IBM SPSS Modeler 16 Python Scripting and Automation Guide

Product Information

This edition applies to version 16, release 0, modification 0 of IBM SPSS Modeler and to all subsequent releases and modifications until otherwise indicated in new editions.

Contents

Chapter 1. Scripting

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 and conditionally execute nodes depending on whether the conditions for execution have been met.
- Create loops to repeatedly execute nodes within the stream.
- 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 IBM SPSS Modeler interface. More information on scripting language, syntax, and commands is provided in the chapters that follow.[1]

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

1. The IBM SPSS Modeler Legacy scripting language is still available for use with IBM SPSS Modeler 16. See the document *IBM SPSS Modeler 16 Scripting and Automation Guide* for more information. See Appendix B, "Migrating from legacy scripting to Python scripting," on page 237 for guidance on mapping your existing IBM SPSS Modeler Legacy scripts to Python scripts.

1. From the Tools menu, choose:
   Stream Properties > **Execution**
2. Click the **Execution** tab to work with scripts for the current stream.
3. Select the Execution mode: **Default (optional scrip)**.

The toolbar icons at the top of the stream script 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.
- Edit a script (undo, cut, copy, paste, and other common edit functions).
- Execute the entire current script.
- Execute selected lines from a script.
- Stop a script during execution. (This icon is only enabled when a script is running.)
- 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.

You can also choose to change the type of scripting from Python scripting to legacy scripting.

The script editor includes the following features that help with script authoring:
- Syntax highlighting; keywords, literal values (such as strings and numbers), and comments are highlighted.
- Line numbering.
- Block matching; when the cursor is placed by the start of a program block, the corresponding end block is also highlighted.
- Suggested auto-completion.

The colors and text styles used by the syntax highlighter can be customized using the IBM SPSS Modeler display preferences. You can access the display preferences by choosing **Tools** > **Options** > **User Options** and clicking the **Syntax** tab.

A list of suggested syntax completions can be accessed by selecting **Auto-Suggest** from the context menu, or pressing Ctrl + Space. Use the cursor keys to move up and down the list, then press Enter to insert the selected text. Press Esc to exit from auto-suggest mode without modifying the existing text.

The **Debug** tab displays debugging messages and can be used to evaluate script state once the script has been executed. The **Debug** tab consists of a read-only text area and a single line input text field. The text area displays text that is sent to either standard output, for example through the Python `print` command, or standard error by the scripts, for example through error message text. The input text field takes input from the user. This input is then evaluated within the context of the script that was most recently executed within the dialog (known as the *scripting context*). The text area contains the command and resulting output so that the user can see a trace of commands. The input text field always contains the command prompt (>>> for Python scripting).

A new scripting context is created in the following circumstances:
- A script is executed using the “Run this script” button or the “Run selected lines” button.

- The scripting language is changed.

If a new scripting context is created, the text area is cleared.

Note: Executing a stream outside of the script panel will not modify the script context of the script panel. The values of any variables created as part of that execution will not be visible within the script dialog.

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**

The same toolbar and script syntax-checking options are available for standalone scripts as for stream scripts. See the topic "Stream Scripts" on page 1 for more information.

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. See the topic Chapter 19, "SuperNode Properties," on page 231 for more information. 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...

Looping and conditional execution in streams

From version 16.0 onwards, SPSS Modeler enables you to create some basic scripts from within a stream by selecting values within various dialog boxes instead of having to write instructions directly in the scripting language. The two main types of scripts you can create in this way are simple loops and a way to execute nodes if a condition has been met.

You can combine both looping and conditional execution rules within a stream. For example, you may have data relating to sales of cars from manufacturers worldwide. You could set up a loop to process the data in a stream, identifying details by the country of manufacture, and output the data to different graphs showing details such as sales volume by model, emissions levels by both manufacturer and engine size, and so on. If you were interested in analyzing European information only, you could also add conditions to the looping that prevented graphs being created for manufacturers based in America and Asia.

Note: Because both looping and conditional execution are based on background scripts they are only applied to a whole stream when it is run.

- **Looping** You can use looping to automate repetitive tasks. For example, this might mean adding a given number of nodes to a stream and changing one node parameter each time. Alternatively, you could control the running of a stream or branch again and again for a given number of times, as in the following examples:
  - Run the stream a given number of times and change the source each time.
  - Run the stream a given number of times, changing the value of a variable each time.
  - Run the stream a given number of times, entering one extra field on each execution.
  - Build a model a given number of times and change a model setting each time.
- **Conditional Execution** You can use this to control how terminal nodes are run, based on conditions that you predefine, examples may include the following:
  - Based on whether a given value is true or false, control if a node will be run.
  - Define whether looping of nodes will be run in parallel or sequentially.

Both looping and conditional execution are set up on the Execution tab within the Stream Properties dialog box. Any nodes that are used in conditional or looping requirements are shown with an additional symbol attached to them on the stream canvas to indicate that they are taking part in looping and conditional execution.

You can access the Execution tab in one of 3 ways:

- Using the menus at the top of the main dialog box:
  1. From the Tools menu, choose:

     Stream Properties > **Execution**
  2. Click the Execution tab to work with scripts for the current stream.
- From within a stream:
  1. Right-click on a node and choose **Looping/Conditional Execution**.
  2. Select the relevant submenu option.
- From the graphic toolbar at the top of the main dialog box, click the stream properties icon.

If this is the first time you have set up either looping or conditional execution details, on the Execution tab select the **Looping/Conditional Execution** execution mode and then select either the **Conditional** or **Looping** subtab.

Looping in streams

With looping you can automate repetitive tasks in streams; examples may include the following:

- Run the stream a given number of times and change the source each time.
- Run the stream a given number of times, changing the value of a variable each time.
- Run the stream a given number of times, entering one extra field on each execution.
- Build a model a given number of times and change a model setting each time.

You set up the conditions to be met on the **Looping** subtab of the stream Execution tab. To display the subtab, select the **Looping/Conditional Execution** execution mode.

Any looping requirements that you define will take effect when you run the stream, if the **Looping/Conditional Execution** execution mode has been set. Optionally, you can generate the script code for your looping requirements and paste it into the script editor by clicking **Paste...** in the bottom right corner of the Looping subtab; the main Execution tab display changes to show the **Default (optional script)** execution mode with the script in the top part of the tab. This means that you can define a looping structure using the various looping dialog box options before generating a script that you can customize further in the script editor. Note that when you click **Paste...** any conditional execution requirements you have defined will also be displayed in the generated script.

To set up a loop:

1. Create an iteration key to define the main looping structure to be carried out in a stream. See Create an iteration key for more information.
2. Where needed, define one or more iteration variables. See Create an iteration variable for more information.
3. The iterations and any variables you created are shown in the main body of the subtab. By default, iterations are executed in the order they appear; to move an iteration up or down the list, click on it to select it then use the up or down arrow in the right hand column of the subtab to change the order.

Creating an iteration key for looping in streams

You use an iteration key to define the main looping structure to be carried out in a stream. For example, if you are analyzing car sales, you could create a stream parameter *Country of manufacture* and use this as the iteration key; when the stream is run this key is set to each different country value in your data during each iteration. Use the Define Iteration Key dialog box to set up the key.

To open the dialog box, either select the **Iteration Key...** button in the bottom left corner of the Looping subtab, or right click on any node in the stream and select either **Looping/Conditional Execution** > **Define Iteration Key (Fields)** or **Looping/Conditional Execution** > **Define Iteration Key (Values)**. If you open the dialog box from the stream, some of the fields may be completed automatically for you, such as the name of the node.

To set up an iteration key, complete the following fields:

Iterate on. You can select from one of the following options:

- **Stream Parameter - Fields**. Use this option to create a loop that sets the value of an existing stream parameter to each specified field in turn.
- **Stream Parameter - Values**. Use this option to create a loop that sets the value of an existing stream parameter to each specified value in turn.
- **Node Property - Fields**. Use this option to create a loop that sets the value of a node property to each specified field in turn.
- **Node Property - Values**. Use this option to create a loop that sets the value of a node property to each specified value in turn.

What to Set. Choose the item that will have its value set each time the loop is executed. You can select from one of the following options:

- **Parameter**. Only available if you select either **Stream Parameter - Fields** or **Stream Parameter - Values**. Select the required parameter from the available list.
- **Node**. Only available if you select either **Node Property - Fields** or **Node Property - Values**. Select the node for which you want to set up a loop. Click the browse button to open the Select Node dialog and choose the node you want; if there are too many nodes listed you can filter the display to only show certain types of nodes by selecting one of the following categories: Source, Process, Graph, Modeling, Output, Export, or Apply Model nodes.
- **Property**. Only available if you select either **Node Property - Fields** or **Node Property - Values**. Select the property of the node from the available list.

Fields to Use. Only available if you select either **Stream Parameter - Fields** or **Node Property - Fields**. Choose the field, or fields, within a node to use to provide the iteration values. You can select from one of the following options:

- **Node**. Only available if you select **Stream Parameter - Fields**. Select the node that contains the details for which you want to set up a loop. Click the browse button to open the Select Node dialog and choose the node you want; if there are too many nodes listed you can filter the display to only show certain types of nodes by selecting one of the following categories: Source, Process, Graph, Modeling, Output, Export, or Apply Model nodes.
- **Field List**. Click the list button in the right column to display the Select Fields dialog box, within which you select the fields in the node to provide the iteration data. See "Selecting fields for iterations" on page 7 for more information.

Values to Use. Only available if you select either **Stream Parameter - Values** or **Node Property - Values**. Choose the value, or values, within the selected field to use as iteration values. You can select from one of the following options:

- **Node**. Only available if you select **Stream Parameter - Values**. Select the node that contains the details for which you want to set up a loop. Click the browse button to open the Select Node dialog and choose the node you want; if there are too many nodes listed you can filter the display to only show certain types of nodes by selecting one of the following categories: Source, Process, Graph, Modeling, Output, Export, or Apply Model nodes.
- **Field List**. Select the field in the node to provide the iteration data.
- **Value List**. Click the list button in the right column to display the Select Values dialog box, within which you select the values in the field to provide the iteration data.

Creating an iteration variable for looping in streams

You can use iteration variables to change the values of stream parameters or properties of selected nodes within a stream each time a loop is executed. For example, if your stream loop is analyzing car sales data and using *Country of manufacture* as the iteration key, you may have one graph output showing sales by model and another graph output showing exhaust emissions information. In these cases you could create iteration variables that create new titles for the resultant graphs, such as *Swedish vehicle emissions* and *Japanese car sales by model*. Use the Define Iteration Variable dialog box to set up any variables that you require.

To open the dialog box, either select the **Iteration Variable...** button in the bottom left corner of the Looping subtab, or right click on any node in the stream and select:**Looping/Conditional Execution** > **Define Iteration Variable**.

To set up an iteration variable, complete the following fields:

Change. Select the type of attribute that you want to amend. You can choose from either **Stream Parameter** or **Node Property**.

- If you select **Stream Parameter**, choose the required parameter and then, by using one of the following options, if available in your stream, define what the value of that parameter should be set to with each iteration of the loop:
  - **Global variable**. Select the global variable that the stream parameter should be set to.
  - **Table output cell**. To set a stream parameter to be the value in a table output cell, select the table from the list and enter the **Row** and **Column** to be used.
  - **Enter manually**. Select this if you want to manually enter a value for this parameter to take in each iteration. When you return to the Looping subtab a new column is created into which you enter the required text.
- If you select **Node Property**, choose the required node and one of its properties and then set the value you want to use for that property. Set the new property value by using one of the following options:
  - **Alone**. The property value will use the iteration key value. See "Creating an iteration key for looping in streams" on page 5 for more information.
  - **As prefix to stem**. Uses the iteration key value as a prefix to what you enter in the **Stem** field.
  - **As suffix to stem**. Uses the iteration key value as a suffix to what you enter in the **Stem** field

  If you select either the prefix or suffix option you are prompted to add the additional text to the **Stem** field. For example, if your iteration key value is *Country of manufacture*, and you select **As prefix to stem**, you might enter *- sales by model* in this field.

Selecting fields for iterations

When creating iterations you can select one or more fields using the Select Fields dialog box.

Sort by. You can sort available fields for viewing by selecting one of the following options:
- **Natural**. View the order of fields as they have been passed down the data stream into the current node.
- **Name**. Use alphabetical order to sort fields for viewing.
- **Type**. View fields sorted by their measurement level. This option is useful when selecting fields with a particular measurement level.

Select fields from the list one at a time or use the Shift-click and Ctrl-click methods to select multiple fields. You can also use the buttons below the list to select groups of fields based on their measurement level, or to select or deselect all fields in the table.

Note that the fields available for selection are filtered to show only the fields that are appropriate for the stream parameter or node property you are using. For example, if you are using a stream parameter that has a storage type of String, only fields that have a storage type of String are shown.

Conditional execution in streams

With conditional execution you can control how terminal nodes are run, based on the stream contents matching conditions that you define; examples may include the following:
- Based on whether a given value is true or false, control if a node will be run.
- Define whether looping of nodes will be run in parallel or sequentially.

You set up the conditions to be met on the **Conditional** subtab of the stream Execution tab. To display the subtab, select the **Looping/Conditional Execution** execution mode.

Any conditional execution requirements that you define will take effect when you run the stream, if the **Looping/Conditional Execution** execution mode has been set. Optionally, you can generate the script code for your conditional execution requirements and paste it into the script editor by clicking **Paste...** in the bottom right corner of the Conditional subtab; the main Execution tab display changes to show the **Default (optional script)** execution mode with the script in the top part of the tab. This means that you can define conditions using the various looping dialog box options before generating a script that you can customize further in the script editor. Note that when you click **Paste...** any looping requirements you have defined will also be displayed in the generated script.

To set up a condition:

1. In the right hand column of the Conditional subtab, click the Add Execution Statement button to open the Conditional Execution Statement dialog box. In this dialog you specify the condition that must be met in order for the node to be executed.
2. In the Conditional Execution Statement dialog box, specify the following:
   a. **Node**. Select the node for which you want to set up conditional execution. Click the browse button to open the Select Node dialog and choose the node you want; if there are too many nodes listed you can filter the display to show nodes by one of the following categories: Export, Graph, Modeling, or Output node.
   b. **Condition based on**. Specify the condition that must be met for the node to be executed. You can choose from one of four options: **Stream parameter**, **Global variable**, **Table output cell**, or **Always true**. The details you enter in the bottom half of the dialog box are controlled by the condition you choose.
      - **Stream parameter**. Select the parameter from the list available and then choose the **Operator** for that parameter; for example, the operator may be More than, Equals, Less than, Between, and so on. You then enter the **Value**, or minimum and maximum values, depending on the operator.
      - **Global variable**. Select the variable from the list available; for example, this might include: Mean, Sum, Minimum value, Maximum value, or Standard deviation. You then select the **Operator** and values required.
      - **Table output cell**. Select the table node from the list available and then choose the **Row** and **Column** in the table. You then select the **Operator** and values required.
      - **Always true**. Select this option if the node must always be executed. If you select this option, there are no further parameters to select.
3. Repeat steps 1 and 2 as often as required until you have set up all the conditions you require. The node you selected and the condition to be met before that node is executed are shown in the main body of the subtab in the **Execute Node** and **If this condition is true** columns respectively.
4. By default, nodes and conditions are executed in the order they appear; to move a node and condition up or down the list, click on it to select it then use the up or down arrow in the right hand column of the subtab to change the order.

In addition, you can set the following options at the bottom of the Conditional subtab:

- **Evaluate all in order**. Select this option to evaluate each condition in the order in which they are shown on the subtab. The nodes for which conditions have been found to be "True" will all be executed once all the conditions have been evaluated.
- **Execute one at a time**. Only available if **Evaluate all in order** is selected. Selecting this means that if a condition is evaluated as "True", the node associated with that condition is executed before the next condition is evaluated.
- **Evaluate until first hit**. Selecting this means that only the first node that returns a "True" evaluation from the conditions you specified will be run.

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 1. 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. 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. See the topic "Using Command Line Arguments" on page 49 for more information.

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, 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.

1. With the cursor in a text area, press Ctrl+F to access the Find/Replace dialog box.
2. Enter the text you want to search for, or choose from the drop-down list of recently searched items.
3. Enter the replacement text, if any.
4. Click **Find Next** to start the search.
5. Click **Replace** to replace the current selection, or **Replace All** to update all or selected instances.
6. 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. A regular expression pattern describes the structure of the string that the expression will try to find in an input string. The following types of regular expression constructs are supported.

Table 1. 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

Table 2. 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]

Table 3. 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]

Table 3. Predefined character classes (continued)

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

Table 4. 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

For more information about using regular expressions, and for some examples, see http://www.ibm.com/developerworks/java/tutorials/j-introtojava2/section9.html.

Examples

The following code searches for and matches the three numbers at the start of a string:

```
^[0-9]{3}
```

The following code searches for and matches the three numbers at the end of a string:

```
[0-9]{3}$
```

Chapter 2. The Scripting Language

Scripting Language Overview

The scripting facility for IBM SPSS Modeler enables you to create scripts that operate on the SPSS Modeler user interface, manipulate output objects, and run command syntax. You can run scripts directly from within SPSS Modeler.

Scripts in IBM SPSS Modeler are written in the scripting language Python. The Java-based implementation of Python that is used by IBM SPSS Modeler is called Jython. The scripting language consists of the following features:

- A format for referencing nodes, streams, projects, output, and other IBM 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.

The following sections describe the Python scripting language, the Jython implementation of Python, and the basic syntax for getting started with scripting within IBM SPSS Modeler. Information about specific properties and commands is provided in the sections that follow.

Python and Jython

Jython is an implementation of the Python scripting language, which is written in the Java language and integrated with the Java platform. Python is a powerful object-oriented scripting language. Jython is useful because it provides the productivity features of a mature scripting language and, unlike Python, runs in any environment that supports a Java virtual machine (JVM). This means that the Java libraries on the JVM are available to use when you are writing programs. With Jython, you can take advantage of this difference, and use the syntax and most of the features of the Python language

As a scripting language, Python (and its Jython implementation) is easy to learn and efficient to code, and has minimal required structure to create a running program. Code can be entered interactively, that is, one line at a time. Python is an interpreted scripting language; there is no precompile step, as there is in Java. Python programs are simply text files that are interpreted as they are input (after parsing for syntax errors). Simple expressions, like defined values, as well as more complex actions, such as function definitions, are immediately executed and available for use. Any changes that are made to the code can be tested quickly. Script interpretation does, however, have some disadvantages. For example, use of an undefined variable is not a compiler error, so it is detected only if (and when) the statement in which the variable is used is executed. In this case, the program can be edited and run to debug the error.

Python sees everything, including all data and code, as an object. You can, therefore, manipulate these objects with lines of code. Some select types, such as numbers and strings, are more conveniently considered as values, not objects; this is supported by Python. There is one null value that is supported. This null value has the reserved name `None`.

For a more in-depth introduction to Python and Jython scripting, and for some example scripts, see www.ibm.com/developerworks/java/tutorials/j-jython1 and www.ibm.com/developerworks/java/tutorials/j-jython2.

Python Scripting

This guide to the Python scripting language is an introduction to the components that are most likely to be used when scripting in IBM SPSS Modeler, including concepts and programming basics. This will provide you with enough knowledge to start developing your own Python scripts to use within IBM SPSS Modeler.

Operations

Assignment is done using an equals sign (=). For example, to assign the value "3" to a variable called "x" you would use the following statement:

```
x = 3
```

The equals sign is also used to assign string type data to a variable. For example, to assign the value "a string value" to the variable "y" you would use the following statement:

```
y = "a string value"
```

The following table lists some commonly used comparison and numeric operations, and their descriptions.

Table 5. Common comparison and numeric operations

Operation	**Description**
`x < y`	Is `x` less than `y`?
`x > y`	Is `x` greater than `y`?
`x <= y`	Is `x` less than or equal to `y`?
`x >= y`	Is `x` greater than or equal to `y`?
`x == y`	Is `x` equal to `y`?
`x != y`	Is `x` not equal to `y`?
`x <> y`	Is `x` not equal to `y`?
`x + y`	Add `y` to `x`
`x - y`	Subtract `y` from `x`
`x * y`	Multiply `x` by `y`
`x / y`	Divide `x` by `y`
`x ** y`	Raise `x` to the `y` power

Lists

Lists are sequences of elements. A list can contain any number of elements, and the elements of the list can be any type of object. Lists can also be thought of as arrays. The number of elements in a list can increase or decrease as elements are added, removed, or replaced.

Examples

`[]`	Any empty list.
`[1]`	A list with a single element, an integer.
`["Mike", 10, "Don", 20]`	A list with four elements, two string elements and two integer elements.
`[[],[7],[8,9]]`	A list of lists. Each sub-list is either an empty list or a list of integer elements.

`x = 7; y = 2; z = 3;` `[1, x, y, x + y]`	A list of integers. This example demonstrates the use of variables and expressions.

You can assign a list to a variable, for example:

```
mylist1 = ["one", "two", "three"]
```

You can then access specific elements of the list, for example:

```
mylist[0]
```

This will result in the following output:

```
one
```

The number in the brackets ([]) is known as an *index* and refers to a particular element of the list. The elements of a list are indexed starting from 0.

You can also select a range of elements of a list; this is called *slicing*. For example, `x[1:3]` selects the second and third elements of `x`. The end index is one past the selection.

Strings

A *string* is an immutable sequence of characters that is treated as a value. Strings support all of the immutable sequence functions and operators that result in a new string. For example, `"abcdef"[1:4]` results in the output `"bcd"`.

In Python, characters are represented by strings of length one.

Strings literals are defined by the use of single or triple quoting. Strings that are defined using single quotes cannot span lines, while strings that are defined using triple quotes can. A string can be enclosed in single quotes (') or double quotes ("). A quoting character may contain the other quoting character un-escaped or the quoting character escaped, that is preceded by the backslash (\) character.

Examples

```
"This is a string"
'This is also a string'
"It's a string"
'This book is called "Python Scripting and Automation Guide".'
"This is an escape quote (\") in a quoted string"
```

Multiple strings separated by white space are automatically concatenated by the Python parser. This makes it easier to enter long strings and to mix quote types in a single string, for example:

```
"This string uses ' and " 'that string uses ".'
```

This results in the following output:

```
This string uses ' and that string uses ".
```

Strings support several useful methods. Some of these methods are given in the following table.

Table 6. String methods

Method	Usage
`s.capitalize()`	Initial capitalize `s`
`s.count(ss {,start {,end}})`	Count the occurrences of `ss` in `s[start:end]`

`s.startswith(str {, start {, end}})`
`s.endswith(str {, start {, end}})`
	Test to see if `s` starts with `str`
	Test to see if `s` ends with `str`

Table 6. String methods (continued)

Method	**Usage**
`s.expandtabs({size})`	Replace tabs with spaces, default `size` is 8

`s.find(str {, start {, end}})`
`s.rfind(str {, start {, end}})`
	Finds first index of `str` in `s`; if not found, the result is -1. `rfind` searches right to left.

`s.index(str {, start {, end}})`
`s.rindex(str {, start {, end}})`
	Finds first index of `str` in `s`; if not found: raise `ValueError`. `rindex` searches right to left.

`s.isalnum`	Test to see if the string is alphanumeric
`s.isalpha`	Test to see if the string is alphabetic
`s.isnum`	Test to see if the string is numeric
`s.isupper`	Test to see if the string is all uppercase
`s.islower`	Test to see if the string is all lowercase
`s.isspace`	Test to see if the string is all whitespace
`s.istitle`	Test to see if the string is a sequence of initial cap alphanumeric strings

`s.lower()`
`s.upper()`
`s.swapcase()`
`s.title()`
	Convert to all lower case
	Convert to all upper case
	Convert to all opposite case
	Convert to all title case

`s.join(seq)`	Join the strings in `seq` with `s` as the separator
`s.splitlines({keep})`	Split `s` into lines, if keep is `true`, keep the new lines
`s.split({sep {, max}})`	Split `s` into "words" using `sep` (default `sep` is a white space) for up to `max` times

`s.ljust(width)`
`s.rjust(width)`
`s.center(width)`
`s.zfill(width)`
	Left justify the string in a field `width` wide
	Right justify the string in a field `width` wide
	center justify the string in a field `width` wide
	Fill with 0.

`s.lstrip()`
`s.rstrip()`
`s.strip()`
	Remove leading white space
	Remove trailing white space
	Remove leading and trailing white space

`s.translate(str {,delc})`	Translate `s` using table, after removing any characters in `delc`. `str` should be a string with length `== 256`.
`s.replace(old, new {, max})`	Replaces all or `max` occurrences of string `old` with string `new`

Remarks

Remarks are comments that are introduced by the pound (or hash) sign (#). All text that follows the pound sign on the same line is considered part of the remark and is ignored. A remark can start in any column. The following example demonstrates the use of remarks:

```
#The HelloWorld application is one of the most simple
print 'Hello World' # print the Hello World line
```

Statement Syntax

The statement syntax for Python is very simple. In general, each source line is a single statement. Except for `expression` and `assignment` statements, each statement is introduced by a keyword name, such as `if` or `for`. Blank lines or remark lines can be inserted anywhere between any statements in the code. If there is more than one statement on a line, each statement must be separated by a semicolon (`;`).

Very long statements can continue on more than one line. In this case the statement that is to continue on to the next line must end with a backslash (\), for example:

```
x = "A loooooooooooooooooooong string" + \
     "another looooooooooooooooooong string"
```

When a structure is enclosed by parentheses (()), brackets ([]), or curly braces ({}), the statement can be continued on to a new line after any comma, without having to insert a backslash, for example:

```
x = (1, 2, 3, "hello",
      "goodbye", 4, 5, 6)
```

Identifiers

Identifiers are used to name variables, functions, classes and keywords. Identifiers can be any length, but must start with either an alphabetical character of upper or lower case, or the underscore character (_). Names that start with an underscore are generally reserved for internal or private names. After the first character, the identifier can contain any number and combination of alphabetical characters, numbers from 0-9, and the underscore character.

There are some reserved words in Jython that cannot be used to name variables, functions, or classes. They fall under the following categories:

- **Statement introducers:** `assert`, `break`, `class`, `continue`, `def`, `del`, `elif`, `else`, `except`, `exec`, `finally`, `for`, `from`, `global`, `if`, `import`, `pass`, `print`, `raise`, `return`, `try`, and `while`
- **Parameter introducers:** `as`, `import`, and `in`
- **Operators:** `and`, `in`, `is`, `lambda`, `not`, and `or`

Improper keyword use generally results in a `SyntaxError`.

Blocks of Code

Blocks of code are groups of statements that are used where single statements are expected. Blocks of code can follow any of the following statements: `if`, `elif`, `else`, `for`, `while`, `try`, `except`, `def`, and `class`. These statements introduce the block of code with the colon character (:), for example:

```
if x == 1:
    y = 2
    z = 3
elif:
    y = 4
    z = 5
```

Indentation is used to delimit code blocks (rather than the curly braces that are used in Java). All lines in a block must be indented to the same position. This is because a change in the indentation indicates the end of a code block. It is usual to indent by four spaces per level. It is recommended that spaces are used to indent the lines, rather than tabs. Spaces and tabs must not be mixed. The lines in the outermost block of a module must start at column one, else a `SyntaxError` will occur.

The statements that make up a code block (and follow the colon) can also be on a single line, separated by semicolons, for example:

```
if x == 1: y = 2; z = 3;
```

Passing Arguments to a Script

Passing arguments to a script is useful as it means a script can be used repeatedly without modification. The arguments that are passed on the command line are passed as values in the list `sys.argv`. The number of values passed can be obtained by using the command `len(sys.argv)`. For example:

```
import sys
print "test1"
print sys.argv[0]
print sys.argv[1]
print len(sys.argv)
```

In this example, the `import` command imports the entire `sys` class so that the methods that exist for this class, such as `argv`, can be used.

The script in this example can be invoked using the following line:

```
/u/mjloos/test1 mike don
```

The result is the following output:

```
/u/mjloos/test1 mike don
test1
mike
don
3
```

Examples

The `print` keyword prints the arguments immediately following it. If the statement is followed by a comma, a new line is not included in the output. For example:

```
print "This demonstrates the use of a",
print " comma at the end of a print statement."
```

This will result in the following output:

```
This demonstrates the use of a comma at the end of a print statement.
```

The `for` statement is used to iterate through a block of code. For example:

```
mylist1 = ["one", "two", "three"]
for lv in mylist1:
    print lv
    continue
```

In this example, three strings are assigned to the list `mylist1`. The elements of the list are then printed, with one element of each line. This will result in the following output:

```
one
two
three
```

In this example, the iterator `lv` takes the value of each element in the list `mylist1` in turn as the for loop implements the code block for each element. An iterator can be any valid identifier of any length.

The `if` statement is a conditional statement. It evaluates the condition and returns either true or false, depending on the result of the evaluation. For example:

```
mylist1 = ["one", "two", "three"]
for lv in mylist1:
    if lv == "two"
        print "The value of lv is ", lv
    else
        print "The value of lv is not two, but ", lv
    continue
```

In this example, the value of the iterator `lv` is evaluated. If the value of `lv` is `two` a different string is returned to the string that is returned if the value of `lv` is not `two`. This results in the following output:

```
The value of lv is not two, but one
The value of lv is two
The value of lv is not two, but three
```

Mathematical Methods

From the math module you can access useful mathematical methods. Some of these methods are given in the following table. Unless specified otherwise, all values are returned as floats.

Table 7. Mathematical methods

Method	Usage
`math.ceil(x)`	Return the ceiling of x as a float, that is the smallest integer greater than or equal to x
`math.copysign(x, y)`	Return x with the sign of y. `copysign(1, -0.0)` returns -1
`math.fabs(x)`	Return the absolute value of x
`math.factorial(x)`	Return x factorial. If x is negative or not an integer, a `ValueError` is raised.
`math.floor(x)`	Return the floor of x as a float, that is the largest integer less than or equal to x
`math.frexp(x)`	Return the mantissa (m) and exponent (e) of x as the pair (m, e). m is a float and e is an integer, such that `x == m * 2**e` exactly. If x is zero, returns (0.0, 0), otherwise `0.5 <= abs(m) < 1`.
`math.fsum(iterable)`	Return an accurate floating point sum of values in `iterable`
`math.isinf(x)`	Check if the float x is positive or negative infinitive
`math.isnan(x)`	Check if the float x is `NaN` (not a number)
`math.ldexp(x, i)`	Return `x * (2**i)`. This is essentially the inverse of the function `frexp`.
`math.modf(x)`	Return the fractional and integer parts of x. Both results carry the sign of x and are floats.
`math.trunc(x)`	Return the `Real` value x, that has been truncated to an `Integral`.
`math.exp(x)`	Return `e**x`
`math.log(x[, base])`	Return the logarithm of x to the given value of base. If base is not specified, the natural logarithm of x is returned.
`math.log1p(x)`	Return the natural logarithm of 1+x (base e)
`math.log10(x)`	Return the base-10 logarithm of x
`math.pow(x, y)`	Return x raised to the power y. `pow(1.0, x)` and `pow(x, 0.0)` always return 1, even when x is zero or NaN.
`math.sqrt(x)`	Return the square root of x

In addition to the mathematical functions, there are some useful trigonometric methods. These methods are shown in the following table.

Table 8. Trigonometric methods

Method	Usage
`math.acos(x)`	Return the arc cosine of x in radians
`math.asin(x)`	Return the arc sine of x in radians
`math.atan(x)`	Return the arc tangent of x in radians
`math.atan2(y, x)`	Return `atan(y / x)` in radians.

Table 8. Trigonometric methods (continued)

Method	**Usage**
`math.cos(x)`	Return the cosine of `x` in radians.
`math.hypot(x, y)`	Return the Euclidean norm `sqrt(x*x + y*y)`. This is the length of the vector from the origin to the point `(x, y)`.
`math.sin(x)`	Return the sine of `x` in radians
`math.tan(x)`	Return the tangent of `x` in radians
`math.degrees(x)`	Convert angle `x` from radians to degrees
`math.radians(x)`	Convert angle `x` from degrees to radians
`math.acosh(x)`	Return the inverse hyperbolic cosine of `x`
`math.asinh(x)`	Return the inverse hyperbolic sine of `x`
`math.atanh(x)`	Return the inverse hyperbolic tangent of `x`
`math.cosh(x)`	Return the hyperbolic cosine of `x`
`math.sinh(x)`	Return the hyperbolic cosine of `x`
`math.tanh(x)`	Return the hyperbolic tangent of `x`

There are also two mathematical constants. The value of `math.pi` is the mathematical constant pi. The value of `math.e` is the mathematical constant e.

Using Non-ASCII characters

In order to use non-ASCII characters, Python requires explicit encoding and decoding of strings into Unicode. In IBM SPSS Modeler, Python scripts are assumed to be encoded in UTF-8, which is a standard Unicode encoding that supports non-ASCII characters. The following script will compile because the Python compiler has been set to UTF-8 by SPSS Modeler.

```
stream = modeler.script.stream()
filenode = stream.createAt("variablefile", "テストノード", 96, 64)
```

However, the resulting node will have an incorrect label.

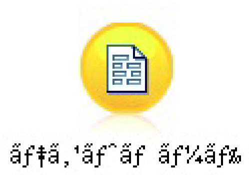

Figure 3. Node label containing non-ASCII characters, displayed incorrectly

The label is incorrect because the string literal itself has been converted to an ASCII string by Python.

Python allows Unicode string literals to be specified by adding a `u` character prefix before the string literal:

```
stream = modeler.script.stream()
filenode = stream.createAt("variablefile", u"テストノード", 96, 64)
```

This will create a Unicode string and the label will be appear correctly.

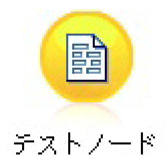

Figure 4. Node label containing non-ASCII characters, displayed correctly

Using Python and Unicode is a large topic which is beyond the scope of this document. Many books and online resources are available that cover this topic in great detail.

Object-Oriented Programming

Object-oriented programming is based on the notion of creating a model of the target problem in your programs. Object-oriented programming reduces programming errors and promotes the reuse of code. Python is an object-oriented language. Objects defined in Python have the following features:

- **Identity.** Each object must be distinct, and this must be testable. The `is` and `is not` tests exist for this purpose.
- **State.** Each object must be able to store state. Attributes, such as fields and instance variables, exist for this purpose.
- **Behavior.** Each object must be able to manipulate its state. Methods exist for this purpose.

Python includes the following features for supporting object-oriented programming:

- **Class-based object creation.** Classes are templates for the creation of objects. Objects are data structures with associated behavior.
- **Inheritance with polymorphism.** Python supports single and multiple inheritance. All Python instance methods are polymorphic and can be overridden by subclasses.
- **Encapsulation with data hiding.** Python allows attributes to be hidden. When hidden, attributes can be accessed from outside the class only through methods of the class. Classes implement methods to modify the data.

Defining a Class

Within a Python class, both variables and methods can be defined. Unlike in Java, in Python you can define any number of public classes per source file (or *module*). Therefore, a module in Python can be thought of similar to a package in Java.

In Python, classes are defined using the `class` statement. The `class` statement has the following form:

```
class name (superclasses): statement
```

or

```
class name (superclasses):
    assignment
    .
    .
    function
    .
    .
```

When you define a class, you have the option to provide zero or more *assignment* statements. These create class attributes that are shared by all instances of the class. You can also provide zero or more *function* definitions. These function definitions create methods. The superclasses list is optional.

The class name should be unique in the same scope, that is within a module, function or class. You can define multiple variables to reference the same class.

Creating a Class Instance

Classes are used to hold class (or shared) attributes or to create class instances. To create an instance of a class, you call the class as if it were a function. For example, consider the following class:

```
class MyClass:
    pass
```

Here, the `pass` statement is used because a statement is required to complete the class, but no action is required programmatically.

The following statement creates an instance of the class `MyClass`:

```
x = MyClass()
```

Adding Attributes to a Class Instance

Unlike in Java, in Python clients can add attributes to an instance of a class. Only the one instance is changed. For example, to add attributes to an instance `x`, set new values on that instance:

```
x.attr1 = 1
x.attr2 = 2
    .
    .
x.attrN = n
```

Defining Class Attributes and Methods

Any variable that is bound in a class is a *class attribute*. Any function defined within a class is a *method*. Methods receive an instance of the class, conventionally called `self`, as the first argument. For example, to define some class attributes and methods, you might enter the following code:

```
class MyClass
    attr1 = 10        #class attributes
    attr2 = "hello"

    def method1(self):
        print MyClass.attr1   #reference the class attribute

    def method2(self):
        print MyClass.attr2   #reference the class attribute

    def method3(self, text):
        self.text = text        #instance attribute
        print text, self.text   #print my argument and my attribute

    method4 = method3   #make an alias for method3
```

Inside a class, you should qualify all references to class attributes with the class name; for example, `MyClass.attr1`. All references to instance attributes should be qualified with the `self` variable; for example, `self.text`. Outside the class, you should qualify all references to class attributes with the class name (for example `MyClass.attr1`) or with an instance of the class (for example `x.attr1`, where `x` is an instance of the class). Outside the class, all references to instance variables should be qualified with an instance of the class; for example, `x.text`.

Hidden Variables

Data can be hidden by creating *Private* variables. Private variables can be accessed only by the class itself. If you declare names of the form `__xxx` or `__xxx_yyy`, that is with two preceding underscores, the Python parser will automatically add the class name to the declared name, creating hidden variables, for example:

```
class MyClass:
    __attr = 10   #private class attribute

    def method1(self):
        pass

    def method2(self, p1, p2):
        pass

    def __privateMethod(self, text):
        self.__text = text    #private attribute
```

Unlike in Java, in Python all references to instance variables must be qualified with `self`; there is no implied use of `this`.

Inheritance

The ability to inherit from classes is fundamental to object-oriented programming. Python supports both single and multiple inheritance. *Single inheritance* means that there can be only one superclass. *Multiple inheritance* means that there can be more than one superclass.

Inheritance is implemented by subclassing other classes. Any number of Python classes can be superclasses. In the Jython implementation of Python, only one Java class can be directly or indirectly inherited from. It is not required for a superclass to be supplied.

Any attribute or method in a superclass is also in any subclass and can be used by the class itself, or by any client as long as the attribute or method is not hidden. Any instance of a subclass can be used wherever and instance of a superclass can be used; this is an example of *polymorphism*. These features enable reuse and ease of extension.

Example

```
class Class1: pass    #no inheritance

class Class2: pass

class Class3(Class1): pass     #single inheritance

class Class4(Class3, Class2): pass     #multiple inheritance
```

Chapter 3. Scripting in IBM SPSS Modeler

Types of scripts

In IBM SPSS Modeler there are three types of script:

- *Stream scripts* are used to control execution of a single stream and are stored within the stream.
- *SuperNode scripts* are used to control the behavior of SuperNodes.
- *Stand-alone or session scripts* can be used to coordinate execution across a number of different streams.

Various methods are available to be used in scripts in IBM SPSS Modeler with which you can access a wide range of SPSS Modeler functionality. These methods are also used in Chapter 4, "The Scripting API," on page 35 to create more advanced functions.

Streams, SuperNode streams, and diagrams

Most of the time, the term *stream* means the same thing, regardless of whether it is a stream that is loaded from a file or used within a SuperNode. It generally means a collection of nodes that are connected together and can be executed. In scripting, however, not all operations are supported in all places, meaning a script author should be aware of which stream variant they are using.

Streams

A stream is the main IBM SPSS Modeler document type. It can be saved, loaded, edited and executed. Streams can also have parameters, global values, a script, and other information associated with them.

SuperNode streams

A *SuperNode stream* is the type of stream used within a SuperNode. Like a normal stream, it contains nodes which are linked together. SuperNode streams have a number of differences from a normal stream:

- Parameters and any scripts are associated with the SuperNode that owns the SuperNode stream, rather than with the SuperNode stream itself.
- SuperNode streams have additional input and output connector nodes, depending on the type of SuperNode. These connector nodes are used to flow information into and out of the SuperNode stream, and are created automatically when the SuperNode is created.

Diagrams

The term *diagram* covers the functions that are supported by both normal streams and SuperNode streams, such as adding and removing nodes, and modifying connections between the nodes.

Executing a stream

The following example runs all executable nodes in the stream, and is the simplest type of stream script:

```
modeler.script.stream().runAll(None)
```

The following example also runs all executable nodes in the stream:

```
stream = modeler.script.stream()
stream.runAll(None)
```

In this example, the stream is stored in a variable called `stream`. Storing the stream in a variable is useful because a script is typically used to modify either the stream or the nodes within a stream. Creating a variable that stores the stream results in a more concise script.

The scripting context

The `modeler.script` module provides the context in which a script is executed. The module is automatically imported into a SPSS Modeler script at run time. The module defines four functions that provide a script with access to its execution environment:

- The `session()` function returns the session for the script. The session defines information such as the locale and the SPSS Modeler backend (either a local process or a networked SPSS Modeler Server) that is being used to run any streams.
- The `stream()` function can be used with stream and SuperNode scripts. This function returns the stream that owns either the stream script or the SuperNode script that is being run.
- The `diagram()` function can be used with SuperNode scripts. This function returns the diagram within the SuperNode. For other script types, this function returns the same as the `stream()` function.
- The `supernode()` function can be used with SuperNode scripts. This function returns the SuperNode that owns the script that is being run.

The four functions and their outputs are summarized in the following table.

Table 9. Summary of `modeler.script` functions

Script type	**`session()`**	**`stream()`**	**`diagram()`**	**`supernode()`**
Standalone	Returns a session	Returns the current managed stream at the time the script was invoked (for example, the stream passed via the batch mode `-stream` option), or `None`.	Same as for `stream()`	Not applicable
Stream	Returns a session	Returns a stream	Same as for `stream()`	Not applicable
SuperNode	Returns a session	Returns a stream	Returns a SuperNode stream	Returns a SuperNode

The `modeler.script` module also defines a way of terminating the script with an exit code. The `exit(`*`exit-code`*`)` function stops the script from executing and returns the supplied integer exit code.

One of the methods that is defined for a stream is `runAll(List)`. This method runs all executable nodes. Any models or outputs that are generated by executing the nodes are added to the supplied list.

It is common for a stream execution to generate outputs such as models, graphs, and other output. To capture this output, a script can supply a variable that is initialized to a list, for example:

```
stream = modeler.script.stream()
results = []
stream.runAll(results)
```

When execution is complete, any objects that are generated by the execution can be accessed from the `results` list.

Referencing existing nodes

A stream is often pre-built with some parameters that must be modified before the stream is executed. Modifying these parameters involves the following tasks:

1. Locating the nodes in the relevant stream.
2. Changing the node or stream settings (or both).

Finding nodes

Streams provide a number of ways of locating an existing node. These methods are summarized in the following table.

Table 10. Methods for locating an existing node

Method	Return type	Description
`s.findAll(type, label)`	Collection	Returns a list of all nodes with the specified type and label. Either the type or label can be `None`, in which case the other parameter is used.
`s.findAll(filter, recursive)`	Collection	Returns a collection of all nodes that are accepted by the specified filter. If the recursive flag is `True`, any SuperNodes within the specified stream are also searched.
`s.findByID(id)`	Node	Returns the node with the supplied ID or `None` if no such node exists. The search is limited to the current stream.
`s.findByType(type, label)`	Node	Returns the node with the supplied type, label, or both. Either the type or name can be `None`, in which case the other parameter is used. If multiple nodes result in a match, then an arbitrary one is chosen and returned. If no nodes result in a match, then the return value is `None`.
`s.findDownstream(fromNodes)`	Collection	Searches from the supplied list of nodes and returns the set of nodes downstream of the supplied nodes. The returned list includes the originally supplied nodes.
`s.findUpstream(fromNodes)`	Collection	Searches from the supplied list of nodes and returns the set of nodes upstream of the supplied nodes. The returned list includes the originally supplied nodes.

As an example, if a stream contained a single Filter node that the script needed to access, the Filter node can be found by using the following script:

```
stream = modeler.script.stream()
node = stream.findByType("filter", None)
...
```

Alternatively, if the ID of the node (as shown on the Annotations tab of the node dialog box) is known, the ID can be used to find the node, for example:

```
stream = modeler.script.stream()
node = stream.findByID("id32FJT71G2")  # the filter node ID
...
```

Setting properties

Nodes, streams, models, and outputs all have properties that can be accessed and, in most cases, set. Properties are typically used to modify the behavior or appearance of the object. The methods that are available for accessing and setting object properties are summarized in the following table.

Table 11. Methods for accessing and setting object properties

Method	Return type	Description
`p.getPropertyValue(propertyName)`	Object	Returns the value of the named property or `None` if no such property exists.
`p.setPropertyValue(propertyName, value)`	Not applicable	Sets the value of the named property.
`p.setPropertyValues(properties)`	Not applicable	Sets the values of the named properties. Each entry in the properties map consists of a key that represents the property name and the value that should be assigned to that property.
`p.getKeyedPropertyValue( propertyName, keyName)`	Object	Returns the value of the named property and associated key or `None` if no such property or key exists.
`p.setKeyedPropertyValue( propertyName, keyName, value)`	Not applicable	Sets the value of the named property and key.

For example, if you wanted to set the value of a Variable File node at the start of a stream, you can use the following script:

```
stream = modeler.script.stream()
node = stream.findByType("variablefile", None)
node.setPropertyValue("full_filename", "$CLEO/DEMOS/DRUG1n")
...
```

Alternatively, you might want to filter a field from a Filter node. In this case, the value is also keyed on the field name, for example:

```
stream = modeler.script.stream()
# Locate the filter node ...
node = stream.findByType("filter", None)
# ... and filter out the "Na" field
node.setKeyedPropertyValue("include", "Na", False)
```

Creating nodes and modifying streams

In some situations, you might want to add new nodes to existing streams. Adding nodes to existing streams typically involves the following tasks:

1. Creating the nodes.
2. Linking the nodes into the existing stream flow.

Creating nodes

Streams provide a number of ways of creating nodes. These methods are summarized in the following table.

Table 12. Methods for creating nodes

Method	Return type	Description
`s.create(nodeType, name)`	Node	Creates a node of the specified type and adds it to the specified stream.
`s.createAt(nodeType, name, x, y)`	Node	Creates a node of the specified type and adds it to the specified stream at the specified location. If either x < 0 or y < 0, the location is not set.

Table 12. Methods for creating nodes (continued)

Method	Return type	Description
`s.createModelApplier(modelOutput, name)`	Node	Creates a model applier node that is derived from the supplied model output object.

For example, to create a new Type node in a stream you can use the following script:

```
stream = modeler.script.stream()
# Create a new type node
node = stream.create("type", "My Type")
```

Linking and unlinking nodes

When a new node is created within a stream, it must be connected into a sequence of nodes before it can be used. Streams provide a number of methods for linking and unlinking nodes. These methods are summarized in the following table.

Table 13. Methods for linking and unlinking nodes

Method	Return type	Description
`s.link(source, target)`	Not applicable	Creates a new link between the source and the target nodes.
`s.link(source, targets)`	Not applicable	Creates new links between the source node and each target node in the supplied list.
`s.linkBetween(inserted, source, target)`	Not applicable	Connects a node between two other node instances (the source and target nodes) and sets the position of the inserted node to be between them. Any direct link between the source and target nodes is removed first.
`s.linkPath(path)`	Not applicable	Creates a new path between node instances. The first node is linked to the second, the second is linked to the third, and so on.
`s.unlink(source, target)`	Not applicable	Removes any direct link between the source and the target nodes.
`s.unlink(source, targets)`	Not applicable	Removes any direct links between the source node and each object in the targets list.
`s.unlinkPath(path)`	Not applicable	Removes any path that exists between node instances.
`s.disconnect(node)`	Not applicable	Removes any links between the supplied node and any other nodes in the specified stream.
`s.isValidLink(source, target)`	*boolean*	Returns `True` if it would be valid to create a link between the specified source and target nodes. This method checks that both objects belong to the specified stream, that the source node can supply a link and the target node can receive a link, and that creating such a link will not cause a circularity in the stream.

The example script that follows performs these five tasks:

1. Creates a Variable File input node, a Filter node, and a Table output node.
2. Connects the nodes together.
3. Sets the file name on the Variable File input node.
4. Filters the field "Drug" from the resulting output.
5. Executes the Table node.

```
stream = modeler.script.stream()
filenode = stream.createAt("variablefile", "My File Input ", 96, 64)
filternode = stream.createAt("filter", "Filter", 192, 64)
tablenode = stream.createAt("table", "Table", 288, 64)
stream.link(filenode, filternode)
stream.link(filternode, tablenode)
filenode.setPropertyValue("full_filename", "$CLEO_DEMOS/DRUG1n")
filternode.setKeyedPropertyValue("include", "Drug", False)
results = []
tablenode.run(results)
```

Importing, replacing, and deleting nodes

As well as creating and connecting nodes, it is often necessary to replace and delete nodes from the stream. The methods that are available for importing, replacing and deleting nodes are summarized in the following table.

Table 14. Methods for importing, replacing, and deleting nodes

Method	Return type	Description
`s.replace(originalNode, replacementNode, discardOriginal)`	Not applicable	Replaces the specified node from the specified stream. Both the original node and replacement node must be owned by the specified stream.
`s.insert(source, nodes, newIDs)`	List	Inserts copies of the nodes in the supplied list. It is assumed that all nodes in the supplied list are contained within the specified stream. The `newIDs` flag indicates whether new IDs should be generated for each node, or whether the existing ID should be copied and used. It is assumed that all nodes in a stream have a unique ID, so this flag must be set to `True` if the source stream is the same as the specified stream. The method returns the list of newly inserted nodes, where the order of the nodes is undefined (that is, the ordering is not necessarily the same as the order of the nodes in the input list).
`s.delete(node)`	Not applicable	Deletes the specified node from the specified stream. The node must be owned by the specified stream.
`s.deleteAll(nodes)`	Not applicable	Deletes all the specified nodes from the specified stream. All nodes in the collection must belong to the specified stream.
`s.clear()`	Not applicable	Deletes all nodes from the specified stream.

Traversing through nodes in a stream

A common requirement is to identify nodes that are either upstream or downstream of a particular node. The stream provides a number of methods that can be used to identify these nodes. These methods are summarized in the following table.

Table 15. Methods to identify upstream and downstream nodes

Method	Return type	Description
`s.iterator()`	Iterator	Returns an iterator over the node objects that are contained in the specified stream. If the stream is modified between calls of the `next()` function, the behavior of the iterator is undefined.
`s.predecessorAt(node, index)`	Node	Returns the specified immediate predecessor of the supplied node or `None` if the index is out of bounds.
`s.predecessorCount(node)`	*int*	Returns the number of immediate predecessors of the supplied node.
`s.predecessors(node)`	List	Returns the immediate predecessors of the supplied node.
`s.successorAt(node, index)`	Node	Returns the specified immediate successor of the supplied node or `None` if the index is out of bounds.
`s.successorCount(node)`	*int*	Returns the number of immediate successors of the supplied node.
`s.successors(node)`	List	Returns the immediate successors of the supplied node.

Getting information about nodes

Nodes fall into a number of different categories such as data import and export nodes, model building nodes, and other types of nodes. Every node provides a number of methods that can be used to find out information about the node.

The methods that can be used to obtain the ID, name, and label of a node are summarized in the following table.

Table 16. Methods to obtain the ID, name, and label of a node

Method	Return type	Description
`n.getLabel()`	*string*	Returns the display label of the specified node. The label is the value of the property `custom_name` only if that property is a non-empty string and the `use_custom_name` property is not set; otherwise, the label is the value of `getName()`.

Table 16. Methods to obtain the ID, name, and label of a node (continued)

Method	Return type	Description
`n.setLabel(label)`	Not applicable	Sets the display label of the specified node. If the new label is a non-empty string it is assigned to the property `custom_name`, and `False` is assigned to the property `use_custom_name` so that the specified label takes precedence; otherwise, an empty string is assigned to the property `custom_name` and `True` is assigned to the property `use_custom_name`.
`n.getName()`	*string*	Returns the name of the specified node.
`n.getID()`	*string*	Returns the ID of the specified node. A new ID is created each time a new node is created. The ID is persisted with the node when it is saved as part of a stream so that when the stream is opened, the node IDs are preserved. However, if a saved node is inserted into a stream, the inserted node is considered to be a new object and will be allocated a new ID.

Methods that can be used to obtain other information about a node are summarized in the following table.

Table 17. Methods for obtaining information about a node

Method	Return type	Description
`n.getTypeName()`	*string*	Returns the scripting name of this node. This is the same name that could be used to create a new instance of this node.
`n.isInitial()`	*Boolean*	Returns `True` if this is an *initial* node, that is one that occurs at the start of a stream.
`n.isInline()`	*Boolean*	Returns `True` if this is an *in-line* node, that is one that occurs mid-stream.
`n.isTerminal()`	*Boolean*	Returns `True` if this is a *terminal* node, that is one that occurs at the end of a stream.
`n.getXPosition()`	*int*	Returns the x position offset of the node in the stream.
`n.getYPosition()`	*int*	Returns the y position offset of the node in the stream.
`n.setXYPosition(x, y)`	Not applicable	Sets the position of the node in the stream.
`n.setPositionBetween(source, target)`	Not applicable	Sets the position of the node in the stream so that it is positioned between the supplied nodes.
`n.isCacheEnabled()`	*Boolean*	Returns `True` if the cache is enabled; returns `False` otherwise.

Table 17. Methods for obtaining information about a node (continued)

Method	**Return type**	**Description**
`n.setCacheEnabled(val)`	Not applicable	Enables or disables the cache for this object. If the cache is full and the caching becomes disabled, the cache is flushed.
`n.isCacheFull()`	*Boolean*	Returns `True` if the cache is full; returns `False` otherwise.
`n.flushCache()`	Not applicable	Flushes the cache of this node. Has no affect if the cache is not enabled or is not full.

Chapter 4. The Scripting API

Introduction to the Scripting API

The Scripting API provides access to a wide range of SPSS Modeler functionality. All the methods described so far are part of the API and can be accessed implicitly within the script without further imports. However, if you want to reference the API classes, you must import the API explicitly with the following statement:

```
import modeler.api
```

This import statement is required by many of the Scripting API examples.

Example: searching for nodes using a custom filter

The section "Finding nodes" on page 27 included an example of searching for a node in a stream using the type name of the node as the search criterion. In some situations, a more generic search is required and this can be implemented using the `NodeFilter` class and the stream `findAll()` method. This kind of search involves the following two steps:

1. Creating a new class that extends `NodeFilter` and that implements a custom version of the `accept()` method.
2. Calling the stream `findAll()` method with an instance of this new class. This returns all nodes that meet the criteria defined in the `accept()` method.

The following example shows how to search for nodes in a stream that have the node cache enabled. The returned list of nodes could be used to either flush or disable the caches of these nodes.

```
import modeler.api

class CacheFilter(modeler.api.NodeFilter):
 """A node filter for nodes with caching enabled"""
 def accept(this, node):
  return node.isCacheEnabled()

cachingnodes = modeler.script.stream().findAll(CacheFilter(), False)
```

Metadata: Information about data

Because nodes are connected together in a stream, information about the columns or fields that are available at each node is available. For example, in the Modeler UI, this allows you to select which fields to sort or aggregate by. This information is called the data model.

Scripts can also access the data model by looking at the fields coming into or out of a node. For some nodes, the input and output data models are the same, for example a Sort node simply reorders the records but doesn't change the data model. Some, such as the Derive node, can add new fields. Others, such as the Filter node can rename or remove fields.

In the following example, the script takes the standard IBM SPSS Modeler `druglearn.str` stream, and for each field, builds a model with one of the input fields dropped. It does this by:

1. Accessing the output data model from the Type node.
2. Looping through each field in the output data model.
3. Modifying the Filter node for each input field.
4. Changing the name of the model being built.
5. Running the model build node.

Note: Before running the script in the `druglean.str` stream, remember to set the scripting language to Python (the stream was created in a previous version of IBM SPSS Modeler so the stream scripting language is set to Legacy).

```
import modeler.api

stream = modeler.script.stream()
filternode = stream.findByType("filter", None)
typenode = stream.findByType("type", None)
c50node = stream.findByType("c50", None)
# Always use a custom model name
c50node.setPropertyValue("use_model_name", True)

lastRemoved = None
fields = typenode.getOutputDataModel()
for field in fields:
    # If this is the target field then ignore it
    if field.getModelingRole() == modeler.api.ModelingRole.OUT:
        continue

    # Re-enable the field that was most recently removed
    if lastRemoved != None:
        filternode.setKeyedPropertyValue("include", lastRemoved, True)

    # Remove the field
    lastRemoved = field.getColumnName()
    filternode.setKeyedPropertyValue("include", lastRemoved, False)

    # Set the name of the new model then run the build
    c50node.setPropertyValue("model_name", "Exclude " + lastRemoved)
    c50node.run([])
```

The DataModel object provides a number of methods for accessing information about the fields or columns within the data model. These methods are summarized in the following table.

Table 18. DataModel object methods for accessing information about fields or columns

Method	Return type	Description
`d.getColumnCount()`	*int*	Returns the number of columns in the data model.
`d.columnIterator()`	Iterator	Returns an iterator that returns each column in the "natural" insert order. The iterator returns instances of Column.
`d.nameIterator()`	Iterator	Returns an iterator that returns the name of each column in the "natural" insert order.
`d.contains(name)`	*Boolean*	Returns `True` if a column with the supplied name exists in this DataModel, `False` otherwise.
`d.getColumn(name)`	Column	Returns the column with the specified name.
`d.getColumnGroup(name)`	ColumnGroup	Returns the named column group or `None` if no such column group exists.
`d.getColumnGroupCount()`	*int*	Returns the number of column groups in this data model.
`d.columnGroupIterator()`	Iterator	Returns an iterator that returns each column group in turn.

Table 18. DataModel object methods for accessing information about fields or columns (continued)

Method	Return type	Description
`d.toArray()`	Column[]	Returns the data model as an array of columns. The columns are ordered in their "natural" insert order.

Each field (Column object) includes a number of methods for accessing information about the column. The table below shows a selection of these.

Table 19. Column object methods for accessing information about the column

Method	Return type	Description
`c.getColumnName()`	*string*	Returns the name of the column.
`c.getColumnLabel()`	*string*	Returns the label of the column or an empty string if there is no label associated with the column.
`c.getMeasureType()`	MeasureType	Returns the measure type for the column.
`c.getStorageType()`	StorageType	Returns the storage type for the column.
`c.isMeasureDiscrete()`	*Boolean*	Returns `True` if the column is discrete. Columns that are either a set or a flag are considered discrete.
`c.isModelOutputColumn()`	*Boolean*	Returns `True` if the column is a model output column.
`c.isStorageDatetime()`	*Boolean*	Returns `True` if the column's storage is a time, date or timestamp value.
`c.isStorageNumeric()`	*Boolean*	Returns `True` if the column's storage is an integer or a real number.
`c.isValidValue(value)`	*Boolean*	Returns `True` if the specified value is valid for this storage, and `valid` when the valid column values are known.
`c.getModelingRole()`	ModelingRole	Returns the modeling role for the column.
`c.getSetValues()`	Object[]	Returns an array of valid values for the column, or `None` if either the values are not known or the column is not a set.
`c.getValueLabel(value)`	*string*	Returns the label for the value in the column, or an empty string if there is no label associated with the value.
`c.getFalseFlag()`	Object	Returns the "false" indicator value for the column, or `None` if either the value is not known or the column is not a flag.
`c.getTrueFlag()`	Object	Returns the "true" indicator value for the column, or `None` if either the value is not known or the column is not a flag.

Table 19. Column object methods for accessing information about the column (continued)

Method	Return type	Description
`c.getLowerBound()`	Object	Returns the lower bound value for the values in the column, or `None` if either the value is not known or the column is not continuous.
`c.getUpperBound()`	Object	Returns the upper bound value for the values in the column, or `None` if either the value is not known or the column is not continuous.

Note that most of the methods that access information about a column have equivalent methods defined on the DataModel object itself. For example the two following statements are equivalent:

```
dataModel.getColumn("someName").getModelingRole()
dataModel.getModelingRole("someName")
```

Accessing Generated Objects

Executing a stream typically involves producing additional output objects. These additional objects might be a new model, or a piece of output that provides information to be used in subsequent executions.

In the example below, the `druglearn.str` stream is used again as the starting point for the stream. In this example, all nodes in the stream are executed and the results are stored in a list. The script then loops through the results, and any model outputs that result from the execution are saved as an IBM SPSS Modeler model (`.gm`) file, and the model is PMML exported.

```
import modeler.api

stream = modeler.script.stream()

# Set this to an existing folder on your system.
# Include a trailing directory separator
modelFolder = "C:/temp/models/"

# Execute the stream
models = []
stream.runAll(models)

# Save any models that were created
taskrunner = modeler.script.session().getTaskRunner()
for model in models:
    # If the stream execution built other outputs then ignore them
    if not(isinstance(model, modeler.api.ModelOutput)):
        continue

    label = model.getLabel()
    algorithm = model.getModelDetail().getAlgorithmName()

    # save each model...
    modelFile = modelFolder + label + algorithm + ".gm"
    taskrunner.saveModelToFile(model, modelFile)

    # ...and export each model PMML...
    modelFile = modelFolder + label + algorithm + ".xml"
    taskrunner.exportModelToFile(model, modelFile, modeler.api.FileFormat.XML)
```

The task runner class provides a convenient way running various common tasks. The methods that are available in this class are summarized in the following table.

Table 20. Methods of the task runner class for performing common tasks

Method	Return type	Description
`t.createStream(name, autoConnect, autoManage)`	Stream	Creates and returns a new stream. Note that code that must create streams privately without making them visible to the user should set the `autoManage` flag to `False`.
`t.exportDocumentToFile( documentOutput, filename, fileFormat)`	Not applicable	Exports the stream description to a file using the specified file format.
`t.exportModelToFile(modelOutput, filename, fileFormat)`	Not applicable	Exports the model to a file using the specified file format.
`t.exportStreamToFile(stream, filename, fileFormat)`	Not applicable	Exports the stream to a file using the specified file format.
`t.insertNodeFromFile(filename, diagram)`	Node	Reads and returns a node from the specified file, inserting it into the supplied diagram. Note that this can be used to read both Node and SuperNode objects.
`t.openDocumentFromFile(filename, autoManage)`	DocumentOutput	Reads and returns a document from the specified file.
`t.openModelFromFile(filename, autoManage)`	ModelOutput	Reads and returns a model from the specified file.
`t.openStreamFromFile(filename, autoManage)`	Stream	Reads and returns a stream from the specified file.
`t.saveDocumentToFile( documentOutput, filename)`	Not applicable	Saves the document to the specified file location.
`t.saveModelToFile(modelOutput, filename)`	Not applicable	Saves the model to the specified file location.
`t.saveStreamToFile(stream, filename)`	Not applicable	Saves the stream to the specified file location.

Handling Errors

The Python language provides error handling via the `try...except` code block. This can be used within scripts to trap exceptions and handle problems that would otherwise cause the script to terminate.

In the example script below, an attempt is made to retrieve a model from a IBM SPSS Collaboration and Deployment Services Repository. This operation can cause an exception to be thrown, for example, the repository login credentials might not have been set up correctly, or the repository path is wrong. In the script, this may cause a `ModelerException` to be thrown (all exceptions that are generated by IBM SPSS Modeler are derived from `modeler.api.ModelerException`).

```
import modeler.api

session = modeler.script.session()
try:
    repo = session.getRepository()
    m = repo.retrieveModel("/some-non-existent-path", None, None, True)
    # print goes to the Modeler UI script panel Debug tab
    print "Everything OK"
except modeler.api.ModelerException, e:
    print "An error occurred:", e.getMessage()
```

Note: Some scripting operations may cause standard Java exceptions to be thrown; these are not derived from `ModelerException`. In order to catch these exceptions, an additional except block can be used to catch all Java exceptions, for example:

```
import modeler.api

session = modeler.script.session()
try:
    repo = session.getRepository()
    m = repo.retrieveModel("/some-non-existent-path", None, None, True)
    # print goes to the Modeler UI script panel Debug tab
    print "Everything OK"
except modeler.api.ModelerException, e:
    print "An error occurred:", e.getMessage()
except java.lang.Exception, e:
    print "A Java exception occurred:", e.getMessage()
```

Stream, Session, and SuperNode Parameters

Parameters provide a useful way of passing values at runtime, rather than hard coding them directly in a script. Parameters and their values are defined in the same as way for streams, that is, as entries in the parameters table of a stream or SuperNode, or as parameters on the command line. The Stream and SuperNode classes implement a set of functions defined by the ParameterProvider object as shown in the following table. Session provides a `getParameters()` call which returns an object that defines those functions.

Table 21. Functions defined by the ParameterProvider object

Method	**Return type**	**Description**
`p.parameterIterator()`	Iterator	Returns an iterator of parameter names for this object.
`p.getParameterDefinition( parameterName)`	ParameterDefinition	Returns the parameter definition for the parameter with the specified name, or `None` if no such parameter exists in this provider. The result may be a snapshot of the definition at the time the method was called and need not reflect any subsequent modifications made to the parameter through this provider.
`p.getParameterLabel(parameterName)`	*string*	Returns the label of the named parameter, or `None` if no such parameter exists.
`p.setParameterLabel(parameterName, label)`	Not applicable	Sets the label of the named parameter.
`p.getParameterStorage( parameterName)`	ParameterStorage	Returns the storage of the named parameter, or `None` if no such parameter exists.
`p.setParameterStorage( parameterName, storage)`	Not applicable	Sets the storage of the named parameter.
`p.getParameterType(parameterName)`	ParameterType	Returns the type of the named parameter, or `None` if no such parameter exists.
`p.setParameterType(parameterName, type)`	Not applicable	Sets the type of the named parameter.
`p.getParameterValue(parameterName)`	Object	Returns the value of the named parameter, or `None` if no such parameter exists.

Table 21. Functions defined by the ParameterProvider object (continued)

Method	**Return type**	**Description**
`p.setParameterValue(parameterName, value)`	Not applicable	Sets the value of the named parameter.

In the following example, the script aggregates some Telco data to find which region has the lowest average income data. A stream parameter is then set with this region. That stream parameter is then used in a Select node to exclude that region from the data, before a churn model is built on the remainder.

The example is artificial because the script generates the Select node itself and could therefore have generated the correct value directly into the Select node expression. However, streams are typically pre-built, so setting parameters in this way provides a useful example.

The first part of the example script creates the stream parameter that will contain the region with the lowest average income. The script also creates the nodes in the aggregation branch and the model building branch, and connects them together.

```
import modeler.api

stream = modeler.script.stream()

# Initialize a stream parameter
stream.setParameterStorage("LowestRegion", modeler.api.ParameterStorage.INTEGER)

# First create the aggregation branch to compute the average income per region
statisticsimportnode = stream.createAt("statisticsimport", "SPSS File", 114, 142)
statisticsimportnode.setPropertyValue("full_filename", "$CLEO_DEMOS/telco.sav")
statisticsimportnode.setPropertyValue("use_field_format_for_storage", True)

aggregatenode = modeler.script.stream().createAt("aggregate", "Aggregate", 294, 142)
aggregatenode.setPropertyValue("keys", ["region"])
aggregatenode.setKeyedPropertyValue("aggregates", "income", ["Mean"])

tablenode = modeler.script.stream().createAt("table", "Table", 462, 142)

stream.link(statisticsimportnode, aggregatenode)
stream.link(aggregatenode, tablenode)

selectnode = stream.createAt("select", "Select", 210, 232)
selectnode.setPropertyValue("mode", "Discard")
# Reference the stream parameter in the selection
selectnode.setPropertyValue("condition", "'region' = '$P-LowestRegion'")

typenode = stream.createAt("type", "Type", 366, 232)
typenode.setKeyedPropertyValue("direction", "churn", "Target")

c50node = stream.createAt("c50", "C5.0", 534, 232)

stream.link(statisticsimportnode, selectnode)
stream.link(selectnode, typenode)
stream.link(typenode, c50node)
```

The example script creates the following stream.

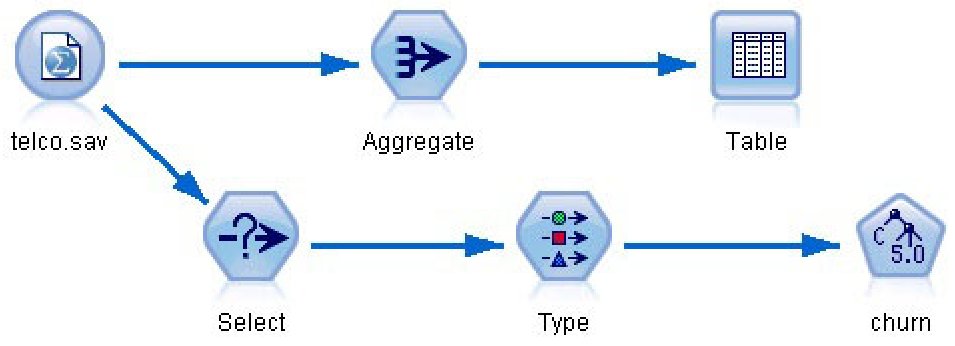

Figure 5. Stream that results from the example script

The following part of the example script executes the Table node at the end of the aggregation branch.

```
# First execute the table node
results = []
tablenode.run(results)
```

The following part of the example script accesses the table output that was generated by the execution of the Table node. The script then iterates through rows in the table, looking for the region with the lowest average income.

```
# Running the table node should produce a single table as output
table = results[0]

# table output contains a RowSet so we can access values as rows and columns
rowset = table.getRowSet()
min_income = 1000000.0
min_region = None

# From the way the aggregate node is defined, the first column
# contains the region and the second contains the average income
row = 0
rowcount = rowset.getRowCount()
while row < rowcount:
    if rowset.getValueAt(row, 1) < min_income:
        min_income = rowset.getValueAt(row, 1)
        min_region = rowset.getValueAt(row, 0)
    row += 1
```

The following part of the script uses the region with the lowest average income to set the "LowestRegion" stream parameter that was created earlier. The script then runs the model builder with the specified region excluded from the training data.

```
# Check that a value was assigned
if min_region != None:
    stream.setParameterValue("LowestRegion", min_region)
else:
    stream.setParameterValue("LowestRegion", -1)

# Finally run the model builder with the selection criteria
c50node.run([])
```

The complete example script is shown below.

```
import modeler.api

stream = modeler.script.stream()
```

```
# Create a stream parameter
stream.setParameterStorage("LowestRegion", modeler.api.ParameterStorage.INTEGER)

# First create the aggregation branch to compute the average income per region
statisticsimportnode = stream.createAt("statisticsimport", "SPSS File", 114, 142)
statisticsimportnode.setPropertyValue("full_filename", "$CLEO_DEMOS/telco.sav")
statisticsimportnode.setPropertyValue("use_field_format_for_storage", True)

aggregatenode = modeler.script.stream().createAt("aggregate", "Aggregate", 294, 142)
aggregatenode.setPropertyValue("keys", ["region"])
aggregatenode.setKeyedPropertyValue("aggregates", "income", ["Mean"])

tablenode = modeler.script.stream().createAt("table", "Table", 462, 142)

stream.link(statisticsimportnode, aggregatenode)
stream.link(aggregatenode, tablenode)

selectnode = stream.createAt("select", "Select", 210, 232)
selectnode.setPropertyValue("mode", "Discard")
# Reference the stream parameter in the selection
selectnode.setPropertyValue("condition", "'region' = '$P-LowestRegion'")

typenode = stream.createAt("type", "Type", 366, 232)
typenode.setKeyedPropertyValue("direction", "churn", "Target")

c50node = stream.createAt("c50", "C5.0", 534, 232)

stream.link(statisticsimportnode, selectnode)
stream.link(selectnode, typenode)
stream.link(typenode, c50node)


# First execute the table node
results = []
tablenode.run(results)

# Running the table node should produce a single table as output
table = results[0]

# table output contains a RowSet so we can access values as rows and columns
rowset = table.getRowSet()
min_income = 1000000.0
min_region = None

# From the way the aggregate node is defined, the first column
# contains the region and the second contains the average income
row = 0
rowcount = rowset.getRowCount()
while row < rowcount:
    if rowset.getValueAt(row, 1) < min_income:
        min_income = rowset.getValueAt(row, 1)
        min_region = rowset.getValueAt(row, 0)
    row += 1

# Check that a value was assigned
if min_region != None:
    stream.setParameterValue("LowestRegion", min_region)
else:
    stream.setParameterValue("LowestRegion", -1)

# Finally run the model builder with the selection criteria
c50node.run([])
```

Global Values

Global values are used to compute various summary statistics for specified fields. These summary values can be accessed anywhere within the stream. Global values are similar to stream parameters in that they are accessed by name through the stream. They are different from stream parameters in that the associated values are updated automatically when a Set Globals node is run, rather than being assigned by scripting or from the command line. The global values for a stream are accessed by calling the stream's `getGlobalValues()` method.

The GlobalValues object defines the functions that are shown in the following table.

Table 22. Functions that are defined by the GlobalValues object

Method	Return type	Description
`g.fieldNameIterator()`	Iterator	Returns an iterator for each field name with at least one global value.
`g.getValue(type, fieldName)`	Object	Returns the global value for the specified type and field name, or `None` if no value can be located. The returned value is generally expected to be a number, although future functionality may return different value types.
`g.getValues(fieldName)`	Map	Returns a map containing the known entries for the specified field name, or `None` if there are no existing entries for the field.

`GlobalValues.Type` defines the type of summary statistics that are available. The following summary statistics are available:

- `MAX`: the maximum value of the field.
- `MEAN`: the mean value of the field.
- `MIN`: the minimum value of the field.
- `STDDEV`: the standard deviation of the field.
- `SUM`: the sum of the values in the field.

For example, the following script accesses the mean value of the "income" field, which is computed by a Set Globals node:

```
import modeler.api

globals = modeler.script.stream().getGlobalValues()
mean_income = globals.getValue(modeler.api.GlobalValues.Type.MEAN, "income")
```

Working with Multiple Streams: Standalone Scripts

To work with multiple streams, a standalone script must be used. The standalone script can be edited and run within the IBM SPSS Modeler UI or passed as a command line parameter in batch mode.

The following standalone script opens two streams. One of these streams builds a model, while the second stream plots the distribution of the predicted values.

```
# Change to the appropriate location for your system
demosDir = "C:/Program Files/IBM/SPSS/Modeler/16/DEMOS/streams/"

session = modeler.script.session()
tasks = session.getTaskRunner()
```

```
# Open the model build stream, locate the C5.0 node and run it
buildstream = tasks.openStreamFromFile(demosDir + "druglearn.str", True)
c50node = buildstream.findByType("c50", None)
results = []
c50node.run(results)

# Now open the plot stream, find the Na_to_K derive and the histogram
plotstream = tasks.openStreamFromFile(demosDir + "drugplot.str", True)
derivenode = plotstream.findByType("derive", None)
histogramnode = plotstream.findByType("histogram", None)

# Create a model applier node, insert it between the derive and histogram nodes
# then run the histgram
applyc50 = plotstream.createModelApplier(results[0], results[0].getName())
applyc50.setPositionBetween(derivenode, histogramnode)
plotstream.linkBetween(applyc50, derivenode, histogramnode)
histogramnode.setPropertyValue("color_field", "$C-Drug")
histogramnode.run([])

# Finally, tidy up the streams
buildstream.close()
plotstream.close()
```

Chapter 5. Scripting Tips

This section provides an overview of tips and techniques for using scripts, including modifying stream execution and using an encoded password in a script.

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 Execution tab of the stream properties dialog box:

1. Begin with an empty script.
2. Click the **Append default script** button on the toolbar to add the default stream script.
3. Change the order of statements in the default stream script to the order in which you want statements to be executed.

Working with models

If automatic model replacement is enabled in IBM SPSS Modeler, and a model builder node is executed through the IBM SPSS Modeler user interface, an existing model nugget that is linked to the model builder node is replaced with the new model nugget. If the model builder node is executed using a script, the existing linked model nugget is not replaced. To replace the existing model nugget, you must explicitly specify the replacement of the nugget in your script.

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 `database` and `databaseexport` stores the encoded password.

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

   Encode Password...
2. Specify a password in the Password text box.
3. Click **Encode** to generate a random encoding of your password.
4. Click the Copy button to copy the encoded password to the Clipboard.
5. 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.

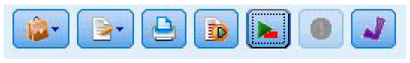

Figure 6. Stream script toolbar icons

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.py -execute
```

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

Specifying File Paths

When specifying file paths to directories and files, you can use either a single forward slash (/) or a double backslash (\\) as the directory separator, for example

```
c:/demos/druglearn.str
```

or

```
c:\\demos\\druglearn.str
```

Compatibility with Previous Releases

Legacy scripts that were created in previous releases of IBM SPSS Modeler should generally work unchanged in the current release. For the scripts to work, **Legacy** must be selected on the stream script tab in the Stream Properties dialog box, or the Standalone Script dialog box. 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.

Python scripts created in the current release will 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.

Chapter 6. Command Line Arguments

Invoking the Software

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

1. On a computer where IBM SPSS Modeler is installed, open a DOS, or command-prompt, window.
2. To launch the IBM 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 23. Types of command line arguments.

Argument type	**Where described**
System arguments	See the topic "System Arguments" on page 50 for more information.
Parameter arguments	See the topic "Parameter Arguments" on page 51 for more information.
Server connection arguments	See the topic "Server Connection Arguments" on page 51 for more information.
IBM SPSS Collaboration and Deployment Services Repository connection arguments	See the topic "IBM SPSS Collaboration and Deployment Services Repository Connection Arguments" on page 52 for more information.

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 IBM SPSS Modeler scripts in this manner, using the `-script` flag.

Debugging Command Line Arguments

To debug a command line, use the `modelerclient` command to launch IBM SPSS Modeler with the desired arguments. This enables 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).

System Arguments

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

Table 24. System arguments

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. See the topic "Combining Multiple Arguments" on page 53 for more information.
`-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. If the script file suffix is `.py`, the file is assumed to be a Python script, otherwise it is assumed to be a legacy script.
`-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).
`-scriptlang` <python | legacy>	This can be used to specify the scripting language associated with `-script` option, regardless of the script file suffix. Example `client -scriptlang python -script scores.txt -execute` This runs the supplied script file using Python even though the file suffix was not `.py`.

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
```

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**
- **Slot parameters**, also referred to as **node properties**. These parameters are used to modify the settings of nodes in the stream. See the topic "Node Properties Overview" on page 56 for more information.
- **Command line parameters**, used to alter the invocation of IBM 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:database.datasource={"ORA 10gR2", user1, mypsw, true}
```

The format is the same as that of the `datasource` parameter of the `database` node property. See the topic "database Node Properties" on page 64 for more information.

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 IBM 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_*`). See the topic "IBM SPSS Collaboration and Deployment Services Repository Connection Arguments" for more information.

Table 25. Server connection arguments.

Argument	**Behavior/Description**
`-server`	Runs IBM 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. See the topic "IBM SPSS Collaboration and Deployment Services Repository Connection Arguments" for more information.
`-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 IBM 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.

Table 26. IBM SPSS Collaboration and Deployment Services Repository connection arguments

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 Repository accepts connections (typically, 8080 by default).
`-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 7. 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 “Setting properties” on page 27.
- 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 IBM 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 2, “The Scripting Language,” on page 13.

Abbreviations

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

Table 27. Standard abbreviations used throughout the syntax.

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 IBM 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.

See the topics “Stream, Session, and SuperNode Parameters” on page 40 and “System Arguments” on page 50 for further scripting examples.

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`. 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.

Table 28. Common node properties.

Property name	Data type	Property description
`use_custom_name`	*boolean*	
`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
`cache_enabled`	*boolean*	

`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 `userinput` or `filter`.

SuperNode-specific properties are discussed separately, as with all other nodes. See the topic Chapter 19, “SuperNode Properties,” on page 231 for more information.

Chapter 8. Stream Properties

A variety of stream properties can be controlled by scripting.

The script can access the current stream using the `stream()` function in the `modeler.script` module, for example:

```
mystream = modeler.script.stream()
```

To reference stream properties, you must use a special stream variable, denoted with a ^ preceding the stream.

The nodes property is used to refer to the nodes in the current stream.

Stream properties are described in the following table.

Table 29. Stream properties.

Property name	Data type	Property description
`execute_method`	`Normal` `Script`	

`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*	

Table 29. Stream properties (continued).

Property name	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"	
time_rollover	*boolean*	
import_datetime_as_string	*boolean*	
decimal_places	*number*	
decimal_symbol	Default Period Comma	
angles_in_radians	*boolean*	
use_max_set_size	*boolean*	
max_set_size	*number*	
ruleset_evaluation	Voting FirstHit	
refresh_source_nodes	*boolean*	Use to refresh source nodes automatically upon stream execution.
script	*string*	
script_language	Python Legacy	Sets the scripting language for the stream script.
annotation	*string*	
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*	

Table 29. Stream properties (continued).

Property name	Data type	Property description
`record_count_suppress_input`	*boolean*	
`record_count_feedback_interval`	*integer*	
`use_stream_auto_create_node_settings`	*boolean*	If true, then stream-specific settings are used, otherwise user preferences are used.
`create_model_applier_for_new_models`	*boolean*	If true, when a model builder creates a new model, and it has no active update links, a new model applier is added.
`create_model_applier_update_links`	`createEnabled` `createDisabled` `doNotCreate`	Defines the type of link created when a model applier node is added automatically.
`create_source_node_from_builders`	*boolean*	If true, when a source builder creates a new source output, and it has no active update links, a new source node is added.
`create_source_node_update_links`	`createEnabled` `createDisabled` `doNotCreate`	Defines the type of link created when a source node is added automatically.

Chapter 9. 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.

Table 30. Source node common properties.

Property name	Data type	Property description

`direction`	`Input`
	`Target`
	`Both`
	`None`
	`Partition`
	`Split`
	`Frequency`
	`RecordID`
		Keyed property for field roles.
		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.

`storage`	`Unknown`
	`String`
	`Integer`
	`Real`
	`Time`
	`Date`
	`Timestamp`
		Read-only keyed property for field storage type.

`check`	`None`
	`Nullify`
	`Coerce`
	`Discard`
	`Warn`
	`Abort`
		Keyed property for field type and range checking.

`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*.

`value_mode`	`Read`
	`Pass`
	`Read+`
	`Current`
	`Specify`
		Determines how values are set for a field on the next data pass.
		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. This setting can be overridden for specific fields by using the `value_mode` property.

Table 30. Source node common properties (continued).

Property name	Data type	Property description
extend_values	*boolean*	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.
value_labels	*string*	Used to specify a value label. Note that values must be specified first.
enable_missing	*boolean*	When set to *T*, activates tracking of missing values for the field.
missing_values	[*value value ...*]	Specifies data values that denote missing data.
range_missing	*boolean*	When this property is set to *T*, 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	*boolean*	When this property is set to *T*, nulls (undefined values that are displayed as $null$ in the software) are considered missing values.
whitespace_missing	*boolean*	When this property is set to *T*, values containing only white space (spaces, tabs, and new lines) are considered missing values.
description	*string*	Used to specify a field label or description.
default_include	*boolean*	Keyed property to specify whether the default behavior is to pass or filter fields:
include	*boolean*	Keyed property used to determine whether individual fields are included or filtered:
new_name	*string*	

asimport Node Properties

The Analytic Server source enables you to run a stream on Hadoop Distributed File System (HDFS).

Table 31. asimport node properties.

asimport node properties	Data type	Property description
data_source	*string*	The name of the data source.
host	*string*	The name of the Analytic Server host.
port	*integer*	The port on which the Analytic Server is listening.
tenant	*string*	In a multi-tenant environment, the name of the tenant to which you belong. In a single-tenant environment, this defaults to **ibm**.
set_credentials	*boolean*	If user authentication on the Analytic Server is the same as on SPSS Modeler server, set this to **false**. Otherwise, set to **true**.

Table 31. asimport node properties (continued).

`asimport` node properties	Data type	Property description
`user_name`	*string*	The user name for logging in to the Analytic Server. Only needed if `set_credentials` is true.
`password`	*string*	The password for logging in to the Analytic Server. Only needed if `set_credentials` is true.

cognosimport Node Properties

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

Table 32. cognosimport node properties.

`cognosimport` node properties	Data type	Property description
`mode`	`Data` `Report`	Specifies whether to import Cognos BI data (default) or reports.
`cognos_connection`	*{"string",boolean, "string","string", "string"}*	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*]
`cognos_filters`	*field*	The name of one or more filters to apply before importing data.
`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*"}]

Table 32. cognosimport node properties (continued).

`cognosimport` node properties	Data type	Property description
`cognos_report_directory`	*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.
`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*"}]

tm1import Node Properties

The IBM Cognos TM1 source node imports data from Cognos TM1 databases.

Table 33. tm1import node properties.

`tm1import` node properties	Data type	Property description
`pm_host`	*string*	The host name. For example: `set TM1_import.pm_host= 'http://9.191.86.82:9510/pmhub/pm'`
`tm1_connection`	*{"field","field", ... ,"field"}*	A list property containing the connection details for the TM1 server. The format is: `{ "TM1_Server_Name" "tm1_ username" "tm1_ password"}` `: set TM1_import.tm1_connection= ['Planning Sample' admin apple]`
`selected_view`	*{"field" "field"}*	A list property containing the details of the selected TM1 cube and the name of the cube view from where data will be imported into SPSS. For example: `: set TM1_import.selected_view= {'plan_BudgetPlan' 'Goal Input'}`

database Node 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.

Table 34. database node properties.

`database` node 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. See the topic "Generating an Encoded Password" on page 47 for more information. This property is read-only during execution.
`tablename`	*string*	Name of the table you want to access.

`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 one or more spaces, periods (also known as a "full stop"), or underscores, you can use the "backslash double quote" format to treat it as string. For example: `\"db2v9.7.6_linux\"` or: `"\"TDATA 131\""`.

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

Table 35. database node properties - datasource specific.

`database` node 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.

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.

datacollectionimport Node Properties

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

Figure 7. Dimensions Data Import node

Table 36. datacollectionimport node properties.

`datacollectionimport` node properties	Data type	Property description

`metadata_name`	*string*	The name of the MDSC. The special value `DimensionsMDD` indicates that the standard IBM SPSS 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.

Table 36. datacollectionimport node properties (continued).

datacollectionimport node properties	Data type	Property description
casedata_udl_string	*string*	When casedata_source_type is *UDL*, specifies the OLD-DB connection string for the data source containing the case data.
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 IBM SPSS 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	*boolean*	
import_sourcefile_variables	*boolean*	
import_multi_response	MultipleFlags Single	

excelimport Node Properties

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

Table 37. excelimport node properties.

excelimport node 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.

evimport Node 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.

Table 38. evimport node properties.

evimport node properties	**Data type**	**Property description**
connection	*list*	Structured property--list of parameters making up an Enterprise View connection.
tablename	*string*	The name of a table in the Application View.

fixedfile Node 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.

Table 39. fixedfile node properties.

fixedfile node properties	Data type	Property description
record_len	*number*	Specifies the number of characters in each record.
line_oriented	*boolean*	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.
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	*boolean*	Specifies whether dates or times are automatically identified in the source data.
lines_to_scan	*number*	
fields	*list*	Structured property.
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	*boolean*	
custom_storage	Unknown String Integer Real Time Date Timestamp	

Table 39. fixedfile node properties (continued).

`fixedfile` node properties	Data type	Property description

`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`
		This property is applicable only if a custom storage has been specified.

`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.

`custom_decimal_symbol`	*field*	Applicable only if a custom storage has been specified.

`encoding`	`StreamDefault`
	`SystemDefault`
	`"UTF-8"`
		Specifies the text-encoding method.

sasimport Node Properties

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

Table 40. sasimport node properties.

`sasimport` node 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`	*boolean*	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.

simgen Node Properties

The Simulation Generate node provides an easy way to generate simulated data—either from scratch using user specified statistical distributions or automatically using the distributions obtained from running a Simulation Fitting node on existing historical data. This is useful when you want to evaluate the outcome of a predictive model in the presence of uncertainty in the model inputs.

Table 41. simgen node properties.

`simgen` node properties	Data type	Property description
`fields`	Structured property	
`correlations`	Structured property	
`max_cases`	*integer*	Minimum value is 1000, maximum value is 2,147,483,647
`create_iteration_field`	*boolean*	
`iteration_field_name`	*string*	
`replicate_results`	*boolean*	
`random_seed`	*integer*	
`overwrite_when_refitting`	*boolean*	
`parameter_xml`	*string*	Returns the parameter Xml as a string

Table 41. simgen node properties (continued).

simgen node properties	Data type	Property description
distribution	Bernoulli Beta Binomial Categorical Exponential Fixed Gamma Lognormal NegativeBinomialFailures NegativeBinomialTrials Normal Poisson Range Triangular Uniform Weibull	
bernoulli_prob	*number*	0 ≤ bernoulli_prob ≤ 1
beta_shape1	*number*	Must be ≥ 0
beta_shape2	*number*	Must be ≥ 0
beta_min	*number*	Optional. Must be less than beta_max.
beta_max	*number*	Optional. Must be greater than beta_min.
binomial_n	*integer*	Must be > 0
binomial_prob	*number*	0 ≤ binomial_prob ≤ 1
binomial_min	*number*	Optional. Must be less than binomial_max.
binomial_max	*number*	Optional. Must be greater than binomial_min.
exponential_scale	*number*	Must be > 0
exponential_min	*number*	Optional. Must be less than exponential_max.
exponential_max	*number*	Optional. Must be greater than exponential_min.
fixed_value	*string*	
gamma_shape	*number*	Must be ≥ 0
gamma_scale	*number*	Must be ≥ 0
gamma_min	*number*	Optional. Must be less than gamma_max.
gamma_max	*number*	Optional. Must be greater than gamma_min.
lognormal_shape1	*number*	Must be ≥ 0
lognormal_shape2	*number*	Must be ≥ 0
lognormal_min	*number*	Optional. Must be less than lognormal_max.
lognormal_max	*number*	Optional. Must be greater than lognormal_min.
negative_bin_failures_threshold	*number*	Must be ≥ 0
negative_bin_failures_prob	*number*	0 ≤ negative_bin_failures_prob ≤ 1

Table 41. simgen node properties (continued).

`simgen` node properties	Data type	Property description
`negative_bin_failures_min`	*number*	Optional. Must be less than `negative_bin_failures_max`.
`negative_bin_failures_max`	*number*	Optional. Must be greater than `negative_bin_failures_min`.
`negative_bin_trials_threshold`	*number*	Must be ≥ 0
`negative_bin_trials_prob`	*number*	0 ≤ `negative_bin_trials_prob` ≤ 1
`negative_bin_trials_min`	*number*	Optional. Must be less than `negative_bin_trials_max`.
`negative_bin_trials_max`	*number*	Optional. Must be less than `negative_bin_trials_min`.
`normal_mean`	*number*	
`normal_sd`	*number*	Must be > 0
`normal_min`	*number*	Optional. Must be less than `normal_max`.
`normal_max`	*number*	Optional. Must be greater than `normal_min`.
`poisson_mean`	*number*	Must be ≥ 0
`poisson_min`	*number*	Optional. Must be less than `poisson_max`.
`poisson_max`	*number*	Optional. Must be greater than `poisson_min`.
`triangular_mode`	*number*	`triangular_min` ≤ `triangular_mode` ≤ `triangular_max`
`triangular_min`	*number*	Must be less than `triangular_mode`
`triangular_max`	*number*	Must be greater than `triangular_mode`
`uniform_min`	*number*	Must be less than `uniform_max`
`uniform_max`	*number*	Must be greater than `uniform_min`
`weibull_rate`	*number*	Must be ≥ 0
`weibull_scale`	*number*	Must be ≥ 0
`weibull_location`	*number*	Must be ≥ 0
`weibull_min`	*number*	Optional. Must be less than `weibull_max`.
`weibull_max`	*number*	Optional. Must be greater than `weibull_min`.

Correlation can be any number between +1 and -1. You can specify as many or as few correlations as you like. Any unspecified correlations are set to zero. If any fields are unknown, the correlation value should be set on the correlation matrix (or table) and is shown in red text. When there are unknown fields, it is not possible to execute the node.

statisticsimport Node Properties

The IBM SPSS 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.

The properties for this node are described under "statisticsimport Node Properties" on page 227.

userinput Node 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.

Table 42. userinput node properties.

`userinput` node properties	Data type	Property description
`data`		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. **Note:** The values that are entered for this property must be strings, not numbers. For example, the numbers 1, 2, 3 and 4 must be entered as `"1 2 3 4"`.
`names`		Structured slot that sets or returns a list of field names generated by the node.

`custom_storage`	`Unknown`
	`String`
	`Integer`
	`Real`
	`Time`
	`Date`
	`Timestamp`
		Keyed slot that sets or returns the storage for a field.

`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* `data` *and should no longer be used.*

variablefile Node 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.

Table 43. variablefile node properties.

`variablefile` node properties	Data type	Property description
`skip_header`	*number*	Specifies the number of characters to ignore at the beginning of the first record.
`num_fields_auto`	*boolean*	Determines the number of fields in each record automatically. Records must be terminated with a new-line character.
`num_fields`	*number*	Manually specifies the number of fields in each record.
`delimit_space`	*boolean*	Specifies the character used to delimit field boundaries in the file.
`delimit_tab`	*boolean*	
`delimit_new_line`	*boolean*	
`delimit_non_printing`	*boolean*	
`delimit_comma`	*boolean*	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`	*boolean*	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`	*boolean*	Treats multiple adjacent blank delimiter characters as a single delimiter.
`read_field_names`	*boolean*	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.
`invalid_char_replacement`	*string*	
`break_case_by_newline`	*boolean*	Specifies that the line delimiter is the newline character.
`lines_to_scan`	*number*	Specifies how many lines to scan for specified data types.

Table 43. variablefile node properties (continued).

`variablefile` node properties	Data type	Property description
`auto_recognize_datetime`	*boolean*	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`	*boolean*	

`custom_storage`	`Unknown`
	`String`
	`Integer`
	`Real`
	`Time`
	`Date`
	`Timestamp`	

`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.

Table 43. variablefile node properties (continued).

variablefile node 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.
custom_decimal_symbol	*field*	Applicable only if a custom storage has been specified.
encoding	StreamDefault SystemDefault "UTF-8"	Specifies the text-encoding method.

xmlimport Node 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.

Table 44. xmlimport node properties.

xmlimport node properties	**Data type**	**Property description**
read	single directory	Reads a single data file (default), or all XML files in a directory.
recurse	*boolean*	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.
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.

Table 44. xmlimport node properties (continued).

xmlimport node properties	Data type	Property description
`fields`		List of items (elements and attributes) to import. Each item in the list is an XPath expression.

Chapter 10. Record Operations Node Properties

append Node Properties

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

Table 45. append node properties.

`append` node 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`	*boolean*	Enables case sensitivity when matching field names.
`include_fields_from`	`Main` `All`	
`create_tag_field`	*boolean*	
`tag_field_name`	*string*	

aggregate Node Properties

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

Table 46. aggregate node properties.

`aggregate` node 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`	*boolean*	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.
`extension`	*string*	Specify a prefix or suffix for duplicate aggregated fields (sample below).
`add_as`	`Suffix` `Prefix`	

Table 46. aggregate node properties (continued).

aggregate node properties	Data type	Property description
`inc_record_count`	*boolean*	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.

balance Node 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.

Table 47. balance node properties.

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

The `directives` 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.

derive_stb Node Properties

The Space-Time-Boxes node derives Space-Time-Boxes from latitude, longitude and timestamp fields. You can also identify frequent Space-Time-Boxes as hangouts.

Table 48. derive_stb node properties.

derive_stb node properties	Data type	Property description
`mode`	`IndividualRecords` `Hangouts`	
`latitude_field`	*field*	
`longitude_field`	*field*	
`timestamp_field`	*field*	
`hangout_density`	*density*	A single density. See `densities` for valid density values.

Table 48. derive_stb node properties (continued).

`derive_stb` **node properties**	**Data type**	**Property description**

`densities`	[*density*,*density*,..., *density*]	Each density is a string, for example `STB_GH8_1DAY`.
		Note: There are limits to which densities are valid. For the geohash, values from `GH1` to `GH15` can be used. For the temporal part, the following values can be used:
		`EVER`
		`1YEAR`
		`1MONTH`
		`1DAY`
		`12HOURS`
		`8HOURS`
		`6HOURS`
		`4HOURS`
		`3HOURS`
		`2HOURS`
		`1HOUR`
		`30MINS`
		`15MINS`
		`10MINS`
		`5MINS`
		`2MINS`
		`1MIN`
		`30SECS`
		`15SECS`
		`10SECS`
		`5SECS`
		`2SECS`
		`1SEC`

`id_field`	*field*	

`qualifying_duration`	`1DAY`
	`12HOURS`
	`8HOURS`
	`6HOURS`
	`4HOURS`
	`3HOURS`
	`2Hours`
	`1HOUR`
	`30MIN`
	`15MIN`
	`10MIN`
	`5MIN`
	`2MIN`
	`1MIN`
	`30SECS`
	`15SECS`
	`10SECS`
	`5SECS`
	`2SECS`
	`1SECS`
		Must be a string.

`min_events`	*integer*	Minimum valid integer value is 2.
`qualifying_pct`	*integer*	Must be in the range of 1 and 100.
`add_extension_as`	`Prefix` `Suffix`	
`name_extension`	*string*	

distinct Node 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.

Table 49. distinct node properties.

`distinct` node 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.
`grouping_fields`	[*field field field*]	Lists fields used to determine whether records are identical. **Note:** This property is deprecated from IBM SPSS Modeler 16 onwards.
`composite_value`	Structured slot	
`composite_values`	Structured slot	
`inc_record_count`	*boolean*	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.
`sort_keys`	Structured slot.	**Note:** This property is deprecated from IBM SPSS Modeler 16 onwards.
`default_ascending`	*boolean*	
`low_distinct_key_count`	*boolean*	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`	*boolean*	Specifies that all records with the same key values are grouped together in the input.
`disable_sql_generation`	*boolean*	

merge Node 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.

Table 50. merge node properties.

`merge` node 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.

Table 50. merge node properties (continued).

merge node properties	Data type	Property description
key_fields	[*field field field*]	
common_keys	*boolean*	
join	Inner FullOuter PartialOuter Anti	
outer_join_tag.n	*boolean*	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	*boolean*	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 (boolean versus string) because only one input dataset can be specified.
use_existing_sort_keys	*boolean*	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.

rfmaggregate Node 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.

Table 51. rfmaggregate node properties.

rfmaggregate node 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	*boolean*	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.

Table 51. rfmaggregate node properties (continued).

`rfmaggregate` node properties	Data type	Property description
add_as	Suffix Prefix	Specify if the `extension` should be added as a suffix or a prefix.
discard_low_value_records	*boolean*	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	*boolean*	Enable use of either the `specify_transaction_date` or `transaction_within_last` settings.
specify_transaction_date	*boolean*	
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	*boolean*	Displays the date of the second most recent transaction for each customer.
save_r3	*boolean*	Only available if `save_r2` is selected. Displays the date of the third most recent transaction for each customer.

Rprocess Node Properties

The R Process node enables you to take data from an IBM(r) SPSS(r) Modeler stream and modify the data using your own custom R script. After the data is modified it is returned to the stream.

Table 52. Rprocess node properties.

`Rprocess` node properties	Data type	Property description
syntax	*string*	
convert_flags	StringsAndDoubles LogicalValues	
convert_datetime	*boolean*	
convert_datetime_class	POSIXct POSIXlt	
convert_missing	*boolean*	

sample Node 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.

Table 53. sample node properties.

sample node 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.
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	*boolean*	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	*boolean*	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	*boolean*	
minimum_stratum_size	*integer*	This option only applies when a Complex sample is taken with Sample units=Proportions.
use_max_stratum_size	*boolean*	
maximum_stratum_size	*integer*	This option only applies when a Complex sample is taken with Sample units=Proportions.

Table 53. sample node properties (continued).

sample node properties	Data type	Property description
clusters	*field*	
stratify_by	*[field1 ... fieldN]*	
specify_input_weight	*boolean*	
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*	

select Node 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.

Table 54. select node properties.

select node properties	Data type	Property description
mode	Include Discard	Specifies whether to include or discard selected records.
condition	*string*	Condition for including or discarding records.

sort Node Properties

The Sort node sorts records into ascending or descending order based on the values of one or more fields.

Table 55. sort node properties.

sort node properties	Data type	Property description
keys	[{*string* Ascending} \ {*string* Descending}]	Specifies the fields you want to sort against. If no direction is specified, the default is used.
default_ascending	*boolean*	Specifies the default sort order.
use_existing_keys	*boolean*	Specifies whether sorting is optimized by using the previous sort order for fields that are already sorted.

Table 55. sort node properties (continued).

sort node properties	Data type	Property description
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.

streamingts Node Properties

The Streaming TS node builds and scores time series models in one step, without the need for a Time Intervals node.

Table 56. streamingts node properties.

streamingts node properties	Data type	Property description
custom_fields	*boolean*	If custom_fields=false, the settings from an upstream Type node are used. If custom_fields=true, targets and inputs must be specified.
targets	[*field1...fieldN*]	
inputs	[*field1...fieldN*]	
method	ExpertModeler Exsmooth Arima	
calculate_conf	*boolean*	
conf_limit_pct	*real*	
use_time_intervals_node	*boolean*	If use_time_intervals_node=true, then the settings from an upstream Time Intervals node are used. Otherwise, interval_offset_position, interval_offset, and interval_type must be specified.
interval_offset_position	LastObservation LastRecord	LastObservation refers to **Last valid observation**. LastRecord refers to **Count back from last record**.
interval_offset	*number*	
interval_type	Periods Years Quarters Months WeeksNonPeriodic DaysNonPeriodic HoursNonPeriodic MinutesNonPeriodic SecondsNonPeriodic	
events	*fields*	
expert_modeler_method	AllModels Exsmooth Arima	
consider_seasonal	*boolean*	
detect_outliers	*boolean*	
expert_outlier_additive	*boolean*	

Table 56. streamingts node properties (continued).

streamingts node properties	**Data type**	**Property description**
expert_outlier_level_shift	*boolean*	
expert_outlier_innovational	*boolean*	
expert_outlier_transient	*boolean*	
expert_outlier_seasonal_additive	*boolean*	
expert_outlier_local_trend	*boolean*	
expert_outlier_additive_patch	*boolean*	
exsmooth_model_type	Simple HoltsLinearTrend BrownsLinearTrend DampedTrend SimpleSeasonal WintersAdditive WintersMultiplicative	
exsmooth_transformation_type	None SquareRoot NaturalLog	
arima_p	*integer*	Same property as for Time Series modeling node
arima_d	*integer*	Same property as for Time Series modeling node
arima_q	*integer*	Same property as for Time Series modeling node
arima_sp	*integer*	Same property as for Time Series modeling node
arima_sd	*integer*	Same property as for Time Series modeling node
arima_sq	*integer*	Same property as for Time Series modeling node
arima_transformation_type	None SquareRoot NaturalLog	Same property as for Time Series modeling node
arima_include_constant	*boolean*	Same property as for Time Series modeling node
tf_arima_p.*fieldname*	*integer*	Same property as for Time Series modeling node. For transfer functions.
tf_arima_d.*fieldname*	*integer*	Same property as for Time Series modeling node. For transfer functions.
tf_arima_q.*fieldname*	*integer*	Same property as for Time Series modeling node. For transfer functions.
tf_arima_sp.*fieldname*	*integer*	Same property as for Time Series modeling node. For transfer functions.
tf_arima_sd.*fieldname*	*integer*	Same property as for Time Series modeling node. For transfer functions.
tf_arima_sq.*fieldname*	*integer*	Same property as for Time Series modeling node. For transfer functions.
tf_arima_delay.*fieldname*	*integer*	Same property as for Time Series modeling node. For transfer functions.
tf_arima_transformation_type.*fieldname*	None SquareRoot NaturalLog	
arima_detect_outlier_mode	None Automatic	
arima_outlier_additive	*boolean*	
arima_outlier_level_shift	*boolean*	

Table 56. streamingts node properties (continued).

streamingts node properties	**Data type**	**Property description**
arima_outlier_innovational	*boolean*	
arima_outlier_transient	*boolean*	
arima_outlier_seasonal_additive	*boolean*	
arima_outlier_local_trend	*boolean*	
arima_outlier_additive_patch	*boolean*	
deployment_force_rebuild	*boolean*	
deployment_rebuild_mode	Count Percent	
deployment_rebuild_count	*number*	
deployment_rebuild_pct	*number*	
deployment_rebuild_field	<*field*>	

Chapter 11. Field Operations Node Properties

anonymize Node 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.

Table 57. anonymize node properties.

`anonymize` node properties	Data type	Property description
`enable_anonymize`	*boolean*	When set to T, activates anonymization of field values (equivalent to selecting **Yes** for that field in the Anonymize Values column).
`use_prefix`	*boolean*	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`	*boolean*	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.

autodataprep Node 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.

Table 58. autodataprep node properties.

`autodataprep` node properties	Data type	Property description

`objective`	`Balanced`
	`Speed`
	`Accuracy`
	`Custom`	

Table 58. autodataprep node properties (continued).

autodataprep node properties	**Data type**	**Property description**
custom_fields	*boolean*	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	*boolean*	
frequency_field	*field*	
use_weight	*boolean*	
weight_field	*field*	
excluded_fields	Filter None	
if_fields_do_not_match	StopExecution ClearAnalysis	
prepare_dates_and_times	*boolean*	Control access to all the date and time fields
compute_time_until_date	*boolean*	
reference_date	Today Fixed	
fixed_date	*date*	
units_for_date_durations	Automatic Fixed	
fixed_date_units	Years Months Days	
compute_time_until_time	*boolean*	
reference_time	CurrentTime Fixed	
fixed_time	*time*	
units_for_time_durations	Automatic Fixed	
fixed_date_units	Hours Minutes Seconds	
extract_year_from_date	*boolean*	
extract_month_from_date	*boolean*	
extract_day_from_date	*boolean*	
extract_hour_from_time	*boolean*	
extract_minute_from_time	*boolean*	
extract_second_from_time	*boolean*	
exclude_low_quality_inputs	*boolean*	
exclude_too_many_missing	*boolean*	
maximum_percentage_missing	*number*	
exclude_too_many_categories	*boolean*	
maximum_number_categories	*number*	

Table 58. autodataprep node properties (continued).

autodataprep node properties	Data type	Property description
exclude_if_large_category	*boolean*	
maximum_percentage_category	*number*	
prepare_inputs_and_target	*boolean*	
adjust_type_inputs	*boolean*	
adjust_type_target	*boolean*	
reorder_nominal_inputs	*boolean*	
reorder_nominal_target	*boolean*	
replace_outliers_inputs	*boolean*	
replace_outliers_target	*boolean*	
replace_missing_continuous_inputs	*boolean*	
replace_missing_continuous_target	*boolean*	
replace_missing_nominal_inputs	*boolean*	
replace_missing_nominal_target	*boolean*	
replace_missing_ordinal_inputs	*boolean*	
replace_missing_ordinal_target	*boolean*	
maximum_values_for_ordinal	*number*	
minimum_values_for_continuous	*number*	
outlier_cutoff_value	*number*	
outlier_method	Replace Delete	
rescale_continuous_inputs	*boolean*	
rescaling_method	MinMax ZScore	
min_max_minimum	*number*	
min_max_maximum	*number*	
z_score_final_mean	*number*	
z_score_final_sd	*number*	
rescale_continuous_target	*boolean*	
target_final_mean	*number*	
target_final_sd	*number*	
transform_select_input_fields	*boolean*	
maximize_association_with_target	*boolean*	
p_value_for_merging	*number*	
merge_ordinal_features	*boolean*	
merge_nominal_features	*boolean*	
minimum_cases_in_category	*number*	
bin_continuous_fields	*boolean*	
p_value_for_binning	*number*	
perform_feature_selection	*boolean*	
p_value_for_selection	*number*	

Table 58. autodataprep node properties (continued).

autodataprep node properties	Data type	Property description
perform_feature_construction	*boolean*	
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*	

binning Node 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.

Table 59. binning node properties.

binning node 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*.

Table 59. binning node properties (continued).

binning node properties	**Data type**	**Property description**
`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.
`tile4`	*boolean*	Generates four quantile bins, each containing 25% of cases.
`tile5`	*boolean*	Generates five quintile bins.
`tile10`	*boolean*	Generates 10 decile bins.
`tile20`	*boolean*	Generates 20 vingtile bins.
`tile100`	*boolean*	Generates 100 percentile bins.
`use_custom_tile`	*boolean*	
`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`	*boolean*	
`rank_name_extension`	*string*	The default extension is *_RANK*.
`rank_fractional`	*boolean*	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`	*boolean*	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.

Table 59. binning node properties (continued).

binning node properties	Data type	Property description
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.
optimal_merge_bins	*boolean*	Specifies that any bins with small case counts will be added to a larger, neighboring bin.
optimal_small_bin_threshold	*integer*	
optimal_pre_bin	*boolean*	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	

derive Node 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.

Table 60. derive node properties.

derive node 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).
add_as	Suffix Prefix	Adds the extension as a prefix (at the beginning) or as a suffix (at the end) of the field name.

Table 60. derive node properties (continued).

`derive` node properties	Data type	Property description

`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.
`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*	

ensemble Node Properties

The Ensemble node combines two or more model nuggets to obtain more accurate predictions than can be gained from any one model.

Table 61. ensemble node properties.

`ensemble` node properties	Data type	Property description
`ensemble_target_field`	*field*	Specifies the target field for all models used in the ensemble.
`filter_individual_model_output`	*boolean*	Specifies whether scoring results from individual models should be suppressed.

Table 61. ensemble node properties (continued).

ensemble node properties	Data type	Property description
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	*boolean*	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.

filler Node 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.

Table 62. filler node properties.

filler node 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*	

filter Node Properties

The Filter node filters (discards) fields, renames fields, and maps fields from one source node to another.

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.

Table 63. filter node properties.

`filter` node properties	Data type	Property description
`default_include`	*boolean*	Keyed property to specify whether the default behavior is to pass or filter fields. 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.
`include`	*boolean*	Keyed property for field inclusion and removal.
`new_name`	*string*	

history Node 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.

Table 64. history node properties.

`history` node 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.
`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.

partition Node 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.

Table 65. partition node properties.

`partition` node properties	Data type	Property description
`new_name`	*string*	Name of the partition field generated by the node.
`create_validation`	*boolean*	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.
`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`.

reclassify Node Properties

The Reclassify node transforms one set of categorical values to another. Reclassification is useful for collapsing categories or regrouping data for analysis.

Table 66. reclassify node properties.

`reclassify` node 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`	*boolean*	
`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.
`reclassify`	*string*	Structured property for field values.
`use_default`	*boolean*	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.

reorder Node 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.

Table 67. reorder node properties.

`reorder` node properties	Data type	Property description
`mode`	`Custom` `Auto`	You can sort values automatically or specify a custom order.
`sort_by`	`Name` `Type` `Storage`	
`ascending`	*boolean*	
`start_fields`	*[field1 field2 ... fieldn]*	New fields are inserted after these fields.
`end_fields`	*[field1 field2 ... fieldn]*	New fields are inserted before these fields.

restructure Node 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.

Table 68. restructure node properties.

restructure node properties	**Data type**	**Property description**
fields_from	[*category category category*] all	
include_field_name	*boolean*	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.

rfmanalysis Node 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).

Table 69. rfmanalysis node properties.

rfmanalysis node 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	

Table 69. rfmanalysis node properties (continued).

`rfmanalysis` **node properties**	**Data type**	**Property description**
`add_outliers`	*boolean*	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`.

settoflag Node Properties

The Set to Flag node derives multiple flag fields based on the categorical values defined for one or more nominal fields.

Table 70. settoflag node properties.

`settoflag` **node properties**	**Data type**	**Property description**
`fields_from`	[*category category category*] `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`	*boolean*	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`	*boolean*	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.

statisticstransform Node 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 IBM SPSS Statistics.

The properties for this node are described under “statisticstransform Node Properties” on page 227.

timeintervals Node 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.

Table 71. timeintervals node properties.

`timeintervals` **node 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.

Table 71. timeintervals node properties (continued).

timeintervals node 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*).

Table 71. timeintervals node properties (continued).

`timeintervals` node 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`	*boolean*	

`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`	

Table 71. timeintervals node properties (continued).

timeintervals node 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.
pad	Blank MeanOfRecentPoints True False	Specifies the padding method for a field.
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	*boolean*	
estimation_to_end	*boolean*	

Table 71. timeintervals node properties (continued).

timeintervals node properties	Data type	Property description
estimation_start_offset	*integer*	
estimation_num_holdouts	*integer*	
create_future_records	*boolean*	
num_future_records	*integer*	
create_future_field	*boolean*	
future_field_name	*string*	

transpose Node Properties

The Transpose node swaps the data in rows and columns so that records become fields and fields become records.

Table 72. transpose node properties.

transpose node 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*	

type Node 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.

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.

Table 73. type node properties.

type node properties	Data type	Property description

`direction`	`Input`
	`Target`
	`Both`
	`None`
	`Partition`
	`Split`
	`Frequency`
	`RecordID`
		Keyed property for field roles.
		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`
	`OrderedSet`
	`Default`
		Measurement level of the field (previously called the "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`.
		Note: The data types used internally differ from those visible in the type node. The correspondence is as follows:
		Range -> Continuous
		Set - > Nominal
		OrderedSet -> Ordinal
		Discrete- > Categorical

`storage`	`Unknown`
	`String`
	`Integer`
	`Real`
	`Time`
	`Date`
	`Timestamp`
		Read-only keyed property for field storage type.

`check`	`None`
	`Nullify`
	`Coerce`
	`Discard`
	`Warn`
	`Abort`
		Keyed property for field type and range checking.

`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`.

`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.

`extend_values`	*boolean*	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.
`enable_missing`	*boolean*	When set to `T`, activates tracking of missing values for the field.
`missing_values`	[*value value ...*]	Specifies data values that denote missing data.

Table 73. type node properties (continued).

type node properties	Data type	Property description
range_missing	*boolean*	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	*boolean*	When set to T, *nulls* (undefined values that are displayed as $null$ in the software) are considered missing values.
whitespace_missing	*boolean*	When set to T, values containing only white space (spaces, tabs, and new lines) are considered missing values.
description	*string*	Specifies the description for a field.
value_labels	*[{Value LabelString} { Value LabelString} ...]*	Used to specify labels for value pairs.
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.
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.
decimal_separator	DEFAULT PERIOD COMMA	Sets the decimal separator for the field (applies only to fields with REAL storage).
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).

Table 73. type node properties (continued).

type node 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).

`number_format`	`DEFAULT`
	`STANDARD`
	`SCIENTIFIC`
	`CURRENCY`
		Sets the number display format for the field.

`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.
`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.
`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.

`grouping_symbol`	`DEFAULT`
	`NONE`
	`LOCALE`
	`PERIOD`
	`COMMA`
	`SPACE`
		Sets the grouping symbol for the field.

`column_width`	*integer*	Sets the column width for the field. A value of –1 will set column width to `Auto`.

`justify`	`AUTO`
	`CENTER`
	`LEFT`
	`RIGHT`
		Sets the column justification for the field.

Chapter 12. 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.

Table 74. Common graph node properties.

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`	*boolean*	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).

Specifying colors. The colors for titles, captions, backgrounds, and labels can be specified by using the hexadecimal strings starting with the hash (#) symbol.

The first two digits specify the red content; the middle two digits specify the green content; and the last two digits 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.

collection Node 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.

Table 75. collection node properties.

`collection` node properties	Data type	Property description
`over_field`	*field*	
`over_label_auto`	*boolean*	
`over_label`	*string*	
`collect_field`	*field*	
`collect_label_auto`	*boolean*	
`collect_label`	*string*	
`three_D`	*boolean*	
`by_field`	*field*	
`by_label_auto`	*boolean*	
`by_label`	*string*	

`operation`	`Sum`
	`Mean`
	`Min`
	`Max`
	`SDev`	

`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`	*boolean*	
`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.

distribution Node 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.

Table 76. distribution node properties.

`distribution` node properties	Data type	Property description
`plot`	`SelectedFields` `Flags`	

Table 76. distribution node properties (continued).

distribution node properties	Data type	Property description
x_field	*field*	
color_field	*field*	Overlay field.
normalize	*boolean*	
sort_mode	ByOccurence Alphabetic	
use_proportional_scale	*boolean*	

evaluation Node 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.

Table 77. evaluation node properties.

evaluation node properties	Data type	Property description
chart_type	Gains Response Lift Profit ROI ROC	
inc_baseline	*boolean*	
field_detection_method	Metadata Name	
use_fixed_cost	*boolean*	
cost_value	*number*	
cost_field	*string*	
use_fixed_revenue	*boolean*	
revenue_value	*number*	
revenue_field	*string*	
use_fixed_weight	*boolean*	
weight_value	*number*	
weight_field	*field*	
n_tile	Quartiles Quintles Deciles Vingtiles Percentiles 1000-tiles	
cumulative	*flag*	
style	Line Point	

Table 77. evaluation node properties (continued).

`evaluation` node 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`	

`export_data`	*boolean*	
`data_filename`	*string*	
`delimiter`	*string*	
`new_line`	*boolean*	
`inc_field_names`	*boolean*	
`inc_best_line`	*boolean*	
`inc_business_rule`	*boolean*	
`business_rule_condition`	*string*	
`plot_score_fields`	*boolean*	
`score_fields`	*[field1 ... fieldN]*	
`target_field`	*field*	
`use_hit_condition`	*boolean*	
`hit_condition`	*string*	
`use_score_expression`	*boolean*	
`score_expression`	*string*	
`caption_auto`	*boolean*	

graphboard Node 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.

Table 78. graphboard node properties.

graphboard node 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 Pie PieCountMap PieCounts PieMap PointOverlayMap PolygonOverlayMap Ribbon Scatterplot SPLOM Surface	Identifies the graph type.
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*	

Table 78. graphboard node properties (continued).

graphboard node properties	Data type	Property description
`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*	
`map_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*	
`map_file`	*field*	
`reference_map_file`	*field*	
`map_layer`	*field*	
`reference_map_layer`	*field*	
`map_attribute`	*field*	
`reference_map_attribute`	*field*	

histogram Node 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.

Table 79. histogram node properties.

histogram node 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*	

Table 79. histogram node properties (continued).

`histogram` node properties	Data type	Property description
`bins`	`ByNumber` `ByWidth`	
`num_bins`	*number*	
`bin_width`	*number*	
`normalize`	*boolean*	
`separate_bands`	*boolean*	
`x_label_auto`	*boolean*	
`x_label`	*string*	
`y_label_auto`	*boolean*	
`y_label`	*string*	
`use_grid`	*boolean*	
`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`	*boolean*	Indicates whether the normal distribution curve should be shown on the output.

multiplot Node 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.

Table 80. multiplot node properties.

`multiplot` node properties	Data type	Property description
`x_field`	*field*	
`y_fields`	[*field field field*]	
`panel_field`	*field*	
`animation_field`	*field*	
`normalize`	*boolean*	
`use_overlay_expr`	*boolean*	
`overlay_expression`	*string*	
`records_limit`	*number*	

`if_over_limit`	`PlotBins`
	`PlotSample`
	`PlotAll`	

`x_label_auto`	*boolean*	
`x_label`	*string*	
`y_label_auto`	*boolean*	
`y_label`	*string*	
`use_grid`	*boolean*	

Table 80. multiplot node properties (continued).

`multiplot` node properties	Data type	Property description
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.

plot Node Properties

The Plot node shows the relationship between numeric fields. You can create a plot by using points (a scatterplot) or lines.

Table 81. plot node properties.

`plot` node 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	*boolean*	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	

Table 81. plot node properties (continued).

plot node 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	*boolean*	
records_limit	*number*	
if_over_limit	PlotBins PlotSample PlotAll	
x_label_auto	*boolean*	
x_label	*string*	
y_label_auto	*boolean*	
y_label	*string*	
z_label_auto	*boolean*	
z_label	*string*	
use_grid	*boolean*	
graph_background	*color*	Standard graph colors are described at the beginning of this section.

Table 81. plot node properties (continued).

plot node properties	Data type	Property description
page_background	*color*	Standard graph colors are described at the beginning of this section.
use_overlay_expr	*boolean*	Deprecated in favor of overlay_type.

timeplot Node 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.

Table 82. timeplot node properties.

timeplot node properties	Data type	Property description
plot_series	Series Models	
use_custom_x_field	*boolean*	
x_field	*field*	
y_fields	[*field field field*]	
panel	*boolean*	
normalize	*boolean*	
line	*boolean*	
points	*boolean*	
point_type	Rectangle Dot Triangle Hexagon Plus Pentagon Star BowTie HorizontalDash VerticalDash IronCross Factory House Cathedral OnionDome ConcaveTriangle OblateGlobe CatEye FourSidedPillow RoundRectangle Fan	
smoother	*boolean*	You can add smoothers to the plot only if you set panel to True.
use_records_limit	*boolean*	
records_limit	*integer*	
symbol_size	*number*	Specifies a symbol size.

Table 82. timeplot node properties (continued).

`timeplot` node properties	**Data type**	**Property description**
`panel_layout`	`Horizontal` `Vertical`	

web Node 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.

Table 83. web node properties.

`web` node properties	**Data type**	**Property description**
`use_directed_web`	*boolean*	
`fields`	[*field field field*]	
`to_field`	*field*	
`from_fields`	[*field field field*]	
`true_flags_only`	*boolean*	

`line_values`	`Absolute`
	`OverallPct`
	`PctLarger`
	`PctSmaller`	

`strong_links_heavier`	*boolean*	

`num_links`	`ShowMaximum`
	`ShowLinksAbove`
	`ShowAll`	

`max_num_links`	*number*	
`links_above`	*number*	
`discard_links_min`	*boolean*	
`links_min_records`	*number*	
`discard_links_max`	*boolean*	
`links_max_records`	*number*	
`weak_below`	*number*	
`strong_above`	*number*	
`link_size_continuous`	*boolean*	

`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.

Chapter 13. 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.

Table 84. Common modeling node properties.

Property	Values	Property description
`custom_fields`	*boolean*	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`	*boolean*	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`	*boolean*	
`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`	*boolean*	Weight and frequency fields are used by specific models as noted for each model type.
`frequency_field`	*field*	
`use_weight`	*boolean*	
`weight_field`	*field*	
`use_model_name`	*boolean*	
`model_name`	*string*	Custom name for new model.
`mode`	`Simple` `Expert`	

anomalydetection Node 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.

Table 85. anomalydetection node properties.

anomalydetection Node 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. See the topic "Common Modeling Node Properties" on page 125 for more information.
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	*boolean*	
adjustment_coeff	*number*	Value used to balance the relative weight given to continuous and categorical fields in calculating the distance.
peer_group_num_auto	*boolean*	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.
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.

apriori Node 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.

Table 86. apriori node properties.

`apriori` Node 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. See the topic "Common Modeling Node Properties" on page 125 for more information.
`antecedents`	[*field1 ... fieldN*]	
`min_supp`	*number*	
`min_conf`	*number*	
`max_antecedents`	*number*	
`true_flags`	*boolean*	
`optimize`	`Speed` `Memory`	
`use_transactional_data`	*boolean*	
`contiguous`	*boolean*	
`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.

autoclassifier Node 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.

Table 87. autoclassifier node properties.

autoclassifier Node 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. See the topic "Common Modeling Node Properties" on page 125 for more information.
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	*boolean*	
enable_accuracy_limit	*boolean*	
accuracy_limit	*integer*	Integer between 0 and 100.
enable_ area_under_curve _limit	*boolean*	
area_under_curve_limit	*number*	Real number between 0.0 and 1.0.
enable_profit_limit	*boolean*	
profit_limit	*number*	Integer greater than 0.
enable_lift_limit	*boolean*	
lift_limit	*number*	Real number greater than 1.0.
enable_number_of_variables_limit	*boolean*	
number_of_variables_limit	*number*	Integer greater than 0.
use_fixed_cost	*boolean*	
fixed_cost	*number*	Real number greater than 0.0.
variable_cost	*field*	
use_fixed_revenue	*boolean*	
fixed_revenue	*number*	Real number greater than 0.0.
variable_revenue	*field*	
use_fixed_weight	*boolean*	
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	*boolean*	
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	*boolean*	
stop_after_time_limit	*number*	Real number set to the number of hours to limit the overall elapsed time for an auto classifier run.

Table 87. autoclassifier node properties (continued).

autoclassifier Node Properties	Values	Property description
enable_stop_after_valid_model_produced	*boolean*	
use_costs	*boolean*	
<algorithm>	*boolean*	Enables or disables the use of a specific algorithm.
<algorithm>.<property>	*string*	Sets a property value for a specific algorithm. See the topic "Setting Algorithm Properties" for more information.

Setting Algorithm Properties

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.

Multiple values can also be assigned for property.

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.

autocluster Node 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.

Table 88. autocluster node properties.

autocluster Node 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.

Table 88. autocluster node properties (continued).

autocluster Node Properties	**Values**	**Property description**
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	*boolean*	
silhouette_limit	*integer*	Integer between 0 and 100.
enable_number_less_limit	*boolean*	
number_less_limit	*number*	Real number between 0.0 and 1.0.
enable_number_greater_limit	*boolean*	
number_greater_limit	*number*	Integer greater than 0.
enable_smallest_cluster_limit	*boolean*	
smallest_cluster_units	Percentage Counts	
smallest_cluster_limit_percentage	*number*	
smallest_cluster_limit_count	*integer*	Integer greater than 0.
enable_largest_cluster_limit	*boolean*	
largest_cluster_units	Percentage Counts	
largest_cluster_limit_percentage	*number*	
largest_cluster_limit_count	*integer*	
enable_smallest_largest_limit	*boolean*	
smallest_largest_limit	*number*	
enable_importance_limit	*boolean*	
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>	*boolean*	Enables or disables the use of a specific algorithm.
<algorithm>.<property>	*string*	Sets a property value for a specific algorithm. See the topic "Setting Algorithm Properties" on page 129 for more information.

autonumeric Node 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.

Table 89. autonumeric node properties.

`autonumeric` **Node Properties**	**Values**	**Property description**
`custom_fields`	*boolean*	If True, custom field settings will be used instead of type node settings.
`target`	*field*	The Auto Numeric node requires a single target and one or more input fields. Weight and frequency fields can also be specified. See the topic "Common Modeling Node Properties" on page 125 for more information.
`inputs`	*[field1 ... field2]*	
`partition`	*field*	
`use_frequency`	*boolean*	
`frequency_field`	*field*	
`use_weight`	*boolean*	
`weight_field`	*field*	
`use_partitioned_data`	*boolean*	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`	*boolean*	
`enable_correlation_limit`	*boolean*	
`correlation_limit`	*integer*	
`enable_number_of_fields_limit`	*boolean*	
`number_of_fields_limit`	*integer*	
`enable_relative_error_limit`	*boolean*	
`relative_error_limit`	*integer*	
`enable_model_build_time_limit`	*boolean*	
`model_build_time_limit`	*integer*	
`enable_stop_after_time_limit`	*boolean*	
`stop_after_time_limit`	*integer*	
`stop_if_valid_model`	*boolean*	

Table 89. autonumeric node properties (continued).

autonumeric Node Properties	Values	Property description
`<algorithm>`	*boolean*	Enables or disables the use of a specific algorithm.
`<algorithm>.<property>`	*string*	Sets a property value for a specific algorithm. See the topic "Setting Algorithm Properties" on page 129 for more information.

bayesnet Node 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.

Table 90. bayesnet node properties.

bayesnet Node 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. See the topic "Common Modeling Node Properties" on page 125 for more information.
`continue_training_existing_model`	*boolean*	
`structure_type`	`TAN` `MarkovBlanket`	Select the structure to be used when building the Bayesian network.
`use_feature_selection`	*boolean*	
`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`	*boolean*	
`all_probabilities`	*boolean*	
`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.

Table 90. bayesnet node properties (continued).

`bayesnet` Node Properties	Values	Property description
`maximum_number_inputs`	*number*	Specifies the maximum number of input fields to be used in building the Bayesian network.
`calculate_variable_importance`	*boolean*	
`calculate_raw_propensities`	*boolean*	
`calculate_adjusted_propensities`	*boolean*	
`adjusted_propensity_partition`	`Test` `Validation`	

buildr Node Properties

The R Building node enables you to enter custom R script to perform model building and model scoring deployed in IBM SPSS Modeler.

Table 91. buildr properties.

`buildr` Node Properties	Values	Property description
`build_syntax`	*string*	R scripting syntax for model building.
`score_syntax`	*string*	R scripting syntax for model scoring.
`convert_flags`	`StringsAndDoubles` `LogicalValues`	Option to convert flag fields.
`convert_datetime`	*boolean*	Option to convert variables with date or datetime formats to R date/time formats.
`convert_datetime_class`	`POSIXct` `POSIXlt`	Options to specify to what format variables with date or datetime formats are converted.
`convert_missing`	*boolean*	Option to convert missing values to R `NA` value.
`output_html`	*boolean*	Option to display graphs on a tab in the R model nugget.
`output_text`	*boolean*	Option to write R console text output to a tab in the R model nugget.

c50 Node 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.

Table 92. c50 node properties.

`c50` Node 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. See the topic "Common Modeling Node Properties" on page 125 for more information.
`output_type`	`DecisionTree` `RuleSet`	
`group_symbolics`	*boolean*	
`use_boost`	*boolean*	
`boost_num_trials`	*number*	
`use_xval`	*boolean*	
`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`	*boolean*	
`costs`	*structured*	This is a structured property.
`use_winnowing`	*boolean*	
`use_global_pruning`	*boolean*	On (`True`) by default.
`calculate_variable_importance`	*boolean*	
`calculate_raw_propensities`	*boolean*	
`calculate_adjusted_propensities`	*boolean*	
`adjusted_propensity_partition`	`Test` `Validation`	

carma Node 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.

Table 93. carma node properties.

`carma` Node 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. See the topic "Common Modeling Node Properties" on page 125 for more information.

Table 93. carma node properties (continued).

`carma` Node Properties	Values	Property description
`id_field`	*field*	Field used as the ID field for model building.
`contiguous`	*boolean*	Used to specify whether IDs in the ID field are contiguous.
`use_transactional_data`	*boolean*	
`content_field`	*field*	
`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`	*boolean*	Excludes rules with multiple consequents. The default is `False`.
`use_pruning`	*boolean*	The default is `False`.
`pruning_value`	*number*	The default is 500.
`vary_support`	*boolean*	
`estimated_transactions`	*integer*	
`rules_without_antecedents`	*boolean*	

cart Node 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).

Table 94. cart node properties.

`cart` Node 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. See the topic "Common Modeling Node Properties" on page 125 for more information.
`continue_training_existing_model`	*boolean*	

`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`	*boolean*	

Table 94. cart node properties (continued).

cart Node Properties	**Values**	**Property description**
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	*boolean*	Prune tree to avoid overfitting.
use_std_err	*boolean*	Use maximum difference in risk (in Standard Errors).
std_err_multiplier	*number*	Maximum difference.
max_surrogates	*number*	Maximum surrogates.
use_percentage	*boolean*	
min_parent_records_pc	*number*	
min_child_records_pc	*number*	
min_parent_records_abs	*number*	
min_child_records_abs	*number*	
use_costs	*boolean*	
costs	*structured*	Structured property.
priors	Data Equal Custom	
custom_priors	*structured*	Structured property.
adjust_priors	*boolean*	
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	*boolean*	Apply boosting to very large data sets.
min_impurity	*number*	
impurity_measure	Gini Twoing Ordered	
train_pct	*number*	Overfit prevention set.
set_random_seed	*boolean*	Replicate results option.
seed	*number*	
calculate_variable_importance	*boolean*	
calculate_raw_propensities	*boolean*	
calculate_adjusted_propensities	*boolean*	

Table 94. cart node properties (continued).

`cart` Node Properties	Values	Property description
`adjusted_propensity_partition`	`Test` `Validation`	

chaid Node 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.

Table 95. chaid node properties.

`chaid` Node 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. See the topic "Common Modeling Node Properties" on page 125 for more information.
`continue_training_existing_model`	*boolean*	

`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`	*boolean*	
`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`	*boolean*	
`min_parent_records_pc`	*number*	
`min_child_records_pc`	*number*	
`min_parent_records_abs`	*number*	
`min_child_records_abs`	*number*	
`use_costs`	*boolean*	
`costs`	*structured*	Structured property.
`trails`	*number*	Number of component models for boosting or bagging.

`set_ensemble_method`	`Voting`
	`HighestProbability`
	`HighestMeanProbability`
		Default combining rule for categorical targets.

Table 95. chaid node properties (continued).

chaid Node Properties	Values	Property description
`range_ensemble_method`	`Mean` `Median`	Default combining rule for continuous targets.
`large_boost`	*boolean*	Apply boosting to very large data sets.
`split_alpha`	*number*	Significance level for splitting.
`merge_alpha`	*number*	Significance level for merging.
`bonferroni_adjustment`	*boolean*	Adjust significance values using Bonferroni method.
`split_merged_categories`	*boolean*	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`	*boolean*	
`calculate_raw_propensities`	*boolean*	
`calculate_adjusted_propensities`	*boolean*	
`adjusted_propensity_partition`	`Test` `Validation`	
`maximum_number_of_models`	*integer*	

coxreg Node 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.

Table 96. coxreg node properties.

coxreg Node Properties	Values	Property description
`survival_time`	*field*	Cox regression models require a single field containing the survival times.
`target`	*field*	Cox regression models require a single target field, and one or more input fields. See the topic "Common Modeling Node Properties" on page 125 for more information.

`method`	`Enter`
	`Stepwise`
	`BackwardsStepwise`	

`groups`	*field*	
`model_type`	`MainEffects` `Custom`	
`custom_terms`	[*"BP*Sex" "BP*Age"*]	

Table 96. coxreg node properties (continued).

coxreg **Node Properties**	**Values**	**Property description**
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	*boolean*	
ci_value	90 95 99	
correlation	*boolean*	
display_baseline	*boolean*	
survival	*boolean*	
hazard	*boolean*	
log_minus_log	*boolean*	
one_minus_survival	*boolean*	
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.

decisionlist Node 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.

Table 97. decisionlist node properties.

`decisionlist` Node 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. See the topic "Common Modeling Node Properties" on page 125 for more information.
`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.
`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`	*boolean*	
`final_results_only`	*boolean*	
`reuse_fields`	*boolean*	Allows attributes (input fields which appear in rules) to be re-used.
`max_alternatives`	*integer*	
`calculate_raw_propensities`	*boolean*	
`calculate_adjusted_propensities`	*boolean*	

Table 97. decisionlist node properties (continued).

`decisionlist` Node Properties	Values	Property description
`adjusted_propensity_partition`	`Test` `Validation`	

discriminant Node 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.

Table 98. discriminant node properties.

`discriminant` Node 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. See the topic “Common Modeling Node Properties” on page 125 for more information.
`method`	`Enter` `Stepwise`	
`mode`	`Simple` `Expert`	
`prior_probabilities`	`AllEqual` `ComputeFromSizes`	
`covariance_matrix`	`WithinGroups` `SeparateGroups`	
`means`	*boolean*	Statistics options in the Advanced Output dialog box.
`univariate_anovas`	*boolean*	
`box_m`	*boolean*	
`within_group_covariance`	*boolean*	
`within_groups_correlation`	*boolean*	
`separate_groups_covariance`	*boolean*	
`total_covariance`	*boolean*	
`fishers`	*boolean*	
`unstandardized`	*boolean*	
`casewise_results`	*boolean*	Classification options in the Advanced Output dialog box.
`limit_to_first`	*number*	Default value is 10.
`summary_table`	*boolean*	
`leave_one_classification`	*boolean*	
`combined_groups`	*boolean*	
`separate_groups_covariance`	*boolean*	Matrices option **Separate-groups covariance**.
`territorial_map`	*boolean*	

Table 98. discriminant node properties (continued).

`discriminant` Node Properties	Values	Property description
combined_groups	*boolean*	Plot option **Combined-groups**.
separate_groups	*boolean*	Plot option **Separate-groups**.
summary_of_steps	*boolean*	
F_pairwise	*boolean*	
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	*boolean*	
calculate_raw_propensities	*boolean*	
calculate_adjusted_propensities	*boolean*	
adjusted_propensity_partition	Test Validation	

factor Node 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.

Table 99. factor node properties.

`factor` Node 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. See the topic "Common Modeling Node Properties" on page 125 for more information.
method	PC ULS GLS ML PAF Alpha Image	

Table 99. factor node properties (continued).

factor Node Properties	Values	Property description
mode	Simple Expert	
max_iterations	*number*	
complete_records	*boolean*	
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	*boolean*	
hide_values	*boolean*	
hide_below	*number*	

featureselection Node 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?

Table 100. featureselection node properties.

featureselection Node Properties	Values	Property description
target	*field*	Feature Selection models rank predictors relative to the specified target. Weight and frequency fields are not used. See the topic "Common Modeling Node Properties" on page 125 for more information.

Table 100. featureselection node properties (continued).

`featureselection` Node Properties	Values	Property description
screen_single_category	*boolean*	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	*boolean*	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	*boolean*	If True, screens fields with too many categories relative to the total number of records.
max_num_categories	*number*	
screen_std_dev	*boolean*	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	*boolean*	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	*boolean*	When selection_mode is set to ImportanceLevel, specifies whether to select important fields.
select_marginal	*boolean*	When selection_mode is set to ImportanceLevel, specifies whether to select marginal fields.
select_unimportant	*boolean*	When selection_mode is set to ImportanceLevel, specifies whether to select unimportant fields.

Table 100. featureselection node properties (continued).

featureselection Node Properties	Values	Property description
`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.

genlin Node 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.

Table 101. genlin node properties.

genlin Node 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. See the topic "Common Modeling Node Properties" on page 125 for more information.
`use_weight`	*boolean*	
`weight_field`	*field*	Field type is only continuous.
`target_represents_trials`	*boolean*	
`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`	
`offset_field`	*field*	Field type is only continuous.
`offset_value`	*number*	Must be a real number.
`base_category`	`Last` `First`	
`include_intercept`	*boolean*	
`mode`	`Simple` `Expert`	

Table 101. genlin node properties (continued).

genlin Node Properties	**Values**	**Property description**
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.
check_separation	*boolean*	
start_iteration	*number*	Default value is 20; only positive integers allowed.
estimates_change	*boolean*	
estimates_change_min	*number*	Default value is 1E-006; only positive numbers allowed.

Table 101. genlin node properties (continued).

genlin Node Properties	**Values**	**Property description**
`estimates_change_type`	`Absolute` `Relative`	
`loglikelihood_change`	*boolean*	
`loglikelihood_change_min`	*number*	Only positive numbers allowed.
`loglikelihood_change_type`	`Absolute` `Relative`	
`hessian_convergence`	*boolean*	
`hessian_convergence_min`	*number*	Only positive numbers allowed.
`hessian_convergence_type`	`Absolute` `Relative`	
`case_summary`	*boolean*	
`contrast_matrices`	*boolean*	
`descriptive_statistics`	*boolean*	
`estimable_functions`	*boolean*	
`model_info`	*boolean*	
`iteration_history`	*boolean*	
`goodness_of_fit`	*boolean*	
`print_interval`	*number*	Default value is 1; must be positive integer.
`model_summary`	*boolean*	
`lagrange_multiplier`	*boolean*	
`parameter_estimates`	*boolean*	
`include_exponential`	*boolean*	
`covariance_estimates`	*boolean*	
`correlation_estimates`	*boolean*	

`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`	

`value_order`	`Ascending`
	`Descending`
	`DataOrder`	

Table 101. genlin node properties (continued).

`genlin` Node Properties	Values	Property description
`calculate_variable_importance`	*boolean*	
`calculate_raw_propensities`	*boolean*	
`calculate_adjusted_propensities`	*boolean*	
`adjusted_propensity_partition`	`Test` `Validation`	

glmm Node 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.

Table 102. glmm node properties.

`glmm` Node 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`	*boolean*	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`	*boolean*	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.
`use_custom_target_reference`	*boolean*	Indicates whether custom reference category is to be used for a categorical target. Default is `false`.

Table 102. glmm node properties (continued).

glmm Node Properties	Values	Property description
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. If target_distribution is Binomial you can use any of the listed link functions. If target_distribution is Multinomial you can use CLOGLOG, CAUCHIT, LOGIT, NLOGLOG, or PROBIT. If target_distribution is anything other than Binomial or Multinomial you can use IDENTITY, LOG, or POWER.
link_function_param	*number*	Link function parameter value to use. Only applicable if normal_link_function or link_function_type is POWER.
use_predefined_inputs	*boolean*	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	*boolean*	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.

Table 102. glmm node properties (continued).

glmm Node 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_p_converge	*boolean*	Option for parameter convergence.
p_converge	*number*	Blank, or any positive value.
p_converge_type	Absolute Relative	
use_l_converge	*boolean*	Option for log-likelihood convergence.
l_converge	*number*	Blank, or any positive value.
l_converge_type	Absolute Relative	
use_h_converge	*boolean*	Option for Hessian convergence.
h_converge	*number*	Blank, or any positive value.
h_converge_type	Absolute Relative	
max_fisher_steps	*integer*	
singularity_tolerance	*number*	
use_model_name	*boolean*	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	onProbability onIncrease	Basis for computing scoring confidence value: highest predicted probability, or difference between highest and second highest predicted probabilities.
score_category_probabilities	*boolean*	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.

Table 102. glmm node properties (continued).

glmm Node Properties	**Values**	**Property description**
score_propensity	*boolean*	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.

kmeans Node 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.

Table 103. kmeans node properties.

kmeans node 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. See the topic "Common Modeling Node Properties" on page 125 for more information.
num_clusters	*number*	
gen_distance	*boolean*	
cluster_label	String Number	
label_prefix	*string*	
mode	Simple Expert	
stop_on	Default Custom	
max_iterations	*number*	
tolerance	*number*	
encoding_value	*number*	

Table 103. kmeans node properties (continued).

kmeans node Properties	Values	Property description
optimize	Speed Memory	Use to specify whether model building should be optimized for speed or for memory.

knn Node 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.

Table 104. knn node properties.

knn Node Properties	Values	Property description
analysis	PredictTarget IdentifyNeighbors	
objective	Balance Speed Accuracy Custom	
normalize_ranges	*boolean*	
use_case_labels	*boolean*	Check box to enable next option.
case_labels_field	*field*	
identify_focal_cases	*boolean*	Check box to enable next option.
focal_cases_field	*field*	
automatic_k_selection	*boolean*	
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	*boolean*	
range_predictions	Mean Median	
perform_feature_selection	*boolean*	
forced_entry_inputs	[*field1 ... fieldN*]	
stop_on_error_ratio	*boolean*	
number_to_select	*integer*	
minimum_change	*number*	
validation_fold_assign_by_field	*boolean*	
number_of_folds	*integer*	Enabled only if validation_fold_assign_by_field is False
set_random_seed	*boolean*	

Table 104. knn node properties (continued).

knn Node Properties	**Values**	**Property description**
`random_seed`	*number*	
`folds_field`	*field*	Enabled only if `validation_fold_assign_by_field` is `True`
`all_probabilities`	*boolean*	
`save_distances`	*boolean*	
`calculate_raw_propensities`	*boolean*	
`calculate_adjusted_propensities`	*boolean*	
`adjusted_propensity_partition`	`Test` `Validation`	

kohonen Node 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.

Table 105. kohonen node properties.

kohonen Node 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. See the topic "Common Modeling Node Properties" on page 125 for more information.
`continue`	*boolean*	
`show_feedback`	*boolean*	
`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`	*boolean*	
`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*	

Table 105. kohonen node properties (continued).

kohonen Node Properties	Values	Property description
phase2_cycles	*number*	

linear Node Properties

Linear regression models predict a continuous target based on linear relationships between the target and one or more predictors.

Table 106. linear node properties.

linear Node Properties	Values	Property description
target	*field*	Specifies a single target field.
inputs	[*field1 ... fieldN*]	Predictor fields used by the model.
continue_training_existing_model	*boolean*	
objective	Standard Bagging Boosting psm	psm is used for very large datasets, and requires a Server connection.
use_auto_data_preparation	*boolean*	
confidence_level	*number*	
model_selection	ForwardStepwise BestSubsets None	
criteria_forward_stepwise	AICC Fstatistics AdjustedRSquare ASE	
probability_entry	*number*	
probability_removal	*number*	
use_max_effects	*boolean*	
max_effects	*number*	
use_max_steps	*boolean*	
max_steps	*number*	
criteria_best_subsets	AICC AdjustedRSquare ASE	
combining_rule_continuous	Mean Median	
component_models_n	*number*	
use_random_seed	*boolean*	
random_seed	*number*	
use_custom_model_name	*boolean*	
custom_model_name	*string*	

Table 106. linear node properties (continued).

`linear` Node Properties	Values	Property description
use_custom_name	*boolean*	
custom_name	*string*	
tooltip	*string*	
keywords	*string*	
annotation	*string*	

logreg Node 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.

Table 107. logreg node properties.

`logreg` Node 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. See the topic "Common Modeling Node Properties" on page 125 for more information.
logistic_procedure	Binomial Multinomial	
include_constant	*boolean*	
mode	Simple Expert	
method	Enter Stepwise Forwards Backwards BackwardsStepwise	
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}*]	
multinomial_base_category	*string*	Specifies how the reference category is determined.
binomial_categorical_input	*string*	

Table 107. logreg node properties (continued).

`logreg` Node Properties	Values	Property description

`binomial_input_contrast`	`Indicator`
	`Simple`
	`Difference`
	`Helmert`
	`Repeated`
	`Polynomial`
	`Deviation`
		Keyed property for categorical input that specifies how the contrast is determined.

`binomial_input_category`	`First` `Last`	Keyed property for categorical input that specifies how the reference category is determined.

`scale`	`None`
	`UserDefined`
	`Pearson`
	`Deviance`	

`scale_value`	*number*	
`all_probabilities`	*boolean*	

`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`	*boolean*	
`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*	

`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`	

Table 107. logreg node properties (continued).

logreg Node Properties	Values	Property description
l_converge	1.0E-1 1.0E-2 1.0E-3 1.0E-4 1.0E-5 0	
delta	*number*	
iteration_history	*boolean*	
history_steps	*number*	
summary	*boolean*	
likelihood_ratio	*boolean*	
asymptotic_correlation	*boolean*	
goodness_fit	*boolean*	
parameters	*boolean*	
confidence_interval	*number*	
asymptotic_covariance	*boolean*	
classification_table	*boolean*	
stepwise_summary	*boolean*	
info_criteria	*boolean*	
monotonicity_measures	*boolean*	
binomial_output_display	at_each_step at_last_step	
binomial_goodness_of_fit	*boolean*	
binomial_parameters	*boolean*	
binomial_iteration_history	*boolean*	
binomial_classification_plots	*boolean*	
binomial_ci_enable	*boolean*	
binomial_ci	*number*	
binomial_residual	outliers all	
binomial_residual_enable	*boolean*	
binomial_outlier_threshold	*number*	
binomial_classification_cutoff	*number*	
binomial_removal_criterion	LR Wald Conditional	
calculate_variable_importance	*boolean*	
calculate_raw_propensities	*boolean*	

neuralnet Node 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.

Table 108. neuralnet node properties.

`neuralnet` Node 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. See the topic "Common Modeling Node Properties" on page 125 for more information.

`method`	`Quick`
	`Dynamic`
	`Multiple`
	`Prune`
	`ExhaustivePrune`
	`RBFN`	

`prevent_overtrain`	*boolean*	
`train_pct`	*number*	
`set_random_seed`	*boolean*	
`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`	*boolean*	
`show_feedback`	*boolean*	
`binary_encode`	*boolean*	
`use_last_model`	*boolean*	
`gen_logfile`	*boolean*	
`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*	

Table 108. neuralnet node properties (continued).

neuralnet Node Properties	Values	Property description
hl_units_two	*number*	
hl_units_three	*number*	
persistence	*number*	
m_topologies	*string*	
m_non_pyramids	*boolean*	
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*	
r_eta_auto	*boolean*	
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	*boolean*	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	*boolean*	
calculate_adjusted_propensities	*boolean*	
adjusted_propensity_partition	Test Validation	

neuralnetwork Node 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.

Table 109. neuralnetwork node properties.

neuralnetwork Node 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	*boolean*	If a partition field is defined, this option ensures that only data from the training partition is used to build the model.
continue	*boolean*	Continue training existing model.
objective	Standard Bagging Boosting psm	psm is used for very large datasets, and requires a Server connection.
method	MultilayerPerceptron RadialBasisFunction	
use_custom_layers	*boolean*	
first_layer_units	*number*	
second_layer_units	*number*	
use_max_time	*boolean*	
max_time	*number*	
use_max_cycles	*boolean*	
max_cycles	*number*	
use_min_accuracy	*boolean*	
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	*boolean*	
random_seed	*number*	
missing_values	listwiseDeletion missingValueImputation	
use_custom_model_name	*boolean*	
custom_model_name	*string*	
confidence	onProbability onIncrease	
score_category_probabilities	*boolean*	
max_categories	*number*	
score_propensity	*boolean*	
use_custom_name	*boolean*	

Table 109. neuralnetwork node properties (continued).

neuralnetwork Node Properties	Values	Property description
custom_name	*string*	
tooltip	*string*	
keywords	*string*	
annotation	*string*	

quest Node 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.

Table 110. quest node properties.

quest Node 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. See the topic "Common Modeling Node Properties" on page 125 for more information.
continue_training_existing_model	*boolean*	
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	*boolean*	
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	*boolean*	Prune tree to avoid overfitting.
use_std_err	*boolean*	Use maximum difference in risk (in Standard Errors).
std_err_multiplier	*number*	Maximum difference.
max_surrogates	*number*	Maximum surrogates.
use_percentage	*boolean*	
min_parent_records_pc	*number*	
min_child_records_pc	*number*	
min_parent_records_abs	*number*	
min_child_records_abs	*number*	
use_costs	*boolean*	
costs	*structured*	Structured property.

Table 110. quest node properties (continued).

`quest` Node Properties	Values	Property description
priors	Data Equal Custom	
custom_priors	*structured*	Structured property.
adjust_priors	*boolean*	
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	*boolean*	Apply boosting to very large data sets.
split_alpha	*number*	Significance level for splitting.
train_pct	*number*	Overfit prevention set.
set_random_seed	*boolean*	Replicate results option.
seed	*number*	
calculate_variable_importance	*boolean*	
calculate_raw_propensities	*boolean*	
calculate_adjusted_propensities	*boolean*	
adjusted_propensity_partition	Test Validation	

regression Node 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.

Table 111. regression node properties.

`regression` Node 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. See the topic "Common Modeling Node Properties" on page 125 for more information.
method	Enter Stepwise Backwards Forwards	
include_constant	*boolean*	

Table 111. regression node properties (continued).

`regression` Node Properties	Values	Property description
use_weight	*boolean*	
weight_field	*field*	
mode	Simple Expert	
complete_records	*boolean*	
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	*boolean*	
confidence_interval	*boolean*	
covariance_matrix	*boolean*	
collinearity_diagnostics	*boolean*	
regression_coefficients	*boolean*	
exclude_fields	*boolean*	
durbin_watson	*boolean*	
model_fit	*boolean*	
r_squared_change	*boolean*	
p_correlations	*boolean*	
descriptives	*boolean*	
calculate_variable_importance	*boolean*	

sequence Node 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.

Table 112. sequence node properties.

`sequence` Node 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. See the topic "Common Modeling Node Properties" on page 125 for more information.
`time_field`	*field*	
`use_time_field`	*boolean*	
`content_fields`	[*field1 ... fieldn*]	
`contiguous`	*boolean*	
`min_supp`	*number*	
`min_conf`	*number*	
`max_size`	*number*	
`max_predictions`	*number*	
`mode`	`Simple` `Expert`	
`use_max_duration`	*boolean*	
`max_duration`	*number*	
`use_gaps`	*boolean*	
`min_item_gap`	*number*	
`max_item_gap`	*number*	
`use_pruning`	*boolean*	
`pruning_value`	*number*	
`set_mem_sequences`	*boolean*	
`mem_sequences`	*integer*	

slrm Node 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.

Table 113. slrm node properties.

`slrm` Node Properties	Values	Property description
`target`	*field*	The target field must be a nominal or flag field. A frequency field can also be specified. See the topic "Common Modeling Node Properties" on page 125 for more information.
`target_response`	*field*	Type must be flag.
`continue_training_existing_model`	*boolean*	
`target_field_values`	*boolean*	Use all: Use all values from source. Specify: Select values required.

Table 113. slrm node properties (continued).

slrm Node Properties	Values	Property description
target_field_values_specify	*[field1 ... fieldN]*	
include_model_assessment	*boolean*	
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	*boolean*	
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	*boolean*	
calculate_variable_importance	*boolean*	

statisticsmodel Node 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 IBM SPSS Statistics.

The properties for this node are described under "statisticsmodel Node Properties" on page 228.

svm Node 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.

Table 114. svm node properties.

svm Node Properties	Values	Property description
all_probabilities	*boolean*	
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.

Table 114. svm node properties (continued).

`svm` Node Properties	Values	Property description

`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`	*boolean*	
`calculate_raw_propensities`	*boolean*	
`calculate_adjusted_propensities`	*boolean*	
`adjusted_propensity_partition`	`Test` `Validation`	

timeseries Node 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.

Table 115. timeseries node properties.

`timeseries` Node 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. See the topic "Common Modeling Node Properties" on page 125 for more information.
`continue`	*boolean*	

`method`	`ExpertModeler`
	`Exsmooth`
	`Arima`
	`Reuse`	

`expert_modeler_method`	*boolean*	
`consider_seasonal`	*boolean*	
`detect_outliers`	*boolean*	
`expert_outlier_additive`	*boolean*	
`expert_outlier_level_shift`	*boolean*	
`expert_outlier_innovational`	*boolean*	
`expert_outlier_level_shift`	*boolean*	

Table 115. timeseries node properties (continued).

`timeseries` Node Properties	Values	Property description
`expert_outlier_transient`	*boolean*	
`expert_outlier_seasonal_additive`	*boolean*	
`expert_outlier_local_trend`	*boolean*	
`expert_outlier_additive_patch`	*boolean*	

`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*	
`arima_sq`	*integer*	

`arima_transformation_type`	`None`
	`SquareRoot`
	`NaturalLog`	

`arima_include_constant`	*boolean*	
`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`	*boolean*	
`arima_outlier_level_shift`	*boolean*	
`arima_outlier_innovational`	*boolean*	
`arima_outlier_transient`	*boolean*	
`arima_outlier_seasonal_additive`	*boolean*	
`arima_outlier_local_trend`	*boolean*	
`arima_outlier_additive_patch`	*boolean*	
`conf_limit_pct`	*real*	

Table 115. timeseries node properties (continued).

`timeseries` Node Properties	Values	Property description
`max_lags`	*integer*	
`events`	*fields*	
`scoring_model_only`	*boolean*	Use for models with very large numbers (tens of thousands) of time series.

twostep Node 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.

Table 116. twostep node properties.

`twostep` Node 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. See the topic "Common Modeling Node Properties" on page 125 for more information.
`standardize`	*boolean*	
`exclude_outliers`	*boolean*	
`percentage`	*number*	
`cluster_num_auto`	*boolean*	
`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 14. Model Nugget Node Properties

Model nugget nodes share the same common properties as other nodes. See the topic “Common Node Properties” on page 56 for more information.

applyanomalydetection Node Properties

Anomaly Detection modeling nodes can be used to generate an Anomaly Detection model nugget. The scripting name of this model nugget is *applyanomalydetection*. For more information on scripting the modeling node itself, see the topic “anomalydetection Node Properties” on page 125.

Table 117. applyanomalydetection node properties.

`applyanomalydetection` Node 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`	*boolean*	Indicates whether records are discarded from the output or not.
`discard_anomalous_records`	*boolean*	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.

applyapriori Node Properties

Apriori modeling nodes can be used to generate an Apriori model nugget. The scripting name of this model nugget is *applyapriori*. For more information on scripting the modeling node itself, see the topic“apriori Node Properties” on page 127.

Table 118. applyapriori node properties.

`applyapriori` Node Properties	Values	Property description
`max_predictions`	*number (integer)*	
`ignore_unmatached`	*boolean*	
`allow_repeats`	*boolean*	

`check_basket`	`NoPredictions`
	`Predictions`
	`NoCheck`	

`criterion`	`Confidence`
	`Support`
	`RuleSupport`
	`Lift`
	`Deployability`	

applyautoclassifier Node Properties

Auto Classifier modeling nodes can be used to generate an Auto Classifier model nugget. The scripting name of this model nugget is *applyautoclassifier*. For more information on scripting the modeling node itself, see the topic"autoclassifier Node Properties" on page 127.

Table 119. applyautoclassifier node properties.

`applyautoclassifier` Node 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.

applyautocluster Node Properties

Auto Cluster modeling nodes can be used to generate an Auto Cluster model nugget. The scripting name of this model nugget is *applyautocluster*. No other properties exist for this model nugget. For more information on scripting the modeling node itself, see the topic "autocluster Node Properties" on page 129.

applyautonumeric Node Properties

Auto Numeric modeling nodes can be used to generate an Auto Numeric model nugget. The scripting name of this model nugget is *applyautonumeric*. For more information on scripting the modeling node itself, see the topic"autonumeric Node Properties" on page 131.

Table 120. applyautonumeric node properties.

`applyautonumeric` Node Properties	Values	Property description
`calculate_standard_error`	*boolean*	

applybayesnet Node Properties

Bayesian network modeling nodes can be used to generate a Bayesian network model nugget. The scripting name of this model nugget is *applybayesnet*. For more information on scripting the modeling node itself, see the topic "bayesnet Node Properties" on page 132.

Table 121. applybayesnet node properties.

`applybayesnet` Node Properties	Values	Property description
`all_probabilities`	*boolean*	

Table 121. applybayesnet node properties (continued).

applybayesnet Node Properties	Values	Property description
raw_propensity	*boolean*	
adjusted_propensity	*boolean*	
calculate_raw_propensities	*boolean*	
calculate_adjusted_propensities	*boolean*	

applyc50 Node Properties

C5.0 modeling nodes can be used to generate a C5.0 model nugget. The scripting name of this model nugget is *applyc50*. For more information on scripting the modeling node itself, see the topic"c50 Node Properties" on page 133.

Table 122. applyc50 node properties.

applyc50 Node Properties	Values	Property description
sql_generate	Never NoMissingValues	Used to set SQL generation options during rule set execution.
calculate_conf	*boolean*	Available when SQL generation is enabled; this property includes confidence calculations in the generated tree.
calculate_raw_propensities	*boolean*	
calculate_adjusted_propensities	*boolean*	

applycarma Node Properties

CARMA modeling nodes can be used to generate a CARMA model nugget. The scripting name of this model nugget is *applycarma*. No other properties exist for this model nugget. For more information on scripting the modeling node itself, see the topic "carma Node Properties" on page 134.

applycart Node 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 *applycart*. For more information on scripting the modeling node itself, see the topic"cart Node Properties" on page 135.

Table 123. applycart node properties.

applycart Node Properties	Values	Property description
sql_generate	Never MissingValues NoMissingValues	Used to set SQL generation options during rule set execution.
calculate_conf	*boolean*	Available when SQL generation is enabled; this property includes confidence calculations in the generated tree.
display_rule_id	*boolean*	Adds a field in the scoring output that indicates the ID for the terminal node to which each record is assigned.
calculate_raw_propensities	*boolean*	
calculate_adjusted_propensities	*boolean*	

applychaid Node Properties

CHAID modeling nodes can be used to generate a CHAID model nugget. The scripting name of this model nugget is *applychaid*. For more information on scripting the modeling node itself, see the topic“chaid Node Properties” on page 137.

Table 124. applychaid node properties.

applychaid Node Properties	Values	Property description
sql_generate	Never MissingValues	
calculate_conf	*boolean*	
display_rule_id	*boolean*	Adds a field in the scoring output that indicates the ID for the terminal node to which each record is assigned.
calculate_raw_propensities	*boolean*	
calculate_adjusted_propensities	*boolean*	

applycoxreg Node Properties

Cox modeling nodes can be used to generate a Cox model nugget. The scripting name of this model nugget is *applycoxreg*. For more information on scripting the modeling node itself, see the topic“coxreg Node Properties” on page 138.

Table 125. applycoxreg node properties.

applycoxreg Node 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	*boolean*	
cumulative_hazard	*boolean*	

applydecisionlist Node Properties

Decision List modeling nodes can be used to generate a Decision List model nugget. The scripting name of this model nugget is *applydecisionlist*. For more information on scripting the modeling node itself, see the topic“decisionlist Node Properties” on page 140.

Table 126. applydecisionlist node properties.

applydecisionlist Node Properties	Values	Property description
enable_sql_generation	*boolean*	When true, IBM SPSS Modeler will try to push back the Decision List model to SQL.
calculate_raw_propensities	*boolean*	
calculate_adjusted_propensities	*boolean*	

applydiscriminant Node Properties

Discriminant modeling nodes can be used to generate a Discriminant model nugget. The scripting name of this model nugget is *applydiscriminant*. For more information on scripting the modeling node itself, see the topic“discriminant Node Properties” on page 141.

Table 127. applydiscriminant node properties.

applydiscriminant Node Properties	Values	Property description
calculate_raw_propensities	*boolean*	
calculate_adjusted_propensities	*boolean*	

applyfactor Node Properties

PCA/Factor modeling nodes can be used to generate a PCA/Factor model nugget. The scripting name of this model nugget is *applyfactor*. No other properties exist for this model nugget. For more information on scripting the modeling node itself, see the topic “factor Node Properties” on page 142.

applyfeatureselection Node Properties

Feature Selection modeling nodes can be used to generate a Feature Selection model nugget. The scripting name of this model nugget is *applyfeatureselection*. For more information on scripting the modeling node itself, see the topic “featureselection Node Properties” on page 143.

Table 128. applyfeatureselection node properties.

applyfeatureselection Node 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.

applygeneralizedlinear Node Properties

Generalized Linear (genlin) modeling nodes can be used to generate a Generalized Linear model nugget. The scripting name of this model nugget is *applygeneralizedlinear*. For more information on scripting the modeling node itself, see the topic “genlin Node Properties” on page 145.

Table 129. applygeneralizedlinear node properties.

applygeneralizedlinear Node Properties	Values	Property description
calculate_raw_propensities	*boolean*	
calculate_adjusted_propensities	*boolean*	

applyglmm node Properties

GLMM modeling nodes can be used to generate a GLMM model nugget. The scripting name of this model nugget is *applyglmm*. For more information on scripting the modeling node itself, see the topic"glmm Node Properties" on page 148.

Table 130. applyglmm node properties.

`applyglmm` **Node Properties**	**Values**	**Property description**
`confidence`	`onProbability` `onIncrease`	Basis for computing scoring confidence value: highest predicted probability, or difference between highest and second highest predicted probabilities.
`score_category_probabilities`	*boolean*	If set to `True`, produces the predicted probabilities for categorical targets. A field is created for each category. Default is `False`.
`max_categories`	*integer*	Maximum number of categories for which to predict probabilities. Used only if `score_category_probabilities` is `True`.
`score_propensity`	*boolean*	If set to `True`, produces raw propensity scores (likelihood of "True" outcome) for models with flag targets. If partitions are in effect, also produces adjusted propensity scores based on the testing partition. Default is `False`.

applykmeans Node Properties

K-Means modeling nodes can be used to generate a K-Means model nugget. The scripting name of this model nugget is *applykmeans*. No other properties exist for this model nugget. For more information on scripting the modeling node itself, see the topic "kmeans Node Properties" on page 151.

applyknn Node Properties

KNN modeling nodes can be used to generate a KNN model nugget. The scripting name of this model nugget is *applyknn*. For more information on scripting the modeling node itself, see the topic"knn Node Properties" on page 152.

Table 131. applyknn node properties.

`applyknn` **Node Properties**	**Values**	**Property description**
`all_probabilities`	*boolean*	
`save_distances`	*boolean*	

applykohonen Node Properties

Kohonen modeling nodes can be used to generate a Kohonen model nugget. The scripting name of this model nugget is *applykohonen*. No other properties exist for this model nugget. For more information on scripting the modeling node itself, see the topic "kohonen Node Properties" on page 153.

applylinear Node Properties

Linear modeling nodes can be used to generate a Linear model nugget. The scripting name of this model nugget is *applylinear*. For more information on scripting the modeling node itself, see the topic"linear Node Properties" on page 154.

Table 132. applylinear node Properties.

`applylinear` Node Properties	Values	Property description
`use_custom_name`	*boolean*	
`custom_name`	*string*	
`enable_sql_generation`	*boolean*	

applylogreg Node Properties

Logistic Regression modeling nodes can be used to generate a Logistic Regression model nugget. The scripting name of this model nugget is *applylogreg*. For more information on scripting the modeling node itself, see the topic "logreg Node Properties" on page 155.

Table 133. applylogreg node properties.

`applylogreg` Node Properties	Values	Property description
`calculate_raw_propensities`	*boolean*	
`calculate_conf`	*boolean*	
`enable_sql_generation`	*boolean*	

applyneuralnet Node Properties

Neural Net modeling nodes can be used to generate a Neural Net model nugget. The scripting name of this model nugget is *applyneuralnet*. For more information on scripting the modeling node itself, see the topic"neuralnet Node Properties" on page 158.

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.

Table 134. applyneuralnet node properties.

`applyneuralnet` Node Properties	Values	Property description
`calculate_conf`	*boolean*	Available when SQL generation is enabled; this property includes confidence calculations in the generated tree.
`enable_sql_generation`	*boolean*	
`nn_score_method`	`Difference` `SoftMax`	
`calculate_raw_propensities`	*boolean*	
`calculate_adjusted_propensities`	*boolean*	

applyneuralnetwork Node Properties

Neural Network modeling nodes can be used to generate a Neural Network model nugget. The scripting name of this model nugget is *applyneuralnetwork*. For more information on scripting the modeling node itself, see the topic "neuralnetwork Node Properties" on page 159.

Table 135. applyneuralnetwork node properties.

applyneuralnetwork Node Properties	Values	Property description
use_custom_name	*boolean*	
custom_name	*string*	
confidence	onProbability onIncrease	
score_category_probabilities	*boolean*	
max_categories	*number*	
score_propensity	*boolean*	

applyquest Node Properties

QUEST modeling nodes can be used to generate a QUEST model nugget. The scripting name of this model nugget is *applyquest*. For more information on scripting the modeling node itself, see the topic"quest Node Properties" on page 161.

Table 136. applyquest node properties.

applyquest Node Properties	Values	Property description
sql_generate	Never MissingValues NoMissingValues	
calculate_conf	*boolean*	
display_rule_id	*boolean*	Adds a field in the scoring output that indicates the ID for the terminal node to which each record is assigned.
calculate_raw_propensities	*boolean*	
calculate_adjusted_propensities	*boolean*	

applyregression Node Properties

Linear Regression modeling nodes can be used to generate a Linear Regression model nugget. The scripting name of this model nugget is *applyregression*. No other properties exist for this model nugget. For more information on scripting the modeling node itself, see the topic "regression Node Properties" on page 162.

applyr Node Properties

R Building nodes can be used to generate an R model nugget. The scripting name of this model nugget is *applyr*. For more information on scripting the modeling node itself, see the topic"buildr Node Properties" on page 133.

Table 137. applyr node properties

applyr Node Properties	**Values**	**Property Description**
score_syntax	*string*	R scripting syntax for model scoring.
convert_flags	StringsAndDoubles LogicalValues	Option to convert flag fields.
convert_datetime	*boolean*	Option to convert variables with date or datetime formats to R date/time formats.
convert_datetime_class	POSIXct POSIXlt	Options to specify to what format variables with date or datetime formats are converted.
convert_missing	*boolean*	Option to convert missing values to R NA value.

applyselflearning Node 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 *applyselflearning*. For more information on scripting the modeling node itself, see the topic "slrm Node Properties" on page 164.

Table 138. applyselflearning node properties.

applyselflearning Node 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	*boolean*	Takes account of model reliability option on Settings tab.

applysequence Node Properties

Sequence modeling nodes can be used to generate a Sequence model nugget. The scripting name of this model nugget is *applysequence*. No other properties exist for this model nugget. For more information on scripting the modeling node itself, see the topic "sequence Node Properties" on page 163.

applysvm Node Properties

SVM modeling nodes can be used to generate an SVM model nugget. The scripting name of this model nugget is *applysvm*. For more information on scripting the modeling node itself, see the topic"svm Node Properties" on page 165.

Table 139. applysvm node properties.

applysvm Node Properties	**Values**	**Property description**
all_probabilities	*boolean*	

Table 139. applysvm node properties (continued).

`applysvm` Node Properties	Values	Property description
`calculate_raw_propensities`	*boolean*	
`calculate_adjusted_propensities`	*boolean*	

applytimeseries Node Properties

Time Series modeling nodes can be used to generate a Time Series model nugget. The scripting name of this model nugget is *applytimeseries*. For more information on scripting the modeling node itself, see the topic"timeseries Node Properties" on page 166.

Table 140. applytimeseries node properties.

`applytimeseries` Node Properties	Values	Property description
`calculate_conf`	*boolean*	
`calculate_residuals`	*boolean*	

applytwostep Node Properties

TwoStep modeling nodes can be used to generate a TwoStep model nugget. The scripting name of this model nugget is *applytwostep*. No other properties exist for this model nugget. For more information on scripting the modeling node itself, see the topic "twostep Node Properties" on page 168.

Chapter 15. 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 IBM SPSS Modeler application. Database models can also be created and manipulated through scripting using the properties described in this section.

Node Properties for Microsoft Modeling

Microsoft Modeling Node Properties

Common Properties

The following properties are common to the Microsoft database modeling nodes.

Table 141. Common Microsoft node properties.

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`	*boolean*	Specifies whether input data is in tabular or transactional format.
`inputs`	*[field field field]*	Input fields for tabular data.
`target`	*field*	Predicted field (not applicable to MS Clustering or Sequence Clustering nodes).
`unique_field`	*field*	Key field.
`msas_parameters`	*structured*	Algorithm parameters. See the topic "Algorithm Parameters" on page 180 for more information.
`with_drillthrough`	*boolean*	With Drillthrough option.

MS Decision Tree

There are no specific properties defined for nodes of type `mstree`. See the common Microsoft properties at the start of this section.

MS Clustering

There are no specific properties defined for nodes of type `mscluster`. See the common Microsoft properties at the start of this section.

MS Association Rules

The following specific properties are available for nodes of type `msassoc`:

Table 142. msassoc node properties.

`msassoc` Node Properties	**Values**	**Property Description**
`id_field`	*field*	Identifies each transaction in the data.
`trans_inputs`	*[field field field]*	Input fields for transactional data.

Table 142. msassoc node properties (continued).

msassoc Node Properties	Values	Property Description
transactional_target	*field*	Predicted field (transactional data).

MS Naive Bayes

There are no specific properties defined for nodes of type msbayes. See the common Microsoft properties at the start of this section.

MS Linear Regression

There are no specific properties defined for nodes of type msregression. See the common Microsoft properties at the start of this section.

MS Neural Network

There are no specific properties defined for nodes of type msneuralnetwork. See the common Microsoft properties at the start of this section.

MS Logistic Regression

There are no specific properties defined for nodes of type mslogistic. See the common Microsoft properties at the start of this section.

MS Time Series

There are no specific properties defined for nodes of type mstimeseries. See the common Microsoft properties at the start of this section.

MS Sequence Clustering

The following specific properties are available for nodes of type mssequencecluster:

Table 143. mssequencecluster node properties.

mssequencecluster Node 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.

These parameters are derived from SQL Server. To see the relevant parameters for each node:
1. Place a database source node on the canvas.
2. Open the database source node.
3. Select a valid source from the **Data source** drop-down list.
4. Select a valid table from the **Table name** list.
5. Click **OK** to close the database source node.
6. Attach the Microsoft database modeling node whose properties you want to list.

7. Open the database modeling node.
8. Select the **Expert** tab.

The available `msas_parameters` properties for this node are displayed.

Microsoft Model Nugget Properties

The following properties are for the model nuggets created using the Microsoft database modeling nodes.

MS Decision Tree

Table 144. MS Decision Tree properties.

`applymstree` Node 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`	*boolean*	Enables SQL generation.

MS Linear Regression

Table 145. MS Linear Regression properties.

`applymsregression` Node 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

Table 146. MS Neural Network properties.

`applymsneuralnetwork` Node 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

Table 147. MS Logistic Regression properties.

`applymslogistic` Node 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.

Table 147. MS Logistic Regression properties (continued).

`applymslogistic` Node Properties	Values	Description
`analysis_server_name`	*string*	Name of the Analysis server host.

MS Time Series

Table 148. MS Time Series properties.

`applymstimeseries` Node 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

Table 149. MS Sequence Clustering properties.

`applymssequencecluster` Node 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.

Table 150. Common Oracle node properties.

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`		

Table 150. Common Oracle node properties (continued).

Common Oracle Node Properties	Values	Property Description
`username`		
`password`		
`epassword`		
`use_model_name`	*boolean*	
`model_name`	*string*	Custom name for new model.
`use_partitioned_data`	*boolean*	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`	*boolean*	Enables or disables the Oracle automatic data preparation feature (11g databases only).
`costs`	*structured*	Structured property.
`mode`	`Simple` `Expert`	Causes certain properties to be ignored if set to `Simple`, as noted in the individual node properties.
`use_prediction_probability`	*boolean*	
`prediction_probability`	*string*	
`use_prediction_set`	*boolean*	

Oracle Naive Bayes

The following properties are available for nodes of type `oranb`.

Table 151. oranb node properties.

`oranb` **Node 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.

* Property ignored if `mode` is set to `Simple`.

Oracle Adaptive Bayes

The following properties are available for nodes of type `oraabn`.

Table 152. oraabn node properties.

`oraabn` **Node Properties**	Values	Property Description

`model_type`	`SingleFeature`
	`MultiFeature`
	`NaiveBayes`	

`use_execution_time_limit`	*boolean*	*
`execution_time_limit`	*integer*	Value must be greater than 0.*
`max_naive_bayes_predictors`	*integer*	Value must be greater than 0.*

Table 152. oraabn node properties (continued).

oraabn Node Properties	**Values**	**Property Description**
max_predictors	*integer*	Value must be greater than 0.*
priors	Data Equal Custom	
custom_priors	*structured*	Structured property.

* Property ignored if mode is set to Simple.

Oracle Support Vector Machines

The following properties are available for nodes of type orasvm.

Table 153. orasvm node properties.

orasvm Node 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	*boolean*	Gaussian kernel only.*
standard_deviation	*number*	Value must be greater than 0.*
use_epsilon	*boolean*	Regression models only.*
epsilon	*number*	Value must be greater than 0.*
use_complexity_factor	*boolean*	*
complexity_factor	*number*	*
use_outlier_rate	*boolean*	One-Class variant only.*
outlier_rate	*number*	One-Class variant only. 0.0–1.0.*
weights	Data Equal Custom	
custom_weights	*structured*	Structured property.

* Property ignored if mode is set to Simple.

Oracle Generalized Linear Models

The following properties are available for nodes of type oraglm.

Table 154. oraglm node properties.

oraglm Node Properties	Values	Property Description
normalization_method	zscore minmax none	
missing_value_handling	ReplaceWithMean UseCompleteRecords	
use_row_weights	*boolean*	*
row_weights_field	*field*	*
save_row_diagnostics	*boolean*	*
row_diagnostics_table	*string*	*
coefficient_confidence	*number*	*
use_reference_category	*boolean*	*
reference_category	*string*	*
ridge_regression	Auto Off On	*
parameter_value	*number*	*
vif_for_ridge	*boolean*	*

* Property ignored if mode is set to Simple.

Oracle Decision Tree

The following properties are available for nodes of type oradecisiontree.

Table 155. oradecisiontree node properties.

oradecisiontree Node Properties	Values	Property Description
use_costs	*boolean*	
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.*
term_minrec_split	*integer*	Value must be greater than 0.*
display_rule_ids	*boolean*	*

* Property ignored if mode is set to Simple.

Oracle O-Cluster

The following properties are available for nodes of type oraocluster.

Table 156. oraocluster node properties.

oraocluster Node Properties	Values	Property Description
max_num_clusters	*integer*	Value must be greater than 0.

Table 156. oraocluster node properties (continued).

`oraocluster` Node Properties	Values	Property Description
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 orakmeans.

Table 157. orakmeans node properties.

`orakmeans` Node 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 oranmf.

Table 158. oranmf node properties.

`oranmf` Node Properties	Values	Property Description
normalization_method	minmax none	
use_num_features	*boolean*	*
num_features	*integer*	0–1. Default value is estimated from the data by the algorithm.*
random_seed	*number*	*
num_iterations	*integer*	0–500.*
conv_tolerance	*number*	0.0–0.5.*
display_all_features	*boolean*	*

* Property ignored if mode is set to Simple.

Oracle Apriori

The following properties are available for nodes of type `oraapriori`.

Table 159. oraapriori node properties.

oraapriori Node 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	*boolean*	

Oracle Minimum Description Length (MDL)

There are no specific properties defined for nodes of type `oramdl`. See the common Oracle properties at the start of this section.

Oracle Attribute Importance (AI)

The following properties are available for nodes of type `oraai`.

Table 160. oraai node properties.

oraai Node Properties	Values	Property Description
custom_fields	*boolean*	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	*boolean*	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	*boolean*	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	*boolean*	When selection_mode is set to ImportanceLevel, specifies whether to select unimportant fields.
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 `applyoranb`.

Oracle Adaptive Bayes

There are no specific properties defined for nodes of type `applyoraabn`.

Oracle Support Vector Machines

There are no specific properties defined for nodes of type `applyorasvm`.

Oracle Decision Tree

The following properties are available for nodes of type `applyoradecisiontree`.

Table 161. applyoradecisiontree node properties.

`applyoradecisiontree` **Node Properties**	**Values**	**Property Description**
`use_costs`	*boolean*	
`display_rule_ids`	*boolean*	

Oracle O-Cluster

There are no specific properties defined for nodes of type `applyoraocluster`.

Oracle KMeans

There are no specific properties defined for nodes of type `applyorakmeans`.

Oracle NMF

The following property is available for nodes of type `applyoranmf`:

Table 162. applyoranmf node properties.

`applyoranmf` **Node Properties**	**Values**	**Property Description**
`display_all_features`	*boolean*	

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.

Table 163. Common ISW node properties.

Common ISW node Properties	Values	Property Description
`inputs`	*List of fields*	
`datasource`		
`username`		
`password`		
`epassword`		
`enable_power_options`	*boolean*	
`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 `db2imtree`.

Table 164. db2imtree node properties.

`db2imtree` Node Properties	Values	Property Description
`target`	*field*	
`perform_test_run`	*boolean*	
`use_max_tree_depth`	*boolean*	
`max_tree_depth`	*integer*	Value greater than 0.
`use_maximum_purity`	*boolean*	
`maximum_purity`	*number*	Number between 0 and 100.
`use_minimum_internal_cases`	*boolean*	
`minimum_internal_cases`	*integer*	Value greater than 1.
`use_costs`	*boolean*	
`costs`	*structured*	Structured property.

ISW Association

The following properties are available for nodes of type `db2imassoc`.

Table 165. db2imassoc node properties.

`db2imassoc` Node Properties	Values	Property Description
`use_transactional_data`	*boolean*	
`id_field`	*field*	
`content_field`	*field*	

Table 165. db2imassoc node properties (continued).

db2imassoc Node Properties	Values	Property Description
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	*boolean*	
item_constraints_type	Include Exclude	
use_taxonomy	*boolean*	
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	*boolean*	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 db2imsequence.

Table 166. db2imsequence node properties.

db2imsequence Node 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	*boolean*	
item_constraints_type	Include Exclude	
use_taxonomy	*boolean*	
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.

Table 166. db2imsequence node properties (continued).

db2imsequence Node Properties	**Values**	**Property Description**
load_taxonomy_to_table	*boolean*	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 Regression

The following properties are available for nodes of type db2imreg.

Table 167. db2imreg node properties.

db2imreg Node 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	*boolean*	
max_rsquared_value	*number*	Value between 0.0 and 1.0.
use_execution_time_limit	*boolean*	
execution_time_limit_mins	*integer*	Value greater than 0.
use_max_degree_polynomial	*boolean*	
max_degree_polynomial	*integer*	
use_intercept	*boolean*	
use_auto_feature_selection_method	*boolean*	
auto_feature_selection_method	normal adjusted	
use_min_significance_level	*boolean*	
min_significance_level	*number*	
use_min_significance_level	*boolean*	

The following properties apply only if regression_method is set to rbf.

Table 168. db2imreg node properties if regression_method is set to rbf.

db2imreg Node Properties	Values	Property Description
use_output_sample_size	*boolean*	If true, auto-set the value to the default.
output_sample_size	*integer*	Default is 2. Minimum is 1.
use_input_sample_size	*boolean*	If true, auto-set the value to the default.
input_sample_size	*integer*	Default is 2. Minimum is 1.

Table 168. db2imreg node properties if regression_method is set to rbf (continued).

`use_max_num_centers`	*boolean*	If true, auto-set the value to the default.
`max_num_centers`	*integer*	Default is 20. Minimum is 1.
`use_min_region_size`	*boolean*	If true, auto-set the value to the default.
`min_region_size`	*integer*	Default is 15. Minimum is 1.
`use_max_data_passes`	*boolean*	If true, auto-set the value to the default.
`max_data_passes`	*integer*	Default is 5. Minimum is 2.
`use_min_data_passes`	*boolean*	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 `db2imcluster`.

Table 169. db2imcluster node properties.

`db2imcluster` Node Properties	**Values**	**Property Description**

`cluster_method`	`demographic`
	`kohonen`
	`birch`	

`kohonen_num_rows`	*integer*	
`kohonen_num_columns`	*integer*	
`kohonen_passes`	*integer*	
`use_num_passes_limit`	*boolean*	
`use_num_clusters_limit`	*boolean*	
`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`	*boolean*	
`execution_time_limit_mins`	*integer*	Value greater than 0.
`min_data_percentage`	*number*	0–100%
`use_similarity_threshold`	*boolean*	
`similarity_threshold`	*number*	Value between 0.0 and 1.0.

ISW Naive Bayes

The following properties are available for nodes of type `db2imnbs`.

Table 170. db2imnb node properties.

db2imnb Node Properties	Values	Property Description
perform_test_run	*boolean*	
probability_threshold	*number*	Default is 0.001. Minimum value is 0; maximum value is 1.000
use_costs	*boolean*	
costs	*structured*	Structured property.

ISW Logistic Regression

The following properties are available for nodes of type db2imlog.

Table 171. db2imlog node properties.

db2imlog Node Properties	Values	Property Description
perform_test_run	*boolean*	
use_costs	*boolean*	
costs	*structured*	Structured property.

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 db2imtimeseries.

Table 172. db2imtimeseries node properties.

db2imtimeseries Node Properties	Values	Property Description
time	*field*	Integer, time, or date allowed.
targets	*list of fields*	
forecasting_algorithm	arima exponential_smoothing seasonal_trend_decomposition	
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 exponential_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 `applydb2imtree`.

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 `applydb2imreg`.

ISW Clustering

There are no specific properties defined for nodes of type `applydb2imcluster`.

ISW Naive Bayes

There are no specific properties defined for nodes of type `applydb2imnb`.

ISW Logistic Regression

There are no specific properties defined for nodes of type `applydb2imlog`.

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.

Table 173. Common Netezza node properties.

Common Netezza Node Properties	Values	Property Description
`custom_fields`	*boolean*	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.
`target`	*field*	Target field (continuous or categorical).
`record_id`	*field*	Field to be used as unique record identifier.
`use_upstream_connection`	*boolean*	If true (default), the connection details specified in an upstream node. Not used if `move_data_to_connection` is specified.

Table 173. Common Netezza node properties (continued).

Common Netezza Node Properties	Values	Property Description
move_data_connection	*boolean*	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	*boolean*	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	*boolean*	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 netezzadectree.

Table 174. netezzadectree node properties.

netezzadectree Node 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. Default is weight of 1 for all classes.
pruning_measure	Acc wAcc	Default is Acc (accuracy). Alternative wAcc (weighted accuracy) takes class weights into account while applying pruning.

Table 174. netezzadectree node properties (continued).

`netezzadectree` Node Properties	Values	Property Description
`prune_tree_options`	`allTrainingData` `partitionTrainingData` `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`	*boolean*	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 `netezzakmeans`.

Table 175. netezzakmeans node properties.

`netezzakmeans` Node 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 `netezzabayes`.

Table 176. netezzabayes node properties.

`netezzabayes` Node 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`	*boolean*	If true, displays additional progress information in a message dialog box.

Table 176. netezzabayes node properties (continued).

netezzabayes Node Properties	Values	Property Description
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 netezzanaivebayes.

Table 177. netezzanaivebayes node properties.

netezzanaivebayes Node Properties	Values	Property Description
compute_probabilities	*boolean*	If true, produces a confidence level (probability) field as well as the prediction field.
use_m_estimation	*boolean*	If true, uses m-estimation technique for avoiding zero probabilities during estimation.

Netezza KNN

The following properties are available for nodes of type netezzaknn.

Table 178. netezzaknn node properties.

netezzaknn Node Properties	Values	Property Description
weights	*structured*	Structured property used to assign weights to individual classes.
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	*boolean*	If true, standardizes measurements for continuous input fields before calculating distance values.
use_coresets	*boolean*	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 netezzadivcluster.

Table 179. netezzadivcluster node properties.

netezzadivcluster Node 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.

Table 179. netezzadivcluster node properties (continued).

`netezzadivcluster` Node Properties	Values	Property Description
`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 `netezzapca`.

Table 180. netezzapca node properties.

`netezzapca` Node Properties	Values	Property Description
`center_data`	*boolean*	If true (default), performs data centering (also known as "mean subtraction") before the analysis.
`perform_data_scaling`	*boolean*	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`	*boolean*	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 `netezzaregtree`.

Table 181. netezzaregtree node properties.

`netezzaregtree` Node 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.

Table 181. netezzaregtree node properties (continued).

netezzaregtree Node Properties	Values	Property Description
prune_tree_options	allTrainingData partitionTrainingData 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	*boolean*	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 netezzalineregression.

Table 182. netezzalineregression node properties.

netezzalineregression Node Properties	Values	Property Description
use_svd	*boolean*	If true, uses Singular Value Decomposition matrix instead of original matrix, for increased speed and numerical accuracy.
include_intercept	*boolean*	If true (default), increases overall accuracy of solution.
calculate_model_diagnostics	*boolean*	If true, calculates diagnostics on the model.

Netezza Time Series

The following properties are available for nodes of type netezzatimeseries.

Table 183. netezzatimeseries node properties.

netezzatimeseries Node 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.

Table 183. netezzatimeseries node properties (continued).

netezzatimeseries Node Properties	Values	Property Description
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.
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>
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.
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		

Table 183. netezzatimeseries node properties (continued).

netezzatimeseries Node Properties	**Values**	**Property Description**
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.
period	*integer*	Length of seasonal cycle, specified in conjunction with units_period. Not applicable for spectral analysis.
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	*boolean*	Indicates if historical values are to be included in output.
include_interpolated_values	*boolean*	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 `netezzaglm`.

Table 184. netezzaglm node properties.

`netezzaglm` Node Properties	**Values**	**Property Description**

`dist_family`	`bernoulli`
	`gaussian`
	`poisson`
	`negativebinomial`
	`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.
`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`.

Table 184. netezzaglm node properties (continued).

`netezzaglm` Node Properties	Values	Property Description
`link_params`	*number*	Link function parameter value to use. Only applicable if `link_function` is `power` or `oddspower`.
`interaction`	[{[*colnames1*],[*levels1*]},{[*colnames2*],[*levels2*]},...,{[*colnamesN*],[*levelsN*]},]	Specifies interactions between fields. *colnames* is a list of input fields, and *level* is always 0 for each field.
`intercept`	*boolean*	If `true`, includes the intercept in the model.

Netezza Model Nugget Properties

The following properties are common to Netezza database model nuggets.

Table 185. Common Netezza model nugget properties.

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.

Table 186. Script names of Netezza model nuggets.

Model Nugget	Script Name
Decision Tree	`applynetezzadectree`
K-Means	`applynetezzakmeans`
Bayes Net	`applynetezzabayes`
Naive Bayes	`applynetezzanaivebayes`
KNN	`applynetezzaknn`
Divisive Clustering	`applynetezzadivcluster`
PCA	`applynetezzapca`
Regression Tree	`applynetezzaregtree`
Linear Regression	`applynetezzalineregression`
Time Series	`applynetezzatimeseries`
Generalized Linear	`applynetezzaglm`

Chapter 16. 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.

analysis Node 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.

Table 187. analysis node properties.

`analysis` **Node properties**	**Data type**	**Property description**
`output_mode`	`Screen` `File`	Used to specify target location for output generated from the output node.
`use_output_name`	*boolean*	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*]	
`full_filename`	*string*	If disk, data, or HTML output, the name of the output file.
`coincidence`	*boolean*	
`performance`	*boolean*	
`evaluation_binary`	*boolean*	
`confidence`	*boolean*	
`threshold`	*number*	
`improve_accuracy`	*number*	
`inc_user_measure`	*boolean*	
`user_if`	*expr*	
`user_then`	*expr*	
`user_else`	*expr*	
`user_compute`	`[Mean Sum Min Max SDev]`	

dataaudit Node 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.

Table 188. dataaudit node properties.

dataaudit Node properties	Data type	Property description
custom_fields	*boolean*	
fields	*[field1 ... fieldN]*	
overlay	*field*	
display_graphs	*boolean*	Used to turn the display of graphs in the output matrix on or off.
basic_stats	*boolean*	
advanced_stats	*boolean*	
median_stats	*boolean*	
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	*boolean*	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	*boolean*	When the output_format is HTML, causes the output to be separated into pages.

Table 188. dataaudit node properties (continued).

`dataaudit` Node properties	Data type	Property description
`lines_per_page`	*number*	When used with `paginate_output`, specifies the lines per page of output.
`full_filename`	*string*	

matrix Node 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.

Table 189. matrix node properties.

`matrix` node properties	Data type	Property description

`fields`	`Selected`
	`Flags`
	`Numerics`	

`row`	*field*	
`column`	*field*	
`include_missing_values`	*boolean*	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`	*boolean*	
`use_output_name`	*boolean*	Specifies whether a custom output name is used.
`output_name`	*string*	If `use_output_name` is true, specifies the name to use.

Table 189. matrix node properties (continued).

matrix node properties	Data type	Property description
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. Both the Formatted and Delimited formats can take the modifier transposed, which transposes the rows and columns in the table.
paginate_output	*boolean*	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*	

means Node 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.

Table 190. means node properties.

means node 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	*boolean*	Specifies whether correlation labels are shown in output. This setting applies only when means_mode is set to BetweenFields.
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.

Table 190. means node properties (continued).

`means` node properties	Data type	Property description
`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`	*boolean*	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.

report Node 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.

Table 191. report node properties.

`report` node 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.

Table 191. report node properties (continued).

report node properties	**Data type**	**Property description**
use_output_name	*boolean*	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	*boolean*	
title	*string*	
lines_per_page	*number*	

Routput Node Properties

The R Output node enables you to analyze data and the results of model scoring using your own custom R script. The output of the analysis can be text or graphical. The output is added to the **Output** tab of the manager pane; alternatively, the output can be redirected to a file.

Table 192. Routput node properties.

Routput node properties	**Data type**	**Property description**
syntax	*string*	
convert_flags	StringsAndDoubles LogicalValues	
convert_datetime	*boolean*	
convert_datetime_class	POSIXct POSIXlt	
convert_missing	*boolean*	
output_name	Auto Custom	
custom_name	*string*	
output_to	Screen File	
output_type	Graph Text	
full_filename	*string*	
graph_file_type	HTML COU	
text_file_type	HTML TXT COU	

setglobals Node 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)`.

Table 193. setglobals node properties.

`setglobals` node properties	Data type	Property description
`globals`	`[Sum Mean Min Max SDev]`	Structured property
`clear_first`	*boolean*	
`show_preview`	*boolean*	

simeval Node Properties

The Simulation Evaluation node evaluates a specified predicted target field, and presents distribution and correlation information about the target field.

Table 194. simeval node properties.

`simeval` node properties	Data type	Property description
`target`	*field*	
`iteration`	*field*	
`presorted_by_iteration`	*boolean*	
`max_iterations`	*number*	
`tornado_fields`	*[field1...fieldN]*	
`plot_pdf`	*boolean*	
`plot_cdf`	*boolean*	
`show_ref_mean`	*boolean*	
`show_ref_median`	*boolean*	
`show_ref_sigma`	*boolean*	
`num_ref_sigma`	*number*	
`show_ref_pct`	*boolean*	
`ref_pct_bottom`	*number*	
`ref_pct_top`	*number*	
`show_ref_custom`	*boolean*	
`ref_custom_values`	*[number1...numberN]*	
`category_values`	`Category` `Probabilities` `Both`	
`category_groups`	`Categories` `Iterations`	
`create_pct_table`	*boolean*	

Table 194. simeval node properties (continued).

simeval node properties	Data type	Property description
pct_table	Quartiles Intervals Custom	
pct_intervals_num	*number*	
pct_custom_values	*[number1...numberN]*	

simfit Node Properties

The Simulation Fitting node examines the statistical distribution of the data in each field and generates (or updates) a Simulation Generate node, with the best fitting distribution assigned to each field. The Simulation Generate node can then be used to generate simulated data.

Table 195. simfit node properties.

simfit node properties	Data type	Property description
build	Node XMLExport Both	
use_source_node_name	*boolean*	
source_node_name	*string*	The custom name of the source node that is either being generated or updated.
use_cases	All LimitFirstN	
use_case_limit	*integer*	
fit_criterion	AndersonDarling KolmogorovSmirnov	
num_bins	*integer*	
parameter_xml_filename	*string*	
generate_parameter_import	*boolean*	

statistics Node Properties

The Statistics node provides basic summary information about numeric fields. It calculates summary statistics for individual fields and correlations between fields.

Table 196. statistics node properties.

statistics node properties	Data type	Property description
use_output_name	*boolean*	Specifies whether a custom output name is used.
output_name	*string*	If use_output_name is true, specifies the name to use.

Table 196. statistics node properties (continued).

statistics node properties	Data type	Property description
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*]	
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	*boolean*	
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.

statisticsoutput Node 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 IBM SPSS Statistics analytical procedures is available. This node requires a licensed copy of IBM SPSS Statistics.

The properties for this node are described under "statisticsoutput Node Properties" on page 228.

table Node 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.

Table 197. table node properties.

`table` node properties	Data type	Property description
`full_filename`	*string*	If disk, data, or HTML output, the name of the output file.
`use_output_name`	*boolean*	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`	*boolean*	Transposes the data before export so that rows represent fields and columns represent records.
`paginate_output`	*boolean*	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.
`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.
`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.

`decimal_separator`	`DEFAULT`
	`PERIOD`
	`COMMA`
		Sets the decimal separator for the field (applies only to fields with REAL storage).

Table 197. table node properties (continued).

table node 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).
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).
column_width	*integer*	Sets the column width for the field. A value of –1 will set column width to Auto.
justify	AUTO CENTER LEFT RIGHT	Sets the column justification for the field.

transform Node Properties

The Transform node allows you to select and visually preview the results of transformations before applying them to selected fields.

Table 198. transform node properties.

`transform` node 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`	*boolean*	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`	*boolean*	Indicates if the $\log_n$ transformation should be used.
`formula_log_n_offset`	*number*	
`formula_log_10`	*boolean*	Indicates if the $\log_{10}$ transformation should be used.
`formula_log_10_offset`	*number*	
`formula_exponential`	*boolean*	Indicates if the exponential transformation (e^x) should be used.
`formula_square_root`	*boolean*	Indicates if the square root transformation should be used.
`use_output_name`	*boolean*	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.
`paginate_output`	*boolean*	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 17. Export Node Properties

Common Export Node Properties

The following properties are common to all export nodes.

Table 199. Common export node properties.

Property	Values	Property description
`publish_path`	*string*	Enter the rootname name to be used for the published image and parameter files.
`publish_metadata`	*boolean*	Specifies if a metadata file is produced that describes the inputs and outputs of the image and their data models.
`publish_use_parameters`	*boolean*	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.

asexport Node Properties

The Analytic Server export enables you to run a stream on Hadoop Distributed File System (HDFS).

Table 200. asexport node properties.

`asexport` node properties	Data type	Property description
`data_source`	*string*	The name of the data source.
`export_mode`	*string*	Specifies whether to **`append`** exported data to the existing data source, or to **`overwrite`** the existing data source.
`host`	*string*	The name of the Analytic Server host.
`port`	*integer*	The port on which the Analytic Server is listening.
`tenant`	*string*	In a multi-tenant environment, the name of the tenant to which you belong. In a single-tenant environment, this defaults to **`ibm`**.
`set_credentials`	*boolean*	If user authentication on the Analytic Server is the same as on SPSS Modeler server, set this to **`false`**. Otherwise, set to **`true`**.
`user_name`	*string*	The user name for logging in to the Analytic Server. Only needed if `set_credentials` is true.
`password`	*string*	The password for logging in to the Analytic Server. Only needed if `set_credentials` is true.

cognosexport Node 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.

Table 201. cognosexport node properties.

`cognosexport` node 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 `""` *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 `databaseexport` in the next section, with the exception that the `datasource` property is not valid.

tm1export Node Properties

The IBM Cognos TM1 Export node exports data in a format that can be read by Cognos TM1 databases.

Table 202. tm1export node properties.

tm1export node properties	Data type	Property description

pm_host	*string*	The host name. For example:
		`set TM1_import.pm_host=`
		`'http://9.191.86.82:9510/pmhub/pm'`

tm1_connection	*{"field","field", ... ,"field"}*	A list property containing the connection details for the TM1 server. The format is:
		`{ "TM1_Server_Name"`
		`"tm1_username" "tm1_ password"}`
		`: set TM1_import.tm1_connection=`
		`['Planning Sample' admin apple]`

selected_cube	*field*	The name of the cube to which you are exporting data. For example:
		`:set TM1_export.selected_cube=`
		`'plan_BudgetPlan'`

spssfield_tm1element_mapping	*list*	The tm1 element to be mapped to must be part of the column dimension for selected cube view.
		Format: `[{"param1", "value"},...,{"paramN",`
		`"value"}]`
		For example:
		`:set TM1_export.spssfield_`
		`tm1element_mapping = [{"plan_version",`
		`"plan_version"},{"plan_department",`
		`"plan_department"}]`

databaseexport Node 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.

Table 203. databaseexport node properties.

databaseexport node 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. See the topic "Generating an Encoded Password" on page 47 for more information.
table_name	*string*	

write_mode	`Create`
	`Append`
	`Merge`	

Table 203. databaseexport node properties (continued).

databaseexport node properties	**Data type**	**Property description**
map	*string*	Maps a stream field name to a database column name (valid only if write_mode is Merge). 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	
drop_existing_table	*boolean*	
delete_existing_rows	*boolean*	
default_string_size	*integer*	
type		Structured property used to set the schema type.
generate_import	*boolean*	
use_custom_create_table_command	*boolean*	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	*boolean*	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.
not_logged	*boolean*	
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*	

Table 203. databaseexport node properties (continued).

databaseexport node properties	**Data type**	**Property description**
specify_data_file	*boolean*	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	*boolean*	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	*boolean*	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.
logfile_name	*string*	
check_table_size	*boolean*	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	*boolean*	Specifies whether a given field is a primary key.
use_custom_create_index_command	*boolean*	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	*boolean*	Used to enable or disable custom SQL for a specific index.
indexes.INDEXNAME.custom_create_ ommand		Specifies the custom SQL used for the specified index.
indexes.INDEXNAME.remove	*boolean*	If true, removes the specified index from the set of indexes.
table_space	*string*	Specifies the table space that will be created.
use_partition	*boolean*	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.

Table 204. databaseexport node properties using compression features.

`databaseexport` node properties	**Data type**	**Property description**
`use_compression`	*boolean*	If set to `true`, creates tables for export with compression.
`compression_mode`	`Row` `Page`	Sets the level of compression for SQL Server databases.

	`Default`
	`Direct_Load_Operations`
	`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.

datacollectionexport Node Properties

The IBM SPSS Data Collection export node outputs data in the format used by IBM SPSS Data Collection market research software. The IBM SPSS Data Collection Data Library must be installed to use this node.

Table 205. datacollectionexport node properties.

`datacollectionexport` node properties	**Data type**	**Property description**
`metadata_file`	*string*	The name of the metadata file to export.
`merge_metadata`	`Overwrite` `MergeCurrent`	
`enable_system_variables`	*boolean*	Specifies whether the exported *.mdd* file should include IBM SPSS Data Collection system variables.
`casedata_file`	*string*	The name of the *.sav* file to which case data is exported.
`generate_import`	*boolean*	

excelexport Node 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.

Table 206. excelexport node properties.

`excelexport` node properties	**Data type**	**Property description**
`full_filename`	*string*	

Table 206. excelexport node properties (continued).

excelexport node properties	Data type	Property description
excel_file_type	Excel2003 Excel2007	
export_mode	Create Append	
inc_field_names	*boolean*	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.
launch_application	*boolean*	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	*boolean*	Specifies whether an Excel Import node should be generated that will read the exported data file.

outputfile Node 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.

Table 207. outputfile node properties.

outputfile node properties	Data type	Property description
full_filename	*string*	Name of output file.
write_mode	Overwrite Append	
inc_field_names	*boolean*	
use_newline_after_records	*boolean*	
delimit_mode	Comma Tab Space Other	
other_delimiter	*char*	
quote_mode	None Single Double Other	
other_quote	*boolean*	
generate_import	*boolean*	

Table 207. outputfile node properties (continued).

`outputfile` node properties	Data type	Property description
encoding	StreamDefault SystemDefault "UTF-8"	

sasexport Node 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.

Table 208. sasexport node properties.

`sasexport` node 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	*boolean*	

statisticsexport Node Properties

The Statistics Export node outputs data in IBM SPSS Statistics *.sav* format. The *.sav* files can be read by IBM 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 “statisticsexport Node Properties” on page 228.

xmlexport Node 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.

Table 209. xmlexport node properties.

`xmlexport` node properties	Data type	Property description
full_filename	*string*	(required) Full path and file name of XML export file.
use_xml_schema	*boolean*	Specifies whether to use an XML schema (XSD or DTD file) to control the structure of the exported data.

Table 209. xmlexport node properties (continued).

`xmlexport` node properties	Data type	Property description
`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`	*boolean*	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.

Chapter 18. IBM SPSS Statistics Node Properties

statisticsimport Node 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.

Table 210. statisticsimport node properties.

`statisticsimport` node properties	Data type	Property description
`full_filename`	*string*	The complete filename, including path.
`password`	*string*	The password. The `password` parameter must be set before the `file_encrypted` parameter.
`file_encrypted`	*flag*	Whether or not the file is password protected.
`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 IBM SPSS Statistics field format information when importing.

statisticstransform Node 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 IBM SPSS Statistics.

Table 211. statisticstransform node properties.

`statisticstransform` node properties	Data type	Property description
`syntax`	*string*	
`check_before_saving`	*boolean*	Validates the entered syntax before saving the entries. Displays an error message if the syntax is invalid.
`default_include`	*boolean*	See the topic “filter Node Properties” on page 99 for more information.
`include`	*boolean*	See the topic “filter Node Properties” on page 99 for more information.
`new_name`	*string*	See the topic “filter Node Properties” on page 99 for more information.

statisticsmodel Node 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 IBM SPSS Statistics.

`statisticsmodel` **node properties**	**Data type**	**Property description**
`syntax`	*string*	
`default_include`	*boolean*	See the topic "filter Node Properties" on page 99 for more information.
`include`	*boolean*	See the topic "filter Node Properties" on page 99 for more information.
`new_name`	*string*	See the topic "filter Node Properties" on page 99 for more information.

statisticsoutput Node 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 IBM SPSS Statistics analytical procedures is available. This node requires a licensed copy of IBM SPSS Statistics.

Table 212. statisticsoutput node properties.

`statisticsoutput` **node properties**	**Data type**	**Property description**
`mode`	`Dialog` `Syntax`	Selects "IBM SPSS Statistics dialog" option or Syntax Editor
`syntax`	*string*	
`use_output_name`	*boolean*	
`output_name`	*string*	
`output_mode`	`Screen` `File`	
`full_filename`	*string*	

`file_type`	`HTML`
	`SPV`
	`SPW`	

statisticsexport Node Properties

The Statistics Export node outputs data in IBM SPSS Statistics *.sav* format. The *.sav* files can be read by IBM SPSS Statistics Base and other products. This is also the format used for cache files in IBM SPSS Modeler.

Table 213. statisticsexport node properties.

statisticsexport node properties	**Data type**	**Property description**
full_filename	*string*	
file_type	Standard Compressed	Save file in *sav* or *zsav* format.
encrypt_file	*flag*	Whether or not the file is password protected.
password	*string*	The password.
launch_application	*boolean*	
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	*boolean*	

Chapter 19. SuperNode Properties

Properties that are specific to SuperNodes are described in the following tables. Note that common node properties also apply to SuperNodes.

Table 214. Terminal supernode properties.

Property name	Property type/List of values	Property description
`execute_method`	`Script` `Normal`	
`script`	*string*	
`script_language`	`Python` `Legacy`	Sets the scripting language for the SuperNode script.

SuperNode Parameters

You can use scripts to create or set SuperNode parameters using the same functions that are used to modify stream parameters. See the topic "Stream, Session, and SuperNode Parameters" on page 40 for more information.

Setting Properties for Encapsulated Nodes

In order to set the properties on nodes within the SuperNode, you must access the diagram owned by that SuperNode, and then use the various `find` methods (such as `findByName()` and `findByID()`) to locate the nodes. For example, in a SuperNode script that includes a single Type node:

```
supernode = modeler.script.supernode()
diagram = supernode.getCompositeProcessorDiagram()
# Find the type node within the supernode internal diagram
typenode = diagram.findByName("type", None)
typenode.setKeyedProperty("direction", "Drug", "Input")
typenode.setKeyedProperty("direction", "Age", "Target")
```

Limitations of SuperNode scripts. SuperNodes cannot manipulate other streams and cannot change the current stream.

Appendix A. Node names reference

This section provides a reference for the scripting names of the nodes in IBM SPSS Modeler.

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. See the topic Chapter 14, "Model Nugget Node Properties," on page 169 for more information.

Note: Under normal circumstances, referencing models by both name *and* type is recommended to avoid confusion.

Table 215. Model Nugget Names (Modeling Palette).

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

Table 215. Model Nugget Names (Modeling Palette) (continued).

Model name	Model
sequence	Sequence
slrm	Self-learning response model
statisticsmodel	IBM SPSS Statistics model
svm	Support vector machine
timeseries	Time Series
twostep	TwoStep

Table 216. 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
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

Table 216. Model Nugget Names (Database Modeling Palette) (continued).

Model name	**Model**
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:

1. From the menus choose:

   Tools > **User Options**
2. Click the **Notifications** tab.
3. 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.

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 "Graph Node Common Properties" on page 113 and Chapter 16, "Output Node Properties," on page 205.

Table 217. Output object types and the nodes that create them.

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

Table 217. Output object types and the nodes that create them (continued).

Output object type	**Node**
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

Appendix B. Migrating from legacy scripting to Python scripting

Legacy script migration overview

This section provides a summary of the differences between Python and legacy scripting in IBM SPSS Modeler, and provides information about how to migrate your legacy scripts to Python scripts. In this section you will find a list of standard SPSS Modeler legacy commands and the equivalent Python commands.

General differences

Legacy scripting owes much of its design to OS command scripts. Legacy scripting is line oriented, and although there are some block structures, for example `if...then...else...endif` and `for...endfor`, indentation is generally not significant.

In Python scripting, indentation is significant and lines belonging to the same logical block must be indented by the same level.

Note: You must take care when copying and pasting Python code. A line that is indented using tabs might look the same in the editor as a line that is indented using spaces. However, the Python script will generate an error because the lines are not considered as equally indented.

The scripting context

The scripting context defines the environment that the script is being executed in, for example the stream or SuperNode that executes the script. In legacy scripting the context is implicit, which means, for example, that any node references in a stream script are assumed to be within the stream that executes the script.

In Python scripting, the scripting context is provided explicitly via the `modeler.script` module. For example, a Python stream script can access the stream that executes the script with the following code:

```
s = modeler.script.stream()
```

Stream related functions can then be invoked through the returned object.

Commands versus functions

Legacy scripting is command oriented. This mean that each line of script typically starts with the command to be run followed by the parameters, for example:

```
connect 'Type':typenode to :filternode
rename :derivenode as "Compute Total"
```

Python uses functions that are usually invoked through an object (a module, class or object) that defines the function, for example:

```
stream = modeler.script.stream()
typenode = stream.findByName("type", "Type)
filternode = stream.findByName("filter", None)
stream.link(typenode, filternode)
derive.setLabel("Compute Total")
```

Literals and comments

Some literal and comment commands that are commonly used in IBM SPSS Modeler have equivalent commands in Python scripting. This might help you to convert your existing SPSS Modeler Legacy scripts to Python scripts for use in IBM SPSS Modeler 16.

Table 218. Legacy scripting to Python scripting mapping for literals and comments.

Legacy scripting	**Python scripting**
Integer, for example `4`	Same
Float, for example `0.003`	Same
Single quoted strings, for example `'Hello'`	Same **Note:** String literals containing non-ASCII characters must be prefixed by a `u` to ensure that they are represented as Unicode.
Double quoted strings, for example `"Hello again"`	Same **Note:** String literals containing non-ASCII characters must be prefixed by a `u` to ensure that they are represented as Unicode.

Long strings, for example
`"""This is a string`
`that spans multiple`
`lines"""`
	Same

Lists, for example `[1 2 3]`	`[1, 2, 3]`
Variable reference, for example `set x = 3`	`x = 3`

Line continuation (\), for example
`set x = [1 2 \`
`  3 4]`
	`x = [ 1, 2,\`
	`  3, 4]`

Block comment, for example
`/* This is a long comment`
`over a line. */`
	`""" This is a long comment`
	`over a line. """`

Line comment, for example `set x = 3  # make x 3`	`x = 3  # make x 3`
`undef`	`None`
`true`	`True`
`false`	`False`

Operators

Some operator commands that are commonly used in IBM SPSS Modeler have equivalent commands in Python scripting. This might help you to convert your existing SPSS Modeler Legacy scripts to Python scripts for use in IBM SPSS Modeler 16.

Table 219. Legacy scripting to Python scripting mapping for operators.

Legacy scripting	**Python scripting**

`NUM1 + NUM2`
`LIST + ITEM`
`LIST1 + LIST2`
	`NUM1 + NUM2`
	`LIST.append(ITEM)`
	`LIST1.extend(LIST2)`

`NUM1 – NUM2`
`LIST - ITEM`
	`NUM1 – NUM2`
	`LIST.remove(ITEM)`

`NUM1 * NUM2`	`NUM1 * NUM2`
`NUM1 / NUM2`	`NUM1 / NUM2`

Table 219. Legacy scripting to Python scripting mapping for operators (continued).

Legacy scripting	**Python scripting**

`=`
`==`
	`==`

`/=`
`/==`
	`!=`

`X ** Y`	`X ** Y`

`X < Y`
`X <= Y`
`X > Y`
`X >= Y`
	`X < Y`
	`X <= Y`
	`X > Y`
	`X >= Y`

`X div Y`
`X rem Y`
`X mod Y`
	`X // Y`
	`X % Y`
	`X % Y`

`and`
`or`
`not(EXPR)`
	`and`
	`or`
	`not EXPR`

Conditionals and looping

Some conditional and looping commands that are commonly used in IBM SPSS Modeler have equivalent commands in Python scripting. This might help you to convert your existing SPSS Modeler Legacy scripts to Python scripts for use in IBM SPSS Modeler 16.

Table 220. Legacy scripting to Python scripting mapping for conditionals and looping.

Legacy scripting	**Python scripting**

`for VAR from INT1 to INT2`
`  ...`
`endfor`
	`for VAR in range(INT1, INT2):`
	`  ...`
	or
	`VAR = INT1`
	`while VAR <= INT2:`
	`  ...`
	`  VAR += 1`

`for VAR in LIST`
`  ...`
`endfor`
	`for VAR in LIST:`
	`  ...`

`for VAR in_fields_to NODE`
`  ...`
`endfor`
	`for VAR in NODE.getInputDataModel():`
	`  ...`

`for VAR in_fields_at NODE`
`  ...`
`endfor`
	`for VAR in NODE.getOutputDataModel():`
	`  ...`

`if...then`
`  ...`
`elseif...then`
`  ...`
`else`
`   ...`
`endif`
	`if ...:`
	`  ...`
	`elif ...:`
	`  ...`
	`else:`
	`  ...`

`with TYPE OBJECT`
`  ...`
`endwith`
	No equivalent

`var VAR1`	Variable declaration is not required

Variables

In legacy scripting, variables are declared before they are referenced, for example:

```
var mynode
set mynode = create typenode at 96 96
```

In Python scripting, variables are created when they are first referenced, for example:

```
mynode = stream.createAt("type", "Type", 96, 96)
```

In legacy scripting, references to variables must be explicitly removed using the ^ operator, for example:

```
var mynode
set mynode = create typenode at 96 96
set ^mynode.direction."Age" = Input
```

Like most scripting languages, this is not necessary is Python scripting, for example:

```
mynode = stream.createAt("type", "Type", 96, 96)
mynode.setKeyedPropertyValue("direction","Age","Input")
```

Node, output and model types

In legacy scripting, the different object types (node, output, and model) typically have the type appended to the type of object. For example, the Derive node has the type `derivenode`:

```
set feature_name_node = create derivenode at 96 96
```

The IBM SPSS Modeler API in Python does not include the `node` suffix, so the Derive node has the type `derive`, for example:

```
feature_name_node = stream.createAt("derive", "Feature", 96, 96)
```

The only difference in type names in legacy and Python scripting is the lack of the type suffix.

Property names

Property names are the same in both legacy and Python scripting. For example, in the Variable File node, the property that defines the file location is `full_filename` in both scripting environments.

Node references

Many legacy scripts use an implicit search to find and access the node to be modified. For example, the following commands search the current stream for a Type node with the label "Type", then set the direction (or modeling role) of the "Age" field to Input and the "Drug" field to be Target, that is the value to be predicted:

```
set 'Type':typenode.direction."Age" = Input
set 'Type':typenode.direction."Drug" = Target
```

In Python scripting, node objects have to be located explicitly before calling the function to set the property value, for example:

```
typenode = stream.findByType("type", "Type")
typenode.setKeyedPropertyValue("direction", "Age", "Input")
typenode.setKeyedPropertyValue("direction", "Drug", "Target")
```

Note: In this case, "Target" must be in string quotes.

Python scripts can alternatively use the `ModelingRole` enumeration in the `modeler.api` package.

Although the Python scripting version can be more verbose, it leads to better runtime performance because the search for the node is usually only done once. In the legacy scripting example, the search for the node is done for each command.

Finding nodes by ID is also supported (the node ID is visible in the Annotations tab of the node dialog). For example, in legacy scripting:

```
# id65EMPB9VL87 is the ID of a Type node
set @id65EMPB9VL87.direction."Age" = Input
```

The following script shows the same example in Python scripting:

```
typenode = stream.findByID("id65EMPB9VL87")
typenode.setKeyedPropertyValue("direction", "Age", "Input")
```

Getting and setting properties

Legacy scripting uses the `set` command to assign a value. The term following the `set` command can be a property definition. The following script shows two possible script formats for setting a property:

```
set <node reference>.<property> = <value>
set <node reference>.<keyed-property>.<key> = <value>
```

In Python scripting, the same result is achieved by using the functions `setPropertyValue()` and `setKeyedPropertyValue()`, for example:

```
object.setPropertyValue(property, value)
object.setKeyedPropertyValue(keyed-property, key, value)
```

In legacy scripting, accessing property values can be achieved using the `get` command, for example:

```
var n v
set n = get node :filternode
set v = ^n.name
```

In Python scripting, the same result is achieved by using the function `getPropertyValue()`, for example:

```
n = stream.findByName("filter", None)
v = n.getPropertyValue("name")
```

Editing streams

In legacy scripting, the `create` command is used to create a new node, for example:

```
var agg select
set agg = create aggregatenode at 96 96
set select = create selectnode at 164 96
```

In Python scripting, streams have various methods for creating nodes, for example:

```
stream = modeler.script.stream()
agg = stream.createAt("aggregate", "Aggregate", 96, 96)
select = stream.createAt("select", "Select", 164, 96)
```

In legacy scripting, the `connect` command is used to create links between nodes, for example:

```
connect ^agg to ^select
```

In Python scripting, the `link` method is used to create links between nodes, for example:

```
stream.link(agg, select)
```

In legacy scripting, the `disconnect` command is used to remove links between nodes, for example:

```
disconnect ^agg from ^select
```

In Python scripting, the `unlink` method is used to remove links between nodes, for example:

```
stream.unlink(agg, select)
```

In legacy scripting, the `position` command is used to position nodes on the stream canvas or between other nodes, for example:

```
position ^agg at 256 256
position ^agg between ^myselect and ^mydistinct
```

In Python scripting, the same result is achieved by using two separate methods; `setXYPosition` and `setPositionBetween`. For example:

```
agg.setXYPosition(256, 256)
agg.setPositionBetween(myselect, mydistinct)
```

Node operations

Some node operation commands that are commonly used in IBM SPSS Modeler have equivalent commands in Python scripting. This might help you to convert your existing SPSS Modeler Legacy scripts to Python scripts for use in IBM SPSS Modeler 16.

Table 221. Legacy scripting to Python scripting mapping for node operations.

Legacy scripting	**Python scripting**

`create` *`nodespec`* `at x y`	*`stream`*`.create(`*`type`*`, `*`name`*`)`
	`stream``.createAt(`*`type`*`, `*`name`*`, x, y)`
	`stream``.createBetween(`*`type`*`, `*`name`*`, preNode, postNode)`
	`stream``.createModelApplier(`*`model`*`, `*`name`*`)`

`connect` *`fromNode`* `to` *`toNode`*	*`stream`*`.link(`*`fromNode`*`, `*`toNode`*`)`
`delete` *`node`*	*`stream`*`.delete(`*`node`*`)`
`disable` *`node`*	*`stream`*`.setEnabled(`*`node`*`, False)`
`enable` *`node`*	*`stream`*`.setEnabled(`*`node`*`, True)`

`disconnect` *`fromNode`* `from` *`toNode`*	*`stream`*`.unlink(`*`fromNode`*`, `*`toNode`*`)`
	`stream``.disconnect(`*`node`*`)`

`duplicate` *`node`*	*`node`*`.duplicate()`

`execute` *`node`*	*`stream`*`.runSelected(`*`nodes`*`, `*`results`*`)`
	`stream``.runAll(`*`results`*`)`

`flush` *`node`*	*`node`*`.flushCache()`
`position` *`node`* `at x y`	*`node`*`.setXYPosition(x, y)`
`position` *`node`* `between` *`node1`* `and` *`node2`*	*`node`*`.setPositionBetween(`*`node1`*`, `*`node2`*`)`
`rename` *`node`* `as` *`name`*	*`node`*`.setLabel(`*`name`*`)`

Looping

In legacy scripting, there are two main looping options that are supported:

- *Counted* loops, where an index variable moves between two integer bounds.
- *Sequence* loops that loop through a sequence of values, binding the current value to the loop variable.

The following script is an example of a counted loop in legacy scripting:

```
for i from 1 to 10
 println ^i
endfor
```

The following script is an example of a sequence loop in legacy scripting:

```
var items
set items = [a b c d]

for i in items
 println ^i
endfor
```

There are also other types of loops that can be used:

- Iterating through the models in the models palette, or through the outputs in the outputs palette.
- Iterating through the fields coming into or out of a node.

Python scripting also supports different types of loops. The following script is an example of a counted loop in Python scripting:

```
i = 1
while i <= 10:
 print i
 i += 1
```

The following script is an example of a sequence loop in Python scripting:

```
items = ["a", "b", "c", "d"]
for i in items:
 print i
```

The sequence loop is very flexible, and when it is combined with IBM SPSS Modeler API methods it can support the majority of legacy scripting use cases. The following example shows how to use a sequence loop in Python scripting to iterate through the fields that come out of a node:

```
node = modeler.script.stream().findByType("filter", None)
for column in node.getOutputDataModel().columnIterator():
 print column.getColumnName()
```

Executing streams

During stream execution, model or output objects that are generated are added to one of the object managers. In legacy scripting, the script must either locate the built objects from the object manager, or access the most recently generated output from the node that generated the output.

Stream execution in Python is different, in that any model or output objects that are generated from the execution are returned in a list that is passed to the execution function. This makes it simpler to access the results of the stream execution.

Legacy scripting supports three stream execution commands:

- `execute_all` executes all executable terminal nodes in the stream.
- `execute_script` executes the stream script regardless of the setting of the script execution.
- `execute` *`node`* executes the specified node.

Python scripting supports a similar set of functions:

- *`stream`*`.runAll(`*`results-list`*`)` executes all executable terminal nodes in the stream.
- *`stream`*`.runScript(`*`results-list`*`)` executes the stream script regardless of the setting of the script execution.
- *`stream`*`.runSelected(`*`node-array`*`,` *`results-list`*`)` executes the specified set of nodes in the order that they are supplied.
- *`node`*`.run(`*`results-list`*`)` executes the specified node.

In legacy script, a stream execution can be terminated using the `exit` command with an optional integer code, for example:

```
exit 1
```

In Python scripting, the same result can be achieved with the following script:

```
modeler.script.exit(1)
```

Accessing objects through the file system and repository

In legacy scripting, you can open an existing stream, model or output object using the `open` command, for example:

```
var s
set s = open stream "c:/my streams/modeling.str"
```

In Python scripting, there is the `TaskRunner` class that is accessible from the session and can be used to perform similar tasks, for example:

```
taskrunner = modeler.script.session().getTaskRunner()
s = taskrunner.openStreamFromFile("c:/my streams/modeling.str", True)
```

To save an object in legacy scripting, you can use the `save` command, for example:

```
save stream s as "c:/my streams/new_modeling.str"
```

The equivalent Python script approach would be using the `TaskRunner` class, for example:

```
taskrunner.saveStreamToFile(s, "c:/my streams/new_modeling.str")
```

IBM SPSS Collaboration and Deployment Services Repository based operations are supported in legacy scripting through the `retrieve` and `store` commands, for example:

```
var s
set s = retrieve stream "/my repository folder/my_stream.str"
store stream ^s as "/my repository folder/my_stream_copy.str"
```

In Python scripting, the equivalent functionality would be accessed through the Repository object that is associated with the session, for example:

```
session = modeler.script.session()
repo = session.getRepository()
s = repo.retrieveStream("/my repository folder/my_stream.str", None, None, True)
repo.storeStream(s, "/my repository folder/my_stream_copy.str", None)
```

Note: Repository access requires that the session has been configured with a valid repository connection.

Stream operations

Some stream operation commands that are commonly used in IBM SPSS Modeler have equivalent commands in Python scripting. This might help you to convert your existing SPSS Modeler Legacy scripts to Python scripts for use in IBM SPSS Modeler 16.

Table 222. Legacy scripting to Python scripting mapping for stream operations.

Legacy scripting	**Python scripting**
`create stream` *`DEFAULT_FILENAME`*	*`taskrunner`*`.createStream(`*`name`*`, `*`autoConnect`*`, `*`autoManage`*`)`
`close stream`	*`stream`*`.close()`
`clear stream`	*`stream`*`.clear()`
`get stream` *`stream`*	No equivalent
`load stream` *`path`*	No equivalent
`open stream` *`path`*	*`taskrunner`*`.openStreamFromFile(`*`path`*`, `*`autoManage`*`)`
`save` *`stream`* `as` *`path`*	*`taskrunner`*`.saveStreamToFile(`*`stream`*`, `*`path`*`)`

Table 222. Legacy scripting to Python scripting mapping for stream operations (continued).

Legacy scripting	Python scripting
`retreive stream` *path*	*repository*`.retreiveStream(`*path*, *version*, *label*, *autoManage*`)`
`store` *stream* `as` *path*	*repository*`.storeStream(`*stream*, *path*, *label*`)`

Model operations

Some model operation commands that are commonly used in IBM SPSS Modeler have equivalent commands in Python scripting. This might help you to convert your existing SPSS Modeler Legacy scripts to Python scripts for use in IBM SPSS Modeler 16.

Table 223. Legacy scripting to Python scripting mapping for model operations.

Legacy scripting	Python scripting
`open model` *path*	*taskrunner*`.openModelFromFile(`*path*, *autoManage*`)`
`save` *model* `as` *path*	*taskrunner*`.saveModelToFile(`*model*, *path*`)`
`retrieve model` *path*	*repository*`.retrieveModel(`*path*, *version*, *label*, *autoManage*`)`
`store` *model* `as` *path*	*repository*`.storeModel(`*model*, *path*, *label*`)`

Document output operations

Some document output operation commands that are commonly used in IBM SPSS Modeler have equivalent commands in Python scripting. This might help you to convert your existing SPSS Modeler Legacy scripts to Python scripts for use in IBM SPSS Modeler 16.

Table 224. Legacy scripting to Python scripting mapping for document output operations.

Legacy scripting	Python scripting
`open output` *path*	*taskrunner*`.openDocumentFromFile(`*path*, *autoManage*`)`
`save` *output* `as` *path*	*taskrunner*`.saveDocumentToFile(`*output*, *path*`)`
`retrieve output` *path*	*repository*`.retrieveDocument(`*path*, *version*, *label*, *autoManage*`)`
`store` *output* `as` *path*	*repository*`.storeDocument(`*output*, *path*, *label*`)`

Other differences between legacy scripting and Python scripting

Legacy scripts provide support for manipulating IBM SPSS Modeler projects. Python scripting does not currently support this.

Legacy scripting provides some support for loading *state* objects (combinations of streams and models). State objects have been deprecated since IBM SPSS Modeler 8.0. Python scripting does not support state objects.

Python scripting offers the following additional features that are not available in legacy scripting:
- Class and function definitions
- Error handling
- More sophisticated input/output support
- External and third party modules

Notices

Trademarks

Index