Paper SAS152-2017

Using Python with SAS® Cloud Analytic Services (CAS)

Kevin D Smith and Xiangxiang Meng, SAS Institute Inc.

ABSTRACT

With SAS® Viya™ and SAS® Cloud Analytic Services (CAS), SAS is moving into a new territory where SAS® Analytics is accessible to popular scripting languages using open APIs. Python is one of those client languages. We demonstrate how to connect to CAS, run CAS actions, explore data, build analytical models, and then manipulate and visualize the results using standard Python packages such as Pandas and Matplotlib. We cover a wide variety of topics to give you a bird's eye view of what is possible when you combine the best of SAS with the best of open source.

INTRODUCTION

This paper is a gentle introduction to using Python to access analytics from CAS. We begin with information on how to obtain the Python client and install it. We then show you how to connect to an existing CAS server and run actions. With those basics out of the way, we move on to more interesting subjects like loading data, performing simple analytics, and basic visualizations. We also demonstrate how to operate on tables in CAS using the popular Pandas DataFrame API. Finally, we cover some basic analytical modeling.

This might seem like a lot of territory to cover, but after working through it you'll have a broad understanding of how to interact with CAS and we hope you will be inspired to start using it in your own processes.

DOWNLOADING AND INSTALLING THE PYTHON CAS CLIENT

The Python client to CAS treads on new ground for SAS. It is actually maintained in an open-source project in GitHub. This means that you can browse the source, submit issues, and contribute code just as with any other open-source project. The code submissions are vetted and verified by SAS before being accepted just as if it were written in-house. Releases of the software are available from GitHub as well as the SAS support website. However, in order to install it we first need to have a running Python installation.

The easiest way to get Python and all of the dependencies installed is to use the Anaconda distribution from Continuum Analytics. This is a Python distribution intended for data science use. It includes dozens of packages that you likely will want to use at some point anyway all packaged together in a single installer. The Anaconda releases are available at the following address:

```
https://www.continuum.io/downloads
```

You simply download and install the appropriate package for your platform. Note that you should be using the 64-bit version of Python. In addition, Python is delivered in two major versions: 2.7 and 3.*x*. There are many people that still use the 2.7 release, but it is in maintenance mode. All current development of Python is done in the 3.*x* track. If you are new to Python, you should probably start with version 3.*x*. If you are already familiar with Python, you can use whichever version you are currently using.

Once you have Python installed, you can move on to installing the Python CAS client. Since the source code and API documentation are available from GitHub, we'll use that as the source for the download. The URL for the Python client, known as the SAS Scripting Wrapper for Analytics Transfer (SWAT), follows:

```
https://github.com/sassoftware/python-swat
```

On this page, you can see the README information about the SAS SWAT package that outlines the requirements, the procedure for installing it, and a very short code example. The API documentation is available at this address:

```
https://sassoftware.github.io/python-swat/
```

This page has much more complete information about the installation and usage of the SAS SWAT package. It includes API documentation for all of the objects in the SAS SWAT package as well. You definitely want to add this URL to your bookmarks.

Because we want to install the SAS SWAT package, we need to go to the page of releases at the following URL:

```
https://github.com/sassoftware/python-swat/releases
```

This page contains the latest releases of the software. In most instances, you will want the latest production release of the package. There are possibly two options for installation packages depending on the platform that you are running Python on. Some platforms support binary and REST interfaces to CAS; others support only REST. If there is a platform-specific installer listed in the release files (for v1.0.0, only Linux 64 had a specific installer), you should use that. It enables you to connect to CAS using either the binary interface or REST interface. If you don't see a platform-specific file, you should just use the Source Code distribution. The Source Code distribution is pure Python and works on any platform that Python runs on, but it can connect only to the REST interface of CAS. The downside is that the REST interface has more overhead when talking to the server, so it will be slower than the binary interface.

In either case, you can simply right-click the link to the installation file and copy the link. You then can paste the link as an argument to the **pip install** command that came with your Python distribution. The command below shows an example. The version number here has been removed. You should use whatever the most recent release is. Note that the same package works with both the 2.7 and 3.*x* versions of Python. The URL below is broken across the line for readability.

```
pip install https://github.com/sassoftware/python-swat
        /releases/download/vX.X.X/python-swat-X.X.X-linux64.tar.gz
```

After you have that installed, you should be able to import the package from Python. We use the **ipython** interpreter in our examples. It's a nice wrapper for the standard Python interpreter that makes interactive use more user-friendly. On UNIX-based platforms, you simply execute the **ipython** command in the terminal. On Windows, you should have an IPython choice in the Anaconda menu.

With Python up and running, you can now load the SWAT package as follows:

```
In [1]: import swat
```

Now that we have Python and SWAT installed, we can connect to CAS.

CONNECTING TO CAS

We assume that you already have a running CAS server you can connect to. Describing the installation and startup of CAS is beyond the scope of this paper. There are four pieces of information that you need to connect to CAS from SWAT: 1) host name, 2) port number, 3) user name, and 4) password. The host name is the name of the server that CAS is running on. This can also be an IP address. The port number is the port that SWAT connects to. As mentioned in the previous section, you might be able to connect to only the REST port of CAS if a platform-specific SWAT installer was not available for your platform. Finally, a user name and password are required to authenticate to the server.

The easiest way to create a connection to CAS is to specify all of these explicitly to the **CAS** class constructor in the SWAT package:

```
In [2]: conn = swat.CAS('cas.mycompany.com', 5570, 'username', 'password')
```

After you have a connection to CAS, you can try running a simple action like **serverstatus** to verify that the connection is working:

```
In [3]: conn.serverstatus()
Out[3]:
[About]

 {'CAS': 'Cloud Analytic Services',
  'Copyright': 'Copyright © 2014-2016 SAS Institute Inc. All Rights
                Reserved.',
  'System': {'Hostname': 'cas.mycompany.com',
   'Model Number': 'x86_64',
   'OS Family': 'LIN X64',
   'OS Name': 'Linux',
   'OS Release': '2.6.32-504.12.2.el6.x86_64',
   'OS Version': '#1 SMP Sun Feb 1 12:14:02 EST 2015'},
  'Version': '3.02',
  'VersionLong': 'V.03.02M0D12082016',
  'license': {'expires': '02Feb2017:00:00:00',
   'gracePeriod': 62,
   'site': 'SAS Institute Inc.',
   'siteNum': 1,
   'warningPeriod': 31}}

[server]

 Server Status

    nodes  actions
 0      1       10

[nodestatus]

 Node Status

                 name        role   uptime  running  stalled
 0  cas.mycompany.com  controller  387.823        0        0

+ Elapsed: 0.000662s, mem: 0.0934mb
```

If you feel uneasy about putting your user name and password in your program, SWAT supports Authinfo files for storing that information so it can be looked up in a more secure manner. We won't go into the details of that here. The documentation in the GitHub project outlines the details of setting that up.

Now that we have a connection to a CAS server, let's try working with some CAS actions.

WORKING WITH CAS ACTIONS

We have seen the output of the **serverstatus** action, but you might be wondering what other actions are available. There are a few ways of displaying them. The first is using the tab completion feature of IPython:

```
In [4]: conn.<tab>
Display all 374 possibilities? (y or n)
conn.about                              conn.listnodes
conn.accesscontrol.addacaction          conn.listresults
conn.accesscontrol.addacactionset       conn.listsasopts
conn.accesscontrol.addaccaslib          conn.listservopts
conn.accesscontrol.addaccolumn          conn.listsessions
conn.accesscontrol.addactable           conn.listsessopts
conn.accesscontrol.assumerole           conn.loadactionset
conn.accesscontrol.checkinallobjects    conn.loaddatasource
conn.accesscontrol.checkoutobject       conn.loadindex
conn.accesscontrol.commitactrans        conn.loadlibrefs
conn.accesscontrol.completebackup       conn.loadsasstate
conn.accesscontrol.createbackup         conn.loadtable
conn.accesscontrol.deletebwlist         conn.log

... truncated ...
```

This displays all CAS action sets, actions, and other attributes on the connection, but it does give you a general idea of what's available. You can also ask CAS directly what actions are available by using the **help** action.

```
In [5]: conn.help()
NOTE: Available Action Sets and Actions:
...
NOTE:    builtins
NOTE:       addNode - Adds a machine to the server
NOTE:       removeNode - Remove one or more machines from the server
NOTE:       help - Shows the parameters for an action or lists all
                   available actions
NOTE:       listNodes - Shows the host names used by the server
NOTE:       loadActionSet - Loads an action set for use in this session
NOTE:       installActionSet - Loads an action set in new sessions
                               automatically
NOTE:       log - Shows and modifies logging levels
NOTE:       queryActionSet - Shows whether an action set is loaded
NOTE:       queryName - Checks whether a name is an action or action set
name
NOTE:       reflect - Shows detailed parameter information for an action or
                      all actions in an action set
NOTE:       serverStatus - Shows the status of the server
NOTE:       about - Shows the status of the server
NOTE:       shutdown - Shuts down the server
NOTE:       userInfo - Shows the user information for your connection
NOTE:       actionSetInfo - Shows the build information from loaded action
sets
NOTE:       history - Shows the actions that were run in this session
NOTE:       casCommon - Provides parameters that are common to many actions
NOTE:       ping - Sends a single request to the server to confirm that the
                   connection is working
NOTE:       echo - Prints the supplied parameters to the client log
NOTE:       modifyQueue - Modifies the action response queue settings
NOTE:       getLicenseInfo - Shows the license information for a SAS
                             product
NOTE:       refreshLicense - Refresh SAS license information from a file
NOTE:       httpAddress - Shows the HTTP address for the server monitor
```

```
... truncated ...
```

The **help** action prints a lot of information like tab-completion, but in this case you also get a short description of each action. To get help for a particular action, the easiest way is to use IPython's **?** operator. This displays the Python docstring on the object.

```
In [6]: conn.addnode?
Type:           builtins.Addnode
String form:    ?.builtins.Addnode()
File:           actions.py
Signature:      conn.addnode(role=None, node=None, **kwargs)
Docstring:
Adds a machine to the server

Parameters
----------
role : string, optional
    specifies the role for the machine. Controllers are added as backup
    controllers. Only two controllers are supported.
    Default: captain
    Values: captain, controller, general, worker

node : list, optional
    specifies the host names of the machines to add to the server.
    Default: []
    Note: Value range is 1 <= n < inf

Returns
-------
Addnode object

... truncated ...
```

All of this information is downloaded from the CAS server and formatted when the actions are loaded on the server. This gives the benefit that the documentation displayed on the client can never be out of date with the actions on the server.

The keyword arguments to a Python method (such as **addnode**, **serverstatus,** or **help**) are used as the parameters to the corresponding CAS action. Let's try a new action called **getsessopt**. This action retrieves the value of a session option. It takes a single argument called **name**. We can get the value of the **locale** option as follows:

```
In [7]: conn.getsessopt(name='locale')
Out[7]:
[locale]

 'en_US'

+ Elapsed: 0.000253s, mem: 0.0634mb
```

Parameters can be in the form of many data types such as strings, integers, floating point numbers, lists, and dictionaries. The documentation for each action specifies which data type is required for each parameter. We will get into more advanced parameters in later sections, but first let's look at the return values of CAS actions.

HANDLING CAS ACTION RESULTS

So far we have simply allowed IPython to display the output of our actions. The result of a CAS action call is always a **CASResults** object. The **CASResults** object is a subclass of Python's **collections.OrderedDict** (which is a dictionary with keys that stay in the order in which they were inserted). Let's look at the output of the **serverstatus** action again. However, this time we will capture the output into a variable first.

```
In [8]: status = conn.serverstatus()

In [9]: status
Out[9]:
[About]

 {'CAS': 'Cloud Analytic Services',
  'Copyright': 'Copyright © 2014-2016 SAS Institute Inc. All Rights
               Reserved.',
  'System': {'Hostname': 'cas.mycompany.com',
   'Model Number': 'x86_64',
   'OS Family': 'LIN X64',
   'OS Name': 'Linux',
   'OS Release': '2.6.32-504.12.2.el6.x86_64',
   'OS Version': '#1 SMP Sun Feb 1 12:14:02 EST 2015'},
  'Version': '3.02',
  'VersionLong': 'V.03.02M0D12082016',
  'license': {'expires': '02Feb2017:00:00:00',
   'gracePeriod': 62,
   'site': 'SAS Institute Inc.',
   'siteNum': 1,
   'warningPeriod': 31}}

[server]

 Server Status

    nodes  actions
 0      1       10

[nodestatus]

 Node Status

                 name        role   uptime  running  stalled
 0  cas.mycompany.com  controller  387.823        0        0

+ Elapsed: 0.000662s, mem: 0.0934mb
```

In the output above, the keys of the result are displayed in brackets. The values of the result are displayed after the key name that they belong to. We can look at the keys programmatically using the **keys** method of the **CASResults** object[1]:

```
In [10]: list(status.keys())
```

[1] We use the **list** function around the call to the **keys** method to cover rendering differences between Python 2 and Python 3.

```
Out[10]: ['About', 'server', 'nodestatus']
```

We can access keys individually using Python's dictionary syntax as well.

```
In [11]: status['server']
Out[11]:
[server]

 Server Status

    nodes  actions
 0      1       10
```

The values of the **CASResults** object vary from action to action. They can be a scalar-valued items such as a string or floating point value, or they can be more complex objects such as dictionaries or Pandas DataFrames. We can print the types of the values of the results above using Python's **type** function.

```
In [12]: for key, value in status.items():
    ...:     print(key, type(value))
Out[12]:
About <class 'dict'>
server <class 'swat.dataframe.SASDataFrame'>
nodestatus <class 'swat.dataframe.SASDataFrame'>
```

In this case, the 'About' key contains a dictionary, and the 'server' and 'nodestatus' keys contain DataFrames. A **SASDataFrame** is equivalent to a Pandas **DataFrame**. It simply contains extra metadata about the table and columns such as labels, formats, and so on.

Since the value in 'nodestatus' is a **DataFrame**, we can perform typical **DataFrame** operations on it just as we would with any other **DataFrame**. In the code below, we show the results of the **columns** attribute and the **info** method.

```
In [13]: status['nodestatus'].columns
Out[13]: Index(['name', 'role', 'uptime', 'running', 'stalled'],
               dtype='object')

In [14]: status['nodestatus'].info()
Out[14]:
<class 'swat.dataframe.SASDataFrame'>
RangeIndex: 1 entries, 0 to 0
Data columns (total 5 columns):
name       1 non-null object
role       1 non-null object
uptime     1 non-null float64
running    1 non-null int32
stalled    1 non-null int32
dtypes: float64(1), int32(2), object(2)
memory usage: 112.0+ bytes
```

In additon to the actual values returned, CAS also returns metrics about the action execution. Let's look at those next.

CAS ACTION METRICS

At the end of each CAS action execution, you might have noticed a line at the end that looks like the following:

```
+ Elapsed: 0.0196s, user: 0.019s, sys: 0.001s, mem: 0.315mb
```

This gives you a brief summary of various timings and memory consumption statistics. There are several other pieces of information available about performance and the disposition of the result as well. The most commonly accessed attributes on the **CASResults** object are **severity**, **status**, and **messages**. The **severity** attribute contains a return code that is either 0 (for no reported problems), 1 (warnings were produced), or 2 (errors were produced). The **status** attribute contains a human-readable message summarizing the reason for any errors; if no errors were produced, the string is empty. The **messages** attribute contains any messages that were generated by the action. These are typically printed to the terminal as well, but it is sometimes handy to have them in a variable that you can use in post-processing.

In addition to basic information about the result of the action, there is also a **performance** attribute on the **CASResults** object. It contains various pieces of information about timings, memory usage, and grid usage.

```
In [15]: status.performance
Out[15]: CASPerformance(cpu_system_time=0.001, cpu_user_time=0.018997,
data_movement_bytes=0, data_movement_time=0.0, elapsed_time=0.019644,
memory=330688, memory_os=8441856, memory_quota=12111872, system_cores=32,
system_nodes=1, system_total_memory=202931654656)
```

Each of the parameters shown in **Out[15]**, is available as an attribute on the **performance** object.

```
In [16]: status.performance.cpu_system_time
Out[16]: 0.001
```

These attributes should give you enough diagnostic information to handle errors, or simply report relevant performance information about your analyses. With all of this information under our belts, we can move on to loading some data and doing some real work.

LOADING DATA

Before we can do any sort of statistical analyses, we need to get some data loaded into CAS first. There are many ways to load data, so we'll just cover the simplest methods here. For larger data sets, you will likely want to have the data located on the same machine that CAS is running on so that you don't have to transfer the data each time it is loaded into a CAS table. For smaller data sets, it might not matter as much. We will start with smaller data sets located on the client side first.

LOADING DATA SETS FROM THE CLIENT SIDE

Loading data from the client side into CAS is fairly easy if it's in a common format such as CSV. You can use the **read_csv** method on your **CAS** connection object to read a CSV file (or URL) and load it into a CAS table.

```
In [17]: tbl = conn.read_csv('https://raw.githubusercontent.com/'
   ....:                     'sassoftware/sas-viya-programming/'
   ....:                     'master/data/cars.csv')

In [18]: tbl
Out[18]: CASTable('_T_5FEPFN5Y_DPXS4G22_6PPE7JDUCF',
                  caslib='CASUSER(kesmit)')
```

Loading a table in this manner creates a table on the server with a generated table name in the active caslib. We won't go into detail about caslibs in this paper. They are essentially resources in the server that configure data sources, authentication, and authorization settings for the data source and loaded tables. They also act as namespaces for in-memory tables, which is what we are using them for here. We use the default caslib for all of our examples in this paper.

It is possible to set a specific name for the output table using the **casout=** parameter so that you don't have to look at obscure generated table names.

```
In [19]: tbl = conn.read_csv('https://raw.githubusercontent.com/'
   ....:                     'sassoftware/sas-viya-programming/'
   ....:                     'master/data/cars.csv', casout='cars')

In [20]: tbl
Out[20]: CASTable('cars', caslib='CASUSER(kesmit)')
```

The result of the **read_csv** method is a **CASTable** object. The **CASTable** object is a very rich interface to tables in the CAS server. CAS actions can be executed through the **CASTable** object, and it supports much of the Pandas **DataFrame** API so that it looks and feels like a **DataFrame**, but the processing is done within CAS.

Now that we have a **CASTable** object that references a table in our CAS server, let's get some information about it. The **tableinfo** and **columninfo** actions give you information about the table as a whole and the columns in the table, respectively.

```
In [21]: tbl.tableinfo()
Out[21]:
[TableInfo]

    Name  Rows  Columns Encoding CreateTimeFormatted  \
 0  CARS   428       15    utf-8  16Dec2016:15:43:47

      ModTimeFormatted JavaCharSet    CreateTime       ModTime  \
 0  16Dec2016:15:43:47        UTF8  1.797522e+09  1.797522e+09

    Global  Repeated  View SourceName SourceCaslib  Compressed  \
 0       0         0     0                                   0

   Creator Modifier
 0  kesmit

+ Elapsed: 0.000625s, mem: 0.1mb

In [22]: tbl.columninfo()
Out[22]:
[ColumnInfo]

          Column  ID     Type  RawLength  FormattedLength  NFL  NFD
 0          Make   1  varchar         13               13    0    0
 1         Model   2  varchar         39               39    0    0
 2          Type   3  varchar          6                6    0    0
 3        Origin   4  varchar          6                6    0    0
 4    DriveTrain   5  varchar          5                5    0    0
 5          MSRP   6   double          8               12    0    0
 6       Invoice   7   double          8               12    0    0
 7    EngineSize   8   double          8               12    0    0
 8     Cylinders   9   double          8               12    0    0
 9    Horsepower  10   double          8               12    0    0
 10     MPG_City  11   double          8               12    0    0
```

```
 11  MPG_Highway  12   double          8                12    0    0
 12       Weight  13   double          8                12    0    0
 13    Wheelbase  14   double          8                12    0    0
 14       Length  15   double          8                12    0    0

+ Elapsed: 0.000753s, user: 0.001s, mem: 0.172mb
```

We can fetch a sample of the data using the **fetch** action.

```
In [23]: tbl.fetch(to=3)
Out[23]:
[Fetch]

 Selected Rows from Table CARS

     Make           Model   Type Origin DriveTrain     MSRP  Invoice  \
 0  Acura             MDX    SUV   Asia        All  36945.0  33337.0
 1  Acura  RSX Type S 2dr  Sedan   Asia      Front  23820.0  21761.0
 2  Acura         TSX 4dr  Sedan   Asia      Front  26990.0  24647.0

    EngineSize  Cylinders  Horsepower  MPG_City  MPG_Highway  Weight  \
 0         3.5        6.0       265.0      17.0         23.0  4451.0
 1         2.0        4.0       200.0      24.0         31.0  2778.0
 2         2.4        4.0       200.0      22.0         29.0  3230.0

    Wheelbase  Length
 0      106.0   189.0
 1      101.0   172.0
 2      105.0   183.0

+ Elapsed: 0.00403s, user: 0.001s, sys: 0.002s, mem: 1.7mb
```

Now that we have verified that the table exists in the server and contains the expected data, let’s look at the next method of loading data.

LOADING DATA SETS FROM THE SERVER SIDE

As we mentioned in the previous section, if you have large data sets, you probably want to load the data files on to the CAS server and load them from there so that you don’t have to transfer the data from the client each time it is loaded. To load data from a file, the file must be in a location that is accessible from a caslib. To keep things simple, we are going to assume that you have the data file in your home directory which is accessible through the Casuser caslib.

To load data files from the server side, you use the **loadtable** action.

```
In [24]: out = conn.loadtable(path='cars.csv', casout='cars2')

In [25]: out
Out[25]:
[caslib]

 'CASUSER(kesmit)'

[tableName]

 'CARS2'
```

```
[casTable]

 CASTable('CARS2', caslib='CASUSER(kesmit)')

+ Elapsed: 0.11s, user: 0.056s, sys: 0.043s, mem: 64.8mb
```

In this case, we are calling a CAS action rather than a method on the connection object so the result is a **CASResults** object. However, we can get the **CASTable** object from the casTable key in the result.

```
In [26]: tbl2 = out['casTable']

In [27]: tbl2
Out[27]: CASTable('CARS2', caslib='CASUSER(kesmit)')
```

Loading tables from the server and getting the **CASTable** from the result is such a common thing to do that a small wrapper method was added to the connection object in order to simplify the process. The method is called **load_path**. It takes the same parameters as the **loadtable** action, but just returns the **CASTable** object.

```
In [28]: tbl3 = conn.load_path(path='cars.csv', casout='cars3')

In [29]: tbl3
Out[29]: CASTable('CARS3', caslib='CASUSER(kesmit)')
```

Of course, once the table is loaded, it works just like the **CASTable** that was loaded from the client side.

```
In [30]: tbl3.tableinfo()
Out[30]:
[TableInfo]

     Name  Rows  Columns Encoding CreateTimeFormatted  \
 0  CARS3   428       15    utf-8  16Dec2016:16:06:30

      ModTimeFormatted JavaCharSet    CreateTime       ModTime  \
 0  16Dec2016:16:06:30        UTF8  1.797524e+09  1.797524e+09

    Global  Repeated  View     SourceName     SourceCaslib  \
 0       0         0     0  data/cars.csv  CASUSER(kesmit)

    Compressed Creator Modifier
 0           0  kesmit

+ Elapsed: 0.00084s, user: 0.001s, mem: 0.102mb

In [31]: tbl3.columninfo()
Out[31]:
[ColumnInfo]

          Column  ID     Type  RawLength  FormattedLength  NFL  NFD
 0          Make   1  varchar         13               13    0    0
 1         Model   2  varchar         39               39    0    0
 2          Type   3  varchar          6                6    0    0
 3        Origin   4  varchar          6                6    0    0
 4    DriveTrain   5  varchar          5                5    0    0
 5          MSRP   6   double          8               12    0    0
```

```
6        Invoice   7   double          8              12     0     0
7     EngineSize   8   double          8              12     0     0
8      Cylinders   9   double          8              12     0     0
9     Horsepower  10   double          8              12     0     0
10      MPG_City  11   double          8              12     0     0
11   MPG_Highway  12   double          8              12     0     0
12        Weight  13   double          8              12     0     0
13     Wheelbase  14   double          8              12     0     0
14        Length  15   double          8              12     0     0

+ Elapsed: 0.000759s, mem: 0.17mb
```

The examples of loading data in this section and the previous section demonstrate only the simplest methods of loading data. There are various data file formats that can be read, many options to modify the data types and column metadata, as well as ways of loading data from non-file-based sources such as databases. These topics are much too large to go into in this paper, so we'll refer you to the SAS documentation for more information.

Now that we have some data to work with, we can move on to some more interesting work of performing analytics on it.

COMPUTING SIMPLE STATISTICS

Before getting into more advanced modeling, we can obtain quite a bit of information about our data using CAS actions for simple statistics. These actions are in an action set called **simple**. The **simple** action set should already be loaded. You can verify this by running the **actionsetinfo** action (in addition to running the action, we are also accessing the 'actionset' column of the **DataFrame** in the 'setinfo' key of the result in the code below).

```
In [32]: conn.actionsetinfo().setinfo.actionset
Out[32]:
0       accessControl
1       accessControl
2            builtins
3       configuration
4      dataPreprocess
5            dataStep
6          percentile
7              search
8             session
9         sessionProp
10             simple
11              table
Name: actionset, dtype: object
```

As you can see, the **simple** action set is already loaded on our system. If you don't see **simple** in your list of action sets, you can load it using the **loadactionset** action.

```
In [33]: conn.loadactionset('simple')
NOTE: Added action set 'simple'.
Out[33]:
[actionset]

 'simple'

+ Elapsed: 0.0192s, user: 0.018s, sys: 0.001s, mem: 0.282mb
```

Using IPython's **?** operator for displaying help, we can display the following on the **simple** attribute of the connection object.

```
In[34]: conn.simple?
...

Analytics

Actions
-------
simple.correlation : Generates a matrix of Pearson product-moment
                     correlation coefficients
simple.crosstab    : Performs one-way or two-way tabulations
simple.distinct    : Computes the distinct number of values of the
                     variables in the variable list
simple.freq        : Generates a frequency distribution for one or
                     more variables
simple.groupby     : Builds BY groups in terms of the variable value
                     combinations given the variables in the variable
                     list
simple.mdsummary   : Calculates multidimensional summaries of numeric
                     variables
simple.numrows     : Shows the number of rows in a Cloud Analytic
                     Services table
simple.paracoord   : Generates a parallel coordinates plot of the
                     variables in the variable list
simple.regression  : Performs a linear regression up to 3rd-order
                     polynomials
simple.summary     : Generates descriptive statistics of numeric
                     variables such as the sample mean, sample
                     variance, sample size, sum of squares, and so on
simple.topk        : Returns the top-K and bottom-K distinct values of
                     each variable included in the variable list based
                     on a user-specified ranking order
```

Now that we have this action set loaded, let's try the **summary** action on our previously loaded table. We display only a few rows of the result below to save space.

```
In [35]: tbl.summary()
Out[35]:
[Summary]

 Descriptive Statistics for CARS

        Column      Min       Max      N  NMiss          Mean  \
 0        MSRP  10280.0  192465.0  428.0    0.0  32774.855140
 1     Invoice   9875.0  173560.0  428.0    0.0  30014.700935
 ..        ...      ...       ...    ...   ...           ...
 8   Wheelbase     89.0     144.0  428.0    0.0    108.154206
 9      Length    143.0     238.0  428.0    0.0    186.362150

            Sum           Std      StdErr           Var  \
 0   14027638.0  19431.716674  939.267478  3.775916e+08
 1   12846292.0  17642.117750  852.763949  3.112443e+08
 ..         ...           ...         ...           ...
```

```
 8      46290.0      8.311813    0.401767  6.908624e+01
 9      79763.0     14.357991    0.694020  2.061519e+02

              USS            CSS          CV      TValue           ProbT
 0   6.209854e+11  1.612316e+11  59.288490   34.894059  4.160412e-127
 1   5.184789e+11  1.329013e+11  58.778256   35.196963  2.684398e-128
 ..           ...            ...        ...         ...            ...
 8   5.035958e+06  2.949982e+04   7.685150  269.196577   0.000000e+00
 9   1.495283e+07  8.802687e+04   7.704349  268.525733   0.000000e+00

 [10 rows x 15 columns]

+ Elapsed: 0.00617s, user: 0.006s, sys: 0.002s, mem: 1.75mb
```

You can see that we get statistics such as the minimum value, the maximum value, the number of observations, the number of missing values, and so on. It is also possible to retrieve only the statistics you want by using the **subset=** option.

```
In [36]: tbl.summary(subset=['Sum', 'Std', 'StdErr'])
Out[36]:
[Summary]

 Descriptive Statistics for CARS

        Column         Sum           Std      StdErr
 0        MSRP  14027638.0  19431.716674  939.267478
 1     Invoice  12846292.0  17642.117750  852.763949
 ..        ...         ...           ...         ...
 8   Wheelbase     46290.0      8.311813    0.401767
 9      Length     79763.0     14.357991    0.694020

 [10 rows x 4 columns]

+ Elapsed: 0.00618s, user: 0.003s, sys: 0.005s, mem: 1.74mb
```

Grouping results by data values can also be done. In the example below, we use the **groupby** method on the **CASTable** object. This works very much like the **groupby** method on Pandas **DataFrames**. In its simplest form, it takes a string or list of strings as the variable values to group by.

```
In [37]: tbl.groupby('Origin').summary(subset=['Sum', 'Std', 'StdErr'])
Out[37]:
[ByGroupInfo]

 ByGroupInfo

    Origin Origin_f   _key_
 0    Asia     Asia    Asia
 1  Europe   Europe  Europe
 2     USA      USA     USA

[ByGroup1.Summary]

 Descriptive Statistics for CARS

            Column        Sum           Std      StdErr
 Origin
```

```
 Asia         MSRP  3909129.0  11321.069675  900.655944
 Asia      Invoice  3571144.0   9842.984880  783.065832
 ...           ...        ...           ...         ...
 Asia    Wheelbase    16730.0      7.735249    0.615383
 Asia       Length    28885.0     12.564148    0.999550

 [10 rows x 4 columns]

...

[ByGroup3.Summary]

 Descriptive Statistics for CARS

           Column        Sum           Std      StdErr
 Origin
 USA          MSRP  4171484.0  11711.982506  965.988036
 USA       Invoice  3814553.0  10518.722194  867.569584
 ...           ...        ...           ...         ...
 USA     Wheelbase    16467.0      8.788590    0.724871
 USA        Length    28511.0     15.305265    1.262357

 [10 rows x 4 columns]

+ Elapsed: 0.0106s, user: 0.008s, sys: 0.005s, mem: 1.74mb
```

From the output above, you can see that we get multiple tables back when using BY groups. The first table is a summary of all of the BY groups contained in the output. This can be useful if the number of BY groups is very large and you want to know at the beginning what to expect in the rest of the output. The remaining tables contain the summary statistics for each BY group. The keys of the **CASResults** object are the output table name (in this case, “Summary”) prefixed by “ByGroup#” where # is the index of the BY group. If you prefer to have all of the BY groups in one table, you can concatenate them using the **concat_bygroups** method of the **CASResults** object.

```
In [38]: out = tbl.groupby('Origin').summary(subset=['Sum', 'Std',
                                                     'StdErr'])

In [39]: out.concat_bygroups()
Out[39]:
[Summary]

 Descriptive Statistics for CARS

               Column        Sum           Std      StdErr
 Origin
 Asia            MSRP  3909129.0  11321.069675  900.655944
 Asia         Invoice  3571144.0   9842.984880  783.065832
 Asia      EngineSize      438.3      0.902310    0.071784
 Asia       Cylinders      809.0      1.269008    0.101602
 Asia      Horsepower    30131.0     59.392627    4.725024
 ...              ...        ...           ...         ...
 USA         MPG_City     2804.0      3.982992    0.328512
 USA      MPG_Highway     3824.0      5.396582    0.445103
 USA           Weight   554183.0    855.305524   70.544411
 USA        Wheelbase    16467.0      8.788590    0.724871
 USA           Length    28511.0     15.305265    1.262357
```

```
[30 rows x 4 columns]
```

Let's look at another action in the **simple** action set: **correlation**. It works in the same way as the **summary** action, it is called like a method on the **CASTable** object. By default, the **correlation** action will also return some of the summary statistics in a separate table, since we have already looked at the **summary** action, we will disable those by setting the **simple=** parameter to **False**.

```
In [40]: tbl.correlation(simple=False)
Out[40]:
[Correlation]

 Pearson Correlation Coefficients for CARS

      Variable      MSRP   Invoice  EngineSize  Cylinders  \
 0        MSRP  1.000000  0.999132    0.571753   0.649742
 1     Invoice  0.999132  1.000000    0.564498   0.645226
 2  EngineSize  0.571753  0.564498    1.000000   0.908002
 3   Cylinders  0.649742  0.645226    0.908002   1.000000
 4  Horsepower  0.826945  0.823746    0.787435   0.810341
 5    MPG_City -0.475020 -0.470442   -0.709471  -0.684402
 6 MPG_Highway -0.439622 -0.434585   -0.717302  -0.676100
 7      Weight  0.448426  0.442332    0.807867   0.742209
 8   Wheelbase  0.152000  0.148328    0.636517   0.546730
 9      Length  0.172037  0.166586    0.637448   0.547783

   Horsepower  MPG_City  MPG_Highway    Weight  Wheelbase    Length
 0   0.826945 -0.475020    -0.439622  0.448426   0.152000  0.172037
 1   0.823746 -0.470442    -0.434585  0.442332   0.148328  0.166586
 2   0.787435 -0.709471    -0.717302  0.807867   0.636517  0.637448
 3   0.810341 -0.684402    -0.676100  0.742209   0.546730  0.547783
 4   1.000000 -0.676699    -0.647195  0.630796   0.387398  0.381554
 5  -0.676699  1.000000     0.941021 -0.737966  -0.507284 -0.501526
 6  -0.647195  0.941021     1.000000 -0.790989  -0.524661 -0.466092
 7   0.630796 -0.737966    -0.790989  1.000000   0.760703  0.690021
 8   0.387398 -0.507284    -0.524661  0.760703   1.000000  0.889195
 9   0.381554 -0.501526    -0.466092  0.690021   0.889195  1.000000

+ Elapsed: 0.00583s, user: 0.005s, sys: 0.003s, mem: 1.74mb
```

Correlation matrices are usually easier to interpret using heatmaps, so let's plot the above data using a Python package called Seaborn. We first want to convert the output above to a lower-triangular matrix, then we'll create the plot. Unfortunately, we don't have the space in this paper to explain in detail everything we are doing in this example, so further study will have to be an exercise for the reader.

```
# Import require packages
In [41]: import numpy as np

In [42]: import seaborn as sns

In [43]: from matplotlib.pyplot import show

# Run the correlation action
In [44]: corr = tbl.correlation(simple=False).Correlation

# Set the Variable column as the row labels
```

```
In [45]: corr = corr.set_index('Variable')

# Create a lower-triangular matrix
In [46]: corrl = corr.where(np.tril(np.ones(corr.shape),
                            -1).astype(np.bool))

# Create the heatmap
In [47]: with sns.axes_style('white'):
    ....:     hm = sns.heatmap(corrl)
    ....:     hm.set_yticklabels(corrl.index.str.replace('_', ' '),
    ....:                        rotation=0)
    ....:     hm.set_xticklabels(corrl.index.str.replace('_', ' '),
    ....:                        rotation=-30)
    ....:     show()
```

The resulting graph from the example code above is shown here.

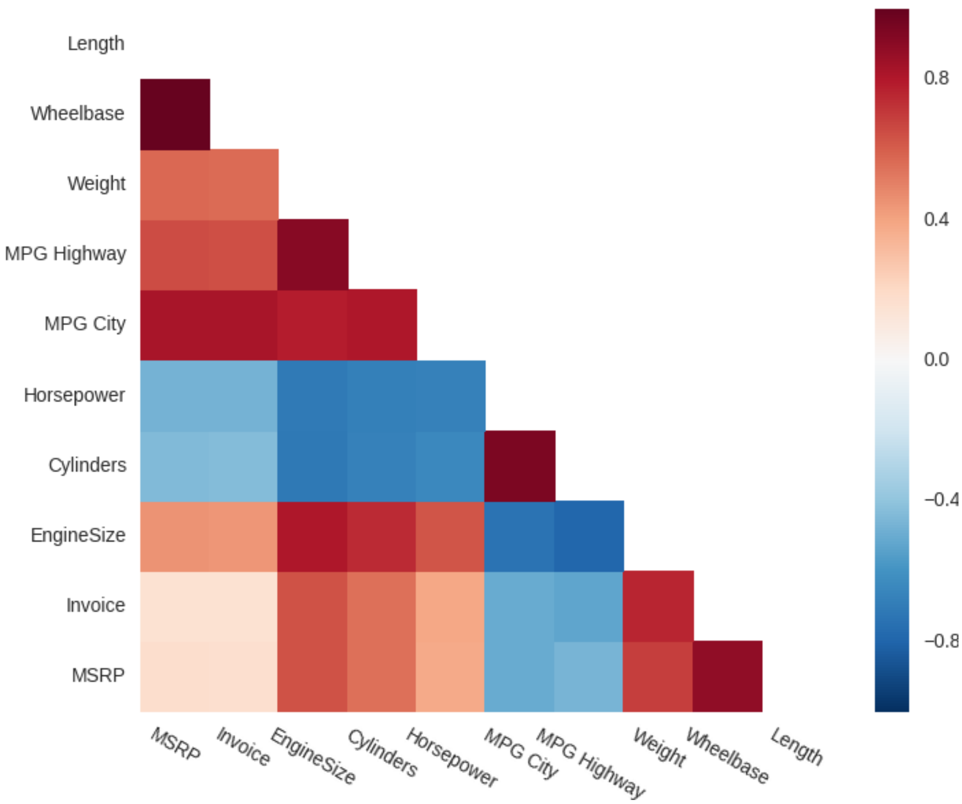

Figure 1. Heatmap displaying the result of the correlation action

With the basics of running CAS actions under our belt, we can move on to some modeling examples.

BUILDING MODELS

CAS also provides a variety of statistical and machine learning models for you to model structured and unstructured data. These models are grouped into action sets based on functionality. For instance, the regression action set contains three different regression models: linear regressions, logistic regressions, and generalized linear models.

```
In [48]: conn.loadactionset('regression')
```

```
In [49]: conn.regression?

Actions
-------
regression.genmod   : Fits generalized linear regression models
regression.glm      : Fits linear regression models using the method of
                      least squares
regression.logistic : Fits logistic regression models
```

Let's continue to work on the cars data set you have loaded to the CAS server and build a simple linear regression model to predict the MSRP value of cars.

```
In [50]: tbl.glm(target='MSRP', inputs=['MPG_City'])
Out[50]:

[ModelInfo]

 Model Information

         RowId        Description Value
 0        DATA        Data Source  CARS
 1  RESPONSEVAR  Response Variable  MSRP

[NObs]

 Number of Observations

   RowId                  Description  Value
 0  NREAD  Number of Observations Read  428.0
 1  NUSED  Number of Observations Used  428.0

[Dimensions]

 Dimensions

      RowId           Description  Value
 0  NEFFECTS     Number of Effects      2
 1    NPARMS  Number of Parameters      2

[ANOVA]

 Analysis of Variance

   RowId           Source     DF            SS            MS  \
 0  MODEL           Model    1.0  3.638090e+10  3.638090e+10
 1  ERROR           Error  426.0  1.248507e+11  2.930768e+08
 2  TOTAL  Corrected Total  427.0  1.612316e+11           NaN

      FValue         ProbF
 0  124.13436  1.783404e-25
 1        NaN           NaN
 2        NaN           NaN

[FitStatistics]

 Fit Statistics
```

```
       RowId Description         Value
 0       RMSE    Root MSE  1.711949e+04
 1    RSQUARE    R-Square  2.256437e-01
 2     ADJRSQ    Adj R-Sq  2.238260e-01
 3        AIC         AIC  8.776260e+03
 4       AICC        AICC  8.776316e+03
 5        SBC         SBC  8.354378e+03
 6  TRAIN_ASE         ASE  2.917073e+08

[ParameterEstimates]

 Parameter Estimates

      Effect  Parameter  DF      Estimate       StdErr     tValue  \
 0  Intercept  Intercept   1  68124.606698  3278.919093  20.776544
 1   MPG_City   MPG_City   1  -1762.135298   158.158758 -11.141560

          Probt
 0  1.006169e-66
 1  1.783404e-25

[Timing]

 Task Timing

             RowId                   Task      Time   RelTime
 0           SETUP     Setup and Parsing  0.391366  0.283194
 1    LEVELIZATION          Levelization  0.315693  0.228437
 2  INITIALIZATION  Model Initialization  0.000099  0.000072
 3            SSCP      SSCP Computation  0.512247  0.370665
 4         FITTING         Model Fitting  0.000415  0.000300
 5         CLEANUP               Cleanup  0.002838  0.002054
 6           TOTAL                 Total  1.381969  1.000000

+ Elapsed: 1.81s, user: 0.032s, sys: 0.066s, mem: 37.9mb
```

Compared to the actions in the **simple** action set, the **glm** action might requires a more complex and deeper parameter structure. In this case, it might be more convenient to define a new GLM model first and then specify the model parameters, step-by-step. In other words, the linear regression above can be rewritten as:

```
linear1 = tbl.Glm()
linear1.target = 'MSRP'
linear1.inputs = ['MPG_City']
linear1()
```

This approach enables you to reuse the code when you need to change only a few parameters of the model. For example, let us add a categorical predictor and display only the parameter estimation table:

```
In[51]: linear1.inputs = ['MPG_City','Origin']
   ...: linear1.nominals = ['Origin']
   ...: linear1.display.names = ['ParameterEstimates']
   ...: linear1()
Out[51]:
```

```
[ParameterEstimates]

 Parameter Estimates

       Effect  Origin      Parameter  DF      Estimate       StdErr  \
 0  Intercept              Intercept   1  57217.013184  2997.826305
 1   MPG_City               MPG_City   1  -1511.917596   143.404229
 2     Origin    Asia    Origin Asia   1    805.634901  1755.483912
 3     Origin  Europe  Origin Europe   1  19453.581452  1817.953561
 4     Origin     USA     Origin USA   0      0.000000          NaN

       tValue         Probt
 0  19.086167  4.565959e-59
 1 -10.543047  3.084694e-23
 2   0.458925  6.465234e-01
 3  10.700813  8.117258e-24
 4        NaN           NaN

+ Elapsed: 2.04s, user: 0.036s, sys: 0.095s, mem: 39.4mb
```

The **decisiontree** action set is another popular analytic action set. It provides three distinct tree-based models: decision tree, random forests, and gradient boosting. Unlike the regression action set, the **decisiontree** action set splits a model into different actions, each represents a typical step of a machine learning process such as training, scoring and score code generation (as SAS DATA step score code).

```
In [52]: conn.loadactionset('decisiontree')

In [52]: conn.decisiontree?

Actions
-------
decisiontree.dtreecode   : Generate DATA step scoring code from a
                           decision tree model
decisiontree.dtreemerge  : Merge decision tree nodes
decisiontree.dtreeprune  : Prune a decision tree
decisiontree.dtreescore  : Score a table using a decision tree model
decisiontree.dtreesplit  : Split decision tree nodes
decisiontree.dtreetrain  : Train a decision tree
decisiontree.forestcode  : Generate DATA step scoring code from a
                           forest model
decisiontree.forestscore : Score a table using a forest model
decisiontree.foresttrain : Train a forest
decisiontree.gbtreecode  : Generate DATA step scoring code from a
                           gradient boosting tree model
decisiontree.gbtreescore : Score a table using a gradient boosting
                           tree model
decisiontree.gbtreetrain : Train a gradient boosting tree
```

The models in the **decisiontree** action set support either continuous, binary or multilevel response variable. Let us fit a random forest model to predict whether a vehicle is from Asia, Europe, or United States.

```
In [53]: forest1 = tbl.Foresttrain()
    ...: forest1.target = 'Origin'
    ...: forest1.inputs = ['MPG_City','MPG_Highway','Type',
```

```
                          'Weight','Length','Cylinders']
    ...: forest1.nominals = ['Type','Cylinders']
    ...: forest1.casout = conn.CASTable('forestModel1', replace=True)
    ...: forest1()
Out[53]:

[ModelInfo]

 Forest for CARS

                                 Descr       Value
 0                     Number of Trees   50.000000
 1   Number of Selected Variables (M)    3.000000
 2                  Random Number Seed    0.000000
 3            Bootstrap Percentage (%)   63.212056
 4                      Number of Bins   20.000000
 5                 Number of Variables    6.000000
 6        Confidence Level for Pruning    0.250000
 7            Max Number of Tree Nodes   29.000000
 8            Min Number of Tree Nodes   11.000000
 9              Max Number of Branches    2.000000
 10             Min Number of Branches    2.000000
 11               Max Number of Levels    6.000000
 12               Min Number of Levels    6.000000
 13               Max Number of Leaves   15.000000
 14               Min Number of Leaves    6.000000
 15             Maximum Size of Leaves  229.000000
 16             Minimum Size of Leaves    5.000000
 17                  Out-of-Bag MCR (%)         NaN

[OutputCasTables]

                 casLib          Name  Rows  Columns  \
 0  CASUSERHDFS(ximeng)  forestModel1   804       38

                               casTable
 0  CASTable('forestModel1', caslib='CAS...

+ Elapsed: 3.8s, user: 0.114s, sys: 0.802s, mem: 25.7mb
```

The **foresttrain** action outputs two result tables: ModelInfo and OutputCasTables. The first table contains parameters that define the forest, parameters that define each individual tree, and tree statistics such as the minimum and maximum number of branches and levels. The second table provides information of the CAS table that stores the actual forest model.

Random forest models are also commonly used in variable selection, which is usually determined by the variable importance of the predictors in training the forest model. In the **foresttrain** action, this importance measure is defined as the total Gini reduction from all of the splits that use this predictor. You can request variable important using the **varimp** option and generate the variable importance using the Matplotlib package.

```
In [54]: forest1.varimp = True
    ...: result = forest1()
    ...: dfVarImp = result['DTreeVarImpInfo']
    ...:
    ...: import matplotlib.pyplot as plt
```

```
...: import numpy as np
...:
...: y_pos = np.arange(len(dfVarImp['Importance']))
...: plt.barh(y_pos, dfVarImp['Importance'], align='center')
...: plt.yticks(y_pos, dfVarImp['Variable'])
...: plt.xlabel('Variable Importance')
...: plt.show()
```

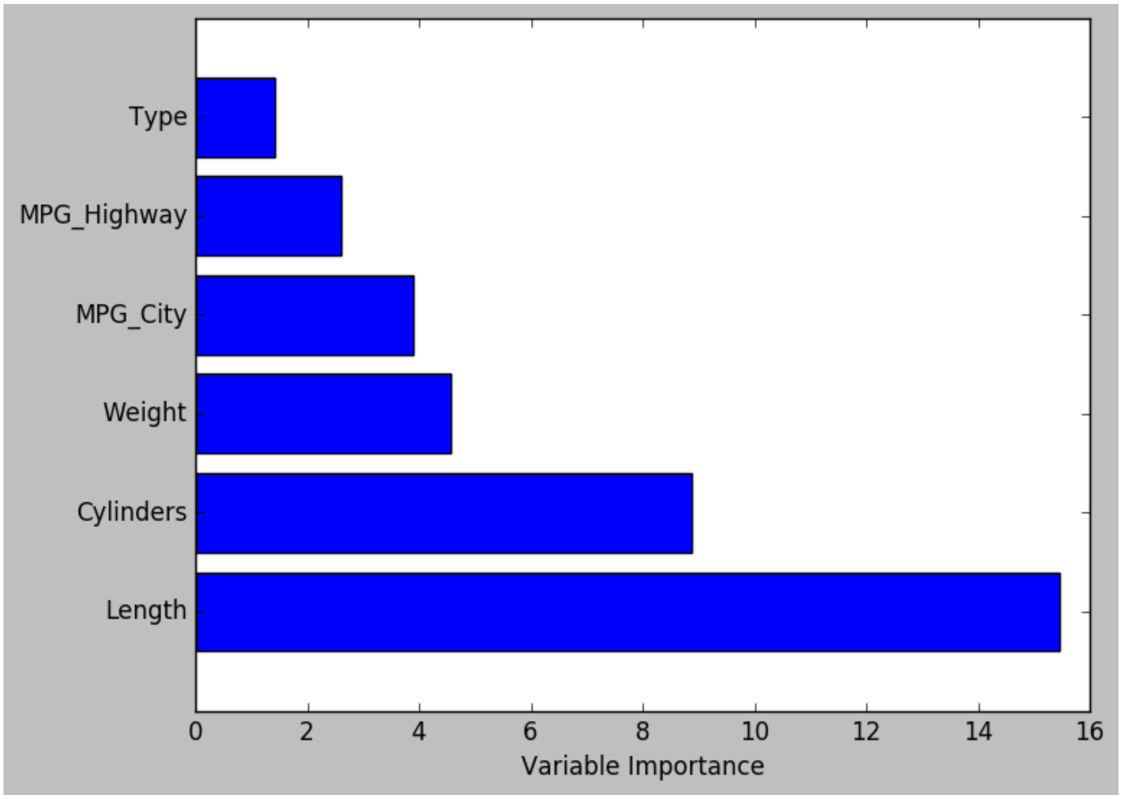

Figure 2. Variable Importance plot from the random forest model

To score the training data or the holdout data using the forest model, you can use the **forestscore** action.

```
In [55]: scored_data = conn.CASTable('scored_output', replace=True)
In [56]: tbl.forestscore(modeltable=conn.CASTable('forestModel1'),
                         casout=scored_data)
In [57]: scored_data.head()
Out[57]:
Selected Rows from Table SCORED_OUTPUT

  _RF_PredName_  _RF_PredP_  _RF_PredLevel_  _MissIt_  _Vote_
0          Asia        0.66             0.0       0.0    33.0
1          Asia        0.70             0.0       0.0    35.0
2          Asia        0.66             0.0       0.0    33.0
3          Asia        0.66             0.0       0.0    33.0
4          Asia        0.50             0.0       0.0    25.0
```

CLOSING THE CONNECTION

When you are finished with a CAS connection, it's always a good idea to close it explicitly.

```
In [58]: conn.close()
```

CONCLUSION

In this paper, we have covered everything from installing the Python client to SAS Viya, loading data into CAS, running CAS actions, to basic analytical modeling. In addition, we demonstrated the integration of CAS results with other Python packages such as the Matplotlib and Seaborn graphics packages. Having access to a third-party language interface to a SAS analytics engine is new territory for SAS, but we hope that we have shown that the integration between the two is quite natural and seamless.

RECOMMENDED READING

- *SAS® Viya: The Python Perspective*

CONTACT INFORMATION

Your comments and questions are valued and encouraged. Contact the authors at:

Kevin D Smith
SAS Institute, Inc.
Kevin.Smith@sas.com

Xiangxiang Meng, PhD
SAS Institute, Inc.
Xiangxiang.Meng@sas.com