Paper SAS668-2017

I Am Multilingual: A Comparison of the Python, Java, Lua, and REST Interfaces to SAS® Viya™

Xiangxiang Meng and Kevin D Smith, SAS Institute Inc.

ABSTRACT

The openness of SAS® Viya™, the new cloud analytic platform centered around SAS® Cloud Analytic Services (CAS), emphasizes a unified experience for data scientists. You can now execute the analytic capabilities of SAS® from different programming languages including Python, Java, and Lua, as well as use a RESTful endpoint to execute CAS actions directly. This paper provides an introduction to these programming language interfaces. For each language, we illustrate how the API is surfaced from the CAS server, the types of data that you can upload to a CAS server, and the result tables that are returned. This paper also provides a comprehensive comparison of using these programming languages to build a common analytical process, including connecting to a CAS server; exploring, manipulating, and visualizing data; and building statistical and machine learning models.

INTRODUCTION

This paper provides an introduction to the different programming interfaces to SAS® Cloud Analytic Services (CAS). CAS is the central analytic environment for SAS® Viya™, which enables a user to submit and execute the same analytic actions from different programming languages or SAS applications. Besides CAS-enabled SAS® procedures, CAS provides interfaces for programming languages such as Python, Java, and Lua. You can also submit actions over the HTTP and HTTPS protocols in other languages using the REST API that is surfaced by CAS.

We compare these interfaces and illustrate how to connect to the CAS server, submit CAS actions, and work with the results returned by CAS actions in Python, Lua, and Java. We then provide examples on data summarization, data exploration, and building analytic models. Finally, we cover some examples of how to use the REST interface to submit CAS actions.

IMPORTING CLIENT SIDE PACKAGES

In this paper, we assume you already have a running CAS server and have some data loaded to the CAS server. For each client (Python, Lua, or Java), you need to import the client side package provided by SAS. These are available for download from support.sas.com. In Python or Lua, This interface is called **SWAT** (Scripting Wrapper for Analytics Transfer). SWAT is a SAS architecture that enables you to interact with a CAS server from different scripting languages such as Python and Lua. The code below demonstrates how to load the SWAT package in Python and Lua.

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

Lua
```
> swat = require 'swat'
```

The Java **CAS Client** provides access to CAS using socket protocols. Unlike the scripting interfaces, the CAS Client is not based on the SWAT architecture; it is pure Java. You can import the Java classes individually:

Java
```
import com.sas.cas.CASActionResults;
import com.sas.cas.CASClient;
import com.sas.cas.CASClientInterface;
import com.sas.cas.CASValue;
```

Alternatively, you can load all classes in com.sas.cas:

```
Java
import com.sas.cas.*;
```

CONNECTING TO CAS

In this paper, we assume that a CAS server is running. To connect to a CAS server, you need to know the host name or the IP address of the server, and the port number. You must have an authenticated user account. In Python and Lua, you can use the **CAS** object to set up a new connection.

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

```
Lua
> conn = swat.CAS('cas.mycompany.com', 5570, 'username', 'password')
```

Java is a strongly typed programming language. In Java, you need to declare and create a new **CASClientInterface** object as a connection to the CAS server.

```
Java
CASClientInterface conn = new CASClient('cas.mycompany.com', 5570,
                                        'username', 'password');
```

The techniques above always create a new CAS session in the CAS server. A CAS session is an isolated execution environment that starts a session process on every machine in the cluster where the CAS server is deployed. All data sets that you upload to CAS stay local to that session unless you promote it to a global scope where it can be visible to other sessions in the server. This design enables multiple users to connect to the same computing cluster with resource tracking and management on the individual sessions. If something goes wrong in your session or the session dies, the CAS server and other sessions connected to the server are not affected.

A CAS session has its own identity and authentication. If you are authenticated, you can specify the session ID to reconnect to an existing CAS session.

```
Python
In [3]: conn = swat.CAS('cas.mycompany.com',5570, 'username',
                        'password', session='sessionId')
```

```
Lua
> conn = swat.CAS('cas.mycompany.com', 5570, 'username', 'password',
                  {session='sessionId'})
```

```
JAVA
CASClient client = new CASClient();
client.setHost('cas.mycompany.com');
client.setPort(5570);
client.setUserName('username');
client.setPassword('password');
client.setSessionID('session-Id');
CASClientInterface conn = new CASClient(client);
```

CAS also includes an embedded web server that hosts the CAS Server Monitor web application. The web application provides a graphical user interface for monitoring the CAS server and the user sessions. If you open the server monitor, you can see how many client side connections have been established to a single CAS session (client count):

	UUID	User	Provider	Name	State	Idle Time	Action Count	Last Action	Client Count	
☐	c3fa4e78-5ea9-234a-adac-194ae343083f	ximeng	Active Directory	py-session-4:Mon Feb 27 15:55:23 2017	connected	00:03	20	session.sessionname	4	⋮

CALLING CAS ACTIONS

A CAS server has both analytic and basic operational action sets. Each action set contains one or more actions. In Python or Lua, you can call a CAS action as a method on the CAS connection object that we created in the previous section. For example, you can call the **actionsetInfo** action to print out the action sets that have been loaded into the server.

Python
```
In [4]: conn.actionsetInfo()
```

Lua
```
> conn:actionsetInfo()
```

Action Output
```
Action set information

          actionset                label  loaded  extension  \
0     accessControl      Access Controls       1     tkacon
1     accessControl      Access Controls       1    casmeta
2          builtins             Builtins       1  tkcasablt
3     configuration    Server Properties       1   tkcascfg
4    dataPreprocess      Data Preprocess       1    tktrans
5          dataStep            DATA Step       1   datastep
6        percentile           Percentile       1   tkcasptl
7            search               Search       1     casidx
8           session      Session Methods       1   tkcsessn
9       sessionProp   Session Properties       1   tkcstate
10           simple     Simple Analytics       1   tkimstat
11            table               Tables       1   tkcastab

              build_time            portdate product_name
0    2017-02-26 20:16:29  V.03.02M0P02262017        tkcas
1    2017-02-26 20:16:29  V.03.02M0P02262017        tkcas
2    2017-02-26 20:16:30  V.03.02M0P02262017        tkcas
3    2017-02-26 20:16:27  V.03.02M0P02262017        tkcas
4    2017-02-26 20:16:29  V.03.02M0P02262017      crsstat
5    2017-02-26 20:15:59  V.03.02M0P02262017        tkcas
6    2017-02-26 20:16:29  V.03.02M0P02262017      crsstat
7    2017-02-26 19:52:11  V.03.02M0P02262017    crssearch
8    2017-02-26 20:16:29  V.03.02M0P02262017        tkcas
9    2017-02-26 20:16:29  V.03.02M0P02262017        tkcas
10   2017-02-26 20:16:29  V.03.02M0P02262017      crsstat
11   2017-02-26 20:16:29  V.03.02M0P02262017        tkcas
```

In Java, you need to invoke an action using the client side **Invoke** method. You also need to explicitly declare the action object (**action1**) and the action result object (**results**). The Java output is skipped because it is identical to the Python/Lua output above.

Java
```
ActionSetInfoOptions action1 = new ActionSetInfoOptions();
CASActionResults<CASValue> results = null;
```

```
try {
    results = client.invoke(action1);
} catch (CASException e) {
    // handle CAS exception here
} catch (IOException ioe){
    // handle other exception here
}
for (int i = 0; i < results.getResultsCount(); i++) {
    System.out.println(results.getResult(i));
}
```

There are several alternative ways to submit CAS actions in Java. For example, you can get the same action result in a fluent programming manner.

```
results = client.getActionSets().builtins().actionSetInfo().invoke();
```

The **help** action is probably the most frequently used CAS action in the beginning. You can use the **help** action to list the actions available in a specific CAS action set, or print the parameter descriptions of a specific action. The following example shows how to use this action to display the actions in the **table** action set, and the parameters of the **tableInfo** action.

Python

```
In [5]: conn.help(actionset='table')
```

Lua

```
> conn:help{actionset='table'}
```

Java

```
HelpOptions help = client.getActionSets().builtins().help();
help.setActionSet('table');
CASActionResults<CASValue> results = help.invoke();
```

Action Output

```
              name                                 description
 0            view      Creates a view from files or tables
 1       attribute        Manages extended table attributes
 2          upload  Transfers binary data to the server ...
 3       loadTable  Loads a table from a caslib's data s...
 4     tableExists   Checks whether a table has been loaded
 5      columnInfo                  Shows column information
 6           fetch        Fetches rows from a table or view
 7            save  Saves a table to a caslib's data source
 8        addTable  Add a table by sending it from the c...
 9       tableInfo          Shows information about a table
 10   tableDetails   Get detailed information about a table
 11      dropTable                            Drops a table
 12   deleteSource  Delete a table or file from a caslib...
 13       fileInfo  Lists the files in a caslib's data s...
 14        promote          Promote a table to global scope
 15      addCaslib  Adds a new caslib to enable access t...
 16     dropCaslib                           Drops a caslib
 17     caslibInfo                  Shows caslib information
 18    queryCaslib           Checks whether a caslib exists
 19      partition                        Partitions a table
 20        shuffle                 Randomly shuffles a table
 21    recordCount  Shows the number of rows in a Cloud ...
```

```
 22  loadDataSource  Loads one or more data source interf...
 23          update                   Updates rows in a table
```

```
Python
In [6]: conn.help(action='tableInfo')
```

```
Lua
> conn:help{action='tableInfo'}
```

```
Java
HelpOptions help = client.getActionSets().builtins().help();
help.setAction('table');
CASActionResults<CASValue> results = help.invoke();
```

```
Action Output
NOTE: Information for action 'table.tableInfo':
NOTE: The following parameters are accepted.  Default values are
shown.
NOTE:    string name=NULL (alias: table),
NOTE:       specifies the table name.
NOTE:    string caslib=NULL,
NOTE:       specifies the caslib containing the table that you want
to use with the action. By default, the active caslib is used.
Specify a value only if you need to access a table from a different
caslib.
NOTE:    boolean quiet=false (alias: silent)
NOTE:       when set to True, attempting to show information for a
table that does not exist returns an OK status and severity. When set
to False, attempting to show information for a table that does not
exist returns an error.
```

When you start a new CAS server, several CAS action sets are preloaded. Except for the **simple** action set, these action sets are mainly for basic operational functionality such as server setup, authentication and authorization, session management, and table operations. To use other action sets available in your CAS server, you need to load them into your CAS session. In Python or Lua, you can load an action set on demand using the **loadActionSet** action. For example, let's load the regression action set that contains linear regression, logistic regression, and generalized linear models.

```
Python
In [7]: conn.loadActionset('regression')
```

```
Lua
> conn:loadActionset{actionset='regression'}
```

```
Java
client.loadActionSet(null, 'regression');
```

UNDERSTANDING CAS ACTION RESULTS

Similar to the ODS tables produced by SAS procedures, CAS actions also produce results that are downloaded to the client. Regardless of which programming interface you use to invoke the action, the information that is returned is the same. However, due to the different capabilities of each language, they are presented to you in different formats. In Python, the results of a CAS action call is actually a **CASResults** object, which is a subclass of the Python **OrderedDict** (a dictionary with keys that remain in the same order as were inserted).

```
Python
In [8]: result = conn.serverstatus()
   ...: type(result)
Out[8]: swat.cas.results.CASResults


In [9]: result.keys()
Out[9]: odict_keys(['About', 'server', 'nodestatus'])
```

We can look at an individual result from an action call using Python's key-value syntax. Unlike SAS ODS tables, the results from CAS action might not always to be a structured table. In the above example, the **About** result is a dictionary, and the other two results (**server**, **nodestatus**) are tabular output. In Python, such results are stored as a **SASDataFrame**, which is a sub-class of the famous Pandas **DataFrame**.

```
Python
In [10]: for key in result:
    ...:     print('Type of ' + key + ' is ' +
                   type(result[key]).__name__)

Type of About is dict
Type of server is SASDataFrame
Type of nodestatus is SASDataFrame
```

A **SASDataFrame** is equivalent to a Pandas **DataFrame** for client side operation. It simply contains extra metadata about the table and columns such as labels, formats, and so on. You can work with a **SASDataFrame** just like working with a Pandas **DataFrame**:

```
Python
In [11]: result['nodestatus'].head(2)
Out[11]:
Node Status

                  name    role    uptime  running  stalled
0  cas02.mycompany.com  worker  9718.606        0        0
1  cas03.mycompany.com  worker  9718.605        0        0
```

In Lua, the collection of results from a CAS action is simply a Lua table. Each result in the collection is a Lua table as well.

```
Lua
> result = conn:serverstatus{}
> type(result)
table

> for key, value in pairs(result) do
>>     print('Type of ' .. key .. ' is ' .. type(value))
>> end
Type of server is table
Type of About is table
Type of nodestatus is table
```

In Java, the collection of results from a CAS action is a **CASResults** object. You use the **getResultsCount** and **getResult** methods to loop through the collection and display each action result.

```
for (int i = 0; i < results.getResultsCount(); i++) {
    System.out.println(results.getResult(i));
}
```

WORKING WITH CAS ACTION RESULTS

If you connect to a CAS server and run a CAS-enabled procedure in a SAS environment, the results that you get are ODS tables. If you want to process the data within an ODS table, you first need to use ODS Output to convert the table into a SAS data set. It can then be processed using SAS procedures or DATA step. In Python or Lua, you can work with the result tables using the native methods that are available for a Pandas DataFrame or a Lua table.

First, let's run the **tableInfo** action to see how many data sets have been loaded into the CAS server.

```
Python
In[12]: result = conn.tableInfo()
   ...: df = result['TableInfo']
   ...: df

Lua
> result = conn:tableInfo()
> tbl = result['TableInfo']

Java
TableInfoOptions info1 = client.getActionSets().table().tableInfo();
CASActionResults<CASValue> results = info1.invoke();

Action Output

        Name     Rows  Columns Encoding CreateTimeFormatted  \
0       CARS      428       15    utf-8  27Feb2017:16:28:45
1   ORGANICS  1688948       36    utf-8  27Feb2017:16:28:51
2  ATTRITION    90831       14    utf-8  27Feb2017:16:32:23

     ModTimeFormatted JavaCharSet    CreateTime       ModTime  \
0  27Feb2017:16:28:45        UTF8  1.803832e+09  1.803832e+09
1  27Feb2017:16:28:51        UTF8  1.803832e+09  1.803832e+09
2  27Feb2017:16:32:23        UTF8  1.803832e+09  1.803832e+09

   Global  Repeated  View SourceName SourceCaslib  Compressed  \
0       1         0     0                                   0
1       1         0     0                                   0
2       1         0     0                                   0

  Creator Modifier
0  sasdemo
1  sasdemo
2  sasdemo
```

You can apply DataFrame methods on this output directly. For example, you can index or filter the table, or apply the **max** method on the Rows column to determine the maximum number of rows of all of the loaded tables.

```
Python
In [13]: resultTable[['Name', 'Rows', 'Columns']]
Out[13]:
        Name     Rows  Columns
0       CARS      428       15
1   ORGANICS  1688948       36
2  ATTRITION    90831       14
```

```
In [14]: df[df['Rows'] > 1000][['Name', 'Rows', 'Columns']]
Out[14]:
        Name     Rows  Columns
1   ORGANICS  1688948       36
2  ATTRITION    90831       14

In [15]: df['Rows'].max()
Out[15]: 1688948
```

The counterpart of Pandas DataFrame in Lua is the Lua table object. The Lua **SWAT** package extends the table object by adding new behaviors to handle the tabular output from CAS actions. For example, you can print out the first row of the **TableInfo** result as follows.

```
Lua
> tbl[1]

{["Name"]="CARS", ["Rows"]=428, ["Columns"]=15, ["Encoding"]="utf-8",
["CreateTimeFormatted"]="27Feb2017:16:28:45",
["ModTimeFormatted"]="27Feb2017:16:28:45", ["JavaCharSet"]="UTF8",
["CreateTime"]=1803832125.8, ["ModTime"]=1803832125.8, ["Global"]=1,
["Repeated"]=0, ["View"]=0, ["SourceName"]="", ["SourceCaslib"]="",
["Compressed"]=0, ["Creator"]="sasdemo", ["Modifier"]=""}
```

The **SWAT** package also provides fancy indexing. You can select a subset of rows, or columns, or both.

```
Lua
> tbl{'Name','Rows','Columns'}

Table Information for Caslib CASUSERHDFS(ximeng)
Name          Rows  Columns
CARS           428       15
ORGANICS   1688948       36
ATTRITION    90831       14

> tbl{1,3}{'Name','Rows','Columns'}

Table Information for Caslib CASUSERHDFS(ximeng)
Name         Rows  Columns
CARS          428       15
ATTRITION  90831       14
```

In Java, all action results are wrapped as a **CASValue** object, which is simply a key-value pair. Similar to the action results returned to the Python or Lua client, the action results in Java could be either tabular or non-tabular format. The Java client also provide a few useful methods to work with tabular action outputs. For example, you can obtain the column names and row counts using the **getColumnNames** and the **getRowCount** methods.

```
Java
TableInfoOptions info1 = client.getActionSets().table().tableInfo();
CASActionResults<CASValue> results = info1.invoke();
CASTable tbl = (CASTable) results.getResult(0).getValue();
System.out.println(tbl);
System.out.println(tbl.getColumnNames());
System.out.println(tbl.getRowCount());
```

Java Output

```
[Name, Rows, Columns, Encoding, CreateTimeFormatted,
ModTimeFormatted, JavaCharSet, CreateTime, ModTime, Global, Repeated,
View, SourceName, SourceCaslib, Compressed, Creator, Modifier]
3
```

You can fetch a row or a cell from a tabular CAS action output as well.

Java

```
System.out.println(tbl.getRow(1));
System.out.println(tbl.getStringAt(1,"Name"));
System.out.println(tbl.getIntAt(1,"Rows"));
```

Java Output

```
[ORGANICS, 1688948, 36, utf-8, 03Mar2017:20:26:09,
03Mar2017:20:26:09, UTF8, 1.80419196925421E9, 1.80419196925421E9, 1,
0, 0, null, null, 0, ximeng, null]
ORGANICS
1688948
```

EXPLORING YOUR DATA

Before we can do any sort of statistical analyses, we need to look at the data to better understand it. CAS provides the **simple** action set that contains data exploration and data summary actions for operations such as computing summary statistics, topK, or distinct counts, one-way or two-way frequency tables, and so on. Let's first try to print some univariate statistics from the attrition table.

Python

```
In [16]: conn.summary(table='cars')
```

Lua

```
> conn:summary{table='cars'}
```

Java

```
SummaryOptions summary1 = client.getActionSets().simple().summary();
Castable castable = new Castable();
castable.setName("cars");
summary1.setTable(castable);
CASActionResults<CASValue> results = summary1.invoke();
```

Action Output

```
 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
2   EngineSize      1.3       8.3  428.0    0.0      3.196729
3    Cylinders      3.0      12.0  426.0    2.0      5.807512
4   Horsepower     73.0     500.0  428.0    0.0    215.885514
5     MPG_City     10.0      60.0  428.0    0.0     20.060748
6  MPG_Highway     12.0      66.0  428.0    0.0     26.843458
7       Weight   1850.0    7190.0  428.0    0.0   3577.953271
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           USS  \
0  14027638.0  19431.716674  939.267478  3.775916e+08  6.209854e+11
1  12846292.0  17642.117750  852.763949  3.112443e+08  5.184789e+11
```

```
2      1368.2      1.108595    0.053586  1.228982e+00  4.898540e+03
3      2474.0      1.558443    0.075507  2.428743e+00  1.540000e+04
4     92399.0     71.836032    3.472326  5.160415e+03  2.215110e+07
5      8586.0      5.238218    0.253199  2.743892e+01  1.839580e+05
6     11489.0      5.741201    0.277511  3.296139e+01  3.224790e+05
7   1531364.0    758.983215   36.686838  5.760555e+05  5.725125e+09
8     46290.0      8.311813    0.401767  6.908624e+01  5.035958e+06
9     79763.0     14.357991    0.694020  2.061519e+02  1.495283e+07

            CSS         CV      TValue          ProbT
0  1.612316e+11  59.288490   34.894059  4.160412e-127
1  1.329013e+11  58.778256   35.196963  2.684398e-128
2  5.247754e+02  34.679034   59.656105  3.133745e-209
3  1.032216e+03  26.834946   76.913766  1.515569e-251
4  2.203497e+06  33.275059   62.173176  4.185344e-216
5  1.171642e+04  26.111777   79.229235  1.866284e-257
6  1.407451e+04  21.387709   96.729204  1.665621e-292
7  2.459757e+08  21.212776   97.526890  5.812547e-294
8  2.949982e+04   7.685150  269.196577   0.000000e+00
9  8.802687e+04   7.704349  268.525733   0.000000e+00
```

In both Python and Lua, you can define a **CASTable** object that references the CAS table in the server. You can then submit CAS actions on the **CASTable** object as if they are methods on the object.

```
Python
In [17]: cars = conn.CASTable('cars')
    ...: cars.summary()

Lua
> cars = conn:CASTable{name:'cars'}
> cars:summary{}
```

Although the **CASTable** object in Python is just a reference to the actual CAS table which does not live in the Python environment, you can still treat it like a **DataFrame** and apply **DataFrame** methods on the CASTable object, such as **groupby**.

```
Python
In [18]: cars.groupby('origin').summary(subset=['min','max'])
Out[18]:
[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      Min      Max
 Origin
 Asia          MSRP  10280.0  89765.0
 Asia       Invoice   9875.0  79978.0
 Asia    EngineSize      1.3      5.6
 Asia     Cylinders      3.0      8.0
```

```
 Asia     Horsepower     73.0    340.0
 Asia       MPG_City     13.0     60.0
 Asia    MPG_Highway     17.0     66.0
 Asia         Weight   1850.0   5590.0
 Asia      Wheelbase     89.0    140.0
 Asia         Length    153.0    224.0

[ByGroup2.Summary]

 Descriptive Statistics for CARS

             Column      Min       Max
 Origin
 Europe         MSRP  16999.0  192465.0
 Europe      Invoice  15437.0  173560.0
 Europe   EngineSize      1.6       6.0
 Europe    Cylinders      4.0      12.0
 Europe   Horsepower    100.0     493.0
 Europe     MPG_City     12.0      38.0
 Europe  MPG_Highway     14.0      46.0
 Europe       Weight   2524.0    5423.0
 Europe    Wheelbase     93.0     123.0
 Europe       Length    143.0     204.0

[ByGroup3.Summary]

 Descriptive Statistics for CARS

             Column      Min      Max
 Origin
 USA            MSRP  10995.0  81795.0
 USA         Invoice  10319.0  74451.0
 USA      EngineSize      1.6      8.3
 USA       Cylinders      4.0     10.0
 USA      Horsepower    103.0    500.0
 USA        MPG_City     10.0     29.0
 USA     MPG_Highway     12.0     37.0
 USA          Weight   2348.0   7190.0
 USA       Wheelbase     93.0    144.0
 USA          Length    150.0    238.0
```

This is equivalent to the following.

```
In [19]: cars.set_param('groupby','origin')
    ...: cars.summary(subset=['min','max'])
```

In contrast, in both Lua and Java, you need to specify the **groupby** variable as a parameter to the table.

Lua

```
> cars2 = conn:CASTable{name='cars',groupby={'origin'}}
> cars2:summary{subset={'min','max'}}
```

Java

```
SummaryOptions summary1 = client.getActionSets().simple().summary();
// define the input CAS table
Castable castable = new Castable();
castable.setName('cars');
```

```
// define the group by variable
Casinvardesc groupbyVar = new Casinvardesc();
groupbyVar.setName("Origin");
castable.setGroupBy(new Casinvardesc[] {groupbyVar});

// define the statistics to compute
summary1.setSubSet(new SummaryOptions.SUBSET[]
                       {SummaryOptions.SUBSET.MIN,
                        SummaryOptions.SUBSET.MAX});

summary1.setTable(castable);
SummaryOptions summary1 = client.getActionSets().simple().summary();
CASActionResults<CASValue> results = summary1.invoke();
```

BUILDING ANALYTIC MODELS

CAS provide a variety of statistical models and machine learning models. These models are grouped into action sets based on functionality. For example, the **regression** action set contains several regression models such as linear regression, logistic regression, and generalized linear models. Let us continue to use the **cars** data and build a simple logistic regression model to predict the origin of the vehicles.

Python

```
In [20]: cars.logistic(
    ...:     target = 'Origin',
    ...:     inputs = ['MSRP', 'MPG_CITY']
    ...: )
```

Lua

```
> cars:logistic{target='Origin',inputs={'MSRP','MPG_City'}}
```

Java

```
LogisticOptions logit1 =
client.getActionSets().regression().logistic();

// define the input CAS table;
Castable castable = new Castable();
castable.setName("cars");
logit1.setTable(castable);

// define the input variable list;
Casinvardesc var1 = new Casinvardesc();
var1.setName("MSRP");
Casinvardesc var2 = new Casinvardesc();
var2.setName("MPG_City");
logit1.setInputs(new Casinvardesc[] {var1, var2});

// define the target variable;
String target = "Origin";
logit1.setTarget(target);

CASActionResults<CASValue> results = logit1.invoke();
```

Action Output

```
 Model Information

          RowId                Description  \
```

0	DATA	Data Source
1	RESPONSEVAR	Response Variable
2	NLEVELS	Number of Response Levels
3	DIST	Distribution
4	LINKTYPE	Link Type
5	LINK	Link Function
6	TECH	Optimization Technique

	Value
0	CARS
1	Origin
2	3
3	Multinomial
4	Cumulative
5	Logit
6	Newton-Raphson with Ridging

[NObs]

Number of Observations

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

[ResponseProfile]

Response Profile

	OrderedValue	Outcome	Origin	Freq
0	1	Asia	Asia	158.0
1	2	Europe	Europe	123.0
2	3	USA	USA	147.0

[ConvergenceStatus]

Convergence Status

	Reason	Status	MaxGradient
0	Convergence criterion (GCONV=1E-8) s...	0	7.492139e-08

[Dimensions]

Dimensions

	RowId	Description	Value
0	NDESIGNCOLS	Columns in Design	4
1	NEFFECTS	Number of Effects	3
2	MAXEFCOLS	Max Effect Columns	2
3	DESIGNRANK	Rank of Design	4
4	OPTPARM	Parameters in Optimization	4

[GlobalTest]

Testing Global Null Hypothesis: BETA=0

Test DF ChiSq ProbChiSq

```
 0  Likelihood Ratio   2  31.881151  1.194252e-07

[FitStatistics]

 Fit Statistics

   RowId               Description       Value
 0  M2LL          -2 Log Likelihood  903.963742
 1   AIC   AIC (smaller is better)  911.963742
 2  AICC  AICC (smaller is better)  912.058305
 3   SBC   SBC (smaller is better)  928.200235

[ParameterEstimates]

 Parameter Estimates

       Effect  Parameter          ParmName Outcome  Origin  DF  \
 0  Intercept  Intercept    Intercept_Asia    Asia    Asia   1
 1  Intercept  Intercept  Intercept_Europe  Europe  Europe   1
 2       MSRP       MSRP              MSRP                   1
 3   MPG_City   MPG_City          MPG_City                   1

    Estimate    StdErr       ChiSq     ProbChiSq
 0 -3.367872  0.658085  26.190709  3.093071e-07
 1 -2.116690  0.646503  10.719456  1.060148e-03
 2  0.000006  0.000005   1.698431  1.924932e-01
 3  0.130489  0.027169  23.067706  1.563956e-06

[Timing]

 Task Timing

             RowId                 Task      Time   RelTime
 0           SETUP    Setup and Parsing  0.044817  0.104601
 1    LEVELIZATION         Levelization  0.022141  0.051676
 2  INITIALIZATION  Model Initialization  0.000426  0.000994
 3            SSCP      SSCP Computation  0.002401  0.005604
 4         FITTING        Model Fitting  0.110876  0.258781
 5          OUTPUT  Creating Output Data  0.237345  0.553955
 6         CLEANUP              Cleanup  0.002248  0.005247
 7           TOTAL                Total  0.428455  1.000000
```

REST INTERFACE

CAS also provides a REST API interface for your web applications to monitor CAS server status and execute CAS actions directly. The REST endpoints are organized into three categories:

- /cas - Information about the CAS server, session creation and action submission.
- /system - Information about the system running CAS.
- /grid - Information about a cluster running CAS.

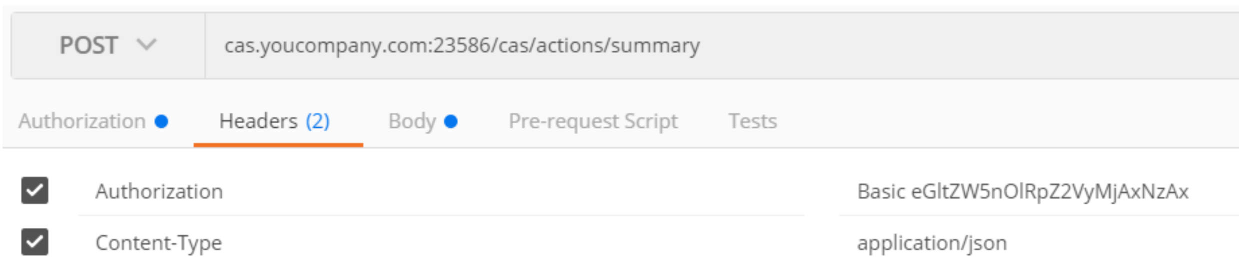
To get the nodes of the cluster running CAS, you can use the ***/cas/nodes*** endpoint.

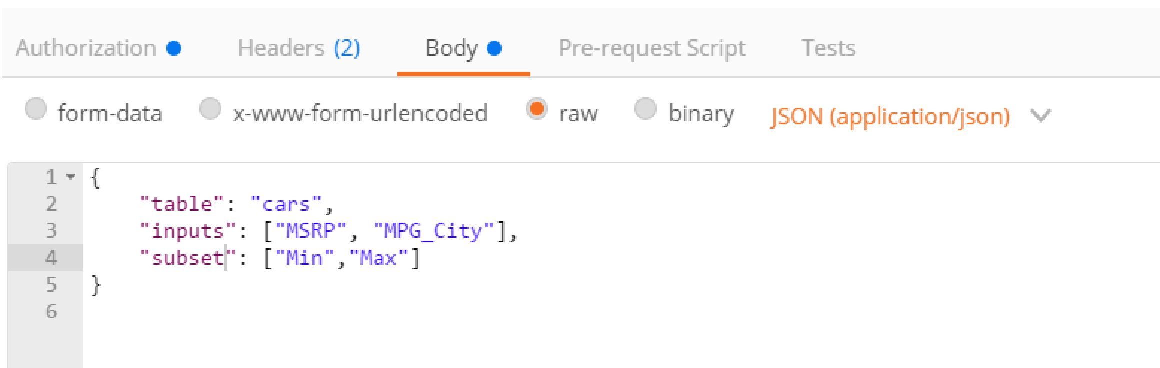
```
GET cas.mycompany.com:[port]/cas/nodes
```

To submit a CAS action, you can post to **/cas/actions/[action name].** The following example shows how to submit a **summary** CAS action on the cars table to obtain the summary statistics for the MSRP values of the vehicles.

```
POST cas.mycompany.com:[port]/cas/actions/summary?table=cars&inputs=MSRP
```

To specify more action parameters such as a list of input variables, you need to add **`Content-Type: application/json`** in the header and pass your parameters in JSON format in the body of your REST call. The following postman screenshots show how to submit a summary action to compute the minimum and maximum values of MPG_City and MPG_Highway columns.





SUPPORTED VERSIONS

- 64-bit Lua 5.2 or later. Install the middleclass (4.0+), csv, and ee5_base64 Lua packages.
- 64-bit Python 2.7.x or 3.4.x
- Java 8 or later

CONCLUSION

This paper provides an introduction to the three programming interfaces to SAS® Cloud Analytic Services (CAS): Python, Lua, and Java. For more information about these CAS clients, you can visit the Viya documentation site for more information. http://support.sas.com/documentation/onlinedoc/viya/index.html

RECOMMENDED READING

- *SAS® Viya – The Python Perspective*

CONTACT INFORMATION

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

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

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