ABSTRACT

As SAS® continues to push boundaries with its cloud-based analytics ecosystem, SAS® Viya®, SAS also continues to break new ground as well! With a new initiative to become more open to developers via a robust API, and current integration with the Python package known as SWAT (Scripting Wrapper for Analytics Transfer), there are opportunities to take your in-house data science initiatives to a higher level. This session looks at incorporating open-source graphing techniques, specifically Python's matplotlib integrated with the popular D3 visualization framework, to generate interactive plots that can spur discovery of the story your data is trying to tell you. We work through some traditional statistical programming examples via calls in Jupyter Notebook to SAS Viya. Finally, within Jupyter, we convert our static graphs into dynamic graphs using mpld3, an open-source Python library that marries D3 to Python.

INTRODUCTION

With the release of SAS Viya, SAS continues its commitment to being a company of open-source inclusiveness. SAS has now adopted a culture that does not require SAS code to be written and called from within a SAS environment. SAS now has the ability to be called as a package and to be included in the most popular open-source languages. This approach has revolutionized commercial and open-source interoperability.

In this paper we are going to look at the use of Python, one of the most popular open-source programming languages. We are also going to examine how using Jupyter Notebook as an integrated development environment (IDE) can enable us to build interactive connections between the two frameworks. This synergy will give us the ability to harness the algorithms and the computing power of SAS, marry it to the data set standardization of Pandas, and ultimately create beautiful new graphs in Python with the D3 library.

To demonstrate the flexibility of this relationship, we will further explore the ability to combine mpld3 as a wrapper that allows Python to dynamically produce D3 charts. The programming examples contained in this paper will be demonstrated in both SAS code and in Python.

SAS SWAT

At the foundation of SAS Viya, is the SAS Cloud Analytic Services Engine, which is more commonly referred to as CAS. Requests to CAS are made from executables that SAS defines as actions. These actions function as an interface into the CAS server. Actions come in many different flavors. Some offer the ability to execute data integration tasks, while others give access to the analytic toolset upon which SAS has built its reputation. When we speak about Python, we are talking about a specific CAS action, SAS SWAT. SWAT enables Python to communicate with SAS CAS, which becomes useful when we encounter issues that comes from working inside of Python data size constraints. With the SWAT package, you can load and analyze data sets of any size on your desktop or the
cloud. Because CAS can be used in any of these environments, it enables you to analyze extremely large data sets with flexible and scalable processing power while still retaining the simplicity of Python.

SAS takes Python integration a step further by leveraging the capability of returning data structures that are consumable by the popular Python pandas package. This allows the Python user the ability to combine the returned data with other data, or to pass the returned data through to other operations within Python, such as visualizations.

**CAS**

When data objects are returned from a SWAT call, these objects are returned as a CASTable. A CASTable is an in-memory table that can be consumable as a DataFrame by a SAS CASResults object. Although these two classes are independent of each other, they work together to produce DataFrame results.

Let’s look at a quick example of this processing. Figure 1 shows how Python first imports a SAS library, and then show how data is pulled from a source and then loaded into a CASTable and a CASResult object.

```
In [15]: import swat
In [24]: c = swat.CAS(host, port, userid, password)
In [25]: print(c)
```

**Figure 1. Data Import and Load**

The code in Figure 1 builds the connection back to SAS. SWAT is the package that allows all the actions from inside of CAS to be called.

We then need to upload the data file into CAS, as shown in Figure 2.

```
In [27]: titanic3 = c.CASTable('titanic3', replace=True)
In [28]: c.upload_file('https://biostat.mc.vanderbilt.edu/wiki/pub/Main/DataSets/titanic3.csv', casout='titanic3')
```

**Figure 2. Upload Data File to CAS**

The code in Figure 2 creates an in-memory table called “titanic3”. In this case we are pulling in an external file into CAS and making that table available to us in memory.

Now that we have a data set in CAS, let’s look at the columnar description of the table. To do that we execute a COLUMNINFO call, as shown in Figure 3.
We are now able to view the properties of the CASTable that we just created. You can see that the variable types are all SAS compatible. This is because the data was uploaded through the CAS and inherits SAS data types.

Let’s now examine a summary of the data that we imported. You will note that this data in Figure 4 comes from the details of the passengers on the ill-fated voyage of the Titanic.

By making a call to the SUMMARY procedure, we can see a summary of the data in the CASTable.

If we want to examine the data grouped by the passengers that survived and the passengers that perished, we use BYGROUP calls to group the data.
Figure 5. Grouping the Data
Figure 6. Grouped Data

We are able to see the main summary statistics for the passengers on the Titanic. This data is summarized by column, giving us the ability to see things such as the mean value of each variable. Looking at this summary, we can immediately see that the passenger class does matter. There is a notable difference in the class of passenger between those who survived and those who perished.

Next, let’s look at what happens when we apply some algorithms to the table to more clearly understand the conclusions. Before we can apply these algorithms, we need to build some sampling into the table and add partitioning, as shown in Figures 7 and 8.
Now that we have partitioned the data, we are ready to add some analytics into the mix. In this case, we choose to use a decisionTree to determine the best characteristics to posses if you wanted to survive the sinking of the Titanic, as shown in Figure 9.
Figure 9. Applying a decisionTree Action Set

The call to LOADACTIONSET in Figure 9 specifies that a decision tree is used as the algorithm of choice. You can see the results of the training of the tree with the variable importance plot. The plot in Figure 9 shows us that the most important variable that predicts survival is not the class of the passenger, but rather the sex of the passenger.

MATPLOTLIB AND D3

Let’s look at how we can now visualize the results from the tree using mpld3. As you recall, mpld3 is the API that enables Python to interact with Matplotlib in order to call D# visualizations. By using mpld3, we can take advantage of the power of SAS to process the analytics and then return the data table back to Python. After the returned data is in Python, it can be visualized by the D3 JavaScript language, which allows for interactive charting.
In [63]: import pandas as pd
import m.plugin

forestAssess_ROC = s.cast_table("forestAssess_ROC", where="l_partind _")
out2 = forestAssess_ROC.to_frame()

from mpld3 import plugins

# Define some CSS to control our custom labels

fig, ax = plt.subplots(figsize=(8, 8))
ax.set_xlabel('False Positive Rate')
ax.set_ylabel('Correct Classification Rate')
ax.set_title('ROC Curve', size=20)
ax.grid(True, alpha=0.3)
points = ax.plot(out2.FPR, out2.Sensitivity, 'bo-', color='k',
                 mec='k', ms=10, mew=1, alpha=0.6, linewidth=1)
ax.plot(pd.Series(range(0,1,1))/10,pd.Series(range(0,1,1))/10,'k--',linewidth=1)

# Get data into lists
my_xpoints = []
my_ypoints = []
for l in points:
    my_xpoints.append(l.get_xdata())
    my_ypoints.append(l.get_ydata())

import numpy as np

labels = []
for (i, k) in zip(np.ndenumerate(my_xpoints), np.ndenumerate(my_ypoints)):
    x = str().split('')
    y = str(k).split('')
    labels.append('<table><tr><th>{0}</th><th>{1}</th></tr><tr><td>{2}</td><td>{3}</td></tr></table>'.format(x[0], y[0], x[1], y[1]))

tooltip = plugins.PointHTMLTooltip(points[0], labels,
                                   offsets=[10, 10], css='css')

plugins.connect(fig, tooltip)

import json

mpld3 hack

class NumpyEncoder(json.JSONEncoder):
    def default(self, obj):
        if isinstance(obj, np.ndarray):
            return obj.tolist()

mpld3 import -display
_display.NumpyEncoder = NumpyEncoder

mpld3.display()
Figure 12. D3 Visualization

12 is a ROC curve chart shown as a D3 visualization. This chart enables you to hover over, click on, and interact with the data that is produced from the code. Not only is the data interactive, but it can be dynamic. By allowing conditional formatting we can use different colors for certain elements of the graph, based on the values of data we retrieve. This feature also allows for dashboarding within a D3 interactive chart.
from mpld3 import plugins

# Define some CSS to control our custom labels

forestAssess = s.CASTable("forestAssess", where="l = _partind_")
    lift = forestAssess.to_frame()

fig, ax = plt.subplots(figsize=(8, 8))
ax.set_xlabel('Percentile')
ax.set_ylabel('Lift')
ax.set_title('Lift Chart', size=20)
ax.grid(True, alpha=0.3)

points = ax.plot(lift._Depth_, lift._Lift_, 'bo-', color='0.5',
                mec='k', ms=10, mew=1, alpha=0.5, linewidth=1)

# Get data into lists
my_xpoints = []
my_ypoints = []
for i in points:
    my_xpoints.append(i.get_xdata())
    my_ypoints.append(i.get_ydata())

import numpy as np

labels = []
for (j, k), (i, l) in zip(np.ndenumerate(my_xpoints), np.ndenumerate(my_ypoints)):
    x = str(i).split(',')
    y = str(j).split(',')
    labels.append("<table><tr><th>{0}</th><th>{1}</th><th>{2}</th></tr><tr><td><a href="{3}>{4}</a></td><td>{5}</td><td>{6}</td></tr></table>".format(x[0][1:].strip(), y[0][1:].strip(), '
    
tooltip = plugins.PointHTMLTooltip(points[3], labels,
                                      voffset=10, hoffset=10, css=css)
plugins.connect(fig, tooltip)

Figure 13. Applying Conditional Formatting

import json

#mpld3 hack

class NumpyEncoder(json.JSONEncoder):
    def default(self, obj):
        if isinstance(obj, np.ndarray):
            return obj.tolist()

return json.JSONEncoder.default(self, obj)

from mpld3 import _diplay
_diplay.NumpyEncoder = NumpyEncoder
mpld3.display()
Figure 15. D3 Visualization of Conditionally Formatted Data

Figure 15 is a lift chart shown as a D3 visualization.
CONCLUSION

The ability to include popular JavaScript charting techniques into SAS gives us new options for building interactive charting inside of Python, while harnessing the power of SAS algorithms. The mpld3 project gives us the capability of joining D3 to Python, and SWAT allows Python to talk to SAS. These techniques allow us to build a system that can dynamically produce charting capability, giving open-source users the ability to build amazing interactive charting that is powered by SAS.

REFERENCES


CONTACT INFORMATION

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

Joe Indelicato
SAS Institute Inc.
713-325-9857
joseph.indelicato@sas.com

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

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