select_important	*flag*	When selection_mode is set to ImportanceLevel, specifies whether to select important fields.

featureselectionnode Properties	Values	Property description
select_marginal	*flag*	When selection_mode is set to ImportanceLevel, specifies whether to select marginal fields.
select_unimportant	*flag*	When selection_mode is set to ImportanceLevel, specifies whether to select unimportant fields.
importance_value	*number*	When selection_mode is set to ImportanceValue, specifies the cutoff value to use. Accepts values from 0 to 100.
top_n	*integer*	When selection_mode is set to TopN, specifies the cutoff value to use. Accepts values from 0 to 1000.

genlinnode Properties



The Generalized Linear model expands the general linear model so that the dependent variable is linearly related to the factors and covariates through a specified link function. Moreover, the model allows for the dependent variable to have a non-normal distribution. It covers the functionality of a wide number of statistical models, including linear regression, logistic regression, loglinear models for count data, and interval-censored survival models.

Example

```
create genlinnode
set :genlinnode.model_type = MainAndAllTwoWayEffects
set :genlinnode.offset_type = Variable
set :genlinnode.offset_field = Claimant
```

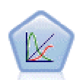
genlinnode Properties	Values	Property description
target	*field*	Generalized Linear models require a single target field which must be a nominal or flag field, and one or more input fields. A weight field can also be specified. For more information, see the topic Common Modeling Node Properties on p. 181.
use_weight	*flag*	
weight_field	*field*	Field type is only continuous.
target_represents_trials	*flag*	
trials_type	Variable FixedValue	
trials_field	*field*	Field type is continuous, flag, or ordinal.
trials_number	*number*	Default value is 10.
model_type	MainEffects MainAndAllTwoWayEffects	
offset_type	Variable FixedValue	

genlinnode Properties	Values	Property description
offset_field	*field*	Field type is only continuous.
offset_value	*number*	Must be a real number.
base_category	Last First	
include_intercept	*flag*	
mode	Simple Expert	
distribution	BINOMIAL GAMMA IGAUSS NEGBIN NORMAL POISSON TWEEDIE MULTINOMIAL	IGAUSS: Inverse Gaussian. NEGBIN: Negative binomial.
negbin_para_type	Specify Estimate	
negbin_parameter	*number*	Default value is 1. Must contain a non-negative real number.
tweedie_parameter	*number*	
link_function	IDENTITY CLOGLOG LOG LOGC LOGIT NEGBIN NLOGLOG ODDSPOWER PROBIT POWER CUMCAUCHIT CUMCLOGLOG CUMLOGIT CUMNLOGLOG CUMPROBIT	CLOGLOG: Complementary log-log. LOGC: log complement. NEGBIN: Negative binomial. NLOGLOG: Negative log-log. CUMCAUCHIT: Cumulative cauchit. CUMCLOGLOG: Cumulative complementary log-log. CUMLOGIT: Cumulative logit. CUMNLOGLOG: Cumulative negative log-log. CUMPROBIT: Cumulative probit.
power	*number*	Value must be real, nonzero number.
method	Hybrid Fisher NewtonRaphson	
max_fisher_iterations	*number*	Default value is 1; only positive integers allowed.
scale_method	MaxLikelihoodEstimate Deviance PearsonChiSquare FixedValue	
scale_value	*number*	Default value is 1; must be greater than 0.
covariance_matrix	ModelEstimator RobustEstimator	
max_iterations	*number*	Default value is 100; non-negative integers only.
max_step_halving	*number*	Default value is 5; positive integers only.

genlinnode Properties	**Values**	**Property description**
check_separation	*flag*	
start_iteration	*number*	Default value is 20; only positive integers allowed.
estimates_change	*flag*	
estimates_change_min	*number*	Default value is 1E-006; only positive numbers allowed.
estimates_change_type	Absolute Relative	
loglikelihood_change	*flag*	
loglikelihood_change_min	*number*	Only positive numbers allowed.
loglikelihood_change_type	Absolute Relative	
hessian_convergence	*flag*	
hessian_convergence_min	*number*	Only positive numbers allowed.
hessian_convergence_type	Absolute Relative	
case_summary	*flag*	
contrast_matrices	*flag*	
descriptive_statistics	*flag*	
estimable_functions	*flag*	
model_info	*flag*	
iteration_history	*flag*	
goodness_of_fit	*flag*	
print_interval	*number*	Default value is 1; must be positive integer.
model_summary	*flag*	
lagrange_multiplier	*flag*	
parameter_estimates	*flag*	
include_exponential	*flag*	
covariance_estimates	*flag*	
correlation_estimates	*flag*	
analysis_type	TypeI TypeIII TypeIAndTypeIII	
statistics	Wald LR	
citype	Wald Profile	
tolerancelevel	*number*	Default value is 0.0001.
confidence_interval	*number*	Default value is 95.
loglikelihood_function	Full Kernel	
singularity_tolerance	1E-007 1E-008 1E-009 1E-010 1E-011 1E-012	

genlinnode Properties	Values	Property description
value_order	Ascending Descending DataOrder	
calculate_variable_importance	*flag*	
calculate_raw_propensities	*flag*	
calculate_adjusted_propensities	*flag*	
adjusted_propensity_partition	Test Validation	

glmmnode Properties



A generalized linear mixed model (GLMM) extends the linear model so that the target can have a non-normal distribution, is linearly related to the factors and covariates via a specified link function, and so that the observations can be correlated. Generalized linear mixed models cover a wide variety of models, from simple linear regression to complex multilevel models for non-normal longitudinal data.

glmmnode Properties	Values	Property description
residual_subject_spec	*structured*	The combination of values of the specified categorical fields that uniquely define subjects within the data set
repeated_measures	*structured*	Fields used to identify repeated observations.
residual_group_spec	[*field1 ... fieldN*]	Fields that define independent sets of repeated effects covariance parameters.
residual_covariance_type	Diagonal AR1 ARMA11 COMPOUND_SYMMETRY IDENTITY TOEPLITZ UNSTRUCTURED VARIANCE_COMPONENTS	Specifies covariance structure for residuals.
custom_target	*flag*	Indicates whether to use target defined in upstream node (false) or custom target specified by target_field (true).
target_field	*field*	Field to use as target if custom_target is true.
use_trials	*flag*	Indicates whether additional field or value specifying number of trials is to be used when target response is a number of events occurring in a set of trials. Default is false.
use_field_or_value	Field Value	Indicates whether field (default) or value is used to specify number of trials.
trials_field	*field*	Field to use to specify number of trials.
trials_value	*integer*	Value to use to specify number of trials. If specified, minimum value is 1.

glmmnode Properties	Values	Property description
use_custom_target_reference	*flag*	Indicates whether custom reference category is to be used for a categorical target. Default is false.
target_reference_value	*string*	Reference category to use if use_custom_target_reference is true.
dist_link_combination	Nominal Logit GammaLog BinomialLogit PoissonLog BinomialProbit NegbinLog BinomialLogC Custom	Common models for distribution of values for target. Choose Custom to specify a distribution from the list provided bytarget_distribution.
target_distribution	Normal Binomial Multinomial Gamma Inverse NegativeBinomial Poisson	Distribution of values for target when dist_link_combination is Custom.
link_function_type	Identity LogC Log CLOGLOG Logit NLOGLOG PROBIT POWER CAUCHIT	Link function to relate target values to predictors.
link_function_param	*number*	Link function parameter value to use. Only applicable if link_function is POWER.
use_predefined_inputs	*flag*	Indicates whether fixed effect fields are to be those defined upstream as input fields (true) or those from fixed_effects_list (false). Default is false.
fixed_effects_list	*structured*	If use_predefined_inputs is false, specifies the input fields to use as fixed effect fields.
use_intercept	*flag*	If true (default), includes the intercept in the model.
random_effects_list	*structured*	List of fields to specify as random effects.
regression_weight_field	*field*	Field to use as analysis weight field.
use_offset	None offset_value offset_field	Indicates how offset is specified. Value None means no offset is used.
offset_value	*number*	Value to use for offset if use_offset is set to offset_value.
offset_field	*field*	Field to use for offset value if use_offset is set to offset_field.

glmmnode Properties	Values	Property description
target_category_order	Ascending Descending Data	Sorting order for categorical targets. Value Data specifies using the sort order found in the data. Default is Ascending.
inputs_category_order	Ascending Descending Data	Sorting order for categorical predictors. Value Data specifies using the sort order found in the data. Default is Ascending.
max_iterations	*integer*	Maximum number of iterations the algorithm will perform. A non-negative integer; default is 100.
confidence_level	*integer*	Confidence level used to compute interval estimates of the model coefficients. A non-negative integer; maximum is 100, default is 95.
degrees_of_freedom_method	Fixed Varied	Specifies how degrees of freedom are computed for significance test.
test_fixed_effects_coeffecients	Model Robust	Method for computing the parameter estimates covariance matrix.
use_model_name	*flag*	Indicates whether to specify a custom name for the model (true) or to use the system-generated name (false). Default is false.
model_name	*string*	If use_model_name is true, specifies the model name to use.
confidence	highest difference	Basis for computing scoring confidence value: highest predicted probability, or difference between highest and second highest predicted probabilities.
score_category_probabilities	*flag*	If true, produces predicted probabilities for categorical targets. Default is false.
max_categories	*integer*	If score_category_probabilities is true, specifies maximum number of categories to save.
score_propensity	*flag*	If true, produces propensity scores for flag target fields that indicate likelihood of “true” outcome for field.
emeans	*structure*	For each categorical field from the fixed effects list, specifies whether to produce estimated marginal means.
covariance_list	*structure*	For each continuous field from the fixed effects list, specifies whether to use the mean or a custom value when computing estimated marginal means.
mean_scale	Original Transformed	Specifies whether to compute estimated marginal means based on the original scale of the target (default) or on the link function transformation.
comparison_adjustment_method	LSD SEQBONFERRONI SEQSIDAK	Adjustment method to use when performing hypothesis tests with multiple contrasts.

kmeansnode Properties

The K-Means node clusters the data set into distinct groups (or clusters). The method defines a fixed number of clusters, iteratively assigns records to clusters, and adjusts the cluster centers until further refinement can no longer improve the model. Instead of trying to predict an outcome, *k*-means uses a process known as unsupervised learning to uncover patterns in the set of input fields.

Example

```
create kmeansnode
# "Fields" tab
set :kmeansnode.custom_fields = True
set :kmeansnode.inputs = ['Cholesterol' 'BP' 'Drug' 'Na' 'K' 'Age']
# "Model" tab
set :kmeansnode.use_model_name = False
set :kmeansnode.model_name = "Kmeans_allinputs"
set :kmeansnode.num_clusters = 9
set :kmeansnode.gen_distance = True
set :kmeansnode.cluster_label = "Number"
set :kmeansnode.label_prefix = "Kmeans_"
set :kmeansnode.optimize = Speed
# "Expert" tab
set :kmeansnode.mode = Expert
set :kmeansnode.stop_on = Custom
set :kmeansnode.max_iterations = 10
set :kmeansnode.tolerance = 3.0
set :kmeansnode.encoding_value = 0.3
```

kmeansnode Properties	Values	Property description
inputs	[*field1 ... fieldN*]	K-means models perform cluster analysis on a set of input fields but do not use a target field. Weight and frequency fields are not used. For more information, see the topic Common Modeling Node Properties on p. 181.
num_clusters	*number*	
gen_distance	*flag*	
cluster_label	String Number	
label_prefix	*string*	
mode	Simple Expert	
stop_on	Default Custom	
max_iterations	*number*	
tolerance	*number*	
encoding_value	*number*	
optimize	Speed Memory	Use to specify whether model building should be optimized for speed or for memory.

knnnode Properties

The *k*-Nearest Neighbor (KNN) node associates a new case with the category or value of the *k* objects nearest to it in the predictor space, where *k* is an integer. Similar cases are near each other and dissimilar cases are distant from each other.

Example

```
create knnnode
# Objectives tab
set: knnnode.objective = Custom
# Settings tab - Neighbors panel
set: knnnode.automatic_k_selection = false
set: knnnode.fixed_k = 2
set: knnnode.weight_by_importance = True
# Settings tab - Analyze panel
set: knnnode.save_distances = True
```

knnnode Properties	Values	Property description
analysis	PredictTarget IdentifyNeighbors	
objective	Balance Speed Accuracy Custom	
normalize_ranges	*flag*	
use_case_labels	*flag*	Check box to enable next option.
case_labels_field	*field*	
identify_focal_cases	*flag*	Check box to enable next option.
focal_cases_field	*field*	
automatic_k_selection	*flag*	
fixed_k	*integer*	Enabled only if automatic_k_selectio is False.
minimum_k	*integer*	Enabled only if automatic_k_selectio is True.
maximum_k	*integer*	
distance_computation	Euclidean CityBlock	
weight_by_importance	*flag*	
range_predictions	Mean Median	
perform_feature_selection	*flag*	
forced_entry_inputs	[*field1 ... fieldN*]	
stop_on_error_ratio	*flag*	
number_to_select	*integer*	
minimum_change	*number*	
validation_fold_assign_by_field	*flag*	

knnnode Properties	Values	Property description
number_of_folds	*integer*	Enabled only if validation_fold_assign_by_field is False
set_random_seed	*flag*	
random_seed	*number*	
folds_field	*field*	Enabled only if validation_fold_assign_by_field is True
all_probabilities	*flag*	
save_distances	*flag*	
calculate_raw_propensities	*flag*	
calculate_adjusted_propensities	*flag*	
adjusted_propensity_partition	Test Validation	

kohonennode Properties

The Kohonen node generates a type of neural network that can be used to cluster the data set into distinct groups. When the network is fully trained, records that are similar should be close together on the output map, while records that are different will be far apart. You can look at the number of observations captured by each unit in the model nugget to identify the strong units. This may give you a sense of the appropriate number of clusters.

Example

```
create kohonennode
# "Model" tab
set :kohonennode.use_model_name = False
set :kohonennode.model_name = "Symbolic Cluster"
set :kohonennode.stop_on = Time
set :kohonennode.time = 1
set :kohonennode.set_random_seed = True
set :kohonennode.random_seed = 12345
set :kohonennode.optimize = Speed
# "Expert" tab
set :kohonennode.mode = Expert
set :kohonennode.width = 3
set :kohonennode.length = 3
set :kohonennode.decay_style = Exponential
set :kohonennode.phase1_neighborhood = 3
set :kohonennode.phase1_eta = 0.5
set :kohonennode.phase1_cycles = 10
set :kohonennode.phase2_neighborhood = 1
set :kohonennode.phase2_eta = 0.2
```

```
set :kohonennode.phase2_cycles = 75
```

kohonennode Properties	Values	Property description
inputs	*[field1 ... fieldN]*	Kohonen models use a list of input fields, but no target. Frequency and weight fields are not used. For more information, see the topic Common Modeling Node Properties on p. 181.
continue	*flag*	
show_feedback	*flag*	
stop_on	Default Time	
time	*number*	
optimize	Speed Memory	Use to specify whether model building should be optimized for speed or for memory.
cluster_label	*flag*	
mode	Simple Expert	
width	*number*	
length	*number*	
decay_style	Linear Exponential	
phase1_neighborhood	*number*	
phase1_eta	*number*	
phase1_cycles	*number*	
phase2_neighborhood	*number*	
phase2_eta	*number*	
phase2_cycles	*number*	

linearnode Properties

Linear regression models predict a continuous target based on linear relationships between the target and one or more predictors.

Example

```
create linearnode
# Build Options tab - Objectives panel
set: linearnode.objective = Standard
# Build Options tab - Model Selection panel
set: linearnode.model_selection = BestSubsets
set: linearnode.criteria_best_subsets = ASE
# Build Options tab - Ensembles panel
```

set: linearnode.combining_rule_categorical = HighestMeanProbability

linearnode Properties	Values	Property description
target	*field*	Specifies a single target field.
inputs	[*field1 ... fieldN*]	Predictor fields used by the model.
continue_training_exist-ing_model	*flag*	
objective	Standard Bagging Boosting psm	psm is used for very large datasets, and requires a Server connection.
use_auto_data_preparation	*flag*	
confidence_level	*number*	
model_selection	ForwardStepwise BestSubsets None	
criteria_forward_stepwise	AICC Fstatistics AdjustedRSquare ASE	
probability_entry	*number*	
probability_removal	*number*	
use_max_effects	*flag*	
max_effects	*number*	
use_max_steps	*flag*	
max_steps	*number*	
criteria_best_subsets	AICC AdjustedRSquare ASE	
combining_rule_continuous	Mean Median	
component_models_n	*number*	
use_random_seed	*flag*	
random_seed	*number*	
use_custom_model_name	*flag*	
custom_model_name	*string*	
use_custom_name	*flag*	
custom_name	*string*	
tooltip	*string*	
keywords	*string*	
annotation	*string*	

logregnode Properties

Logistic regression is a statistical technique for classifying records based on values of input fields. It is analogous to linear regression but takes a categorical target field instead of a numeric range.

Multinomial Example

```
create logregnode
# "Fields" tab
set :logregnode.custom_fields = True
set :logregnode.target = 'Drug'
set :logregnode.inputs = ['BP' 'Cholesterol' 'Age']
set :logregnode.partition = Test
# "Model" tab
set :logregnode.use_model_name = False
set :logregnode.model_name = "Log_reg Drug"
set :logregnode.use_partitioned_data = True
set :logregnode.method = Stepwise
set :logregnode.logistic_procedure = Multinomial
set :logregnode.multinomial_base_category = BP
set :logregnode.model_type = FullFactorial
set :logregnode.custom_terms = [{BP Sex}{Age}{Na K}]
set :logregnode.include_constant = False
# "Expert" tab
set :logregnode.mode = Expert
set :logregnode.scale = Pearson
set :logregnode.scale_value = 3.0
set :logregnode.all_probabilities = True
set :logregnode.tolerance = "1.0E-7"
# "Convergence..." section
set :logregnode.max_iterations = 50
set :logregnode.max_steps = 3
set :logregnode.l_converge = "1.0E-3"
set :logregnode.p_converge = "1.0E-7"
set :logregnode.delta = 0.03
# "Output..." section
set :logregnode.summary = True
set :logregnode.likelihood_ratio = True
set :logregnode.asymptotic_correlation = True
set :logregnode.goodness_fit = True
set :logregnode.iteration_history = True
set :logregnode.history_steps = 3
set :logregnode.parameters = True
set :logregnode.confidence_interval = 90
set :logregnode.asymptotic_covariance = True
set :logregnode.classification_table = True
# "Stepping" options
set :logregnode.min_terms = 7
set :logregnode.use_max_terms = true
set :logregnode.max_terms = 10
set :logregnode.probability_entry = 3
set :logregnode.probability_removal = 5
set :logregnode.requirements = Containment
```

Binomial Example

```
create logregnode
# "Fields" tab
```

```
set :logregnode.custom_fields = True
set :logregnode.target = 'Cholesterol'
set :logregnode.inputs = ['BP' 'Drug' 'Age']
set :logregnode.partition = Test
# "Model" tab
set :logregnode.use_model_name = False
set :logregnode.model_name = "Log_reg Cholesterol"
set :logregnode.multinomial_base_category = BP
set :logregnode.use_partitioned_data = True
set :logregnode.binomial_method = Forwards
set :logregnode.logistic_procedure = Binomial
set :logregnode.binomial_categorical_input = Sex
set :logregnode.binomial_input_contrast.Sex = Simple
set :logregnode.binomial_input_category.Sex = Last
set :logregnode.include_constant = False
# "Expert" tab
set :logregnode.mode = Expert
set :logregnode.scale = Pearson
set :logregnode.scale_value = 3.0
set :logregnode.all_probabilities = True
set :logregnode.tolerance = "1.0E-7"
# "Convergence..." section
set :logregnode.max_iterations = 50
set :logregnode.l_converge = "1.0E-3"
set :logregnode.p_converge = "1.0E-7"
# "Output..." section
set :logregnode.binomial_output_display = at_each_step
set :logregnode.binomial_goodness_fit = True
set :logregnode.binomial_iteration_history = True
set :logregnode.binomial_parameters = True
set :logregnode.binomial_ci_enable = True
set :logregnode.binomial_ci = 85
# "Stepping" options
set :logregnode.binomial_removal_criterion = LR
set :logregnode.binomial_probability_removal = 0.2
```

logregnode Properties	**Values**	**Property description**
target	*field*	Logistic regression models require a single target field and one or more input fields. Frequency and weight fields are not used. For more information, see the topic Common Modeling Node Properties on p. 181.
logistic_procedure	Binomial Multinomial	
include_constant	*flag*	
mode	Simple Expert	
method	Enter Stepwise Forwards Backwards BackwardsStepwise	

logregnode Properties	**Values**	**Property description**
binomial_method	Enter Forwards Backwards	
model_type	MainEffects FullFactorial Custom	When FullFactorial is specified as the model type, stepping methods will not be run, even if specified. Instead, Enter will be the method used. If the model type is set to Custom but no custom fields are specified, a main-effects model will be built.
custom_terms	[*{BP Sex}{BP}{Age}*]	Example: set :logregnode. custom_terms = [{Na} {K} {Na K}]
multinomial_base_category	*string*	Specifies how the reference category is determined.
binomial_categorical_input	*string*	
binomial_input_contrast	Indicator Simple Difference Helmert Repeated Polynomial Deviation	Keyed property for categorical input that specifies how the contrast is determined. Usage format: NODE.binomial_input_contrast.FIELD-NAME
binomial_input_category	First Last	Keyed property for categorical input that specifies how the reference category is determined. Usage format: NODE.binomial_input_category.FIELD-NAME
scale	None UserDefined Pearson Deviance	
scale_value	*number*	
all_probabilities	*flag*	
tolerance	1.0E-5 1.0E-6 1.0E-7 1.0E-8 1.0E-9 1.0E-10	
min_terms	*number*	
use_max_terms	*flag*	
max_terms	*number*	
entry_criterion	Score LR	
removal_criterion	LR Wald	
probability_entry	*number*	
probability_removal	*number*	
binomial_probability_entry	*number*	
binomial_probability_removal	*number*	

logregnode Properties	Values	Property description
requirements	HierarchyDiscrete HierarchyAll Containment None	
max_iterations	*number*	
max_steps	*number*	
p_converge	1.0E-4 1.0E-5 1.0E-6 1.0E-7 1.0E-8 0	
l_converge	1.0E-1 1.0E-2 1.0E-3 1.0E-4 1.0E-5 0	
delta	*number*	
iteration_history	*flag*	
history_steps	*number*	
summary	*flag*	
likelihood_ratio	*flag*	
asymptotic_correlation	*flag*	
goodness_fit	*flag*	
parameters	*flag*	
confidence_interval	*number*	
asymptotic_covariance	*flag*	
classification_table	*flag*	
stepwise_summary	*flag*	
info_criteria	*flag*	
monotonicity_measures	*flag*	
binomial_output_display	at_each_step at_last_step	
binomial_goodness_of_fit	*flag*	
binomial_parameters	*flag*	
binomial_iteration_history	*flag*	
binomial_classification_plots	*flag*	
binomial_ci_enable	*flag*	
binomial_ci	*number*	
binomial_residual	outliers all	
binomial_residual_enable	*flag*	
binomial_outlier_threshold	*number*	
binomial_classification_cutoff	*number*	
binomial_removal_criterion	LR Wald Conditional	

logregnode Properties	Values	Property description
calculate_variable_importance	*flag*	
calculate_raw_propensities	*flag*	

neuralnetnode Properties

Caution: A newer version of the Neural Net modeling node, with enhanced features, is available in this release and is described in the next section (*neuralnetwork*). Although you can still build and score a model with the previous version, we recommend updating your scripts to use the new version. Details of the previous version are retained here for reference.

Example

```
create neuralnetnode
# "Fields" tab
set :neuralnetnode.custom_fields = True
set :neuralnetnode.targets = ['Drug']
set :neuralnetnode.inputs = ['Age' 'Na' 'K' 'Cholesterol' 'BP']
# "Model" tab
set :neuralnetnode.use_partitioned_data = True
set :neuralnetnode.method = Dynamic
set :neuralnetnode.train_pct = 30
set :neuralnetnode.set_random_seed = True
set :neuralnetnode.random_seed = 12345
set :neuralnetnode.stop_on = Time
set :neuralnetnode.accuracy = 95
set :neuralnetnode.cycles = 200
set :neuralnetnode.time = 3
set :neuralnetnode.optimize = Speed
# "Multiple Method Expert Options" section
set :neuralnetnode.m_topologies = "5 30 5; 2 20 3, 1 10 1"
set :neuralnetnode.m_non_pyramids = False
set :neuralnetnode.m_persistence = 100
```

neuralnetnode Properties	Values	Property description
targets	[*field1* ... *fieldN*]	The Neural Net node expects one or more target fields and one or more input fields. Frequency and weight fields are ignored. For more information, see the topic Common Modeling Node Properties on p. 181.
method	Quick Dynamic Multiple Prune ExhaustivePrune RBFN	
prevent_overtrain	*flag*	
train_pct	*number*	
set_random_seed	*flag*	

neuralnetnode Properties	Values	Property description
random_seed	*number*	
mode	Simple Expert	
stop_on	Default Accuracy Cycles Time	Stopping mode.
accuracy	*number*	Stopping accuracy.
cycles	*number*	Cycles to train.
time	*number*	Time to train (minutes).
continue	*flag*	
show_feedback	*flag*	
binary_encode	*flag*	
use_last_model	*flag*	
gen_logfile	*flag*	
logfile_name	*string*	
alpha	*number*	
initial_eta	*number*	
high_eta	*number*	
low_eta	*number*	
eta_decay_cycles	*number*	
hid_layers	One Two Three	
hl_units_one	*number*	
hl_units_two	*number*	
hl_units_three	*number*	
persistence	*number*	
m_topologies	*string*	
m_non_pyramids	*flag*	
m_persistence	*number*	
p_hid_layers	One Two Three	
p_hl_units_one	*number*	
p_hl_units_two	*number*	
p_hl_units_three	*number*	
p_persistence	*number*	
p_hid_rate	*number*	
p_hid_pers	*number*	
p_inp_rate	*number*	
p_inp_pers	*number*	
p_overall_pers	*number*	
r_persistence	*number*	
r_num_clusters	*number*	

neuralnetnode Properties	Values	Property description
r_eta_auto	*flag*	
r_alpha	*number*	
r_eta	*number*	
optimize	Speed Memory	Use to specify whether model building should be optimized for speed or for memory.
calculate_variable_importance	*flag*	Note: The sensitivity_analysis property used in previous releases is deprecated in favor of this property. The old property is still supported, but calculate_variable_importance is recommended.
calculate_raw_propensities	*flag*	
calculate_adjusted_propensities	*flag*	
adjusted_propensity_partition	Test Validation	

neuralnetworknode Properties

The Neural Net node uses a simplified model of the way the human brain processes information. It works by simulating a large number of interconnected simple processing units that resemble abstract versions of neurons. Neural networks are powerful general function estimators and require minimal statistical or mathematical knowledge to train or apply.

Example

```
create neuralnetworknode
# Build Options tab - Objectives panel
set: neuralnetworknode.objective = Standard
# Build Options tab - Stopping Rules panel
set: neuralnetworknode.model_selection = BestSubsets
set: neuralnetworknode.criteria_best_subsets = ASE
# Build Options tab - Ensembles panel
set: neuralnetworknode.combining_rule_categorical = HighestMeanProbability
```

neuralnetworknode Properties	Values	Property description
targets	[*field1* ... *fieldN*]	Specifies target fields.
inputs	[*field1* ... *fieldN*]	Predictor fields used by the model.
splits	[*field1* ... *fieldN*	Specifies the field or fields to use for split modeling.
use_partition	*flag*	If a partition field is defined, this option ensures that only data from the training partition is used to build the model.
continue	*flag*	Continue training existing model.
objective	Standard Bagging Boosting psm	psm is used for very large datasets, and requires a Server connection.

neuralnetworknode Properties	Values	Property description
method	MultilayerPerceptron RadialBasisFunction	
use_custom_layers	*flag*	
first_layer_units	*number*	
second_layer_units	*number*	
use_max_time	*flag*	
max_time	*number*	
use_max_cycles	*flag*	
max_cycles	*number*	
use_min_accuracy	*flag*	
min_accuracy	*number*	
combining_rule_categorical	Voting HighestProbability HighestMeanProbability	
combining_rule_continuous	Mean Median	
component_models_n	*number*	
overfit_prevention_pct	*number*	
use_random_seed	*flag*	
random_seed	*number*	
missing_values	listwiseDeletion missingValueImputation	
use_custom_model_name	*flag*	
custom_model_name	*string*	
confidence	onProbability onIncrease	
score_category_probabilities	*flag*	
max_categories	*number*	
score_propensity	*flag*	
use_custom_name	*flag*	
custom_name	*string*	
tooltip	*string*	
keywords	*string*	
annotation	*string*	

questnode Properties

The QUEST node provides a binary classification method for building decision trees, designed to reduce the processing time required for large C&R Tree analyses while also reducing the tendency found in classification tree methods to favor inputs that allow more splits. Input fields can be numeric ranges (continuous), but the target field must be categorical. All splits are binary.

Example

```
create questnode at 200 100
create variablefilenode at 100 100
connect :variablefilenode to :questnode
set :variablefilenode.full_filename = "$CLEO_DEMOS/DRUG1n"

set :questnode.custom_fields = True
set :questnode.target = Drug
set :questnode.inputs = [Age Na K Cholesterol BP]
set :questnode.model_output_type = InteractiveBuilder
set :questnode.use_tree_directives = True
set :questnode.max_surrogates = 5
set :questnode.split_alpha = 0.03
set :questnode.use_percentage = False
set :questnode.min_parent_records_abs = 40
set :questnode.min_child_records_abs = 30
set :questnode.prune_tree = True
set :questnode.use_std_err = True
set :questnode.std_err_multiplier = 3
```

questnode Properties	Values	Property description
target	*field*	QUEST models require a single target and one or more input fields. A frequency field can also be specified. For more information, see the topic Common Modeling Node Properties on p. 181.
continue_training_existing_model	*flag*	
objective	Standard Boosting Bagging psm	psm is used for very large datasets, and requires a Server connection.
model_output_type	Single InteractiveBuilder	
use_tree_directives	*flag*	
tree_directives	*string*	
use_max_depth	Default Custom	
max_depth	*integer*	Maximum tree depth, from 0 to 1000. Used only if use_max_depth = Custom.
prune_tree	*flag*	Prune tree to avoid overfitting.
use_std_err	*flag*	Use maximum difference in risk (in Standard Errors).
std_err_multiplier	*number*	Maximum difference.
max_surrogates	*number*	Maximum surrogates.
use_percentage	*flag*	
min_parent_records_pc	*number*	
min_child_records_pc	*number*	
min_parent_records_abs	*number*	

questnode Properties	Values	Property description
min_child_records_abs	*number*	
use_costs	*flag*	
costs	*structured*	Structured property using the form: [{drugA drugB 1.5} {drugA drugC 2.1}] where the arguments within braces ({ }) are actual predicted costs.
priors	Data Equal Custom	
custom_priors	*structured*	Structured property using the form: set :cartnode. custom_priors = [{ drugA 0.3 } { drugB 0.6 }]
adjust_priors	*flag*	
trails	*number*	Number of component models for boosting or bagging.
set_ensemble_method	Voting HighestProbability HighestMeanProbability	Default combining rule for categorical targets.
range_ensemble_method	Mean Median	Default combining rule for continuous targets.
large_boost	*flag*	Apply boosting to very large data sets.
split_alpha	*number*	Significance level for splitting.
train_pct	*number*	Overfit prevention set.
set_random_seed	*flag*	Replicate results option.
seed	*number*	
calculate_variable_importance	*flag*	
calculate_raw_propensities	*flag*	
calculate_adjusted_propensities	*flag*	
adjusted_propensity_partition	Test Validation	

regressionnode Properties

Linear regression is a common statistical technique for summarizing data and making predictions by fitting a straight line or surface that minimizes the discrepancies between predicted and actual output values.

Note: The Regression node is due to be replaced by the Linear node in a future release. We recommend using Linear models for linear regression from now on.

Example

```
create regressionnode
# "Fields" tab
set :regressionnode.custom_fields = True
set :regressionnode.target = 'Age'
set :regressionnode.inputs = ['Na' 'K']
```

```
set :regressionnode.partition = Test
set :regressionnode.use_weight = True
set :regressionnode.weight_field = 'Drug'
# "Model" tab
set :regressionnode.use_model_name = False
set :regressionnode.model_name = "Regression Age"
set :regressionnode.use_partitioned_data = True
set :regressionnode.method = Stepwise
set :regressionnode.include_constant = False
# "Expert" tab
set :regressionnode.mode = Expert
set :regressionnode.complete_records = False
set :regressionnode.tolerance = "1.0E-3"
# "Stepping..." section
set :regressionnode.stepping_method = Probability
set :regressionnode.probability_entry = 0.77
set :regressionnode.probability_removal = 0.88
set :regressionnode.F_value_entry = 7.0
set :regressionnode.F_value_removal = 8.0
# "Output..." section
set :regressionnode.model_fit = True
set :regressionnode.r_squared_change = True
set :regressionnode.selection_criteria = True
set :regressionnode.descriptives = True
set :regressionnode.p_correlations = True
set :regressionnode.collinearity_diagnostics = True
set :regressionnode.confidence_interval = True
set :regressionnode.covariance_matrix = True
set :regressionnode.durbin_watson = True
```

regressionnode Properties	**Values**	**Property description**
target	*field*	Regression models require a single target field and one or more input fields. A weight field can also be specified. For more information, see the topic Common Modeling Node Properties on p. 181.
method	Enter Stepwise Backwards Forwards	
include_constant	*flag*	
use_weight	*flag*	
weight_field	*field*	
mode	Simple Expert	
complete_records	*flag*	

regressionnode Properties	Values	Property description
tolerance	1.0E-1 1.0E-2 1.0E-3 1.0E-4 1.0E-5 1.0E-6 1.0E-7 1.0E-8 1.0E-9 1.0E-10 1.0E-11 1.0E-12	Use double quotes for arguments.
stepping_method	useP useF	useP : use probability of F useF: use F value
probability_entry	*number*	
probability_removal	*number*	
F_value_entry	*number*	
F_value_removal	*number*	
selection_criteria	*flag*	
confidence_interval	*flag*	
covariance_matrix	*flag*	
collinearity_diagnostics	*flag*	
regression_coefficients	*flag*	
exclude_fields	*flag*	
durbin_watson	*flag*	
model_fit	*flag*	
r_squared_change	*flag*	
p_correlations	*flag*	
descriptives	*flag*	
calculate_variable_importance	*flag*	

sequencenode Properties

The Sequence node discovers association rules in sequential or time-oriented data. A sequence is a list of item sets that tends to occur in a predictable order. For example, a customer who purchases a razor and aftershave lotion may purchase shaving cream the next time he shops. The Sequence node is based on the CARMA association rules algorithm, which uses an efficient two-pass method for finding sequences.

Example

```
create sequencenode
connect :databasenode to :sequencenode
# "Fields" tab
set :sequencenode.id_field = 'Age'
set :sequencenode.contiguous = True
set :sequencenode.use_time_field = True
set :sequencenode.time_field = 'Date1'
set :sequencenode.content_fields = ['Drug' 'BP']
```

```
set :sequencenode.partition = Test
# "Model" tab
set :sequencenode.use_model_name = True
set :sequencenode.model_name = "Sequence_test"
set :sequencenode.use_partitioned_data = False
set :sequencenode.min_supp = 15.0
set :sequencenode.min_conf = 14.0
set :sequencenode.max_size = 7
set :sequencenode.max_predictions = 5
# "Expert" tab
set :sequencenode.mode = Expert
set :sequencenode.use_max_duration = True
set :sequencenode.max_duration = 3.0
set :sequencenode.use_pruning = True
set :sequencenode.pruning_value = 4.0
set :sequencenode.set_mem_sequences = True
set :sequencenode.mem_sequences = 5.0
set :sequencenode.use_gaps = True
set :sequencenode.min_item_gap = 20.0
set :sequencenode.max_item_gap = 30.0
```

sequencenode Properties	**Values**	**Property description**
id_field	*field*	To create a Sequence model, you need to specify an ID field, an optional time field, and one or more content fields. Weight and frequency fields are not used. For more information, see the topic Common Modeling Node Properties on p. 181.
time_field	*field*	
use_time_field	*flag*	
content_fields	[*field1* ... *fieldn*]	
contiguous	*flag*	
min_supp	*number*	
min_conf	*number*	
max_size	*number*	
max_predictions	*number*	
mode	Simple Expert	
use_max_duration	*flag*	
max_duration	*number*	
use_gaps	*flag*	
min_item_gap	*number*	
max_item_gap	*number*	
use_pruning	*flag*	
pruning_value	*number*	
set_mem_sequences	*flag*	
mem_sequences	*integer*	

slrmnode Properties

The Self-Learning Response Model (SLRM) node enables you to build a model in which a single new case, or small number of new cases, can be used to reestimate the model without having to retrain the model using all data.

Example

```
create slrmnode
set :slrmnode.target = Offer
set :slrmnode.target_response = Response
set :slrmnode.inputs = ['Cust_ID' 'Age' 'Ave_Bal']
```

slrmnode Properties	Values	Property description
target	*field*	The target field must be a nominal or flag field. A frequency field can also be specified. For more information, see the topic Common Modeling Node Properties on p. 181.
target_response	*field*	Type must be flag.
continue_training_existing_model	*flag*	
target_field_values	*flag*	Use all: Use all values from source. Specify: Select values required.
target_field_values_specify	*[field1 ... fieldN]*	
include_model_assessment	*flag*	
model_assessment_random_seed	*number*	Must be a real number.
model_assessment_sample_size	*number*	Must be a real number.
model_assessment_iterations	*number*	Number of iterations.
display_model_evaluation	*flag*	
max_predictions	*number*	
randomization	*number*	
scoring_random_seed	*number*	
sort	Ascending Descending	Specifies whether the offers with the highest or lowest scores will be displayed first.
model_reliability	*flag*	
calculate_variable_importance	*flag*	

statisticsmodelnode Properties

The Statistics Model node enables you to analyze and work with your data by running IBM® SPSS® Statistics procedures that produce PMML. This node requires a licensed copy of SPSS Statistics.

The properties for this node are described under statisticsmodelnode Properties on p. 293.

svmnode Properties

The Support Vector Machine (SVM) node enables you to classify data into one of two groups without overfitting. SVM works well with wide data sets, such as those with a very large number of input fields.

Example

```
create svmnode
# Expert tab
set :svmnode.mode=Expert
set :svmnode.all_probabilities=True
set :svmnode.kernel=Polynomial
set :svmnode.gamma=1.5
```

svmnode Properties	Values	Property description
all_probabilities	*flag*	
stopping_criteria	1.0E-1 1.0E-2 1.0E-3 (default) 1.0E-4 1.0E-5 1.0E-6	Determines when to stop the optimization algorithm.
regularization	*number*	Also known as the C parameter.
precision	*number*	Used only if measurement level of target field is Continuous.
kernel	RBF(default) Polynomial Sigmoid Linear	Type of kernel function used for the transformation.
rbf_gamma	*number*	Used only if kernel is RBF.
gamma	*number*	Used only if kernel is Polynomial or Sigmoid.
bias	*number*	
degree	*number*	Used only if kernel is Polynomial.
calculate_variable_importance	*flag*	
calculate_raw_propensities	*flag*	
calculate_adjusted_propensities	*flag*	
adjusted_propensity_partition	Test Validation	

timeseriesnode Properties

The Time Series node estimates exponential smoothing, univariate Autoregressive Integrated Moving Average (ARIMA), and multivariate ARIMA (or transfer function) models for time series data and produces forecasts of future performance. A Time Series node must always be preceded by a Time Intervals node.

Example

```
create timeseriesnode
set :timeseriesnode.method = Exsmooth
set :timeseriesnode.exsmooth_model_type = HoltsLinearTrend
set :timeseriesnode.exsmooth_transformation_type = None
```

timeseriesnode Properties	Values	Property description
targets	*field*	The Time Series node forecasts one or more targets, optionally using one or more input fields as predictors. Frequency and weight fields are not used. For more information, see the topic Common Modeling Node Properties on p. 181.
continue	*flag*	
method	ExpertModeler Exsmooth Arima Reuse	
expert_modeler_method	*flag*	
consider_seasonal	*flag*	
detect_outliers	*flag*	
expert_outlier_additive	*flag*	
expert_outlier_level_shift	*flag*	
expert_outlier_innovational	*flag*	
expert_outlier_level_shift	*flag*	
expert_outlier_transient	*flag*	
expert_outlier_seasonal_additive	*flag*	
expert_outlier_local_trend	*flag*	
expert_outlier_additive_patch	*flag*	
exsmooth_model_type	Simple HoltsLinearTrend BrownsLinearTrend DampedTrend SimpleSeasonal WintersAdditive WintersMultiplicative	
exsmooth_transformation_type	None SquareRoot NaturalLog	
arima_p	*integer*	
arima_d	*integer*	
arima_q	*integer*	
arima_sp	*integer*	
arima_sd	*integer*	

timeseriesnode Properties	Values	Property description
arima_sq	*integer*	
arima_transformation_type	None SquareRoot NaturalLog	
arima_include_constant	*flag*	
tf_arima_p.*fieldname*	*integer*	For transfer functions.
tf_arima_d.*fieldname*	*integer*	For transfer functions.
tf_arima_q.*fieldname*	*integer*	For transfer functions.
tf_arima_sp.*fieldname*	*integer*	For transfer functions.
tf_arima_sd.*fieldname*	*integer*	For transfer functions.
tf_arima_sq.*fieldname*	*integer*	For transfer functions.
tf_arima_delay.*fieldname*	*integer*	For transfer functions.
tf_arima_transformation_type.*fieldname*	None SquareRoot NaturalLog	For transfer functions.
arima_detect_outlier_mode	None Automatic	
arima_outlier_additive	*flag*	
arima_outlier_level_shift	*flag*	
arima_outlier_innovational	*flag*	
arima_outlier_transient	*flag*	
arima_outlier_seasonal_additive	*flag*	
arima_outlier_local_trend	*flag*	
arima_outlier_additive_patch	*flag*	
conf_limit_pct	*real*	
max_lags	*integer*	
events	*fields*	
scoring_model_only	*flag*	Use for models with very large numbers (tens of thousands) of time series.

twostepnode Properties

The TwoStep node uses a two-step clustering method. The first step makes a single pass through the data to compress the raw input data into a manageable set of subclusters. The second step uses a hierarchical clustering method to progressively merge the subclusters into larger and larger clusters. TwoStep has the advantage of automatically estimating the optimal number of clusters for the training data. It can handle mixed field types and large data sets efficiently.

Example

```
create twostep
set :twostep.custom_fields = True
set :twostep.inputs = ['Age' 'K' 'Na' 'BP']
set :twostep.partition = Test
set :twostep.use_model_name = False
```

```
set :twostep.model_name = "TwoStep_Drug"
set :twostep.use_partitioned_data = True
set :twostep.exclude_outliers = True
set :twostep.cluster_label = "String"
set :twostep.label_prefix = "TwoStep_"
set :twostep.cluster_num_auto = False
set :twostep.max_num_clusters = 9
set :twostep.min_num_clusters = 3
set :twostep.num_clusters = 7
```

twostepnode Properties	**Values**	**Property description**
inputs	[*field1 ... fieldN*]	TwoStep models use a list of input fields, but no target. Weight and frequency fields are not recognized. For more information, see the topic Common Modeling Node Properties on p. 181.
standardize	*flag*	
exclude_outliers	*flag*	
percentage	*number*	
cluster_num_auto	*flag*	
min_num_clusters	*number*	
max_num_clusters	*number*	
num_clusters	*number*	
cluster_label	String Number	
label_prefix	*string*	
distance_measure	Euclidean Loglikelihood	
clustering_criterion	AIC BIC	

Chapter

17

Model Nugget Node Properties

Model nugget nodes share the same common properties as other nodes. For more information, see the topic Common Node Properties in Chapter 9 on p. 109.

applyanomalydetectionnode Properties

Anomaly Detection modeling nodes can be used to generate an Anomaly Detection model nugget. The scripting name of this model nugget is *applyanomalydetectionnode*. For more information on scripting the modeling node itself, see anomalydetectionnode Properties in Chapter 16 on p. 181.

applyanomalydetectionnode Properties	Values	Property description
anomaly_score_method	FlagAndScore FlagOnly ScoreOnly	Determines which outputs are created for scoring.
num_fields	*integer*	Fields to report.
discard_records	*flag*	Indicates whether records are discarded from the output or not.
discard_anomalous_records	*flag*	Indicator of whether to discard the anomalous or *non*-anomalous records. The default is off, meaning that *non*-anomalous records are discarded. Otherwise, if on, anomalous records will be discarded. This property is enabled only if the discard_records property is enabled.

applyapriorinode Properties

Apriori modeling nodes can be used to generate an Apriori model nugget. The scripting name of this model nugget is *applyapriorinode*. For more information on scripting the modeling node itself, see apriorinode Properties in Chapter 16 on p. 183.

applyapriorinode Properties	Values	Property description
max_predictions	*number (integer)*	
ignore_unmatached	*flag*	
allow_repeats	*flag*	
check_basket	NoPredictions Predictions NoCheck	
criterion	Confidence Support RuleSupport Lift Deployability	

applyautoclassifiernode Properties

Auto Classifier modeling nodes can be used to generate an Auto Classifier model nugget. The scripting name of this model nugget is *applyautoclassifiernode*.For more information on scripting the modeling node itself, see autoclassifiernode Properties in Chapter 16 on p. 184.

applyautoclassifiernode Properties	Values	Property description
flag_ensemble_method	Voting ConfidenceWeightedVoting RawPropensityWeightedVoting HighestConfidence AverageRawPropensity	Specifies the method used to determine the ensemble score. This setting applies only if the selected target is a flag field.
flag_voting_tie_selection	Random HighestConfidence RawPropensity	If a voting method is selected, specifies how ties are resolved. This setting applies only if the selected target is a flag field.
set_ensemble_method	Voting ConfidenceWeightedVoting HighestConfidence	Specifies the method used to determine the ensemble score. This setting applies only if the selected target is a set field.
set_voting_tie_selection	Random HighestConfidence	If a voting method is selected, specifies how ties are resolved. This setting applies only if the selected target is a nominal field.

applyautoclusternode Properties

Auto Cluster modeling nodes can be used to generate an Auto Cluster model nugget. The scripting name of this model nugget is *applyautoclusternode*. No other properties exist for this model nugget. For more information on scripting the modeling node itself, see autoclusternode Properties in Chapter 16 on p. 187.

applyautonumericnode Properties

Auto Numeric modeling nodes can be used to generate an Auto Numeric model nugget. The scripting name of this model nugget is *applyautonumericnode*.For more information on scripting the modeling node itself, see autonumericnode Properties in Chapter 16 on p. 188.

applyautonumericnode Properties	Values	Property description
calculate_standard_error	*flag*	

applybayesnetnode Properties

Bayesian network modeling nodes can be used to generate a Bayesian network model nugget. The scripting name of this model nugget is *applybayesnetnode*. For more information on scripting the modeling node itself, see bayesnetnode Properties in Chapter 16 on p. 189.

applybayesnetnode Properties	Values	Property description
all_probabilities	*flag*	
raw_propensity	*flag*	
adjusted_propensity	*flag*	
calculate_raw_propensities	*flag*	
calculate_adjusted_propensities	*flag*	

applyc50node Properties

C5.0 modeling nodes can be used to generate a C5.0 model nugget. The scripting name of this model nugget is *applyc50node*. For more information on scripting the modeling node itself, see c50node Properties in Chapter 16 on p. 191.

applyc50node Properties	Values	Property description
sql_generate	Never NoMissingValues	Used to set SQL generation options during rule set execution.
calculate_conf	*flag*	Available when SQL generation is enabled; this property includes confidence calculations in the generated tree.
calculate_raw_propensities	*flag*	
calculate_adjusted_propensities	*flag*	

applycarmanode Properties

CARMA modeling nodes can be used to generate a CARMA model nugget. The scripting name of this model nugget is *applycarmanode*. No other properties exist for this model nugget. For more information on scripting the modeling node itself, see carmanode Properties in Chapter 16 on p. 192.

applycartnode Properties

C&R Tree modeling nodes can be used to generate a C&R Tree model nugget. The scripting name of this model nugget is *applycartnode*. For more information on scripting the modeling node itself, see cartnode Properties in Chapter 16 on p. 193.

applycartnode Properties	Values	Property description
sql_generate	Never MissingValues NoMissingValues	Used to set SQL generation options during rule set execution.

applycartnode Properties	Values	Property description
calculate_conf	*flag*	Available when SQL generation is enabled; this property includes confidence calculations in the generated tree.
display_rule_id	*flag*	Adds a field in the scoring output that indicates the ID for the terminal node to which each record is assigned.
calculate_raw_propensities	*flag*	
calculate_adjusted_propensities	*flag*	

applychaidnode Properties

CHAID modeling nodes can be used to generate a CHAID model nugget. The scripting name of this model nugget is *applychaidnode*. For more information on scripting the modeling node itself, see chaidnode Properties in Chapter 16 on p. 195.

applychaidnode Properties	Values	Property description
sql_generate	Never MissingValues	
calculate_conf	*flag*	
display_rule_id	*flag*	Adds a field in the scoring output that indicates the ID for the terminal node to which each record is assigned.
calculate_raw_propensities	*flag*	
calculate_adjusted_propensities	*flag*	

applycoxregnode Properties

Cox modeling nodes can be used to generate a Cox model nugget. The scripting name of this model nugget is *applycoxregnode*. For more information on scripting the modeling node itself, see coxregnode Properties in Chapter 16 on p. 197.

applycoxregnode Properties	Values	Property description
future_time_as	Intervals Fields	
time_interval	*number*	
num_future_times	*integer*	
time_field	*field*	
past_survival_time	*field*	
all_probabilities	*flag*	
cumulative_hazard	*flag*	

applydecisionlistnode Properties

Decision List modeling nodes can be used to generate a Decision List model nugget. The scripting name of this model nugget is *applydecisionlistnode*. For more information on scripting the modeling node itself, see decisionlistnode Properties in Chapter 16 on p. 199.

applydecisionlistnode Properties	Values	Property description
enable_sql_generation	*flag*	When true, IBM® SPSS® Modeler will try to push back the Decision List model to SQL.
calculate_raw_propensities	*flag*	
calculate_adjusted_propensities	*flag*	

applydiscriminantnode Properties

Discriminant modeling nodes can be used to generate a Discriminant model nugget. The scripting name of this model nugget is *applydiscriminantnode*. For more information on scripting the modeling node itself, see discriminantnode Properties in Chapter 16 on p. 200.

applydiscriminantnode Properties	Values	Property description
calculate_raw_propensities	*flag*	
calculate_adjusted_propensities	*flag*	

applyfactornode Properties

PCA/Factor modeling nodes can be used to generate a PCA/Factor model nugget. The scripting name of this model nugget is *applyfactornode*. No other properties exist for this model nugget. For more information on scripting the modeling node itself, see factornode Properties in Chapter 16 on p. 202.

applyfeatureselectionnode Properties

Feature Selection modeling nodes can be used to generate a Feature Selection model nugget. The scripting name of this model nugget is *applyfeatureselectionnode*. For more information on scripting the modeling node itself, see featureselectionnode Properties in Chapter 16 on p. 203.

applyfeatureselectionnode Properties	Values	Property description
selected_ranked_fields		Specifies which ranked fields are checked in the model browser.
selected_screened_fields		Specifies which screened fields are checked in the model browser.

applygeneralizedlinearnode Properties

Generalized Linear (genlin) modeling nodes can be used to generate a Generalized Linear model nugget. The scripting name of this model nugget is *applygeneralizedlinearnode*. For more information on scripting the modeling node itself, see genlinnode Properties in Chapter 16 on p. 205.

applygeneralizedlinearnode Properties	Values	Property description
calculate_raw_propensities	*flag*	
calculate_adjusted_propensities	*flag*	

applykmeansnode Properties

K-Means modeling nodes can be used to generate a K-Means model nugget. The scripting name of this model nugget is *applykmeansnode*. No other properties exist for this model nugget. For more information on scripting the modeling node itself, see kmeansnode Properties in Chapter 16 on p. 211.

applyknnnode Properties

KNN modeling nodes can be used to generate a KNN model nugget. The scripting name of this model nugget is *applyknnnode*. For more information on scripting the modeling node itself, see knnnode Properties in Chapter 16 on p. 212.

applyknnnode Properties	Values	Property description
all_probabilities	*flag*	
save_distances	*flag*	

applykohonennode Properties

Kohonen modeling nodes can be used to generate a Kohonen model nugget. The scripting name of this model nugget is *applykohonennode*. No other properties exist for this model nugget. For more information on scripting the modeling node itself, see c50node Properties in Chapter 16 on p. 191.

applylinearnode Properties

Linear modeling nodes can be used to generate a Linear model nugget. The scripting name of this model nugget is *applylinearnode*. For more information on scripting the modeling node itself, see linearnode Properties in Chapter 16 on p. 214.

linear Properties	Values	Property description
use_custom_name	*flag*	
custom_name	*string*	
enable_sql_generation	*flag*	

applylogregnode Properties

Logistic Regression modeling nodes can be used to generate a Logistic Regression model nugget. The scripting name of this model nugget is *applylogregnode*. For more information on scripting the modeling node itself, see logregnode Properties in Chapter 16 on p. 215.

applylogregnode Properties	Values	Property description
calculate_raw_propensities	*flag*	

applyneuralnetnode Properties

Neural Net modeling nodes can be used to generate a Neural Net model nugget. The scripting name of this model nugget is *applyneuralnetnode*. For more information on scripting the modeling node itself, see neuralnetnode Properties in Chapter 16 on p. 220.

Caution: A newer version of the Neural Net nugget, with enhanced features, is available in this release and is described in the next section (*applyneuralnetwork*). Although the previous version is still available, we recommend updating your scripts to use the new version. Details of the previous version are retained here for reference, but support for it will be removed in a future release.

applyneuralnetnode Properties	Values	Property description
calculate_conf	*flag*	Available when SQL generation is enabled; this property includes confidence calculations in the generated tree.
enable_sql_generation	*flag*	
nn_score_method	Difference SoftMax	
calculate_raw_propensities	*flag*	
calculate_adjusted_propensities	*flag*	

applyneuralnetworknode Properties

Neural Network modeling nodes can be used to generate a Neural Network model nugget. The scripting name of this model nugget is *applyneuralnetworknode*. For more information on scripting the modeling node itself, see neuralnetworknode Properties in Chapter 16 on p. 222.

applyneuralnetworknode Properties	Values	Property description
use_custom_name	*flag*	
custom_name	*string*	
confidence	onProbability onIncrease	
score_category_probabilities	*flag*	
max_categories	*number*	
score_propensity	*flag*	

applyquestnode Properties

QUEST modeling nodes can be used to generate a QUEST model nugget. The scripting name of this model nugget is *applyquestnode*. For more information on scripting the modeling node itself, see questnode Properties in Chapter 16 on p. 223.

applyquestnode Properties	Values	Property description
sql_generate	Never MissingValues NoMissingValues	
calculate_conf	*flag*	
display_rule_id	*flag*	Adds a field in the scoring output that indicates the ID for the terminal node to which each record is assigned.
calculate_raw_propensities	*flag*	
calculate_adjusted_propensities	*flag*	

applyregressionnode Properties

Linear Regression modeling nodes can be used to generate a Linear Regression model nugget. The scripting name of this model nugget is *applyregressionnode*. No other properties exist for this model nugget. For more information on scripting the modeling node itself, see regressionnode Properties in Chapter 16 on p. 225.

applyselflearningnode Properties

Self-Learning Response Model (SLRM) modeling nodes can be used to generate a SLRM model nugget. The scripting name of this model nugget is *applyselflearningnode*. For more information on scripting the modeling node itself, see slrmnode Properties in Chapter 16 on p. 229.

applyselflearningnode Properties	Values	Property description
max_predictions	*number*	
randomization	*number*	
scoring_random_seed	*number*	
sort	ascending descending	Specifies whether the offers with the highest or lowest scores will be displayed first.
model_reliability	*flag*	Takes account of model reliability option on Settings tab.

applysequencenode Properties

Sequence modeling nodes can be used to generate a Sequence model nugget. The scripting name of this model nugget is *applysequencenode*. No other properties exist for this model nugget. For more information on scripting the modeling node itself, see sequencenode Properties in Chapter 16 on p. 227.

applysvmnode Properties

SVM modeling nodes can be used to generate an SVM model nugget. The scripting name of this model nugget is *applysvmnode*. For more information on scripting the modeling node itself, see svmnode Properties in Chapter 16 on p. 230.

applysvmnode Properties	Values	Property description
all_probabilities	*flag*	
calculate_raw_propensities	*flag*	
calculate_adjusted_propensities	*flag*	

applytimeseriesnode Properties

Time Series modeling nodes can be used to generate a Time Series model nugget. The scripting name of this model nugget is *applytimeseriesnode*. For more information on scripting the modeling node itself, see timeseriesnode Properties in Chapter 16 on p. 230.

applytimeseriesnode Properties	Values	Property description
calculate_conf	*flag*	
calculate_residuals	*flag*	

applytwostepnode Properties

TwoStep modeling nodes can be used to generate a TwoStep model nugget. The scripting name of this model nugget is *applytwostepnode*. No other properties exist for this model nugget. For more information on scripting the modeling node itself, see twostepnode Properties in Chapter 16 on p. 232.

Chapter 18

Database Modeling Node Properties

IBM® SPSS® Modeler supports integration with data mining and modeling tools available from database vendors, including Microsoft SQL Server Analysis Services, Oracle Data Mining, IBM® DB2® InfoSphere Warehouse, and IBM® Netezza® Analytics. You can build and score models using native database algorithms, all from within the SPSS Modeler application. Database models can also be created and manipulated through scripting using the properties described in this section.

For example, the following script excerpt illustrates the creation of a Microsoft Decision Trees model by using SPSS Modeler's scripting interface:

```
create mstreenode
rename :mstreenode as msbuilder
set msbuilder.analysis_server_name = 'localhost'
set msbuilder.analysis_database_name = 'TESTDB'
set msbuilder.mode = 'Expert'
set msbuilder.datasource = 'LocalServer'
set msbuilder.target = 'Drug'
set msbuilder.inputs = ['Age' 'Sex']
set msbuilder.unique_field = 'IDX'
set msbuilder.custom_fields = true
set msbuilder.model_name = 'MSDRUG'

connect :typenode to msbuilder
execute msbuilder

insert model MSDRUG connected between :typenode and :tablenode
set MSDRUG.sql_generate = true
execute :tablenode
```

Node Properties for Microsoft Modeling

Microsoft Modeling Node Properties

Common Properties

The following properties are common to the Microsoft database modeling nodes.

Common Microsoft Node Properties	Values	Property Description
analysis_database_name	*string*	Name of the Analysis Services database.
analysis_server_name	*string*	Name of the Analysis Services host.
use_transactional_data	*flag*	Specifies whether input data is in tabular or transactional format.
inputs	*[field field field]*	Input fields for tabular data.

Common Microsoft Node Properties	Values	Property Description
target	*field*	Predicted field (not applicable to MS Clustering or Sequence Clustering nodes).
unique_field	*field*	Key field.
msas_parameters	*structured*	Algorithm parameters. For more information, see the topic Algorithm Parameters on p. 245.
with_drillthrough	*flag*	With Drillthrough option.

MS Decision Tree

There are no specific properties defined for nodes of type mstreenode. See the common Microsoft properties at the start of this section.

MS Clustering

There are no specific properties defined for nodes of type msclusternode. See the common Microsoft properties at the start of this section.

MS Association Rules

The following specific properties are available for nodes of type msassocnode:

msassocnode Properties	Values	Property Description
id_field	*field*	Identifies each transaction in the data.
trans_inputs	*[field field field]*	Input fields for transactional data.
transactional_target	*field*	Predicted field (transactional data).

MS Naive Bayes

There are no specific properties defined for nodes of type msbayesnode. See the common Microsoft properties at the start of this section.

MS Linear Regression

There are no specific properties defined for nodes of type msregressionnode. See the common Microsoft properties at the start of this section.

MS Neural Network

There are no specific properties defined for nodes of type msneuralnetworknode. See the common Microsoft properties at the start of this section.

MS Logistic Regression

There are no specific properties defined for nodes of type mslogisticnode. See the common Microsoft properties at the start of this section.

MS Time Series

There are no specific properties defined for nodes of type mstimeseriesnode. See the common Microsoft properties at the start of this section.

MS Sequence Clustering

The following specific properties are available for nodes of type mssequenceclusternode:

mssequenceclusternode Properties	Values	Property Description
id_field	*field*	Identifies each transaction in the data.
input_fields	*[field field field]*	Input fields for transactional data.
sequence_field	*field*	Sequence identifier.
target_field	*field*	Predicted field (tabular data).

Algorithm Parameters

Each Microsoft database model type has specific parameters that can be set using the msas_parameters property—for example:

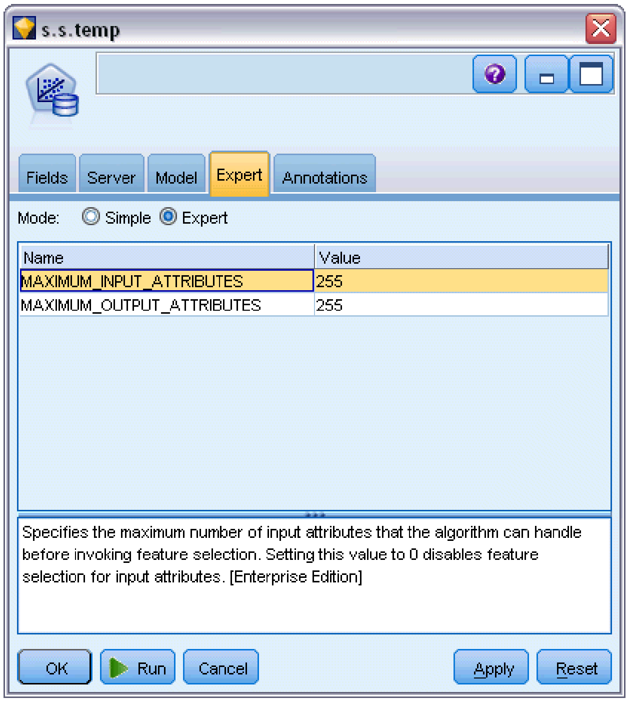
```
set :msregressionnode.msas_parameters =
[{"MAXIMUM_INPUT_ATTRIBUTES" 255}{"MAXIMUM_OUTPUT_ATTRIBUTES" 255}]
```

These parameters are derived from SQL Server. To see the relevant parameters for each node:

- ▶ Place a database source node on the canvas.
- ▶ Open the database source node.
- ▶ Select a valid source from the Data source drop-down list.
- ▶ Select a valid table from the Table name list.
- ▶ Click OK to close the database source node.
- ▶ Attach the Microsoft database modeling node whose properties you want to list.
- ▶ Open the database modeling node.
- ▶ Select the Expert tab.

The available msas_parameters properties for this node are displayed.

Figure 18-1
Example of algorithm parameter display



Microsoft Model Nugget Properties

The following properties are for the model nuggets created using the Microsoft database modeling nodes.

MS Decision Tree

applymstreenode Properties	Values	Description
analysis_database_name	*string*	This node can be scored directly in a stream. This property is used to identify the name of the Analysis Services database.
analysis_server_name	*string*	Name of the Analysis server host.
datasource	*string*	Name of the SQL Server ODBC data source name (DSN).
sql_generate	*flag*	Enables SQL generation.

MS Linear Regression

applymsregressionnode Properties	Values	Description
analysis_database_name	*string*	This node can be scored directly in a stream. This property is used to identify the name of the Analysis Services database.
analysis_server_name	*string*	Name of the Analysis server host.

MS Neural Network

applymsneuralnetworknode Properties	Values	Description
analysis_database_name	*string*	This node can be scored directly in a stream. This property is used to identify the name of the Analysis Services database.
analysis_server_name	*string*	Name of the Analysis server host.

MS Logistic Regression

applymslogisticnode Properties	Values	Description
analysis_database_name	*string*	This node can be scored directly in a stream. This property is used to identify the name of the Analysis Services database.
analysis_server_name	*string*	Name of the Analysis server host.

MS Time Series

applymstimeseriesnode Properties	Values	Description
analysis_database_name	*string*	This node can be scored directly in a stream. This property is used to identify the name of the Analysis Services database.
analysis_server_name	*string*	Name of the Analysis server host.
start_from	new_prediction historical_prediction	Specifies whether to make future predictions or historical predictions.
new_step	*number*	Defines starting time period for future predictions.
historical_step	*number*	Defines starting time period for historical predictions.
end_step	*number*	Defines ending time period for predictions.

MS Sequence Clustering

applymssequenceclusternode Properties	Values	Description
analysis_database_name	*string*	This node can be scored directly in a stream. This property is used to identify the name of the Analysis Services database.
analysis_server_name	*string*	Name of the Analysis server host.

Node Properties for Oracle Modeling

Oracle Modeling Node Properties

The following properties are common to Oracle database modeling nodes.

Common Oracle Node Properties	Values	Property Description
target	*field*	
inputs	*List of fields*	
partition	*field*	Field used to partition the data into separate samples for the training, testing, and validation stages of model building.
datasource		
username		
password		
epassword		
use_model_name	*flag*	
model_name	*string*	Custom name for new model.
use_partitioned_data	*flag*	If a partition field is defined, this option ensures that only data from the training partition is used to build the model.
unique_field	*field*	
auto_data_prep	*flag*	Enables or disables the Oracle automatic data preparation feature (11g databases only).
costs	*structured*	Structured property in the form: [{drugA drugB 1.5} {drugA drugC 2.1}], where the arguments in { } are actual predicted costs.
mode	Simple Expert	Causes certain properties to be ignored if set to Simple, as noted in the individual node properties.
use_prediction_probability	*flag*	
prediction_probability	*string*	
use_prediction_set	*flag*	

Oracle Naive Bayes

The following properties are available for nodes of type oranbnode.

oranbnode Properties	Values	Property Description
singleton_threshold	*number*	0.0–1.0.*
pairwise_threshold	*number*	0.0–1.0.*
priors	Data Equal Custom	
custom_priors	*structured*	Structured property in the form: set :oranbnode.custom_priors = [{drugA 1}{drugB 2}{drugC 3}{drugX 4}{drugY 5}]

* Property ignored if mode is set to Simple.

Oracle Adaptive Bayes

The following properties are available for nodes of type oraabnnode.

oraabnnode Properties	Values	Property Description
model_type	SingleFeature MultiFeature NaiveBayes	
use_execution_time_limit	*flag*	*
execution_time_limit	*integer*	Value must be greater than 0.*
max_naive_bayes_predictors	*integer*	Value must be greater than 0.*
max_predictors	*integer*	Value must be greater than 0.*
priors	Data Equal Custom	
custom_priors	*structured*	Structured property in the form: set :oraabnnode.custom_priors = [{drugA 1}{drugB 2}{drugC 3}{drugX 4}{drugY 5}]

* Property ignored if mode is set to Simple.

Oracle Support Vector Machines

The following properties are available for nodes of type orasvmnode.

orasvmnode Properties	Values	Property Description
active_learning	Enable Disable	
kernel_function	Linear Gaussian System	
normalization_method	zscore minmax none	
kernel_cache_size	*integer*	Gaussian kernel only. Value must be greater than 0.*
convergence_tolerance	*number*	Value must be greater than 0.*
use_standard_deviation	*flag*	Gaussian kernel only.*
standard_deviation	*number*	Value must be greater than 0.*
use_epsilon	*flag*	Regression models only.*
epsilon	*number*	Value must be greater than 0.*
use_complexity_factor	*flag*	*
complexity_factor	*number*	*
use_outlier_rate	*flag*	One-Class variant only.*
outlier_rate	*number*	One-Class variant only. 0.0–1.0.*

orasvmnode Properties	Values	Property Description
weights	Data Equal Custom	
custom_weights	*structured*	Structured property in the form: set :orasvmnode.custom_weights = [{drugA 1}{drugB 2}{drugC 3}{drugX 4}{drugY 5}]

* Property ignored if mode is set to Simple.

Oracle Generalized Linear Models

The following properties are available for nodes of type oraglmnode.

oraglmnode Properties	Values	Property Description
normalization_method	zscore minmax none	
missing_value_handling	ReplaceWith-Mean UseCompleteRe-cords	
use_row_weights	*flag*	*
row_weights_field	*field*	*
save_row_diagnostics	*flag*	*
row_diagnostics_table	*string*	*
coefficient_confidence	*number*	*
use_reference_category	*flag*	*
reference_category	*string*	*
ridge_regression	Auto Off On	*
parameter_value	*number*	*
vif_for_ridge	*flag*	*

* Property ignored if mode is set to Simple.

Oracle Decision Tree

The following properties are available for nodes of type oradecisiontreenode.

oradecisiontreenode Properties	Values	Property Description
use_costs	*flag*	
impurity_metric	Entropy Gini	
term_max_depth	*integer*	2–20.*
term_minpct_node	*number*	0.0–10.0.*
term_minpct_split	*number*	0.0–20.0.*
term_minrec_node	*integer*	Value must be greater than 0.*

oradecisiontreenode Properties	Values	Property Description
term_minrec_split	*integer*	Value must be greater than 0.*
display_rule_ids	*flag*	*

* Property ignored if mode is set to Simple.

Oracle O-Cluster

The following properties are available for nodes of type oraoclusternode.

oraoclusternode Properties	Values	Property Description
max_num_clusters	*integer*	Value must be greater than 0.
max_buffer	*integer*	Value must be greater than 0.*
sensitivity	*number*	0.0–1.0.*

* Property ignored if mode is set to Simple.

Oracle KMeans

The following properties are available for nodes of type orakmeansnode.

orakmeansnode Properties	Values	Property Description
num_clusters	*integer*	Value must be greater than 0.
normalization_method	zscore minmax none	
distance_function	Euclidean Cosine	
iterations	*integer*	0–20.*
conv_tolerance	*number*	0.0–0.5.*
split_criterion	Variance Size	Default is Variance.*
num_bins	*integer*	Value must be greater than 0.*
block_growth	*integer*	1–5.*
min_pct_attr_support	*number*	0.0–1.0.*

* Property ignored if mode is set to Simple.

Oracle NMF

The following properties are available for nodes of type oranmfnode.

oranmfnode Properties	Values	Property Description
normalization_method	minmax none	
use_num_features	*flag*	*
num_features	*integer*	0–1. Default value is estimated from the data by the algorithm.*

oranmfnode Properties	Values	Property Description
random_seed	*number*	*
num_iterations	*integer*	0–500.*
conv_tolerance	*number*	0.0–0.5.*
display_all_features	*flag*	*

* Property ignored if mode is set to Simple.

Oracle Apriori

The following properties are available for nodes of type oraapriorinode.

oraapriorinode Properties	Values	Property Description
content_field	*field*	
id_field	*field*	
max_rule_length	*integer*	2–20.
min_confidence	*number*	0.0–1.0.
min_support	*number*	0.0–1.0.
use_transactional_data	*flag*	

Oracle Minimum Description Length (MDL)

There are no specific properties defined for nodes of type oramdlnode. See the common Oracle properties at the start of this section.

Oracle Attribute Importance (AI)

The following properties are available for nodes of type oraainode.

oraainode Properties	Values	Property Description
custom_fields	*flag*	If true, allows you to specify target, input, and other fields for the current node. If false, the current settings from an upstream Type node are used.
selection_mode	ImportanceLevel ImportanceValue TopN	
select_important	*flag*	When selection_mode is set to ImportanceLevel, specifies whether to select important fields.
important_label	*string*	Specifies the label for the "important" ranking.
select_marginal	*flag*	When selection_mode is set to ImportanceLevel, specifies whether to select marginal fields.
marginal_label	*string*	Specifies the label for the "marginal" ranking.
important_above	*number*	0.0–1.0.
select_unimportant	*flag*	When selection_mode is set to ImportanceLevel, specifies whether to select unimportant fields.

oraainode Properties	Values	Property Description
unimportant_label	*string*	Specifies the label for the "unimportant" ranking.
unimportant_below	*number*	0.0–1.0.
importance_value	*number*	When selection_mode is set to ImportanceValue, specifies the cutoff value to use. Accepts values from 0 to 100.
top_n	*number*	When selection_mode is set to TopN, specifies the cutoff value to use. Accepts values from 0 to 1000.

Oracle Model Nugget Properties

The following properties are for the model nuggets created using the Oracle models.

Oracle Naive Bayes

There are no specific properties defined for nodes of type applyoranbnode.

Oracle Adaptive Bayes

There are no specific properties defined for nodes of type applyoraabnnode.

Oracle Support Vector Machines

There are no specific properties defined for nodes of type applyorasvmnode.

Oracle Decision Tree

The following properties are available for nodes of type applyoradecisiontreenode.

applyoradecisiontreenode Properties	Values	Property Description
use_costs	*flag*	
display_rule_ids	*flag*	

Oracle O-Cluster

There are no specific properties defined for nodes of type applyoraoclusternode.

Oracle KMeans

There are no specific properties defined for nodes of type applyorakmeansnode.

Oracle NMF

The following property is available for nodes of type applyoranmfnode:

applyoranmfnode Properties	Values	Property Description
display_all_features	*flag*	

Oracle Apriori

This model nugget cannot be applied in scripting.

Oracle MDL

This model nugget cannot be applied in scripting.

Node Properties for IBM DB2 Modeling

IBM DB2 Modeling Node Properties

The following properties are common to IBM InfoSphere Warehouse (ISW) database modeling nodes.

Common ISW node Properties	Values	Property Description
inputs	*List of fields*	
datasource		
username		
password		
epassword		
enable_power_options	*flag*	
power_options_max_memory	*integer*	Value must be greater than 32.
power_options_cmdline	*string*	
mining_data_custom_sql	*string*	
logical_data_custom_sql	*string*	
mining_settings_custom_sql		

ISW Decision Tree

The following properties are available for nodes of type db2imtreenode.

db2imtreenode Properties	Values	Property Description
target	*field*	
perform_test_run	*flag*	
use_max_tree_depth	*flag*	
max_tree_depth	*integer*	Value greater than 0.
use_maximum_purity	*flag*	
maximum_purity	*number*	Number between 0 and 100.
use_minimum_internal_cases	*flag*	
minimum_internal_cases	*integer*	Value greater than 1.
use_costs	*flag*	
costs	*structured*	Structured property in the form: [{drugA drugB 1.5} {drugA drugC 2.1}], where the arguments in { } are actual predicted costs.

ISW Association

The following properties are available for nodes of type db2imassocnode.

db2imassocnode Properties	Values	Property Description
use_transactional_data	*flag*	
id_field	*field*	
content_field	*field*	
data_table_layout	basic limited_length	
max_rule_size	*integer*	Value must be greater than 2.
min_rule_support	*number*	0–100%
min_rule_confidence	*number*	0–100%
use_item_constraints	*flag*	
item_constraints_type	Include Exclude	
use_taxonomy	*flag*	
taxonomy_table_name	*string*	The name of the DB2 table to store taxonomy details.
taxonomy_child_column_name	*string*	The name of the child column in the taxonomy table. The child column contains the item names or category names.
taxonomy_parent_column_name	*string*	The name of the parent column in the taxonomy table. The parent column contains the category names.
load_taxonomy_to_table	*flag*	Controls if taxonomy information stored in IBM® SPSS® Modeler should be uploaded to the taxonomy table at model build time. Note that the taxonomy table is dropped if it already exists. Taxonomy information is stored with the model build node and can be edited using the Edit Categories and Edit Taxonomy buttons.

ISW Sequence

The following properties are available for nodes of type db2imsequencenode.

db2imsequencenode Properties	Values	Property Description
id_field	*field*	
group_field	*field*	
content_field	*field*	
max_rule_size	*integer*	Value must be greater than 2.
min_rule_support	*number*	0–100%
min_rule_confidence	*number*	0–100%
use_item_constraints	*flag*	
item_constraints_type	Include Exclude	
use_taxonomy	*flag*	

db2imsequencenode Properties	Values	Property Description
taxonomy_table_name	*string*	The name of the DB2 table to store taxonomy details.
taxonomy_child_column_name	*string*	The name of the child column in the taxonomy table. The child column contains the item names or category names.
taxonomy_parent_column_name	*string*	The name of the parent column in the taxonomy table. The parent column contains the category names.
load_taxonomy_to_table	*flag*	Controls if taxonomy information stored in SPSS Modeler should be uploaded to the taxonomy table at model build time. Note that the taxonomy table is dropped if it already exists. Taxonomy information is stored with the model build node and can be edited using the Edit Categories and Edit Taxonomy buttons.

ISW Regression

The following properties are available for nodes of type db2imregnode.

db2imregnode Properties	Values	Property Description
target	*field*	
regression_method	transform linear polynomial rbf	See next table for properties that apply only if regression_method is set to rbf.
perform_test_run	*field*	
limit_rsquared_value	*flag*	
max_rsquared_value	*number*	Value between 0.0 and 1.0.
use_execution_time_limit	*flag*	
execution_time_limit_mins	*integer*	Value greater than 0.
use_max_degree_polynomial	*flag*	
max_degree_polynomial	*integer*	
use_intercept	*flag*	
use_auto_feature_selec-tion_method	*flag*	
auto_feature_selection_method	normal adjusted	
use_min_significance_level	*flag*	
min_significance_level	*number*	
use_min_significance_level	*flag*	

The following properties apply only if regression_method is set to rbf.

db2imregnode Properties	Values	Property Description
use_output_sample_size	*flag*	If true, auto-set the value to the default.
output_sample_size	*integer*	Default is 2. Minimum is 1.

use_input_sample_size	*flag*	If true, auto-set the value to the default.
input_sample_size	*integer*	Default is 2. Minimum is 1.
use_max_num_centers	*flag*	If true, auto-set the value to the default.
max_num_centers	*integer*	Default is 20. Minimum is 1.
use_min_region_size	*flag*	If true, auto-set the value to the default.
min_region_size	*integer*	Default is 15. Minimum is 1.
use_max_data_passes	*flag*	If true, auto-set the value to the default.
max_data_passes	*integer*	Default is 5. Minimum is 2.
use_min_data_passes	*flag*	If true, auto-set the value to the default.
min_data_passes	*integer*	Default is 5. Minimum is 2.

ISW Clustering

The following properties are available for nodes of type db2imclusternode.

db2imclusternode Properties	Values	Property Description
cluster_method	demographic kohonen birch	
kohonen_num_rows	*integer*	
kohonen_num_columns	*integer*	
kohonen_passes	*integer*	
use_num_passes_limit	*flag*	
use_num_clusters_limit	*flag*	
max_num_clusters	*integer*	Value greater than 1.
birch_dist_measure	log_likelihood euclidean	Default is log_likelihood.
birch_num_cfleaves	*integer*	Default is 1000.
birch_num_refine_passes	*integer*	Default is 3; minimum is 1.
use_execution_time_limit	*flag*	
execution_time_limit_mins	*integer*	Value greater than 0.
min_data_percentage	*number*	0–100%
use_similarity_threshold	*flag*	
similarity_threshold	*number*	Value between 0.0 and 1.0.

ISW Naive Bayes

The following properties are available for nodes of type db2imnbsnode.

db2imnbnode Properties	Values	Property Description
perform_test_run	*flag*	
probability_threshold	*number*	Default is 0.001. Minimum value is 0; maximum value is 1.000

db2imnbnode Properties	Values	Property Description
use_costs	*flag*	
costs	*structured*	Structured property in the form: [{drugA drugB 1.5} {drugA drugC 2.1}], where the arguments in { } are actual predicted costs.

ISW Logistic Regression

The following properties are available for nodes of type db2imlognode.

db2imlognode Properties	Values	Property Description
perform_test_run	*flag*	
use_costs	*flag*	
costs	*structured*	Structured property in the form: [{drugA drugB 1.5} {drugA drugC 2.1}], where the arguments in { } are actual predicted costs.

ISW Time Series

Note: The input fields parameter is not used for this node. If the input fields parameter is found in the script a warning is displayed to say that the node has *time* and *targets* as incoming fields, but no input fields.

The following properties are available for nodes of type db2imtimeseriesnode.

db2imtimeseriesnode Properties	Values	Property Description
time	*field*	Integer, time, or date allowed.
targets	*list of fields*	
forecasting_algorithm	arima exponen-tial_smoothing sea-sonal_trend_de-composition	
forecasting_end_time	auto integer date time	
use_records_all	*boolean*	If false, use_records_start and use_records_end must be set.
use_records_start	*integer / time / date*	Depends on type of time field
use_records_end	*integer / time / date*	Depends on type of time field
interpolation_method	none linear exponen-tial_splines cubic_splines	

IBM DB2 Model Nugget Properties

The following properties are for the model nuggets created using the IBM DB2 ISW models.

ISW Decision Tree

There are no specific properties defined for nodes of type applydb2imtreenode.

ISW Association

This model nugget cannot be applied in scripting.

ISW Sequence

This model nugget cannot be applied in scripting.

ISW Regression

There are no specific properties defined for nodes of type applydb2imregnode.

ISW Clustering

There are no specific properties defined for nodes of type applydb2imclusternode.

ISW Naive Bayes

There are no specific properties defined for nodes of type applydb2imnbnode.

ISW Logistic Regression

There are no specific properties defined for nodes of type applydb2imlognode.

ISW Time Series

This model nugget cannot be applied in scripting.

Node Properties for IBM Netezza Analytics Modeling

Netezza Modeling Node Properties

The following properties are common to IBM Netezza database modeling nodes.

Common Netezza Node Properties	Values	Property Description
custom_fields	*flag*	If true, allows you to specify target, input, and other fields for the current node. If false, the current settings from an upstream Type node are used.
inputs	*[field1 ... fieldN]*	Input or predictor fields used by the model.

Common Netezza Node Properties	Values	Property Description
target	*field*	Target field (continuous or categorical).
record_id	*field*	Field to be used as unique record identifier.
use_upstream_connection	*flag*	If true (default), the connection details specified in an upstream node. Not used if move_data_to_connection is specified.
move_data_connection	*flag*	If true, moves the data to the database specified by connection. Not used if use_upstream_connection is specified.
connection	*structured*	The connection string for the Netezza database where the model is stored. Structured property in the form: ['odbc' '<dsn>' '<username>' '<psw>' '<catname>' '<conn_attribs>' {true|false}] where: <dsn> is the data source name <username> and <psw> are the username and password for the database <catname> is the catalog name <conn_attribs> are the connection attributes true | false indicates whether the password is needed.
table_name	*string*	Name of database table where model is to be stored.
use_model_name	*flag*	If true, uses the name specified by model_name as the name of the model, otherwise model name is created by the system.
model_name	*string*	Custom name for new model.
include_input_fields	*flag*	If true, passes all input fields downstream, otherwise passes only record_id and fields generated by model.

Netezza Decision Tree

The following properties are available for nodes of type netezzadectreenode.

netezzadectreenode Properties	Values	Property Description
impurity_measure	Entropy Gini	The measurement of impurity, used to evaluate the best place to split the tree.
max_tree_depth	*integer*	Maximum number of levels to which tree can grow. Default is 62 (the maximum possible).
min_improvement_splits	*number*	Minimum improvement in impurity for split to occur. Default is 0.01.
min_instances_split	*integer*	Minimum number of unsplit records remaining before split can occur. Default is 2 (the minimum possible).
weights	*structured*	Relative weightings for classes. Structured property in the form: set :netezza_dectree.weights = [{drugA 0.3}{drugB 0.6}] Default is weight of 1 for all classes.

netezzadectreenode Properties	Values	Property Description
pruning_measure	Acc wAcc	Default is Acc (accuracy). Alternative wAcc (weighted accuracy) takes class weights into account while applying pruning.
prune_tree_options	allTrainingData partitionTrain-ingData useOtherTable	Default is to use allTrainingData to estimate model accuracy. Use partitionTrainingData to specify a percentage of training data to use, or useOtherTable to use a training data set from a specified database table.
perc_training_data	*number*	If prune_tree_options is set to partitionTrainingData, specifies percentage of data to use for training.
prune_seed	*integer*	Random seed to be used for replicating analysis results when prune_tree_options is set to partitionTrainingData; default is 1.
pruning_table	*string*	Table name of a separate pruning dataset for estimating model accuracy.
compute_probabilities	*flag*	If true, produces a confidence level (probability) field as well as the prediction field.

Netezza K-Means

The following properties are available for nodes of type netezzakmeansnode.

netezzakmeansnode Properties	Values	Property Description
distance_measure	Euclidean Manhattan Canberra maximum	Method to be used for measuring distance between data points.
num_clusters	*integer*	Number of clusters to be created; default is 3.
max_iterations	*integer*	Number of algorithm iterations after which to stop model training; default is 5.
rand_seed	*integer*	Random seed to be used for replicating analysis results; default is 12345.

Netezza Bayes Net

The following properties are available for nodes of type netezzabayesnode.

netezzabayesnode Properties	Values	Property Description
base_index	*integer*	Numeric identifier assigned to first input field for internal management; default is 777.
sample_size	*integer*	Size of sample to take if number of attributes is very large; default is 10,000.
display_additional_information	*flag*	If true, displays additional progress information in a message dialog box.
type_of_prediction	best neighbors nn-neighbors	Type of prediction algorithm to use: best (most correlated neighbor), neighbors (weighted prediction of neighbors), or nn-neighbors (non null-neighbors).

Netezza Naive Bayes

The following properties are available for nodes of type netezzanaivebayesnode.

netezzanaivebayesnode Properties	Values	Property Description
compute_probabilities	*flag*	If true, produces a confidence level (probability) field as well as the prediction field.
use_m_estimation	*flag*	If true, uses m-estimation technique for avoiding zero probabilities during estimation.

Netezza KNN

The following properties are available for nodes of type netezzaknnnode.

netezzaknnnode Properties	Values	Property Description
weights	*structured*	Structured property used to assign weights to individual classes. Example: set :netezzaknnnode.weights = [{drugA 0.3}{drugB 0.6}]
distance_measure	Euclidean Manhattan Canberra Maximum	Method to be used for measuring the distance between data points.
num_nearest_neighbors	*integer*	Number of nearest neighbors for a particular case; default is 3.
standardize_measurements	*flag*	If true, standardizes measurements for continuous input fields before calculating distance values.
use_coresets	*flag*	If true, uses core set sampling to speed up calculation for large data sets.

Netezza Divisive Clustering

The following properties are available for nodes of type netezzadivclusternode.

netezzadivclusternode Properties	Values	Property Description
distance_measure	Euclidean Manhattan Canberra Maximum	Method to be used for measuring the distance between data points.
max_iterations	*integer*	Maximum number of algorithm iterations to perform before model training stops; default is 5.
max_tree_depth	*integer*	Maximum number of levels to which data set can be subdivided; default is 3.
rand_seed	*integer*	Random seed, used to replicate analyses; default is 12345.
min_instances_split	*integer*	Minimum number of records that can be split, default is 5.
level	*integer*	Hierarchy level to which records are to be scored; default is -1.

Netezza PCA

The following properties are available for nodes of type netezzapcanode.

netezzapcanode Properties	Values	Property Description
center_data	*flag*	If true (default), performs data centering (also known as "mean subtraction") before the analysis.
perform_data_scaling	*flag*	If true, performs data scaling before the analysis. Doing so can make the analysis less arbitrary when different variables are measured in different units.
force_eigensolve	*flag*	If true, uses less accurate but faster method of finding principal components.
pc_number	*integer*	Number of principal components to which data set is to be reduced; default is 1.

Netezza Regression Tree

The following properties are available for nodes of type netezzaregtreenode.

netezzaregtreenode Properties	Values	Property Description
max_tree_depth	*integer*	Maximum number of levels to which the tree can grow below the root node; default is 10.
split_evaluation_measure	Variance	Class impurity measure, used to evaluate the best place to split the tree; default (and currently only option) is Variance.
min_improvement_splits	*number*	Minimum amount to reduce impurity before new split is created in tree.
min_instances_split	*integer*	Minimum number of records that can be split.
pruning_measure	mse r2 pearson spearman	Method to be used for pruning.
prune_tree_options	allTrainingData partitionTrain-ingData useOtherTable	Default is to use allTrainingData to estimate model accuracy. Use partitionTrainingData to specify a percentage of training data to use, or useOtherTable to use a training data set from a specified database table.
perc_training_data	*number*	If prune_tree_options is set to PercTrainingData, specifies percentage of data to use for training.
prune_seed	*integer*	Random seed to be used for replicating analysis results when prune_tree_options is set to PercTrainingData; default is 1.
pruning_table	*string*	Table name of a separate pruning dataset for estimating model accuracy.
compute_probabilities	*flag*	If true, specifies that variances of assigned classes should be included in output.

Netezza Linear Regression

The following properties are available for nodes of type netezzalineregressionnode.

netezzalineregressionnode Properties	Values	Property Description
use_svd	*flag*	If true, uses Singular Value Decomposition matrix instead of original matrix, for increased speed and numerical accuracy.
include_intercept	*flag*	If true (default), increases overall accuracy of solution.
calculate_model_diagnostics	*flag*	If true, calculates diagnostics on the model.

Netezza Time Series

The following properties are available for nodes of type netezzatimeseriesnode.

netezzatimeseriesnode Properties	Values	Property Description
time_points	*field*	Input field containing the date or time values for the time series.
time_series_ids	*field*	Input field containing time series IDs; used if input contains more than one time series.
model_table	*field*	Name of database table where Netezza time series model will be stored.
description_table	*field*	Name of input table that contains time series names and descriptions.
seasonal_adjustment_table	*field*	Name of output table where seasonally adjusted values computed by exponential smoothing or seasonal trend decomposition algorithms will be stored.
algorithm_name	SpectralAnalysis or spectral ExponentialSmoothing or esmoothing ARIMA SeasonalTrendDecomposition or std	Algorithm to be used for time series modeling.
trend_name	N A DA M DM	Trend type for exponential smoothing: N - none A - additive DA - damped additive M - multiplicative DM - damped multiplicative
seasonality_type	N A M	Seasonality type for exponential smoothing: N - none A - additive M - multiplicative
interpolation_method	linear cubicspline exponentialspline	Interpolation method to be used.

netezzatimeseriesnode Properties	Values	Property Description
timerange_setting	SD SP	Setting for time range to use: SD - system-determined (uses full range of time series data) SP - user-specified via earliest_time and latest_time
earliest_time	*Date*	Start and end times, if timerange_setting is SP. Format: <yyyy>-<mm>-<dd> Example: set NZ_DT1.timerange_setting = 'SP' set NZ_DT1.earliest_time = '1921-01-01' set NZ_DT1.latest_time = '2121-01-01'
latest_time		
arima_setting	SD SP	Setting for the ARIMA algorithm (used only if algorithm_name is set to ARIMA): SD - system-determined SP - user-specified If arima_setting = SP, use the following parameters to set the seasonal and non-seasonal values. Example (non-seasonal only): set NZ_DT1.algorithm_name = 'arima' set NZ_DT1.arima_setting = 'SP' set NZ_DT1.p_symbol = 'lesseq' set NZ_DT1.p = '4' set NZ_DT1.d_symbol = 'lesseq' set NZ_DT1.d = '2' set NZ_DT1.q_symbol = 'lesseq' set NZ_DT1.q = '4'
p_symbol	less eq lesseq	ARIMA - operator for parameters p, d, q, sp, sd, and sq: less - less than eq - equals lesseq - less than or equal to
d_symbol		
q_symbol		
sp_symbol		
sd_symbol		
sq_symbol		
p	*integer*	ARIMA - non-seasonal degrees of autocorrelation.
q	*integer*	ARIMA - non-seasonal derivation value.
d	*integer*	ARIMA - non-seasonal number of moving average orders in the model.
sp	*integer*	ARIMA - seasonal degrees of autocorrelation.
sq	*integer*	ARIMA - seasonal derivation value.
sd	*integer*	ARIMA - seasonal number of moving average orders in the model.
advanced_setting	SD SP	Determines how advanced settings are to be handled: SD - system-determined SP - user-specified via period , units_period and forecast_setting. Example: set NZ_DT1.advanced_setting = 'SP' set NZ_DT1.period = 5 set NZ_DT1.units_period = 'd'
period	*integer*	Length of seasonal cycle, specified in conjunction with units_period. Not applicable for spectral analysis.

netezzatimeseriesnode Properties	Values	Property Description
units_period	ms s min h d wk q y	Units in which period is expressed: ms - milliseconds s - seconds min - minutes h - hours d - days wk - weeks q - quarters y - years For example, for a weekly time series use 1 for period and wk for units_period.
forecast_setting	forecasthorizon forecasttimes	Specifies how forecasts are to be made.
forecast_horizon	*string*	If forecast_setting = forecasthorizon, specifies end point for forecasting. Format: <yyyy>-<mm>-<dd>
forecast_times	[{'*date*'}, {'*date*'},..., {'*date*'}]	If forecast_setting = forecasttimes, specifies times to use for making forecasts. Format: <yyyy>-<mm>-<dd>
include_history	*flag*	Indicates if historical values are to be included in output.
include_interpolated_values	*flag*	Indicates if interpolated values are to be included in output. Not applicable if include_history is false.

Netezza Generalized Linear

The following properties are available for nodes of type netezzaglmnode.

netezzaglmnode Properties	Values	Property Description
dist_family	bernoulli gaussian poisson negativebino- mial wald gamma	Distribution type; default is bernoulli.
dist_params	*number*	Distribution parameter value to use. Only applicable if distribution is Negativebinomial.
trials	*integer*	Only applicable if distribution is Binomial. When target response is a number of events occurring in a set of trials, target field contains number of events, and trials field contains number of trials.
model_table	*field*	Name of database table where Netezza generalized linear model will be stored.
maxit	*integer*	Maximum number of iterations the algorithm should perform; default is 20.
eps	*number*	Maximum error value (in scientific notation) at which algorithm should stop finding best fit model. Default is -3, meaning 1E-3, or 0.001.

netezzaglmnode Properties	Values	Property Description
tol	*number*	Value (in scientific notation) below which errors are treated as having a value of zero. Default is -7, meaning that error values below 1E-7 (or 0.0000001) are counted as insignificant.
link_func	identity inverse invnegative invsquare sqrt power oddspower log clog loglog cloglog logit probit gaussit cauchit canbinom cangeom cannegbinom	Link function to use; default is logit.
link_params	*number*	Link function parameter value to use. Only applicable if link_function is power or oddspower.
interaction	[{[*col-names1*],[*lev-els1*]},{[*col-names2*],[*lev-els2*]},...,{[*col-namesN*],[*levelsN*]},]	Specifies interactions between fields. *colnames* is a list of input fields, and *level* is always 0 for each field. Example: [{["K","BP","Sex","K"],[0,0,0,0]},{["Age","Na"],[0,0]}]
intercept	*flag*	If true, includes the intercept in the model.

Netezza Model Nugget Properties

The following properties are common to Netezza database model nuggets.

Common Netezza Model Nugget Properties	Values	Property Description
connection	*string*	The connection string for the Netezza database where the model is stored.
table_name	*string*	Name of database table where model is stored.

Other model nugget properties are the same as those for the corresponding modeling node.

The script names of the model nuggets are as follows.

Model Nugget	Script Name
Decision Tree	applynetezzadectreenode
K-Means	applynetezzakmeansnode
Bayes Net	applynetezzabayesnode
Naive Bayes	applynetezzanaivebayesnode

Model Nugget	Script Name
KNN	applynetezzaknnnode
Divisive Clustering	applynetezzadivclusternode
PCA	applynetezzapcanode
Regression Tree	applynetezzaregtreenode
Linear Regression	applynetezzalineregressionnode
Time Series	applynetezzatimeseriesnode
Generalized Linear	applynetezzaglmnode

Chapter
19
Output Node Properties

Output node properties differ slightly from those of other node types. Rather than referring to a particular node option, output node properties store a reference to the output object. This is useful in taking a value from a table and then setting it as a stream parameter.

This section describes the scripting properties available for output nodes.

analysisnode Properties

The Analysis node evaluates predictive models' ability to generate accurate predictions. Analysis nodes perform various comparisons between predicted values and actual values for one or more model nuggets. They can also compare predictive models to each other.

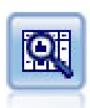
Example

```
create analysisnode
# "Analysis" tab
set :analysisnode.coincidence = True
set :analysisnode.performance = True
set :analysisnode.confidence = True
set :analysisnode.threshold = 75
set :analysisnode.improve_accuracy = 3
set :analysisnode.inc_user_measure = True
# "Define User Measure..."
set :analysisnode.user_if = "@TARGET = @PREDICTED"
set :analysisnode.user_then = "101"
set :analysisnode.user_else = "1"
set :analysisnode.user_compute = [Mean Sum]
set :analysisnode.by_fields = ['Drug']
# "Output" tab
set :analysisnode.output_format = HTML
set :analysisnode.full_filename = "C:/output/analysis_out.html"
```

analysisnode properties	Data type	Property description
output_mode	Screen File	Used to specify target location for output generated from the output node.
use_output_name	*flag*	Specifies whether a custom output name is used.
output_name	*string*	If use_output_name is true, specifies the name to use.
output_format	Text (*.txt*) HTML (*.html*) Output (*.cou*)	Used to specify the type of output.
by_fields	[*field field field*]	

analysisnode properties	Data type	Property description
full_filename	*string*	If disk, data, or HTML output, the name of the output file.
coincidence	*flag*	
performance	*flag*	
confidence	*flag*	
threshold	*number*	
improve_accuracy	*number*	
inc_user_measure	*flag*	
user_if	*expr*	
user_then	*expr*	
user_else	*expr*	
user_compute	[Mean Sum Min Max SDev]	

dataauditnode Properties



The Data Audit node provides a comprehensive first look at the data, including summary statistics, histograms and distribution for each field, as well as information on outliers, missing values, and extremes. Results are displayed in an easy-to-read matrix that can be sorted and used to generate full-size graphs and data preparation nodes.

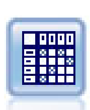
Example

```
create dataauditnode
connect :variablefilenode to :dataauditnode
set :dataauditnode.custom_fields = True
set :dataauditnode.fields = [Age Na K]
set :dataauditnode.display_graphs = True
set :dataauditnode.basic_stats = True
set :dataauditnode.advanced_stats = True
set :dataauditnode.median_stats = False
set :dataauditnode.calculate = [Count Breakdown]
set :dataauditnode.outlier_detection_method = std
set :dataauditnode.outlier_detection_std_outlier = 1.0
set :dataauditnode.outlier_detection_std_extreme = 3.0
set :dataauditnode.output_mode = Screen
```

dataauditnode properties	Data type	Property description
custom_fields	*flag*	
fields	*[field1 … fieldN]*	
overlay	*field*	
display_graphs	*flag*	Used to turn the display of graphs in the output matrix on or off.
basic_stats	*flag*	
advanced_stats	*flag*	
median_stats	*flag*	

dataauditnode properties	**Data type**	**Property description**
calculate	Count Breakdown	Used to calculate missing values. Select either, both, or neither calculation method.
outlier_detection_method	std iqr	Used to specify the detection method for outliers and extreme values.
outlier_detection_std_outlier	*number*	If outlier_detection_method is std, specifies the number to use to define outliers.
outlier_detection_std_extreme	*number*	If outlier_detection_method is std, specifies the number to use to define extreme values.
outlier_detection_iqr_outlier	*number*	If outlier_detection_method is iqr, specifies the number to use to define outliers.
outlier_detection_iqr_extreme	*number*	If outlier_detection_method is iqr, specifies the number to use to define extreme values.
use_output_name	*flag*	Specifies whether a custom output name is used.
output_name	*string*	If use_output_name is true, specifies the name to use.
output_mode	Screen File	Used to specify target location for output generated from the output node.
output_format	Formatted (*.tab*) Delimited (*.csv*) HTML (*.html*) Output (*.cou*)	Used to specify the type of output.
paginate_output	*flag*	When the output_format is HTML, causes the output to be separated into pages.
lines_per_page	*number*	When used with paginate_output, specifies the lines per page of output.
full_filename	*string*	

matrixnode Properties



The Matrix node creates a table that shows relationships between fields. It is most commonly used to show the relationship between two symbolic fields, but it can also show relationships between flag fields or numeric fields.

Example

```
create matrixnode
# "Settings" tab
set :matrixnode.fields = Numerics
set :matrixnode.row = 'K'
set :matrixnode.column = 'Na'
```

```
set :matrixnode.cell_contents = Function
set :matrixnode.function_field = 'Age'
set :matrixnode.function = Sum
# "Appearance" tab
set :matrixnode.sort_mode = Ascending
set :matrixnode.highlight_top = 1
set :matrixnode.highlight_bottom = 5
set :matrixnode.display = [Counts Expected Residuals]
set :matrixnode.include_totals = True
# "Output" tab
set :matrixnode.full_filename = "C:/output/matrix_output.html"
set :matrixnode.output_format = HTML
set :matrixnode.paginate_output = true
set :matrixnode.lines_per_page = 50
```

matrixnode properties	Data type	Property description
fields	Selected Flags Numerics	
row	*field*	
column	*field*	
include_missing_values	*flag*	Specifies whether user-missing (blank) and system missing (null) values are included in the row and column output.
cell_contents	CrossTabs Function	
function_field	*string*	
function	Sum Mean Min Max SDev	
sort_mode	Unsorted Ascending Descending	
highlight_top	*number*	If non-zero, then true.
highlight_bottom	*number*	If non-zero, then true.
display	[Counts Expected Residuals RowPct ColumnPct TotalPct]	
include_totals	*flag*	
use_output_name	*flag*	Specifies whether a custom output name is used.
output_name	*string*	If use_output_name is true, specifies the name to use.
output_mode	Screen File	Used to specify target location for output generated from the output node.

matrixnode properties	**Data type**	**Property description**
output_format	Formatted (*.tab*) Delimited (*.csv*) HTML (*.html*) Output (*.cou*)	Used to specify the type of output. Both the Formatted and Delimited formats can take the modifier transposed, which transposes the rows and columns in the table; for example: NODE.output_format=trans-posed Delimited
paginate_output	*flag*	When the output_format is HTML, causes the output to be separated into pages.
lines_per_page	*number*	When used with paginate_output, specifies the lines per page of output.
full_filename	*string*	

meansnode Properties

The Means node compares the means between independent groups or between pairs of related fields to test whether a significant difference exists. For example, you could compare mean revenues before and after running a promotion or compare revenues from customers who did not receive the promotion with those who did.

Example

```
create meansnode
set :meansnode.means_mode = BetweenFields
set :meansnode.paired_fields = [{'OPEN_BAL' 'CURR_BAL'}]
set :meansnode.label_correlations = true
set :meansnode.output_view = Advanced
set :meansnode.output_mode = File
set :meansnode.output_format = HTML
set :meansnode.full_filename = "C:/output/means_output.html"
```

meansnode properties	**Data type**	**Property description**
means_mode	BetweenGroups BetweenFields	Specifies the type of means statistic to be executed on the data.
test_fields	[field1 ... fieldn]	Specifies the test field when means_mode is set to BetweenGroups.
grouping_field	*field*	Specifies the grouping field.
paired_fields	[{field1 field2} {field3 field4} ...]	Specifies the field pairs to use when means_mode is set to BetweenFields.
label_correlations	*flag*	Specifies whether correlation labels are shown in output. This setting applies only when means_mode is set to BetweenFields.

meansnode properties	Data type	Property description
correlation_mode	Probability Absolute	Specifies whether to label correlations by probability or absolute value.
weak_label	*string*	
medium_label	*string*	
strong_label	*string*	
weak_below_probability	*number*	When correlation_mode is set to Probability, specifies the cutoff value for weak correlations. This must be a value between 0 and 1—for example, 0.90.
strong_above_probability	*number*	Cutoff value for strong correlations.
weak_below_absolute	*number*	When correlation_mode is set to Absolute, specifies the cutoff value for weak correlations. This must be a value between 0 and 1—for example, 0.90.
strong_above_absolute	*number*	Cutoff value for strong correlations.
unimportant_label	*string*	
marginal_label	*string*	
important_label	*string*	
unimportant_below	*number*	Cutoff value for low field importance. This must be a value between 0 and 1—for example, 0.90.
important_above	*number*	
use_output_name	*flag*	Specifies whether a custom output name is used.
output_name	*string*	Name to use.
output_mode	Screen File	Specifies the target location for output generated from the output node.
output_format	Formatted (*.tab*) Delimited (*.csv*) HTML (*.html*) Output (*.cou*)	Specifies the type of output.
full_filename	*string*	
output_view	Simple Advanced	Specifies whether the simple or advanced view is displayed in the output.

reportnode Properties

The Report node creates formatted reports containing fixed text as well as data and other expressions derived from the data. You specify the format of the report using text templates to define the fixed text and data output constructions. You can provide custom text formatting by using HTML tags in the template and by setting options on the Output tab. You can include data values and other conditional output by using CLEM expressions in the template.

Example

```
create reportnode
set :reportnode.output_format = HTML
set :reportnode.full_filename = "C:/report_output.html"
set :reportnode.lines_per_page = 50
set :reportnode.title = "Report node created by a script"
set :reportnode.highlights = False
```

reportnode properties	Data type	Property description
output_mode	Screen File	Used to specify target location for output generated from the output node.
output_format	HTML (*.html*) Text (*.txt*) Output (*.cou*)	Used to specify the type of output.
use_output_name	*flag*	Specifies whether a custom output name is used.
output_name	*string*	If use_output_name is true, specifies the name to use.
text	*string*	
full_filename	*string*	
highlights	*flag*	
title	*string*	
lines_per_page	*number*	

setglobalsnode Properties

The Set Globals node scans the data and computes summary values that can be used in CLEM expressions. For example, you can use this node to compute statistics for a field called *age* and then use the overall mean of *age* in CLEM expressions by inserting the function @GLOBAL_MEAN(age).

Example

```
create setglobalsnode
connect :typenode to :setglobalsnode
set :setglobalsnode.globals.Na = [Max Sum Mean]
set :setglobalsnode.globals.K = [Max Sum Mean]
set :setglobalsnode.globals.Age = [Max Sum Mean SDev]
set :setglobalsnode.clear_first = False
```

```
set :setglobalsnode.show_preview = True
```

setglobalsnode properties	Data type	Property description
globals	[Sum Mean Min Max SDev]	Structured property where fields to be set must be referenced with the following syntax: set :setglobalsnode.globals.Age = [Sum Mean Min Max SDev]
clear_first	*flag*	
show_preview	*flag*	

statisticsnode Properties

The Statistics node provides basic summary information about numeric fields. It calculates summary statistics for individual fields and correlations between fields.

Example

```
create statisticsnode
# "Settings" tab
set :statisticsnode.examine = ['Age' 'BP' 'Drug']
set :statisticsnode.statistics = [Mean Sum SDev]
set :statisticsnode.correlate = ['BP' 'Drug']
# "Correlation Labels..." section
set :statisticsnode.label_correlations = True
set :statisticsnode.weak_below_absolute = 0.25
set :statisticsnode.weak_label = "lower quartile"
set :statisticsnode.strong_above_absolute = 0.75
set :statisticsnode.medium_label = "middle quartiles"
set :statisticsnode.strong_label = "upper quartile"
# "Output" tab
set :statisticsnode.full_filename = "c:/output/statistics_output.html"
set :statisticsnode.output_format = HTML
```

statisticsnode properties	Data type	Property description
use_output_name	*flag*	Specifies whether a custom output name is used.
output_name	*string*	If use_output_name is true, specifies the name to use.
output_mode	Screen File	Used to specify target location for output generated from the output node.
output_format	Text (*.txt*) HTML (*.html*) Output (*.cou*)	Used to specify the type of output.
full_filename	*string*	
examine	[*field field field*]	
correlate	[*field field field*]	

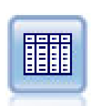
statisticsnode properties	**Data type**	**Property description**
statistics	[Count Mean Sum Min Max Range Variance SDev SErr Median Mode]	
correlation_mode	Probability Absolute	Specifies whether to label correlations by probability or absolute value.
label_correlations	*flag*	
weak_label	*string*	
medium_label	*string*	
strong_label	*string*	
weak_below_probability	*number*	When correlation_mode is set to Probability, specifies the cutoff value for weak correlations. This must be a value between 0 and 1—for example, 0.90.
strong_above_probability	*number*	Cutoff value for strong correlations.
weak_below_absolute	*number*	When correlation_mode is set to Absolute, specifies the cutoff value for weak correlations. This must be a value between 0 and 1—for example, 0.90.
strong_above_absolute	*number*	Cutoff value for strong correlations.

statisticsoutputnode Properties

The Statistics Output node allows you to call an IBM® SPSS® Statistics procedure to analyze your IBM® SPSS® Modeler data. A wide variety of SPSS Statistics analytical procedures is available. This node requires a licensed copy of SPSS Statistics.

The properties for this node are described under statisticsoutputnode Properties on p. 293.

tablenode Properties



The Table node displays the data in table format, which can also be written to a file. This is useful anytime that you need to inspect your data values or export them in an easily readable form.

Example

```
create tablenode
set :tablenode.highlight_expr = "Age > 30"
set :tablenode.output_format = HTML
set :tablenode.transpose_data = true
```

```
set :tablenode.full_filename = "C:/output/table_output.htm"
set :tablenode.paginate_output = true
set :tablenode.lines_per_page = 50
```

tablenode properties	Data type	Property description
full_filename	*string*	If disk, data, or HTML output, the name of the output file.
use_output_name	*flag*	Specifies whether a custom output name is used.
output_name	*string*	If use_output_name is true, specifies the name to use.
output_mode	Screen File	Used to specify target location for output generated from the output node.
output_format	Formatted (*.tab*) Delimited (*.csv*) HTML (*.html*) Output (*.cou*)	Used to specify the type of output.
transpose_data	*flag*	Transposes the data before export so that rows represent fields and columns represent records.
paginate_output	*flag*	When the output_format is HTML, causes the output to be separated into pages.
lines_per_page	*number*	When used with paginate_output, specifies the lines per page of output.
highlight_expr	*string*	
output	*string*	A read-only property that holds a reference to the last table built by the node.
value_labels	*[{Value LabelString} {Value LabelString} ...]*	Used to specify labels for value pairs. For example, set :typenode.value_labels. 'Drug'=[{drugA label1} {drugB label2}]
display_places	*integer*	Sets the number of decimal places for the field when displayed (applies only to fields with REAL storage). A value of –1 will use the stream default. Usage format: NODE.display_places. FIELDNAME
export_places	*integer*	Sets the number of decimal places for the field when exported (applies only to fields with REAL storage). A value of –1 will use the stream default. Usage format: NODE.export_places.FIELDNAME
decimal_separator	DEFAULT PERIOD COMMA	Sets the decimal separator for the field (applies only to fields with REAL storage). Usage format: NODE.decimal_separator. FIELDNAME

tablenode properties	Data type	Property description
date_format	"DDMMYY" "MMDDYY" "YYMMDD" "YYYYMMDD" "YYYYDDD" DAY MONTH "DD-MM-YY" "DD-MM-YYYY" "MM-DD-YY" "MM-DD-YYYY" "DD-MON-YY" "DD-MON-YYYY" "YYYY-MM-DD" "DD.MM.YY" "DD.MM.YYYY" "MM.DD.YY" "MM.DD.YYYY" "DD.MON.YY" "DD.MON.YYYY" "DD/MM/YY" "DD/MM/YYYY" "MM/DD/YY" "MM/DD/YYYY" "DD/MON/YY" "DD/MON/YYYY" MON YYYY q Q YYYY ww WK YYYY	Sets the date format for the field (applies only to fields with DATE or TIMESTAMP storage). Usage format: NODE.date_format.FIELDNAME For example, set :tablenode.date_format. 'LaunchDate' = "DDMMYY"
time_format	"HHMMSS" "HHMM" "MMSS" "HH:MM:SS" "HH:MM" "MM:SS" "(H)H:(M)M:(S)S" "(H)H:(M)M" "(M)M:(S)S" "HH.MM.SS" "HH.MM" "MM.SS" "(H)H.(M)M.(S)S" "(H)H.(M)M" "(M)M.(S)S"	Sets the time format for the field (applies only to fields with TIME or TIMESTAMP storage). Usage format: NODE.time_format.FIELDNAME For example, set :tablenode.time_format. set 'BOF_enter' = "HHMMSS"
column_width	*integer*	Sets the column width for the field. A value of –1 will set column width to Auto. Usage format: NODE.column_width.FIELDNAME
justify	AUTO CENTER LEFT RIGHT	Sets the column justification for the field. Usage format: NODE.justify.FIELDNAME

transformnode Properties

The Transform node allows you to select and visually preview the results of transformations before applying them to selected fields.

Example

```
create transformnode
set :transformnode.fields = [AGE INCOME]
set :transformnode.formula = Select
set :transformnode.formula_log_n = true
set :transformnode.formula_log_n_offset = 1
```

transformnode properties	Data type	Property description
fields	[*field1... fieldn*]	The fields to be used in the transformation.
formula	All Select	Indicates whether all or selected transformations should be calculated.
formula_inverse	*flag*	Indicates if the inverse transformation should be used.
formula_inverse_offset	*number*	Indicates a data offset to be used for the formula. Set as 0 by default, unless specified by user.
formula_log_n	*flag*	Indicates if the $\log_n$ transformation should be used.
formula_log_n_offset	*number*	
formula_log_10	*flag*	Indicates if the $\log_{10}$ transformation should be used.
formula_log_10_offset	*number*	
formula_exponential	*flag*	Indicates if the exponential transformation (e^x) should be used.
formula_square_root	*flag*	Indicates if the square root transformation should be used.
use_output_name	*flag*	Specifies whether a custom output name is used.
output_name	*string*	If use_output_name is true, specifies the name to use.
output_mode	Screen File	Used to specify target location for output generated from the output node.
output_format	HTML (*.html*) Output (*.cou*)	Used to specify the type of output.

transformnode properties	Data type	Property description
paginate_output	*flag*	When the output_format is HTML, causes the output to be separated into pages.
lines_per_page	*number*	When used with paginate_output, specifies the lines per page of output.
full_filename	*string*	Indicates the file name to be used for the file output.

Chapter

20

Export Node Properties

Common Export Node Properties

The following properties are common to all export nodes.

Property	Values	Property description
publish_path	*string*	Enter the rootname name to be used for the published image and parameter files.
publish_metadata	*flag*	Specifies if a metadata file is produced that describes the inputs and outputs of the image and their data models.
publish_use_parameters	*flag*	Specifies if stream parameters are included in the *.par file.
publish_parameters	*string list*	Specify the parameters to be included.
execute_mode	export_data publish	Specifies whether the node executes without publishing the stream, or if the stream is automatically published when the node is executed.

cognosexportnode Properties

The IBM Cognos BI Export node exports data in a format that can be read by Cognos BI databases.

Note: For this node, you must define a Cognos connection and an ODBC connection.

Cognos connection

The properties for the Cognos connection are as follows.

cognosexportnode properties	Data type	Property description
cognos_connection	*{"field","field", ... ,"field"}*	A list property containing the connection details for the Cognos server. The format is: {"*Cognos_server_URL*", *login_mode*, "*namespace*", "*username*", "*password*"} where: *Cognos_server_URL* is the URL of the Cognos server to which you are exporting *login_mode* indicates whether anonymous login is used, and is either true or false; if set to true, the following fields should be set to ""

cognosexportnode properties	Data type	Property description
		namespace specifies the security authentication provider used to log on to the server *username* and *password* are those used to log on to the Cognos server
cognos_package_name	*string*	The path and name of the Cognos package to which you are exporting data, for example: /Public Folders/MyPackage
cognos_datasource	*string*	
cognos_export_mode	Publish ExportFile	
cognos_filename	*string*	

ODBC connection

The properties for the ODBC connection are identical to those listed for databaseexportnode in the next section, with the exception that the datasource property is not valid.

databaseexportnode Properties

The Database export node writes data to an ODBC-compliant relational data source. In order to write to an ODBC data source, the data source must exist and you must have write permission for it.

Example

```
/*
Use this sample with fraud.str from demo folder
Assumes a datasource named "MyDatasource" has been configured
*/
create databaseexport
connect claimvalue:applyneuralnetwork to :databaseexport
 # Export tab
set :databaseexport.username = "user"
set :databaseexport.datasource = "MyDatasource"
set :databaseexport.password = "password"
set :databaseexport.table_name = "predictions"
set :databaseexport.write_mode = Create
set :databaseexport.generate_import = true
set :databaseexport.drop_existing_table = true
set :databaseexport.delete_existing_rows = true
set :databaseexport.default_string_size = 32

# Schema dialog
set :databaseexport.type.region = "VARCHAR(10)"
set :databaseexport.export_db_primarykey.id = true
set :databaseexportnode.use_custom_create_table_command = true
set :databaseexportnode.custom_create_table_command = "My SQL Code"
```

```
# Indexes dialog
set :databaseexport.use_custom_create_index_command = true
set :databaseexport.custom_create_index_command = \
  "CREATE BITMAP INDEX <index-name> ON <table-name> <(index-columns)>"
set :databaseexport.indexes.MYINDEX.fields = [id region]
```

databaseexportnode properties	Data type	Property description
datasource	*string*	
username	*string*	
password	*string*	
epassword	*string*	This slot is read-only during execution. To generate an encoded password, use the Password Tool available from the Tools menu. For more information, see the topic Generating an Encoded Password in Chapter 5 on p. 59.
table_name	*string*	
write_mode	Create Append Merge	
map	*string*	Maps a stream field name to a database column name (valid only if write_mode is Merge). Example: set :databaseexportnode.map.streamBP = 'databaseBP' Multiple mapping is supported, according to the field position, for example: set :databaseexportnode.map=[{streamfield1 field1}{streamfield2 field2}{streamfield3 field3}] For a merge, all fields must be mapped in order to be exported. Field names that do not exist in the database are added as new columns.
key_fields	*[field field ... field]*	Specifies the stream field that is used for key; map property shows what this corresponds to in the database.
join	Database Add	Example: set : databaseexportnode.join = Database
drop_existing_table	*flag*	
delete_existing_rows	*flag*	
default_string_size	*integer*	

databaseexportnode properties	Data type	Property description
type		Structured property used to set the schema type. Usage format: set :databaseexportnode. type.BP = 'VARCHAR(10)'
generate_import	*flag*	
use_custom_create_table_command	*flag*	Use the *custom_create_table* slot to modify the standard CREATE TABLE SQL command.
custom_create_table_command	*string*	Specifies a string command to use in place of the standard CREATE TABLE SQL command.
use_batch	*flag*	The following properties are advanced options for database bulk-loading. A true value for Use_batch turns off row-by-row commits to the database.
batch_size	*number*	Specifies the number of records to send to the database before committing to memory.
bulk_loading	Off ODBC External	Specifies the type of bulk-loading. Additional options for ODBC and External are listed below.
odbc_binding	Row Column	Specify row-wise or column-wise binding for bulk-loading via ODBC.
loader_delimit_mode	Tab Space Other	For bulk-loading via an external program, specify type of delimiter. Select Other in conjunction with the loader_other_delimiter property to specify delimiters, such as the comma (,).
loader_other_delimiter	*string*	
specify_data_file	*flag*	A true flag activates the data_file property below, where you can specify the filename and path to write to when bulk-loading to the database.
data_file	*string*	
specify_loader_program	*flag*	A true flag activates the loader_program property below, where you can specify the name and location of an external loader script or program.
loader_program	*string*	
gen_logfile	*flag*	A true flag activates the logfile_name below, where you can specify the name of a file on the server to generate an error log.

databaseexportnode properties	Data type	Property description
logfile_name	*string*	
check_table_size	*flag*	A true flag allows table checking to ensure that the increase in database table size corresponds to the number of rows exported from IBM® SPSS® Modeler.
loader_options	*string*	Specify additional arguments, such as -comment and -specialdir, to the loader program.
export_db_primarykey	*flag*	Specifies whether a given field is a primary key.
use_custom_create_index_command	*flag*	If true, enables custom SQL for all indexes.
custom_create_index_command	*string*	Specifies the SQL command used to create indexes when custom SQL is enabled. (This value can be overridden for specific indexes as indicated below.)
indexes.INDEXNAME.fields		Creates the specified index if necessary and lists field names to be included in that index.
indexes.INDEXNAME.use_custom_create_index_command	*flag*	Used to enable or disable custom SQL for a specific index.
indexes.INDEXNAME.custom_create_command		Specifies the custom SQL used for the specified index.
indexes.INDEXNAME.remove	*flag*	If true, removes the specified index from the set of indexes.
table_space	*string*	Specifies the table space that will be created.
use_partition	*flag*	Specifies that the distribute hash field will be used.
partition_field	*string*	Specifies the contents of the distribute hash field.

Note: For some databases, you can specify that database tables are created for export with compression (for example, the equivalent of CREATE TABLE MYTABLE (...) COMPRESS YES; in SQL). The properties use_compression and compression_mode are provided to support this feature, as follows.

databaseexportnode properties	Data type	Property description
use_compression	*Boolean*	If set to true, creates tables for export with compression.

databaseexportnode properties	Data type	Property description
compression_mode	Row Page	Sets the level of compression for SQL Server databases.
	Default Direct_Load_Opera-tions All_Operations Basic OLTP Query_High Query_Low Archive_High Archive_Low	Sets the level of compression for Oracle databases. Note that the values OLTP, Query_High, Query_Low, Archive_High, and Archive_Low require a minimum of Oracle 11gR2.

Example - SQL Server

```
var DBSource
set DBSource = get node TestCompressionSQL
set ^DBSource.use_compression = true
set ^DBSource.compression_mode = Page

execute DBSource
```

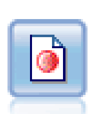
Example - Oracle 11gR1

```
var DBSource
set DBSource = get node TestCompressionOracle11gR1
set ^DBSource.use_compression = true
set ^DBSource.compression_mode = Direct_Load_Operations

execute DBSource
```

Example - Oracle 11gR2

```
var DBSource
set DBSource = get node TestCompressionOracle11gR2
set ^DBSource.use_compression = true
set ^DBSource.compression_mode = Basic

execute DBSource
```

datacollectionexportnode Properties



The IBM® SPSS® Data Collection export node outputs data in the format used by Data Collection market research software. The Data Collection Data Library must be installed to use this node.

Example

```
create datacollectionexportnode
set :datacollectionexportnode.metadata_file = "c:\museums.mdd"
```

```
set :datacollectionexportnode.merge_metadata = Overwrite
set :datacollectionexportnode.casedata_file = "c:\museumdata.sav"
set :datacollectionexportnode.generate_import = true
set :datacollectionexportnode.enable_system_variables = true
```

datacollectionexportnode properties	Data type	Property description
metadata_file	*string*	The name of the metadata file to export.
merge_metadata	Overwrite MergeCurrent	
enable_system_variables	*flag*	Specifies whether the exported *.mdd* file should include Data Collection system variables.
casedata_file	*string*	The name of the *.sav* file to which case data is exported.
generate_import	*flag*	

excelexportnode Properties

The Excel export node outputs data in Microsoft Excel format (*.xls*). Optionally, you can choose to launch Excel automatically and open the exported file when the node is executed.

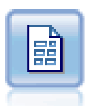
Example

```
create excelexportnode
set :excelexportnode.full_filename = "C:/output/myexport.xls"
set :excelexportnode.excel_file_type = Excel2007
set :excelexportnode.inc_field_names = True
set :excelexportnode.inc_labels_as_cell_notes = False
set :excelexportnode.launch_application = True
set :excelexportnode.generate_import = True
```

excelexportnode properties	Data type	Property description
full_filename	*string*	
excel_file_type	Excel2003 Excel2007	
export_mode	Create Append	
inc_field_names	*flag*	Specifies whether field names should be included in the first row of the worksheet.
start_cell	*string*	Specifies starting cell for export.
worksheet_name	*string*	Name of the worksheet to be written.

excelexportnode properties	Data type	Property description
launch_application	*flag*	Specifies whether Excel should be invoked on the resulting file. Note that the path for launching Excel must be specified in the Helper Applications dialog box (Tools menu, Helper Applications).
generate_import	*flag*	Specifies whether an Excel Import node should be generated that will read the exported data file.

outputfilenode Properties



The Flat File export node outputs data to a delimited text file. It is useful for exporting data that can be read by other analysis or spreadsheet software.

Example

```
create outputfile
set :outputfile.full_filename = "c:/output/flatfile_output.txt"
set :outputfile.write_mode = Append
set :outputfile.inc_field_names = False
set :outputfile.use_newline_after_records = False
set :outputfile.delimit_mode = Tab
set :outputfile.other_delimiter = ","
set :outputfile.quote_mode = Double
set :outputfile.other_quote = "*"
set :outputfile.decimal_symbol = Period
set :outputfile.generate_import = True
```

outputfilenode properties	Data type	Property description
full_filename	*string*	Name of output file.
write_mode	Overwrite Append	
inc_field_names	*flag*	
use_newline_after_records	*flag*	
delimit_mode	Comma Tab Space Other	
other_delimiter	*char*	
quote_mode	None Single Double Other	
other_quote	*flag*	

outputfilenode properties	Data type	Property description
generate_import	*flag*	
encoding	StreamDefault SystemDefault "UTF-8"	

sasexportnode Properties

The SAS export node outputs data in SAS format, to be read into SAS or a SAS-compatible software package. Three SAS file formats are available: SAS for Windows/OS2, SAS for UNIX, or SAS Version 7/8.

Example

```
create sasexportnode
set :sasexportnode.full_filename = "c:/output/SAS_output.sas7bdat"
set :sasexportnode.format = SAS8
set :sasexportnode.export_names = NamesAndLabels
set :sasexportnode.generate_import = True
```

sasexportnode properties	Data type	Property description
format	Windows UNIX SAS7 SAS8	Variant property label fields.
full_filename	*string*	
export_names	NamesAndLabels NamesAsLabels	Used to map field names from IBM® SPSS® Modeler upon export to IBM® SPSS® Statistics or SAS variable names.
generate_import	*flag*	

statisticsexportnode Properties

The Statistics Export node outputs data in IBM® SPSS® Statistics*.sav* format. The *.sav* files can be read by SPSS Statistics Base and other products. This is also the format used for cache files in IBM® SPSS® Modeler.

The properties for this node are described under statisticsexportnode Properties on p. 294.

xmlexportnode Properties

The XML export node outputs data to a file in XML format. You can optionally create an XML source node to read the exported data back into the stream.

Example

```
create xmlexportnode
set :xmlexportnode.full_filename = "c:\export\data.xml"
set :xmlexportnode.map = [{"/catalog/book/genre" genre}{"/catalog/book/title" title}]
```

xmlexportnode properties	Data type	Property description
full_filename	*string*	(required) Full path and file name of XML export file.
use_xml_schema	*flag*	Specifies whether to use an XML schema (XSD or DTD file) to control the structure of the exported data.
full_schema_filename	*string*	Full path and file name of XSD or DTD file to use. Required if use_xml_schema is set to true.
generate_import	*flag*	Generates an XML source node that will read the exported data file back into the stream.
records	*string*	XPath expression denoting the record boundary.
map	*string*	Maps field name to XML structure. Example: set :xmlexportnode.map = [{"/top/node1" field1}{"/top/node2" field2}] This maps the stream field field1 to the XML element /top/node1, and so on.

Chapter

21

IBM SPSS Statistics Node Properties

statisticsimportnode Properties

The Statistics File node reads data from the *.sav* file format used by IBM® SPSS® Statistics, as well as cache files saved in IBM® SPSS® Modeler, which also use the same format.

Example

```
create statisticsimportnode
set :statisticsimportnode.full_filename = "C:/data/drug1n.sav"
set :statisticsimportnode.import_names = true
set :statisticsimportnode.import_data = true
```

statisticsimportnode properties	Data type	Property description
full_filename	*string*	The complete filename, including path.
import_names	NamesAndLabels LabelsAsNames	Method for handling variable names and labels.
import_data	DataAndLabels LabelsAsData	Method for handling values and labels.
use_field_format_for_storage	*Boolean*	Specifies whether to use SPSS Statistics field format information when importing.

statisticstransformnode Properties

The Statistics Transform node runs a selection of IBM® SPSS® Statistics syntax commands against data sources in IBM® SPSS® Modeler. This node requires a licensed copy of SPSS Statistics.

Example

```
create statisticstransformnode
set :statisticstransformnode.syntax = "COMPUTE NewVar = Na + K."
set :statisticstransformnode.new_name.NewVar = "Mixed Drugs"
set :statisticstransformnode.check_before_saving = true
```

statisticstransformnode properties	Data type	Property description
syntax	*string*	
check_before_saving	*flag*	Validates the entered syntax before saving the entries. Displays an error message if the syntax is invalid.

statisticstransformnode properties	Data type	Property description
default_include	*flag*	For more information, see the topic filternode Properties in Chapter 14 on p. 150.
include	*flag*	For more information, see the topic filternode Properties in Chapter 14 on p. 150.
new_name	*string*	For more information, see the topic filternode Properties in Chapter 14 on p. 150.

statisticsmodelnode Properties

The Statistics Model node enables you to analyze and work with your data by running IBM® SPSS® Statistics procedures that produce PMML. This node requires a licensed copy of SPSS Statistics.

Example

```
create statisticsmodelnode
set :statisticsmodelnode.syntax = "COMPUTE NewVar = Na + K."
set :statisticsmodelnode.new_name.NewVar = "Mixed Drugs"
```

statisticsmodelnode properties	Data type	Property description
syntax	*string*	
default_include	*flag*	For more information, see the topic filternode Properties in Chapter 14 on p. 150.
include	*flag*	For more information, see the topic filternode Properties in Chapter 14 on p. 150.
new_name	*string*	For more information, see the topic filternode Properties in Chapter 14 on p. 150.

statisticsoutputnode Properties

The Statistics Output node allows you to call an IBM® SPSS® Statistics procedure to analyze your IBM® SPSS® Modeler data. A wide variety of SPSS Statistics analytical procedures is available. This node requires a licensed copy of SPSS Statistics.

Example

```
create statisticsoutputnode
set :statisticsoutputnode.syntax = "SORT CASES BY Age(A) Sex(A) BP(A) Cholesterol(A)"
set :statisticsoutputnode.use_output_name = False
```

```
set :statisticsoutputnode.output_mode = File
set :statisticsoutputnode.full_filename = "Cases by Age, Sex and Medical History"
set :statisticsoutputnode.file_type = HTML
```

statisticsoutputnode properties	**Data type**	**Property description**
mode	Dialog Syntax	Selects “SPSS Statistics dialog” option or Syntax Editor
syntax	*string*	
use_output_name	*flag*	
output_name	*string*	
output_mode	Screen File	
full_filename	*string*	
file_type	HTML SPV SPW	

statisticsexportnode Properties

The Statistics Export node outputs data in IBM® SPSS® Statistics*.sav* format. The *.sav* files can be read by SPSS Statistics Base and other products. This is also the format used for cache files in IBM® SPSS® Modeler.

Example

```
create statisticsexportnode
set :statisticsexportnode.full_filename = "c:/output/SPSS_Statistics_out.sav"
set :statisticsexportnode.field_names = Names
set :statisticsexportnode.launch_application = True
set :statisticsexportnode.generate_import = True
```

statisticsexportnode properties	**Data type**	**Property description**
full_filename	*string*	
launch_application	*flag*	
export_names	NamesAndLabels NamesAsLabels	Used to map field names from SPSS Modeler upon export to SPSS Statistics or SAS variable names.
generate_import	*flag*	

Chapter 22

SuperNode Properties

Properties that are specific to SuperNodes are described in the following tables. Note that common node properties also apply to SuperNodes.

Table 22-1
source_supernode

Property name	Property type/List of values	Property description
parameters	*any*	Use this property to create and access parameters specified in a SuperNode's parameter table. See details below.

Table 22-2
process_supernode

Property name	Property type/List of values	Property description
parameters	*any*	Use this property to create and access parameters specified in a SuperNode's parameter table. See details below.

Table 22-3
terminal_supernode

Property name	Property type/List of values	Property description
parameters	*any*	Use this property to create and access parameters specified in a SuperNode's parameter table. See details below.
execute_method	Script Normal	
script	*string*	

SuperNode Parameters

You can use scripts to create or set SuperNode parameters using the general format:

```
set mySuperNode.parameters.minvalue = 30
```

Alternatively, you can specify the type of SuperNode in addition to (or instead of) the name:

```
set :process_supernode.parameters.minvalue = 30
```

```
set mySuperNode:process_supernode.parameters.minvalue = 30
```

You can also set the parameter value using a CLEM expression:

```
set :process_supernode.parameters.minvalue = "<expression>"
```

Setting Properties for Encapsulated Nodes

You can set properties for specific nodes encapsulated within a SuperNode by creating a SuperNode parameter to match the literal name of the node and property you want to set. For example, suppose you have a source SuperNode with an encapsulated Variable File node to read in the data. You can pass the name of the file to read (specified using the full_filename property) as follows:

```
set :source_supernode.parameters.':variablefilenode.full_filename' = "c:/mydata.txt"
```

This creates a SuperNode parameter named :variablefilenode.full_filename with a value of c:/mydata.txt. Assuming a node of the specified type exists in the SuperNode, its value for the named property will be set accordingly. Note that this is done in the stream script—that is, the script for the stream that *includes* the SuperNode—rather than the SuperNode script. Be sure to use single quotation marks to specify the parameter name.

This approach can be used with any encapsulated node, as long as a valid node and property reference results. For example, to set the rand_pct property for an encapsulated Sample node, any of the following could be used:

```
set mySuperNode.parameters.':samplenode.rand_pct' = 50
```

or

```
set mySuperNode.parameters.'Sample.rand_pct'= 50
```

or

```
set mySuperNode.parameters.'Sample:samplenode.rand_pct'= 50
```

The first reference above assumes that there is only one Sample node in the stream; the second, that there is only one node named “Sample” regardless of type. The third reference is the most explicit in that it specifies both the name and type for the node.

Limitations of SuperNode scripts. SuperNodes cannot manipulate other streams and cannot change the current stream. Therefore, commands that apply to streams, such as open stream, get stream, execute_script, and so on, cannot be used in SuperNode scripts.

Appendix

A

Notices

This information was developed for products and services offered worldwide.

IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing, IBM Corporation, North Castle Drive, Armonk, NY 10504-1785, U.S.A.

For license inquiries regarding double-byte character set (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

Intellectual Property Licensing, Legal and Intellectual Property Law, IBM Japan Ltd., 1623-14, Shimotsuruma, Yamato-shi, Kanagawa 242-8502 Japan.

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Software Group, Attention: Licensing, 233 S. Wacker Dr., Chicago, IL 60606, USA.

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurements may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

If you are viewing this information softcopy, the photographs and color illustrations may not appear.

Trademarks

IBM, the IBM logo, ibm.com, and SPSS are trademarks of IBM Corporation, registered in many jurisdictions worldwide. A current list of IBM trademarks is available on the Web at *http://www.ibm.com/legal/copytrade.shtml*.

Intel, Intel logo, Intel Inside, Intel Inside logo, Intel Centrino, Intel Centrino logo, Celeron, Intel Xeon, Intel SpeedStep, Itanium, and Pentium are trademarks or registered trademarks of Intel Corporation or its subsidiaries in the United States and other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

UNIX is a registered trademark of The Open Group in the United States and other countries.

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other product and service names might be trademarks of IBM or other companies.

Index

