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**

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

**Lua**

```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 CASClientInterface('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**

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

**Lua**

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

**Java**

```java
CASClient client = new CASClient();
client.setHost('cas.mycompany.com');
client.setPort(5570);
client.setUserName('username');
client.setPassword('password');
client.setSessionID('sessionId');
CASClientInterface conn = new CASClientInterface(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):
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

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

Lua

```lua
> conn:actionsetInfo()
```

**Action Output**

Action set information

<table>
<thead>
<tr>
<th>actionset</th>
<th>label</th>
<th>loaded</th>
<th>extension</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>accessControl</td>
<td>1</td>
<td>tkacon</td>
</tr>
<tr>
<td>1</td>
<td>accessControl</td>
<td>1</td>
<td>casmeta</td>
</tr>
<tr>
<td>2</td>
<td>builtins</td>
<td>1</td>
<td>tkcasablt</td>
</tr>
<tr>
<td>3</td>
<td>configuration</td>
<td>1</td>
<td>tkcascfg</td>
</tr>
<tr>
<td>4</td>
<td>dataPreprocess</td>
<td>1</td>
<td>tktrans</td>
</tr>
<tr>
<td>5</td>
<td>dataStep</td>
<td>1</td>
<td>datastep</td>
</tr>
<tr>
<td>6</td>
<td>percentile</td>
<td>1</td>
<td>tkcasptl</td>
</tr>
<tr>
<td>7</td>
<td>search</td>
<td>1</td>
<td>casidx</td>
</tr>
<tr>
<td>8</td>
<td>session</td>
<td>1</td>
<td>tkcsessn</td>
</tr>
<tr>
<td>9</td>
<td>sessionProp</td>
<td>1</td>
<td>tkcstate</td>
</tr>
<tr>
<td>10</td>
<td>simple</td>
<td>1</td>
<td>tkimstat</td>
</tr>
<tr>
<td>11</td>
<td>table</td>
<td>1</td>
<td>tkcastab</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>build_time</th>
<th>portdate</th>
<th>product_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2017-02-26 20:16:29</td>
<td>V.03.02M0P02262017 tkcas</td>
</tr>
<tr>
<td>1</td>
<td>2017-02-26 20:16:29</td>
<td>V.03.02M0P02262017 tkcas</td>
</tr>
<tr>
<td>2</td>
<td>2017-02-26 20:16:30</td>
<td>V.03.02M0P02262017 tkcas</td>
</tr>
<tr>
<td>3</td>
<td>2017-02-26 20:16:27</td>
<td>V.03.02M0P02262017 tkcas</td>
</tr>
<tr>
<td>4</td>
<td>2017-02-26 20:16:29</td>
<td>V.03.02M0P02262017 crsstat</td>
</tr>
<tr>
<td>5</td>
<td>2017-02-26 20:15:59</td>
<td>V.03.02M0P02262017 tkcas</td>
</tr>
<tr>
<td>6</td>
<td>2017-02-26 20:16:29</td>
<td>V.03.02M0P02262017 crsstat</td>
</tr>
<tr>
<td>7</td>
<td>2017-02-26 19:52:11</td>
<td>V.03.02M0P02262017 crssearch</td>
</tr>
<tr>
<td>8</td>
<td>2017-02-26 20:16:29</td>
<td>V.03.02M0P02262017 tkcas</td>
</tr>
<tr>
<td>9</td>
<td>2017-02-26 20:16:29</td>
<td>V.03.02M0P02262017 tkcas</td>
</tr>
<tr>
<td>10</td>
<td>2017-02-26 20:16:29</td>
<td>V.03.02M0P02262017 crsstat</td>
</tr>
<tr>
<td>11</td>
<td>2017-02-26 20:16:29</td>
<td>V.03.02M0P02262017 tkcas</td>
</tr>
</tbody>
</table>

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

```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();

<table>
<thead>
<tr>
<th>Action Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
</tr>
<tr>
<td>0</td>
</tr>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>13</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>15</td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td>17</td>
</tr>
<tr>
<td>18</td>
</tr>
<tr>
<td>19</td>
</tr>
<tr>
<td>20</td>
</tr>
<tr>
<td>21</td>
</tr>
</tbody>
</table>
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]:
    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.

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

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

**Lua**

```lua
result = conn:tableInfo()
tbl = result['TableInfo']
```

**Java**

```java
TableInfoOptions infol = client.getActionSets().table().tableInfo();
CASAActionResults<CASValue> results = infol.invoke();
```

### Action Output

<table>
<thead>
<tr>
<th>Name</th>
<th>Rows</th>
<th>Columns</th>
<th>Encoding</th>
<th>CreateTimeFormatted</th>
</tr>
</thead>
<tbody>
<tr>
<td>CARS</td>
<td>428</td>
<td>15</td>
<td>utf-8</td>
<td>27Feb2017:16:28:45</td>
</tr>
<tr>
<td>ORGANICS</td>
<td>1688948</td>
<td>36</td>
<td>utf-8</td>
<td>27Feb2017:16:28:51</td>
</tr>
<tr>
<td>ATTRITION</td>
<td>90831</td>
<td>14</td>
<td>utf-8</td>
<td>27Feb2017:16:32:23</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ModTimeFormatted</th>
<th>JavaCharSet</th>
<th>CreateTime</th>
<th>ModTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>27Feb2017:16:28:45</td>
<td>UTF8</td>
<td>1.803832e+09</td>
<td>1.803832e+09</td>
</tr>
<tr>
<td>27Feb2017:16:28:51</td>
<td>UTF8</td>
<td>1.803832e+09</td>
<td>1.803832e+09</td>
</tr>
<tr>
<td>27Feb2017:16:32:23</td>
<td>UTF8</td>
<td>1.803832e+09</td>
<td>1.803832e+09</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Global</th>
<th>Repeated</th>
<th>View</th>
<th>SourceName</th>
<th>SourceCaslib</th>
<th>Compressed</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

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**

```python
In [13]: resultTable[['Name', 'Rows', 'Columns']]
Out[13]:
```

<table>
<thead>
<tr>
<th>Name</th>
<th>Rows</th>
<th>Columns</th>
</tr>
</thead>
<tbody>
<tr>
<td>CARS</td>
<td>428</td>
<td>15</td>
</tr>
<tr>
<td>ORGANICS</td>
<td>1688948</td>
<td>36</td>
</tr>
<tr>
<td>ATTRITION</td>
<td>90831</td>
<td>14</td>
</tr>
</tbody>
</table>
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]:

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"]=180383125.8, ["ModTime"]=180383125.8, ["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();
CASActionResult<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
You can fetch a row or a cell from a tabular CAS action output as well.

**Java**

```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, null, null, 0, ximeng, null]

**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**

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

**Lua**

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

**Java**

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

**Action Output**

<table>
<thead>
<tr>
<th>Column</th>
<th>Min</th>
<th>Max</th>
<th>N</th>
<th>NMiss</th>
<th>Mean</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>MSRP</td>
<td>10280.0</td>
<td>192465.0</td>
<td>428.0</td>
<td>0.0</td>
</tr>
<tr>
<td>1</td>
<td>Invoice</td>
<td>9875.0</td>
<td>173560.0</td>
<td>428.0</td>
<td>0.0</td>
</tr>
<tr>
<td>2</td>
<td>EngineSize</td>
<td>1.3</td>
<td>8.3</td>
<td>428.0</td>
<td>0.0</td>
</tr>
<tr>
<td>3</td>
<td>Cylinders</td>
<td>3.0</td>
<td>12.0</td>
<td>428.0</td>
<td>0.0</td>
</tr>
<tr>
<td>4</td>
<td>Horsepower</td>
<td>73.0</td>
<td>500.0</td>
<td>428.0</td>
<td>0.0</td>
</tr>
<tr>
<td>5</td>
<td>MPG_City</td>
<td>10.0</td>
<td>60.0</td>
<td>428.0</td>
<td>0.0</td>
</tr>
<tr>
<td>6</td>
<td>MPG_Highway</td>
<td>12.0</td>
<td>66.0</td>
<td>428.0</td>
<td>0.0</td>
</tr>
<tr>
<td>7</td>
<td>Weight</td>
<td>1850.0</td>
<td>7190.0</td>
<td>428.0</td>
<td>0.0</td>
</tr>
<tr>
<td>8</td>
<td>Wheelbase</td>
<td>89.0</td>
<td>144.0</td>
<td>428.0</td>
<td>0.0</td>
</tr>
<tr>
<td>9</td>
<td>Length</td>
<td>143.0</td>
<td>238.0</td>
<td>428.0</td>
<td>0.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Sum</th>
<th>Std</th>
<th>StdErr</th>
<th>Var</th>
<th>USS</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>14027638.0</td>
<td>19431.716674</td>
<td>939.267478</td>
<td>3.775916e+08</td>
</tr>
<tr>
<td>1</td>
<td>12846292.0</td>
<td>17642.1117750</td>
<td>852.763949</td>
<td>3.112443e+08</td>
</tr>
</tbody>
</table>

9
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

<table>
<thead>
<tr>
<th>Column</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Origin</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Europe</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSRP</td>
<td>16999.0</td>
<td>192465.0</td>
</tr>
<tr>
<td>Invoice</td>
<td>15437.0</td>
<td>173560.0</td>
</tr>
<tr>
<td>EngineSize</td>
<td>1.6</td>
<td>6.0</td>
</tr>
<tr>
<td>Cylinders</td>
<td>4.0</td>
<td>12.0</td>
</tr>
<tr>
<td>Horsepower</td>
<td>100.0</td>
<td>493.0</td>
</tr>
<tr>
<td>MPG_City</td>
<td>12.0</td>
<td>38.0</td>
</tr>
<tr>
<td>MPG_Highway</td>
<td>14.0</td>
<td>46.0</td>
</tr>
<tr>
<td>Weight</td>
<td>2524.0</td>
<td>5423.0</td>
</tr>
<tr>
<td>Wheelbase</td>
<td>93.0</td>
<td>123.0</td>
</tr>
<tr>
<td>Length</td>
<td>143.0</td>
<td>204.0</td>
</tr>
</tbody>
</table>

[ByGroup3.Summary]

Descriptive Statistics for CARS

<table>
<thead>
<tr>
<th>Column</th>
<th>Min</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Origin</td>
<td></td>
<td></td>
</tr>
<tr>
<td>USA</td>
<td></td>
<td></td>
</tr>
<tr>
<td>MSRP</td>
<td>10995.0</td>
<td>81795.0</td>
</tr>
<tr>
<td>Invoice</td>
<td>10319.0</td>
<td>74451.0</td>
</tr>
<tr>
<td>EngineSize</td>
<td>1.6</td>
<td>8.3</td>
</tr>
<tr>
<td>Cylinders</td>
<td>4.0</td>
<td>10.0</td>
</tr>
<tr>
<td>Horsepower</td>
<td>103.0</td>
<td>500.0</td>
</tr>
<tr>
<td>MPG_City</td>
<td>10.0</td>
<td>29.0</td>
</tr>
<tr>
<td>MPG_Highway</td>
<td>12.0</td>
<td>37.0</td>
</tr>
<tr>
<td>Weight</td>
<td>2348.0</td>
<td>7190.0</td>
</tr>
<tr>
<td>Wheelbase</td>
<td>93.0</td>
<td>144.0</td>
</tr>
<tr>
<td>Length</td>
<td>150.0</td>
<td>238.0</td>
</tr>
</tbody>
</table>

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

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

Java

```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

<table>
<thead>
<tr>
<th>RowId</th>
<th>Description</th>
</tr>
</thead>
</table>
DATA Data Source
RESPONSEVAR Response Variable
NLEVELS Number of Response Levels
DIST Distribution
LINKTYPE Link Type
LINK Link Function
TECH Optimization Technique

Value
CARS
Origin
3
Multinomial
Cumulative
Logit
Newton-Raphson with Ridging

Number of Observations

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

Response Profile

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

Convergence Status

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

Dimensions

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

Global Test

Testing Global Null Hypothesis: BETA=0

Test DF ChiSq ProbChiSq

[FitStatistics]

Fit Statistics

<table>
<thead>
<tr>
<th>RowId</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>M2LL -2 Log Likelihood</td>
<td>903.963742</td>
</tr>
<tr>
<td>1</td>
<td>AIC AIC (smaller is better)</td>
<td>911.963742</td>
</tr>
<tr>
<td>2</td>
<td>AICC AICC (smaller is better)</td>
<td>912.058305</td>
</tr>
<tr>
<td>3</td>
<td>SBC SBC (smaller is better)</td>
<td>928.200235</td>
</tr>
</tbody>
</table>

[ParameterEstimates]

Parameter Estimates

<table>
<thead>
<tr>
<th>Effect</th>
<th>Parameter</th>
<th>ParmName</th>
<th>Outcome</th>
<th>Origin</th>
<th>DF</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Intercept</td>
<td>Intercept</td>
<td>Inter</td>
<td>Asia</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>Intercept</td>
<td>Intercept</td>
<td>Europe</td>
<td>Europe</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>MSRP</td>
<td>MSRP</td>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>MPG_City</td>
<td>MPG_City</td>
<td></td>
<td></td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Estimate</th>
<th>StdErr</th>
<th>ChiSq</th>
<th>ProbChiSq</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-3.367872</td>
<td>0.658085</td>
<td>26.190709</td>
</tr>
<tr>
<td>1</td>
<td>-2.116690</td>
<td>0.646503</td>
<td>10.719456</td>
</tr>
<tr>
<td>2</td>
<td>0.000006</td>
<td>0.000005</td>
<td>1.698431</td>
</tr>
<tr>
<td>3</td>
<td>0.130489</td>
<td>0.027169</td>
<td>23.067706</td>
</tr>
</tbody>
</table>

[Timing]

Task Timing

<table>
<thead>
<tr>
<th>RowId</th>
<th>Task</th>
<th>Time</th>
<th>RelTime</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>SETUP</td>
<td>0.044817</td>
<td>0.104601</td>
</tr>
<tr>
<td>1</td>
<td>LEVELIZATION</td>
<td>0.022141</td>
<td>0.051676</td>
</tr>
<tr>
<td>2</td>
<td>INITIALIZATION</td>
<td>0.000426</td>
<td>0.000994</td>
</tr>
<tr>
<td>3</td>
<td>SSCP</td>
<td>0.002401</td>
<td>0.005604</td>
</tr>
<tr>
<td>4</td>
<td>FITTING</td>
<td>0.110876</td>
<td>0.258781</td>
</tr>
<tr>
<td>5</td>
<td>OUTPUT</td>
<td>0.237345</td>
<td>0.553955</td>
</tr>
<tr>
<td>6</td>
<td>CLEANUP</td>
<td>0.002248</td>
<td>0.005247</td>
</tr>
<tr>
<td>7</td>
<td>TOTAL</td>
<td>0.428455</td>
<td>1.000000</td>
</tr>
</tbody>
</table>

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.

```json
{
    "table": "cars",
    "inputs": ["MSRP", "MPG_City"],
    "subset": ["Min", "Max"]
}
```

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](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: