SAS® GLOBAL FORUM 2020 #SASGF

Paper SAS4376-2020

Tips for Writing Custom SAS® Studio Tasks That Use CAS Actions for Advanced Analytics

Brian R. Gaines, SAS Institute Inc., Cary, NC

Abstract

The SAS® Studio programming interface includes point-and-click user interfaces, called tasks, that enable you to use SAS® software without writing any code. As you navigate a task's menus and options, the underlying SAS code is automatically generated for you in real time, making SAS analytics accessible even to users who are not familiar with SAS code. SAS Studio tasks cover a wide range of analytical areas, but you can also create custom tasks to suit your needs. For example, the code that a task generates can include the CASL programming language that you can use to interact with SAS® Cloud Analytic Services (CAS). CAS is a cloud-enabled, in-memory analytics engine from SAS that uses distributed computing for highly scalable, very fast execution. Because CASL is a relatively new language, custom tasks that generate CASL code make it easier for you to use CAS actions and learn the CASL syntax as you migrate your analytics to CAS. This paper starts by briefly presenting some background on SAS Studio tasks and CAS. Then it uses analytical actions in SAS® Econometrics software to provide several tips and tricks for writing CASL-generating custom tasks.

Introduction

In recent years, one of the major trends in data analysis has been the democratization of analytics. Easy-to-use tools make data and analytics more accessible to a wider range of people, enabling them to use data to improve decision making. SAS Studio tasks are one of the many ways in which SAS is democratizing analytics. These tasks provide point-and-click user interfaces (UIs) that enable you to perform data preparation and various analyses without the need to write code. As you go through the menus and options for a task, the underlying SAS code is automatically generated and updated for you in real time. Not only do tasks make SAS analytics accessible to noncoders, but the real-time code generation can also help users learn the syntax for the method of interest. At the same time, even experienced SAS programmers can use these tasks to accelerate new code development. In just a few keystrokes and clicks, you can have several lines of formatted, typo-free code that is not missing any semicolons!

The predefined SAS Studio tasks cover a wide range of analytical areas, from data preparation and visualization to machine learning, forecasting, and text analytics. This coverage includes many of the advanced analytic capabilities available in SAS® Viya®, which is a modern, high-performance extension of the SAS® Platform. The underlying in-memory analytics engine that powers SAS Viya is SAS Cloud Analytic Services (CAS). CAS is a high-performance server that uses distributed computing for highly scalable, very fast execution.

As you migrate your analytics to CAS, the SAS Viya tasks in SAS Studio make it much easier to learn and develop the code that uses new Viya procedures or the CAS language. The CAS language, or CASL for short, is a new scripting language that enables you to interact with a CAS server by executing CAS actions. A CAS action performs a single task and is the smallest unit of work for the CAS server. When you use a CAS action via the CAS procedure, CASL gives you more granular control and flexibility, in addition to the ability to set more advanced options. The syntax for CASL also resembles the syntax that you use to access CAS from an open source language such as R or Python, so learning CASL can also make it easier to collaborate with your colleagues. For more information about CAS actions and CASL, see Gass (2018) and Sober (2019).

Several prebuilt SAS Viya tasks include an option to generate code that uses a CAS-enabled procedure, or you can generate code that uses PROC CAS and CASL (Figure 1). These tasks are especially helpful for learning CASL syntax because CASL is a new language that differs from traditional SAS procedure code. If you are an experienced SAS programmer, you can toggle between procedure code and the equivalent CASL syntax to see how the more familiar procedure code maps to CASL. As you become more comfortable with developing CASL code, you can also create and customize tasks to suit your needs. For example, if you or your colleagues routinely use a particular set of CAS actions with different data sets and slightly different options, you can create and share a custom task to easily do

exactly that. A task essentially provides you with a web-based interface for flexibly developing CASL code. This paper uses SAS Econometrics analytical actions to provide a few tips and tricks for writing CASL-generating custom tasks.

Figure 1 Hidden Markov Models Task Code Generation Options

CODE GENERATION

- ○ Use HMM procedure
- ● Use CAS procedure

```
proc hmm data=MYCAS.GNPHAMILTON;
    id time=date;
    model dgnp / type=gaussian method=ml;
run;
```

CODE GENERATION

- ● Use HMM procedure
- ○ Use CAS procedure

```
proc cas;
    session %sysfunc(getlsessref(MYCAS));
    action hiddenMarkovModel.hmm / table={name="GNPHAMILTON"
        caslib="%sysfunc(getlcaslib(MYCAS))"} id={time="date"}
        model={depvars={"dgnp"} type="gaussian" method="ml"};
    run;
quit;
```

Writing Custom Tasks

SAS Studio tasks are based on the common task model (CTM), which is an XML-based framework that defines the template for the task. In a CTM file, you specify various options and how they will appear in the task, and you use the Apache Velocity Template Language to dynamically generate SAS code. The main sections of a CTM file are as follows:

- Registration
  - You use the Registration section to specify the basic information of the task, such as the name, description, and version number. This information is displayed on the **Information** tab in the user interface (UI).
- Metadata
  - You use the Metadata section to define the task's input fields, or *controls*. This section has two subsections:
  - DataSources
    - You use the DataSources subsection to define input data sources. For each DataSource, you can define Roles to select variables from the corresponding data set.
  - Options
    - You use the Options subsection to include a variety of different UI controls in the task, such as a check box, radio button, or drop-down list (combo box). For a complete list of the available controls, see SAS Institute Inc. (2019e).
- UI
  - You use the UI section to specify how the controls in the Metadata section appear in the user interface for the task.
- CodeTemplate
  - You use the CodeTemplate section to define the output that the task produces, which is almost always SAS code. This section uses Velocity 2.0, a simple but powerful open source scripting language based on Java, to dynamically incorporate user-made selections from the UI into the generated code.

Velocity variables are typically assigned from selections that you make in the task UI. They are similar to SAS macro variables, but you prefix Velocity variables with `$` instead of `&` and reference the control's name that you defined in the Metadata section. You can further manipulate the CodeTemplate output by using Velocity *directives*. Directives begin with `#` and enable you to incorporate loops and conditional statements to govern the code generation. For example, you can use the `#set` directive to assign a value to a new Velocity variable. The `#set` directive is similar to the %LET statement for macro variables. For more information about the Velocity language, see Apache Software Foundation (2020).

In addition to the sections described previously, the CTM file has two optional sections:

- Dependencies
  - You use the Dependencies section to specify how controls rely on one another. For example, selecting a particular check box could show or hide other options in the task interface.
- Requirements
  - You use the Requirements section to specify conditions that must be met in order for SAS code to be generated.

Here is a general workflow that you can use to develop a custom task:

1. In the Metadata section, you first define a control that you want to include in the task. The control can be a DataSource, a Role, or one of the many Option input types.
2. After you define a new control, you add it to the UI section so that it appears in the task UI.
3. Optionally use the Dependencies or Requirements section as needed.
4. In the CodeTemplate section, you use Velocity to determine how an interaction with the new control translates to the generated code.
5. Repeat this process until you are satisfied with the task.

SAS offers a free e-learning course, "Writing a Custom Task for SAS Studio," that covers the basics of custom task writing. For more information about custom task writing, see the "Recommended Resources" section. A basic understanding of custom task writing is sufficient for this paper, so let's look at a couple of examples of customizing CASL-generating tasks.

Case Study: The Hidden Markov Models Task

The hidden Markov model (HMM) is a popular machine learning model for analyzing sequential data such as time series. The model consists of two main parts: a sequence or chain of observed data (also called *emissions*) and a sequence of hidden or latent states that generate the observed data. The hidden states follow a Markov process, which means that the probability distribution of a state at a given time point depends only on the state at the previous time point. Additionally, the probability of an observation is governed only by the current hidden state and not by any other hidden state or observation. The goal of a hidden Markov model is to use the observed data to learn about the underlying hidden states. For more information about HMMs, see Rabiner (1989) and Chen and Shen (2018).

To be more concrete, let's consider a canonical weather example. You are interested in learning about the weather, which for simplicity can be either sunny, cloudy, or rainy. However, suppose you cannot directly observe the weather but instead you can see only the clothes that people are wearing at different points in time. In this example, the clothing that you see is the observed data, and the three types of weather are the hidden states. You could use a hidden Markov model to infer about the weather conditions on the basis of people's clothing.

This modeling framework has been used for many applications across different disciplines, such as economics, finance, biology, and natural language processing. With SAS Econometrics, you can use the distributed computing power of the in-memory analytics CAS engine to train hidden Markov models. You can write the code by using the HMM procedure, or equivalently by using the CAS procedure and CASL with the Hidden Markov Model action set. SAS Studio includes a Hidden Markov Models task that enables you to quickly develop code that uses either the HMM procedure or the CAS procedure (Figure 1).

Example 1: Analytic Store

For example, suppose you are interested in using the Hidden Markov Models task to develop the code for a model that identifies the different hidden states of a stock market, such as whether it is a bull market or a bear market. You could use the predicted future state of the stock market to derive an optimal trading strategy and portfolio allocation. (For a more detailed treatment of this example, see Chen and Shen 2018.)

Specifying the number of hidden states to use in the model is an important consideration when you train a hidden Markov model. A reasonable approach to this model selection problem is to train multiple models in which each model estimates a different number of hidden states. You can then use information criteria, such as Akaike's information criterion (AIC) or the Bayesian information criterion (BIC), to determine the optimal number of hidden states. After you select your final model, you can use it to forecast future market states and adjust your trading strategy accordingly.

One way to facilitate the scoring of new data is to save the information for the final, trained model as an analytic store, or ASTORE file. That way, you do not have to perform time-consuming model training again when you want to use the model for inference or forecasting. Models that are saved as analytic stores also make it easy to deploy and manage your models with SAS® Model Manager software or to make real-time decisions with SAS® Event Stream Processing software. For more information about analytic stores, see Ebersole (2017).

The SAS Studio task for hidden Markov models does have an option on the **Output** tab for creating a scoring model, but it does not yet include an option for specifically creating an analytic store. Often a task is designed such that it does not include every possible option available in the underlying procedure or CAS action. This approach ensures that the interface remains uncluttered and does not overwhelm you with too many options. However, when you are using a task, you can click the **Edit** button in the code editor to access the generated code. This opens the code in a new editor tab so that you can edit it further or save it to reproduce your analysis later. That way, even if a task does not include an option that you would like to use, the task can still generate 95% of the code that you want before you manually add the finishing touches.

Although the ability to edit the generated code is a helpful feature of the tasks, sometimes you would prefer to have the option directly in the task to bypass the need to manually edit the code each time. This is especially useful if there are other analysts in your organization who rely on a task because they are not familiar with writing SAS code, or for an analyst who is learning the CASL syntax. You might be worried that, in order to include a new option in a predefined SAS task, you need to create your own custom hidden Markov models task from scratch, which might seem like a daunting job. Fortunately, this is not the case—which brings you to Tip #1.

Tip #1: Do Not Reinvent the Wheel

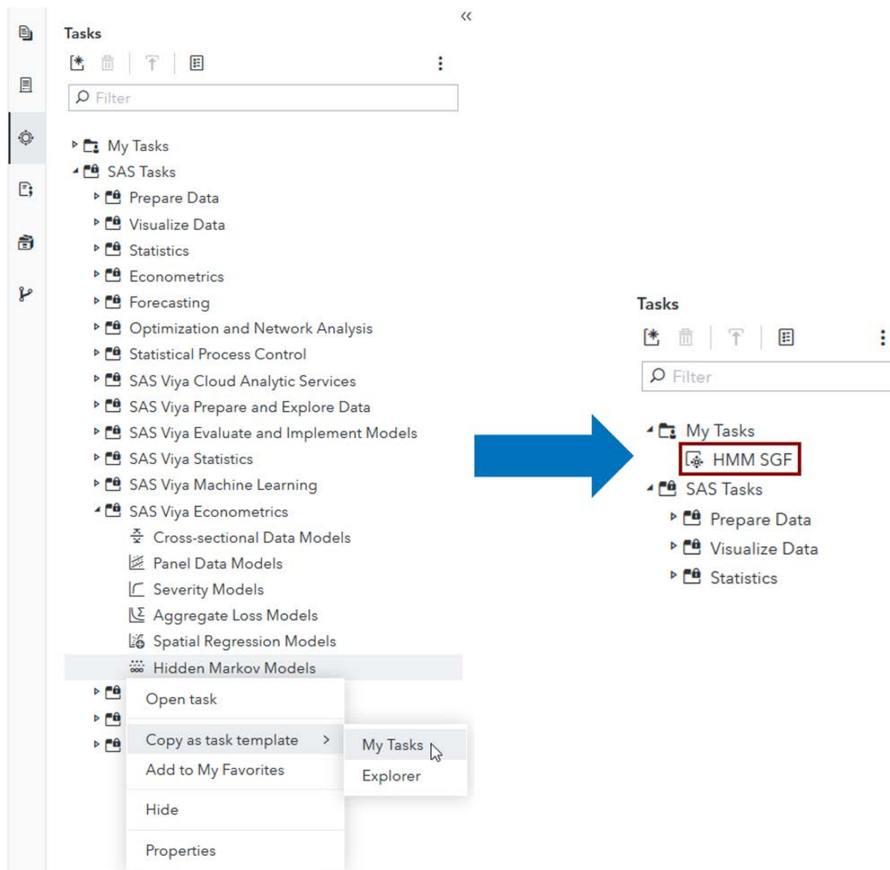
The best way to substantially reduce the time that it takes to develop a task is to rely on existing CTM code instead of writing your task code from scratch. You can actually access the CTM code for any of the predefined SAS tasks that SAS Studio includes. You can create a copy of a predefined SAS task and then tweak it to suit your needs; this is a great way to dip your toe into the custom-task-writing water. In addition, you can copy a task and extract the relevant CTM code to borrow parts of the task. The Sample task is especially useful for this purpose because it contains most of the available controls. Browsing the interfaces of the existing tasks helps generate ideas and reduce the code development time when you are designing and developing a custom task.

To access the CTM code in a predefined task, you first right-click the name of the task of interest, select **Copy as task template**, and then select **My Tasks** (Figure 2). After you specify the name of the new task, a copy of the task is added to your **My Tasks** folder (Figure 2). Let's do this for the Hidden Markov Models task and name the new task "HMM SGF" so it is easy to distinguish from the original predefined Hidden Markov Models task. Alternatively, you can save the CTM template file to SAS Content or a file system that is accessible in the Explorer section.

Now that you have saved the task to your **My Tasks** folder, you can right-click the name of the task and select **Edit task template** to view the CTM code for the task. If you edit this code, you can view the updated task interface by clicking **Open** on the toolbar. When you are satisfied with your changes, you save them and then view the interface for your customized task by double-clicking the name of the task within your **My Tasks** folder.

With a better understanding of the logistics of accessing and editing a predefined task's CTM code, let's return to adding the analytic store option to your custom hidden Markov models task, HMM SGF. Within the task template, you use an *outputdata* control to add an input field that enables a user to specify the name of the output data set that a task creates. You first need to create a new Option element for the outputdata control in the Options section, and then add a corresponding element to the UI section. To maintain consistency with the task's existing outputdata controls, let's also add a Dependency so that the outputdata control is enabled only if a user selects **Create analytic**

Figure 2 Copying a Predefined Task to the My Tasks Section



store model. For this, you can copy and modify the XML code for an existing outputdata control, so the code looks something like the following code. Throughout the paper, the example code snippets that contain XML code highlight the new code in blue. Additional XML code is included in the snippets to provide context, because the location where you insert the code into the task is important for the UI and Dependencies sections. You can also download the complete task code for both examples from the SAS® Global Forum 2020 GitHub repository, located at https://github.com/sascommunities/sas-global-forum-2020.

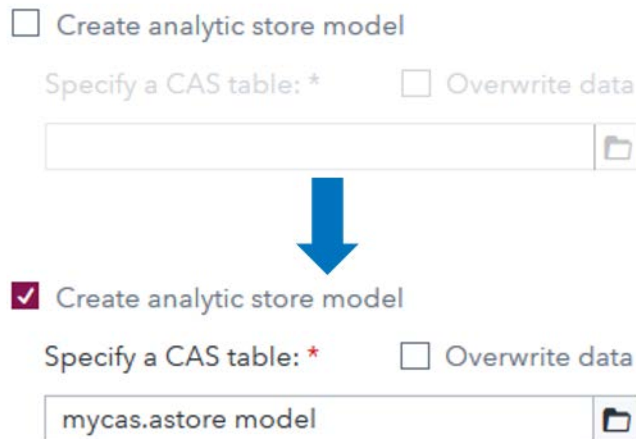
```
<Options>
   ...
   <Option name="outputAstore" inputType="checkbox" defaultValue="0">
      Create analytic store model</Option>
   <Option name="outputAstoreName" inputType="outputdata" libraryEngineInclude="cas"
      required="true" unique="true" indent="1">Specify a CAS table:</Option>
   ...
</Options>
```

```
<UI>
   ...
   <Container option="outputTab">
      <Group open="true" option="outputGroup">
         ...
         <OptionItem option="outputModelName"/>
         <OptionItem option="outputAstore"/>
         <OptionItem option="outputAstoreName"/>
         ...
      </Group>
   </Container>
</UI>
```

```
<Dependencies>
   ...
   <Dependency condition="($outputModel=='1')">
      <Target action="enable" conditionResult="true" option="outputModelName"/>
      <Target action="disable" conditionResult="false" option="outputModelName"/>
   </Dependency>
   <Dependency condition="($outputAstore == '1')">
      <Target option="outputAstoreName" conditionResult="true" action="enable"/>
      <Target option="outputAstoreName" conditionResult="false" action="disable"/>
   </Dependency>
   ...
</Dependencies>
```

The task interface now includes an option for saving the model as an analytic store, in which you must select the check box to enable the outputdata control (Figure 3).

Figure 3 New Analytic Store Outputdata Control



Now that the UI has been updated, you need to incorporate this new control in the generated CASL code in order for the option to be functional. As mentioned previously, this is done in the CodeTemplate section of the CTM code, where the Velocity scripting language dynamically incorporates the CAS table name that a user specifies in the task UI. You can see from the documentation for the Hidden Markov Models action set that the relevant syntax for saving the model as an analytic store is `store={out={caslib="caslibName", name="tableName", replace="true"}}` (SAS Institute Inc. 2019d).

To generate this CASL code, it is important to note that the outputdata control requires a two-level name of the form *LibrefName.TableName*, which is how you typically refer to data set in a SAS DATA step or PROC step. However, CASL code requires the name of the underlying caslib instead of the associated libref. Caslibs are the mechanism that the CAS server uses to interact with data, and they consist of in-memory tables, optional data source files, and metadata (SAS Institute Inc. 2019b). You assign a libref to a caslib by using the LIBNAME statement with the CAS option. For example, you use the following syntax to assign the libref *Mycas* to the *Casuser* caslib:

```
libname mycas cas caslib="CASUSER";
```

You can use this *Mycas* libref and the DATA step to load a data set as an in-memory table into the *Casuser* library, such as the following:

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

Table 1 shows the equivalent code to display the first 11 rows of this in-memory data table when you use either the PRINT procedure or the CAS procedure with the table.fetch action. You can see that the PROC PRINT code uses the *Mycas* libref to refer to the table, whereas table.fetch directly uses the name of the caslib, *Casuser*.

Table 1 Equivalent Code for Displaying Data

<table>
<tr><th>PRINT Procedure</th><th>CAS Procedure and table.fetch Action</th></tr>
<tr><td><pre>proc print data=mycas.cars(obs=11);
run;</pre></td><td><pre>proc cas;
   action table.fetch /
      table={caslib="casuser", name="cars"}
      to=11;
   run;
quit;</pre></td></tr>
</table>

This brings you to Tip #2.

Tip #2: Use Functions and Methods

It is often helpful to take advantage of special functions and methods to correctly generate CASL code. For the issue at hand, the system function GETLCASLIB returns the name of the caslib that is bound to a libref. In this example, the code `%put %sysfunc(getlcaslib(MYCAS));` would return CASUSER.

However, before you can use this helpful system function, you first need to use one of the useful CTM methods. The Velocity variable for an outputdata control (`$outputAstoreName` in this example) returns the two-level name that a user supplies in the task interface, but here you want to reference only the libref. One of the available methods for this control is the `getLibrary()` method, which returns only the name of the libref; this is exactly what you want. That is, if you specify `mycas.astoreModel` in the UI, the Velocity variable `$outputAstoreName` returns `mycas.astoreModel`, whereas `$outputAstoreName.getLibrary()` returns `mycas`. For more information about the CTM methods available, see SAS Institute Inc. (2019e).

By combining this function and method, you can take the user-supplied libref from the outputdata control in the task UI and convert it to a caslib name for the CASL code. The syntax to do this in the CodeTemplate section is `caslib="%sysfunc(getlcaslib($outputAstoreName.getLibrary()))"`.

Tip #3: Be Cognizant of Handling Special Characters

You should also exercise care when you reference the one-level name for the output data table or when you reference a role that a user can assign variables to. Another important distinction between the syntax for CASL code and the syntax for the DATA step (and for most procedures) is how special characters are handled in SAS names. You set the VALIDVARNAME system option to ANY to use special characters in variable names, and you specify VALIDMEMNAME=EXTEND to use them in data set names. These are the default values for both system options in SAS Studio, so it is a good practice for your task-generated code to be robust to the use of special characters.

Typically, a SAS name literal enables you to use special characters that are not otherwise allowed in SAS names when you specify the name of a SAS data set or variable in a PROC or DATA step. A name literal is displayed as a string in quotation marks that is followed by an upper- or lowercase letter n. For example, `data=mycas.'my data'n` refers to a data set that has the name *My data* by using the *Mycas* libref that is bound to the *Casuser* caslib. However, in CASL code, the name of a variable or data table is directly enclosed in quotes, whether or not the name contains special characters. That is, the equivalent CASL code is `table={caslib="CASUSER", name="MY DATA"}`, so you can see that you do not use the name-literal notation. For more information, see the section “Names in the SAS Language” in SAS Institute Inc. (2019a).

Fortunately, there are additional CTM methods available that enable you to keep this straight and to generate code that is robust to special characters in SAS names. By default, if a special character is included in the name of an input or output data set in a task, the corresponding Velocity variable (for example, `$outputAstoreName`) uses the name-literal notation. To reference only the data table name, you can use the `get()` method with the `table` parameter or the `getTable()` method. Table 2 summarizes how these references resolve in the generated code if you enter **mycas.astore model** for the output table name (Figure 3).

Returning to the outputdata control for the model saved as an analytic store, you want to use the `get("table")` method to reference only the table name without the name-literal notation. The finishing touch for the code generation is to incorporate the previously defined Dependency that enables the output data field only if a user selects **Create**

Table 2 Velocity References for the Outputdata Control

Velocity Reference	Description	Generated Code
`$outputAstoreName`	Two-level table name in name-literal form	`mycas.'astore model'n`
`$outputAstoreName.getLibrary()`	Library name	`mycas`
`$outputAstoreName.getTable()`	Table name with name-literal notation	`'astore model'n`
`$outputAstoreName.get("table")`	Table name without name-literal notation	`astore model`

analytic store model. You achieve this by using an **#if** directive in the Velocity code. Putting this all together, the Velocity code to generate the CASL code for the new analytic store option is as follows:

```
#if($outputAstore=='1')
   store={out={caslib="%sysfunc(getlcaslib($outputAstoreName.getLibrary()))",
      name="$outputAstoreName.get("table")", replace="true"}}
#end
```

Bonus tip Here is one last note about the methods that you can use in a task's Velocity code. Velocity is a Java-based template engine, so you can also use standard Java methods for each class. You can use the **getClass()** method to find out which Java class a Velocity variable belongs to, such as **$variable.getClass()**. When you know the Java class, you can reference the Java documentation to see which methods are available. For the string class, for example, methods such as **trim()** and **equalsIgnoreCase()** are helpful for comparing different strings.

Example 2: Regime-Switching Autoregression

As another example, let's use the Hidden Markov Models task to train a regime-switching mean-adjusted autoregression model to identify recessionary periods in the US economy. Hamilton (1989) proposed this model for this very purpose and used it to analyze quarterly US real gross national product (GNP) data from 1951 to 1984. (For a more detailed treatment of this example, see Example 13.3 in the HMM procedure documentation in SAS Institute Inc. 2019c.)

As with Example 1, this is a situation in which some of the bells and whistles of the model that you want to use are not already available in the predefined Hidden Markov Models task. However, you can apply Tip #1 to create a copy of the task and customize it so that generating CASL code for this model is much easier for you and your colleagues.

After adding another copy of the Hidden Markov Models task to your **My Tasks** folder, you first need to add an option for estimating the mean-adjusted form of the regime-switching autoregression model. Let's add a check box for this option to the **Model** tab under the **Number of hidden states** option, keeping in mind that this option should be available only if a user selects **Regime-switching autoregression** for the **Model type**. The new task code to add this option to the task UI is highlighted in the following code. Figure 4 displays the updated UI.

```
<Options>
   ...
   <Option name="armean" inputType="checkbox">Use mean-adjusted form</Option>
   ...
</Options>


<UI>
   ...
   <Container option="modelTab">
      <Group open="true" option="modelGroup">
         ...
         <OptionItem option="nState"/>
         <OptionItem option="armean"/>
         ...
      </Group>
   </Container>
</UI>
```

```
<Dependencies>
   ...
   <Dependency condition="($modelTypeCombo=='ar')">
      <Target action="show" conditionResult="true" option="addYLag"/>
      <Target action="hide" conditionResult="false" option="addYLag"/>
      <Target option="armean" conditionResult="true" action="show"/>
      <Target option="armean" conditionResult="false" action="hide"/>
   </Dependency>
   ...
</Dependencies>
```

Figure 4 New Option for Mean-Adjusted Form

MODEL

Model type:

Regime-switching autoregression

Number of hidden states (default: 2):

2

Use mean-adjusted form

You can see in the documentation that you use the **`arMean`** parameter within the **`model`** parameter to specify this option in the CASL code. The two possible values for the **`arMean`** parameter are ADJUSTED and STANDARD, with a default value of STANDARD. Because STANDARD is the default, including it in the generated code is optional. On the one hand, you might want to exclude the code for default options to keep the generated code more compact. On the other hand, you might want to include the default value in the code to make it easier to identify the options and specifications for the model that you train. Let's add **`arMean="adjusted"`** to the generated code only when **Use mean-adjusted form** is selected. The Velocity code to generate this CASL code is as follows:

```
#if($modelTypeCombo == 'ar' && $armean == '1') arMean="adjusted"#end
```

Another feature of the model proposed by Hamilton (1989) is the ability to estimate certain model parameters independently of the hidden states. Typically, a hidden Markov model produces different parameter estimates for each hidden state. However, sometimes the model is a better reflection of reality or a better fit to the data if some parameters are the same across the different hidden states. For example, you might want your Gaussian HMM model's covariance to be the same for each state. Imposing this type of constraint also reduces the number of parameters that you need to estimate; this can improve performance.

Let's add this functionality to the **Model** tab of your custom Hidden Markov Models task because this option relates to specifying your model. Before doing so, let's first rename the **Estimate the nonstationary Markov chain** option, which is currently located on the **Options** tab, and move it to the **Model** tab. One of the state-independent parameter options is available only if a user specifies a nonstationary Markov chain for the model, so moving this option improves the logical flow of the task. The updated task code is as follows:

```
<Options>
   ...
   <Option defaultValue="0" inputType="checkbox" name="estimateISPV">
      Nonstationary Markov chain</Option>
   ...
</Options>
```

```
<UI>
   ...
   <Container option="modelTab">
      <Group open="true" option="modelGroup">
         ...
         <OptionItem option="nComponent"/>
         <OptionItem option="estimateISPV"/>
         <Group open="true" option="regressorsGroup">
            ...
         </Group>
         ...
      </Group>
   </Container>
   <Container option="optionsTab">
      <Group open="true" option="methodsGroup">
         ...
         <OptionItem option="maxPosteriori"/>
         <OptionItem option="useRandomSeed"/>
         ...
      </Group>
   </Container>
   ...
</UI>
```

Now let's shift the focus to adding the state-independent parameters option to the task. According to the documentation, there are a total of 12 different model parameters that you can possibly estimate independently of state (SAS Institute Inc. 2019d). The XML code to add these options to the task as check boxes is similar to what you used in Example 1 to add the analytic store option to the first custom task, so the code is omitted here but is available in the SAS Global Forum 2020 GitHub repository.

Bonus tip Because you need to add several check boxes to the task, it is a good practice to organize them so that the task UI is more user-friendly. For example, you can place the check boxes inside a closed **State independent parameters** group to reduce the amount of real estate the check boxes initially occupy in the task. In addition, because some of the options are relevant for only certain model types (SAS Institute Inc. 2019c, "The HMM Procedure," Table 4), you can put them into subgroups. As an example, the MU, SIGMA, and MCP options are applicable for Gaussian models, so these can go into a **Gaussian models** subgroup.

You can also improve the organization of the check boxes by arranging them in an order that is intuitive and maintains consistency within the task. To this end, let's arrange the Gaussian model options MU, SIGMA, and MCP in that order because the mean (MU) and covariance (SIGMA) are generally referred to in that order and because the MCP option is available only for Gaussian mixed models. For regime-switching models, you could arrange the check boxes to be consistent with the order in which the related options appear in the task. For example, in the task interface, the option to include an intercept in the model already precedes the option to add seasonal dummy regressors, so the options to estimate these parameters independently of state should maintain this ordering.

As mentioned previously, some of these options are relevant for only certain model types, so let's create Dependency elements to show and hide the check boxes when appropriate. As an example, consider the new **Linear trend** check box that is located in the new **Regime-switching models** subgroup. This option is appropriate if you select either of the regime-switching models as the model type and if you also select the check box to add time trends as regressors. Both options appear on the **Model** tab. You can use the following XML code to create a Dependency that enforces this restriction:

```
<Dependencies>
   ...
   <Dependency condition="(($modelTypeCombo == 'reg' || $modelTypeCombo == 'ar')
            &amp;&amp; $addTrend == '1')">
      <Target option="stateIndependentLtrend" conditionResult="true" action="show"/>
      <Target option="stateIndependentLtrend" conditionResult="false" action="hide"/>
   </Dependency>
   ...
</Dependencies>
```

The condition for the preceding Dependency includes two parts. The first part requires that a user select either of the regime-switching models, which are named in the Options section as “reg” and “ar.” The second part of the condition requires that the check box for adding time trends as regressors to the model be selected.

Bonus tip For Dependency conditions, you use the character entity references `&&` instead of `&&` for the logical AND operator because the `&` symbol is a reserved character in XML. This rule applies only to XML and not to Velocity.

You implement the dependencies for the other state-independent check boxes similarly. You can see the complete XML code for these Dependency elements in the SAS Global Forum 2020 GitHub repository. Keep in mind that a task evaluates dependencies in top-down order, so it matters where you place them in the task code. The previous **Linear trend** Dependency cannot precede the Dependency that surfaces the `addTrend` option, for example. A natural location for the new state-independent Dependency elements is above the Dependency that shows and hides the **Random seed** number text box on the **Options** tab.

With the UI updated, you can now integrate these new check boxes into the CodeTemplate section. As per the documentation, you use the `stateIndependent` parameter within the CASL parameter `model` to specify a comma-separated list of parameters selected in the UI, as in the following code:

```
stateIndependent={"AR", "CONST", "COV", "ISPV", "LTREND", "MCP", "MU", "QTREND",
   "SDUMMY", "SIGMA", "TPM", "XL"}
```

The `stateIndependent` parameter should appear in the generated code only if a user selects at least one of the corresponding 12 check boxes. Although you can enforce this condition by inserting an `#if` directive next to the `stateIndependent` parameter in the Velocity code, an alternative approach is to define a new Velocity variable that indicates whether to generate this section of the code. This approach improves the readability of the Velocity code and can also make it easier to develop and debug a task that includes more than one type of code generation.

The following example shows what the Velocity code for this check looks like. It checks to see whether any of the 12 `stateIndependent` options were selected, and if so, it defines a new Velocity variable, `$showStateIndependent`, with a value of 1.

```
#if($stateIndependentIspv == '1' || $stateIndependentTpm == '1' ||
   $stateIndependentSigma == '1' || $stateIndependentMu == '1' ||
   $stateIndependentMcp == '1' || $stateIndependentAr == '1' ||
   $stateIndependentCov == '1' || $stateIndependentXl == '1' ||
   $stateIndependentConst == '1' || $stateIndependentLtrend == '1' ||
   $stateIndependentQtrend == '1' || $stateIndependentSdummy == '1')
   #set($showStateIndependent = '1')
#end
```

Another thing to consider is that, to specify the options within the `stateIndependent` parameter, you use a comma-separated list of the model parameters selected in the UI. A naive approach is to simply use an `#if` directive for each check box, as shown in the following code snippet:

```
stateIndependent={#if($stateIndependentIspv == '1') "ispv"#end,
   #if($stateIndependentTpm == '1') "tpm"#end, ...}
```

However, this approach generates code that contains unnecessary leading or trailing commas, resulting in an error when you submit the code. Technically, these commas are not required in the code, but it is a good practice to include them to be consistent with the documentation and with the similar syntax that you use to interact with CAS from an open source language where the commas are not optional.

Tip #4: Organize Items into a List

One way to handle the commas in the generated CASL code is to first organize the possible options within the **stateIndependent** parameter into a Velocity list. When the options are in a list, you can use a **#foreach** loop to extract only the options that are selected in the UI, and you use a clever **#if** directive to insert commas as needed. The following is a snippet of the Velocity code to create the new list and add items to it. The complete code is available in the SAS Global Forum 2020 GitHub repository.

```
#if($showStateIndependent == '1')
   #set($stateIndependentList = [])
   #if($stateIndependentIspv == '1')
      #set($success = $stateIndependentList.add("ispv"))
   #end
   ...
#end
```

In the preceding Velocity code, you first use the Velocity variable **$showStateIndependent** that you created previously to determine whether at least one parameter was selected in the UI. If so, you create an empty list, **$stateIndependentList**. For each check box in the UI, you check to see whether it was selected, and if it was, you add an item to the list that corresponds to the check box.

The preceding Velocity code uses a trick for using the **add()** method in this manner. The **add()** method returns a Boolean value that indicates whether the add operation was successful. Because you do not want this value to appear in the generated code, you can instead assign it to a temporary Velocity variable (**$success** in this case). This way the element is still added to the list without unnecessarily displaying the Boolean value that is generated by the **add()** method.

After you organize the selected UI options into a list, you use a **#foreach** loop to traverse the list and generate the code for the corresponding **stateIndependent** options. The following Velocity code also uses an **#if** directive to ensure that the appropriate number of commas appear in the generated CASL code:

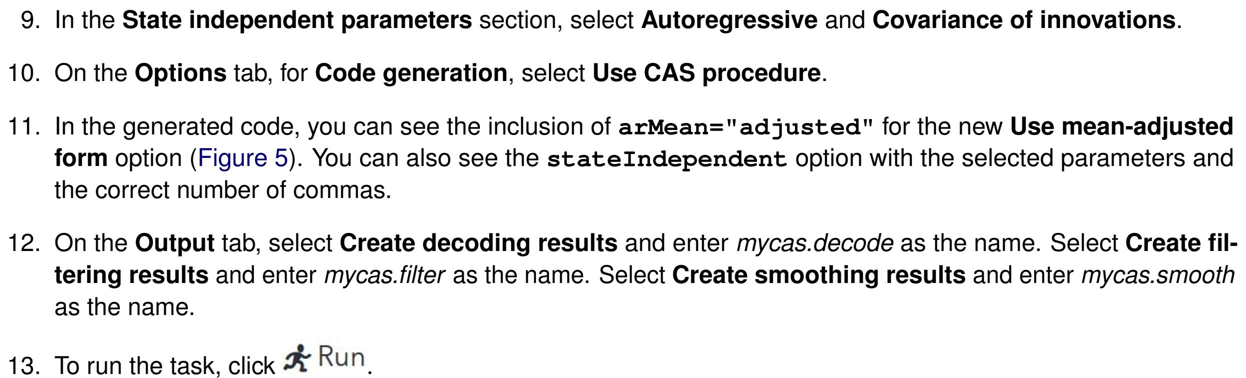
```
#if($showStateIndependent == '1')
   stateIndependent={#foreach($item in $stateIndependentList)
      $CTMUtil.doubleQuoteString($item)#if($foreach.hasNext),#end#end}
#end
```

Your custom Hidden Markov Models task now includes the additional options that you need in order to estimate the model from Hamilton (1989) to identify recessionary periods in the US economy. You can do this by following these steps:

1. Select **Open** to create the task.
2. In a code editor, create the *mycas.gnphamilton* data table. The code to do so is available in the SAS Global Forum 2020 GitHub repository.
3. On the **Data** tab, select the **MYCAS.GNPHAMILTON** data table.
4. To the **Dependent variables** role, assign the **dgnp** variable.
5. To the **Time ID** role, assign the **date** variable.
6. On the **Model** tab, from the **Model type** list, select **Regime-switching autoregression**.
7. Select **Use mean-adjusted form**.
8. Under **Add lags of the dependent variables**, increase **Number of lags** to 4.

9. In the **State independent parameters** section, select **Autoregressive** and **Covariance of innovations**.
10. On the **Options** tab, for **Code generation**, select **Use CAS procedure**.
11. In the generated code, you can see the inclusion of **`arMean="adjusted"`** for the new **Use mean-adjusted form** option (Figure 5). You can also see the **`stateIndependent`** option with the selected parameters and the correct number of commas.
12. On the **Output** tab, select **Create decoding results** and enter *mycas.decode* as the name. Select **Create filtering results** and enter *mycas.filter* as the name. Select **Create smoothing results** and enter *mycas.smooth* as the name.
13. To run the task, click Run.

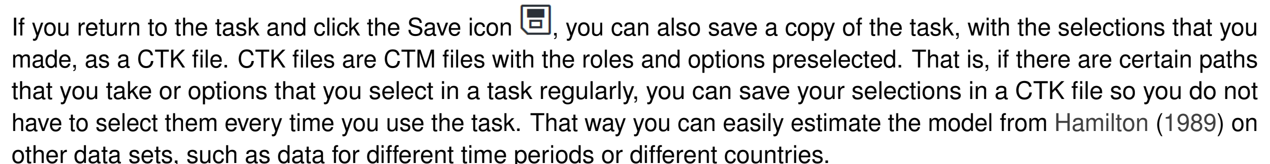
Figure 5 Task-Generated Code



```
proc cas;
    session %sysfunc(getlsessref(MYCAS));
    action hiddenMarkovModel.hmm / table={name="GNPHAMILTON"
        caslib="%sysfunc(getlcaslib(MYCAS))"} id={time="date"}
        model={depvars={"dgnp"} type="ar" arMean="adjusted" method="ml" ylag=4
        stateIndependent={"ar", "cov"}} decode={out={name="decode"
        caslib="%sysfunc(getlcaslib(mycas))" replace=true}}
        filter={out={name="filter" caslib="%sysfunc(getlcaslib(mycas))"
        replace=true}} smooth={out={name="smooth"
        caslib="%sysfunc(getlcaslib(mycas))" replace=true}};
    run;
quit;
```

After you use the task to estimate the model, you can use the following code to generate a plot of the model's recession probabilities across time (Figure 6).[1] The results deviate slightly from the results in the documentation example and in Hamilton (1989) because you did not specify the same initial values.

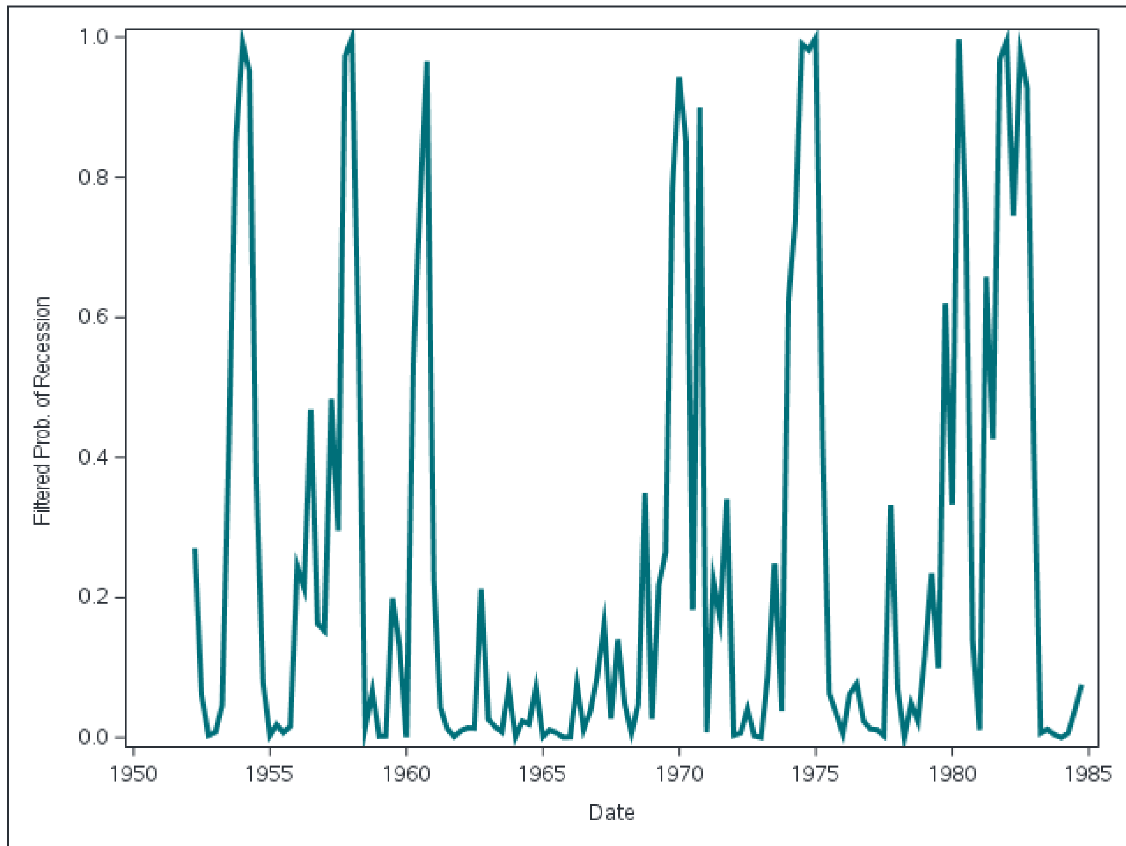
```
data filter;
   set mycas.filter;
run;
proc sort data=filter;
   by date;
run;
proc sgplot data=filter;
   series x=date y=state2;
   yaxis label="Filtered Prob. of Recession";
run;
```



If you return to the task and click the Save icon, you can also save a copy of the task, with the selections that you made, as a CTK file. CTK files are CTM files with the roles and options preselected. That is, if there are certain paths that you take or options that you select in a task regularly, you can save your selections in a CTK file so you do not have to select them every time you use the task. That way you can easily estimate the model from Hamilton (1989) on other data sets, such as data for different time periods or different countries.

[1] This code is from Example 13.3 in the HMM procedure documentation in SAS Institute Inc. (2019c).

Figure 6 Estimated Recession Probabilities



Conclusion

Your new custom task enables you and your colleagues to quickly develop CASL code to estimate hidden Markov models that use the power of CAS with only a few clicks and keystrokes. The task makes some of the new functionality in SAS Econometrics, such as the ability to save models in analytic store format or to estimate certain model parameters independently of state, much more accessible. More generally, custom tasks provide a flexible framework for creating user interfaces to expedite code development for any repeated process, without the need to use Java or JavaScript. You can easily share your custom tasks with colleagues and collaborators in the same way that you share SAS code files. Whether you are learning SAS code for the first time, using a new procedure, or adopting the new CAS language, or you simply want to improve code development time, SAS Studio tasks make SAS analytics accessible to a broader range of users.

References

Apache Software Foundation (2020). *Velocity 2.0 User Guide*. Wakefield, MA: Apache Software Foundation. https://velocity.apache.org/engine/2.0/user-guide.html.

Chen, X., and Shen, J. (2018). “Regime-Switching Models: Capturing Structural Changes in Time Series.” In *Proceedings of the SAS Global Forum 2018 Conference*. Cary, NC: SAS Institute Inc. https://support.sas.com/content/dam/SAS/support/en/sas-global-forum-proceedings/2018/1879-2018.pdf.

Ebersole, B. (2017). “PROC ASTORE in SAS Viya.” June 28. https://communities.sas.com/t5/SAS-Communities-Library/PROC-ASTORE-in-SAS-Viya/ta-p/370778.

Gass, M. (2018). "Cloud Analytic Services Actions: A Holistic View." In *Proceedings of the SAS Global Forum 2018 Conference*. Cary, NC: SAS Institute Inc. https://www.sas.com/content/dam/SAS/support/en/sas-global-forum-proceedings/2018/1981-2018.pdf.

Hamilton, J. D. (1989). "A New Approach to the Economic Analysis of Nonstationary Time Series and the Business Cycle." *Econometrica* 57:357–384.

Rabiner, L. R. (1989). "A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition." *Proceedings of the IEEE* 77:257–286.

SAS Institute Inc. (2019a). *SAS 9.4 Language Reference: Concepts*. Cary, NC: SAS Institute Inc. https://go.documentation.sas.com/?docsetId=lrcon&docsetTarget=titlepage.htm&docsetVersion=9.4&locale=en.

SAS Institute Inc. (2019b). *SAS Cloud Analytic Services 3.5: Fundamentals*. Cary, NC: SAS Institute Inc. https://go.documentation.sas.com/?docsetId=casfun&docsetTarget=titlepage.htm&docsetVersion=3.5&locale=en.

SAS Institute Inc. (2019c). *SAS Econometrics 8.5: Econometrics Procedures*. Cary, NC: SAS Institute Inc. https://go.documentation.sas.com/?docsetId=casecon&docsetTarget=titlepage.htm&docsetVersion=8.5&locale=en.

SAS Institute Inc. (2019d). *SAS Econometrics 8.5: Programming Guide*. Cary, NC: SAS Institute Inc. https://go.documentation.sas.com/?docsetId=casactecon&docsetTarget=casactecon_contents.htm&docsetVersion=8.5.

SAS Institute Inc. (2019e). *SAS Studio 5.2: Developer's Guide to Writing Custom Tasks*. Cary, NC: SAS Institute Inc. https://go.documentation.sas.com/?docsetId=webeditordg&docsetTarget=titlepage.htm&docsetVersion=5.2.

Sober, S. (2019). "What Is CASL?" In *Proceedings of the SAS Global Forum 2019 Conference*. Cary, NC: SAS Institute Inc. https://www.sas.com/content/dam/SAS/support/en/sas-global-forum-proceedings/2019/3040-2019.pdf.

Acknowledgments

The author is grateful to Ed Huddleston at SAS Institute Inc. for his valuable editorial assistance in the preparation of this paper. The author thanks Qingsong Yang, the developer of the Hidden Markov Models task, in addition to Xilong Chen and Ji Shen in the SAS Econometrics group for their useful feedback on this project. Thanks also to Sharad Prabhu, Dave DeNardis, and Olivia Wright for their helpful comments on an early draft.

Recommended Resources

In addition to SAS Institute Inc. (2019e), the following references are great resources for custom task writing:

- Corcoran, C., and Peters, A. (2015). "Teach Them to Fish—How to Use Tasks in SAS Studio to Enable Coworkers to Run Your Reports Themselves." In *Proceedings of the SAS Global Forum 2016 Conference*. Cary, NC: SAS Institute Inc. https://support.sas.com/resources/papers/proceedings15/SAS1831-2015.pdf.
- Dexter, M., Kiser, K., Peters, A., and Corcoran, C. (2016). "Create Web-Based SAS Reports without Having to Be a Web Developer." In *Proceedings of the SAS Global Forum 2016 Conference*. Cary, NC: SAS Institute Inc. https://support.sas.com/resources/papers/proceedings16/SAS4381-2016.pdf.
- Inman, E., and Wright, O. (2017). "Developing Your Own SAS Studio Custom Tasks for Advanced Analytics." In *Proceedings of the SAS Global Forum 2017 Conference*. Cary, NC: SAS Institute Inc. https://support.sas.com/resources/papers/proceedings17/SAS0677-2017.pdf.
- SAS Institute Inc. (2019e). "Writing a Custom Task for SAS Studio." Cary, NC: SAS Institute Inc. https://support.sas.com/edu/schedules.html?ctry=us&crs=CWTSS.

- Wright, O. (2019). “Custom Task Cheat Sheet.” https://communities.sas.com/t5/SAS-Communities-Library/Custom-Task-Tuesday-Download-the-Ultimate-Custom-Task-Cheat/ta-p/552910.
- Wright, O. (2019). *Custom Task Tuesday* (blog). https://communities.sas.com/t5/tag/Custom%20Task%20Tuesday/tg-p/board-id/library.
- Wright, O. (2020). “SAS Studio Custom Tasks: Tips and Tricks for the Adventurous Task Author.” In *Proceedings of the SAS Global Forum 2020 Conference*. Cary, NC: SAS Institute Inc. https://www.sas.com/content/dam/SAS/support/en/sas-global-forum-proceedings/2020/4537-2020.pdf.

Contact Information

Your comments and questions are valued and encouraged. Contact the author:

Brian R. Gaines
SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513
Brian.Gaines@sas.com
http://brgaines.github.io/

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

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