Paper SAS3455-2019

Open Visualization with SAS® Viya® and Python

Joe Indelicato, SAS Institute Inc.

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

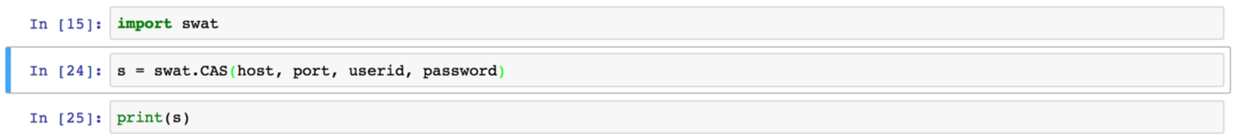
At the foundation of SAS Viya, is the SAS Cloud Analytic Services Engine, which is more commonly referred to as CAS. Requests to CAS are made from executables that SAS defines as actions. These actions function as an interface into the CAS server. Actions come in many different flavors. Some offer the 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.



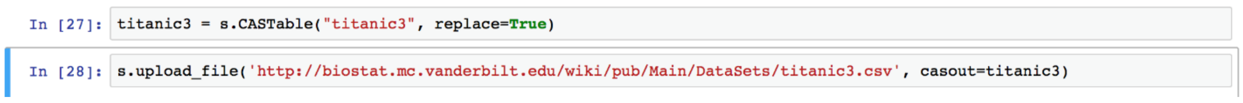
```
In [15]: import swat

In [24]: s = swat.CAS(host, port, userid, password)

In [25]: print(s)
```

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.

f

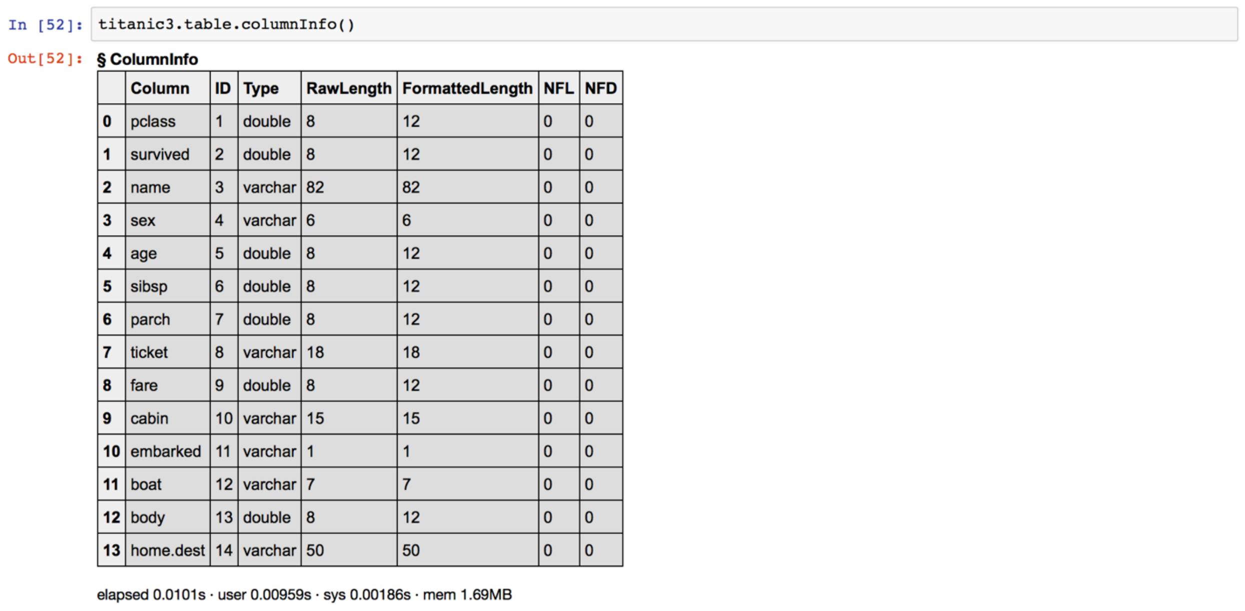


```
In [27]: titanic3 = s.CASTable("titanic3", replace=True)

In [28]: s.upload_file('http://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.



```
In [52]: titanic3.table.columnInfo()
Out[52]:
```

§ ColumnInfo

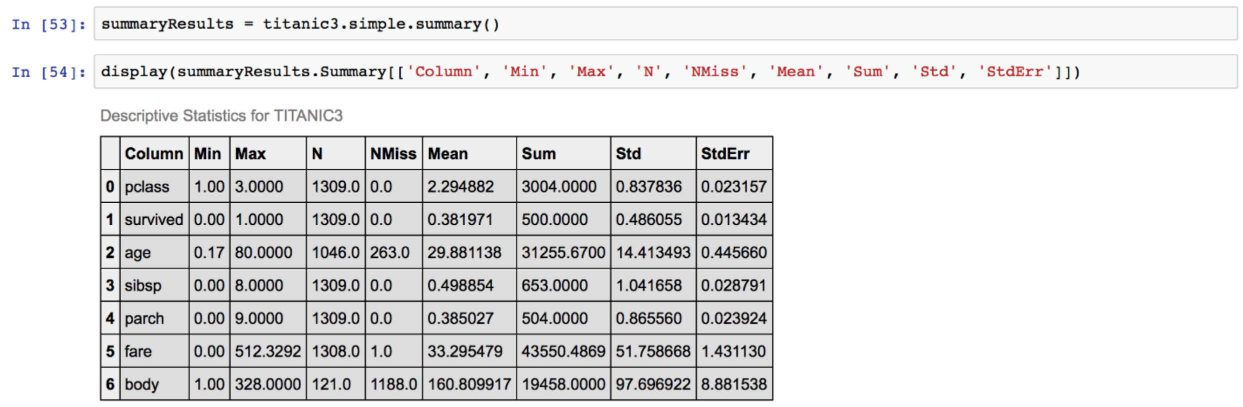
	Column	ID	Type	RawLength	FormattedLength	NFL	NFD
0	pclass	1	double	8	12	0	0
1	survived	2	double	8	12	0	0
2	name	3	varchar	82	82	0	0
3	sex	4	varchar	6	6	0	0
4	age	5	double	8	12	0	0
5	sibsp	6	double	8	12	0	0
6	parch	7	double	8	12	0	0
7	ticket	8	varchar	18	18	0	0
8	fare	9	double	8	12	0	0
9	cabin	10	varchar	15	15	0	0
10	embarked	11	varchar	1	1	0	0
11	boat	12	varchar	7	7	0	0
12	body	13	double	8	12	0	0
13	home.dest	14	varchar	50	50	0	0

elapsed 0.0101s · user 0.00959s · sys 0.00186s · mem 1.69MB

Figure 3. COLUMNINFO Call

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.



```
In [53]: summaryResults = titanic3.simple.summary()

In [54]: display(summaryResults.Summary[['Column', 'Min', 'Max', 'N', 'NMiss', 'Mean', 'Sum', 'Std', 'StdErr']])
```

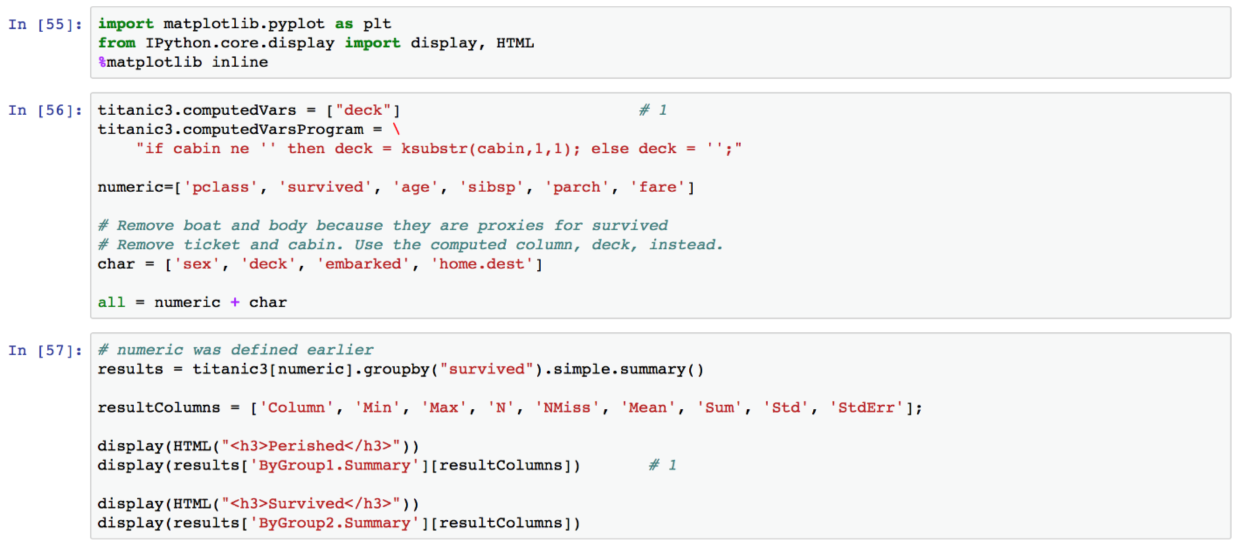
Descriptive Statistics for TITANIC3

	Column	Min	Max	N	NMiss	Mean	Sum	Std	StdErr
0	pclass	1.00	3.0000	1309.0	0.0	2.294882	3004.0000	0.837836	0.023157
1	survived	0.00	1.0000	1309.0	0.0	0.381971	500.0000	0.486055	0.013434
2	age	0.17	80.0000	1046.0	263.0	29.881138	31255.6700	14.413493	0.445660
3	sibsp	0.00	8.0000	1309.0	0.0	0.498854	653.0000	1.041658	0.028791
4	parch	0.00	9.0000	1309.0	0.0	0.385027	504.0000	0.865560	0.023924
5	fare	0.00	512.3292	1308.0	1.0	33.295479	43550.4869	51.758668	1.431130
6	body	1.00	328.0000	121.0	1188.0	160.809917	19458.0000	97.696922	8.881538

Figure 4. Data Summary

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.

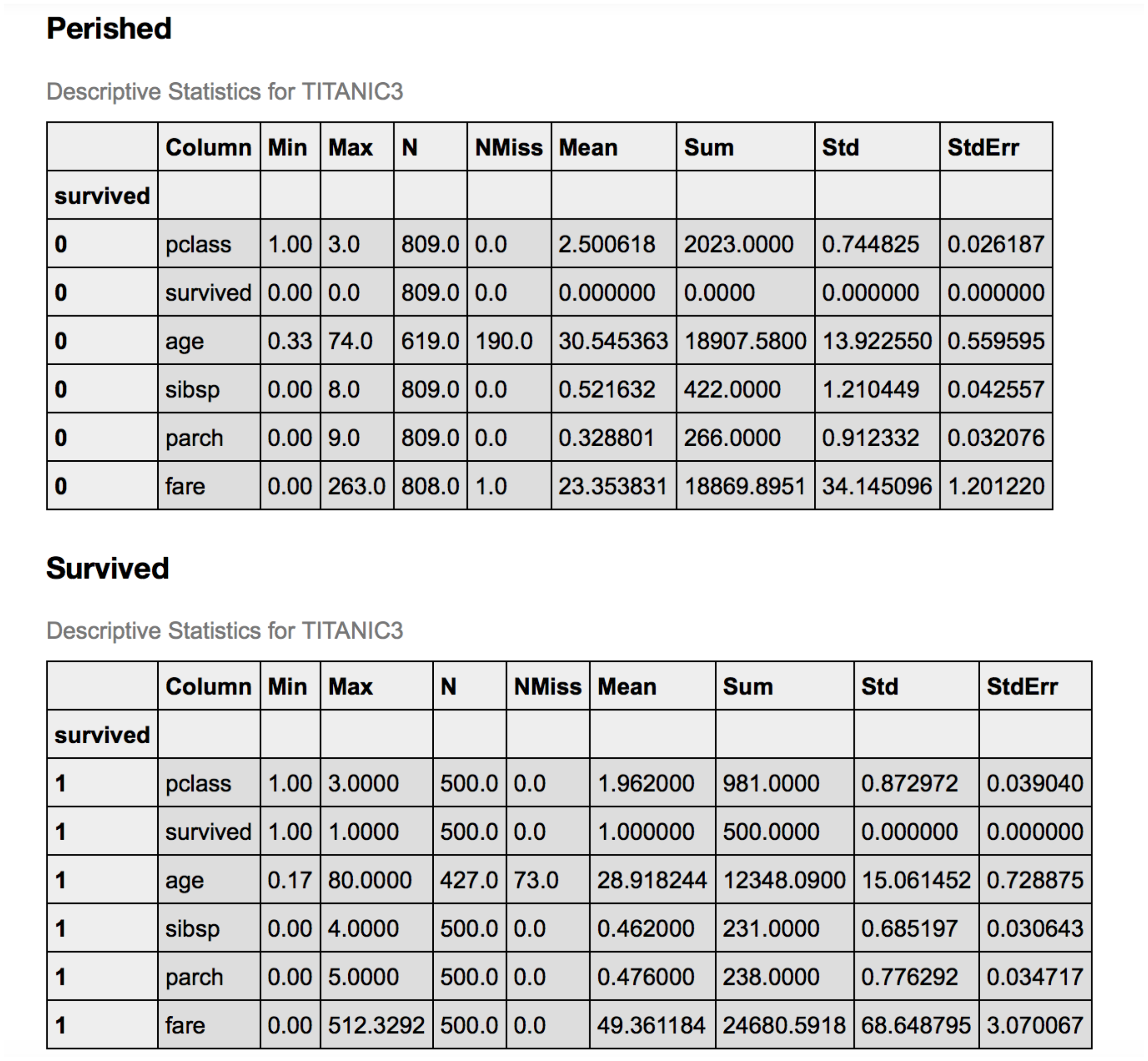


```
In [55]: import matplotlib.pyplot as plt
         from IPython.core.display import display, HTML
         %matplotlib inline
```

```
In [56]: titanic3.computedVars = ["deck"]                          # 1
         titanic3.computedVarsProgram = \
             "if cabin ne '' then deck = ksubstr(cabin,1,1); else deck = '';"

         numeric=['pclass', 'survived', 'age', 'sibsp', 'parch', 'fare']

         # Remove boat and body because they are proxies for survived
         # Remove ticket and cabin. Use the computed column, deck, instead.
         char = ['sex', 'deck', 'embarked', 'home.dest']

         all = numeric + char
```

```
In [57]: # numeric was defined earlier
         results = titanic3[numeric].groupby("survived").simple.summary()

         resultColumns = ['Column', 'Min', 'Max', 'N', 'NMiss', 'Mean', 'Sum', 'Std', 'StdErr'];

         display(HTML("<h3>Perished</h3>"))
         display(results['ByGroup1.Summary'][resultColumns])       # 1

         display(HTML("<h3>Survived</h3>"))
         display(results['ByGroup2.Summary'][resultColumns])
```

Figure 5. Grouping the Data



Perished

Descriptive Statistics for TITANIC3

	Column	Min	Max	N	NMiss	Mean	Sum	Std	StdErr
survived									
0	pclass	1.00	3.0	809.0	0.0	2.500618	2023.0000	0.744825	0.026187
0	survived	0.00	0.0	809.0	0.0	0.000000	0.0000	0.000000	0.000000
0	age	0.33	74.0	619.0	190.0	30.545363	18907.5800	13.922550	0.559595
0	sibsp	0.00	8.0	809.0	0.0	0.521632	422.0000	1.210449	0.042557
0	parch	0.00	9.0	809.0	0.0	0.328801	266.0000	0.912332	0.032076
0	fare	0.00	263.0	808.0	1.0	23.353831	18869.8951	34.145096	1.201220

Survived

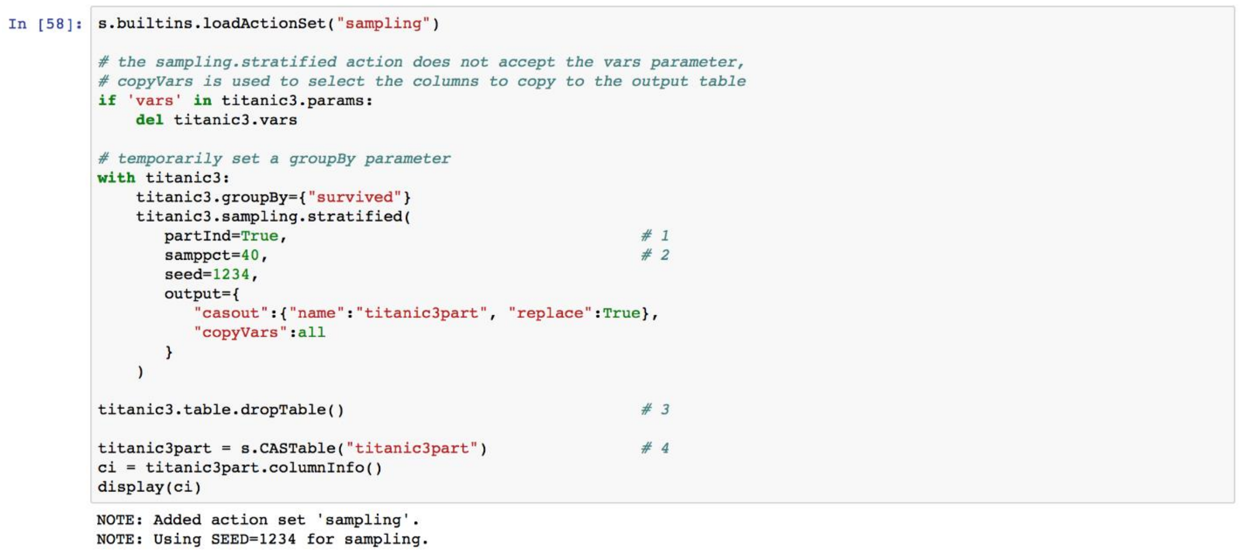
Descriptive Statistics for TITANIC3

	Column	Min	Max	N	NMiss	Mean	Sum	Std	StdErr
survived									
1	pclass	1.00	3.0000	500.0	0.0	1.962000	981.0000	0.872972	0.039040
1	survived	1.00	1.0000	500.0	0.0	1.000000	500.0000	0.000000	0.000000
1	age	0.17	80.0000	427.0	73.0	28.918244	12348.0900	15.061452	0.728875
1	sibsp	0.00	4.0000	500.0	0.0	0.462000	231.0000	0.685197	0.030643
1	parch	0.00	5.0000	500.0	0.0	0.476000	238.0000	0.776292	0.034717
1	fare	0.00	512.3292	500.0	0.0	49.361184	24680.5918	68.648795	3.070067

Figure 6. Grouped Data

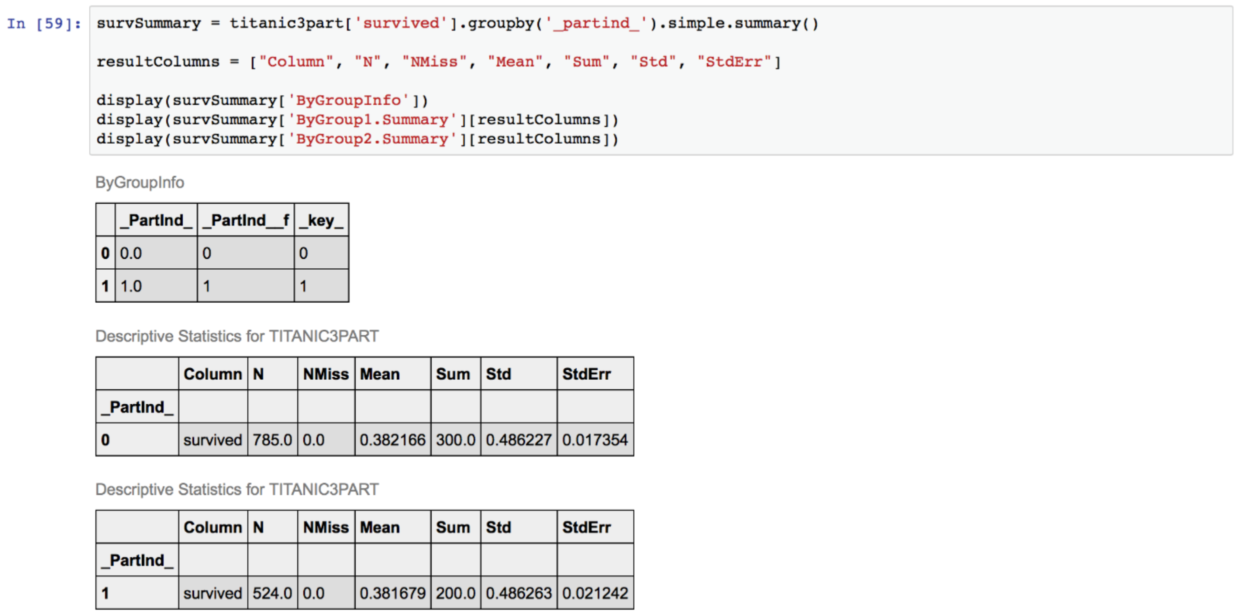
We are able to see the main summary statistics for the passengers on the Titanic. This data is summarized by column, giving us the ability to see things such as the mean value of each variable. Looking at this summary, we can immediately see that the passenger class does matter. There is a notable difference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



```
In [58]: s.builtins.loadActionSet("sampling")

         # the sampling.stratified action does not accept the vars parameter,
         # copyVars is used to select the columns to copy to the output table
         if 'vars' in titanic3.params:
             del titanic3.vars

         # temporarily set a groupBy parameter
         with titanic3:
             titanic3.groupBy={"survived"}
             titanic3.sampling.stratified(
                partInd=True,                                # 1
                samppct=40,                                  # 2
                seed=1234,
                output={
                   "casout":{"name":"titanic3part", "replace":True},
                   "copyVars":all
                }
             )

         titanic3.table.dropTable()                          # 3

         titanic3part = s.CASTable("titanic3part")           # 4
         ci = titanic3part.columnInfo()
         display(ci)

         NOTE: Added action set 'sampling'.
         NOTE: Using SEED=1234 for sampling.
```

Figure 7. Adding Sampling



```
In [59]: survSummary = titanic3part['survived'].groupby('_partind_').simple.summary()

         resultColumns = ["Column", "N", "NMiss", "Mean", "Sum", "Std", "StdErr"]

         display(survSummary['ByGroupInfo'])
         display(survSummary['ByGroup1.Summary'][resultColumns])
         display(survSummary['ByGroup2.Summary'][resultColumns])
```

ByGroupInfo

	PartInd	_PartInd__f	_key_
0	0.0	0	0
1	1.0	1	1

Descriptive Statistics for TITANIC3PART

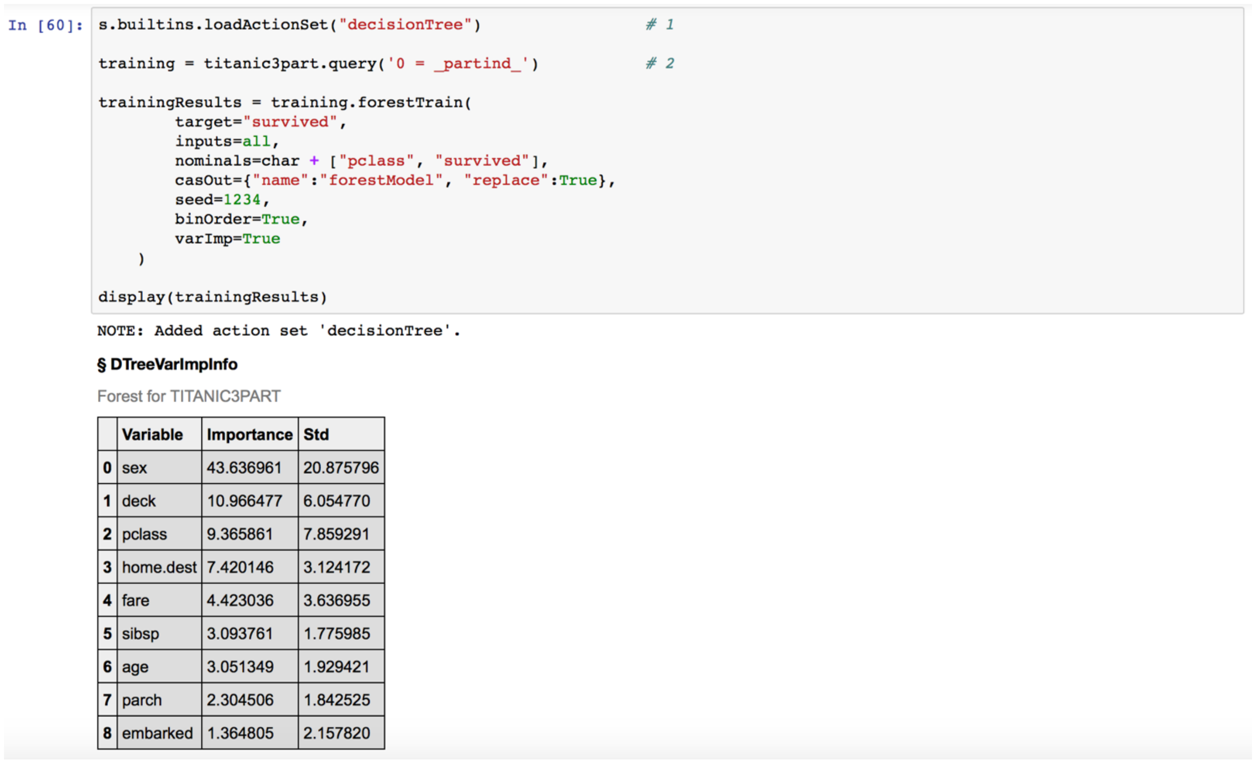
	Column	N	NMiss	Mean	Sum	Std	StdErr
PartInd							
0	survived	785.0	0.0	0.382166	300.0	0.486227	0.017354

Descriptive Statistics for TITANIC3PART

	Column	N	NMiss	Mean	Sum	Std	StdErr
PartInd							
1	survived	524.0	0.0	0.381679	200.0	0.486263	0.021242

Figure 8. Adding Partitioning

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.



```
In [60]: s.builtins.loadActionSet("decisionTree")                    # 1

         training = titanic3part.query('0 = _partind_')              # 2

         trainingResults = training.forestTrain(
                 target="survived",
                 inputs=all,
                 nominals=char + ["pclass", "survived"],
                 casOut={"name":"forestModel", "replace":True},
                 seed=1234,
                 binOrder=True,
                 varImp=True
             )

         display(trainingResults)
```

```
NOTE: Added action set 'decisionTree'.
```

§ DTreeVarImpInfo

Forest for TITANIC3PART

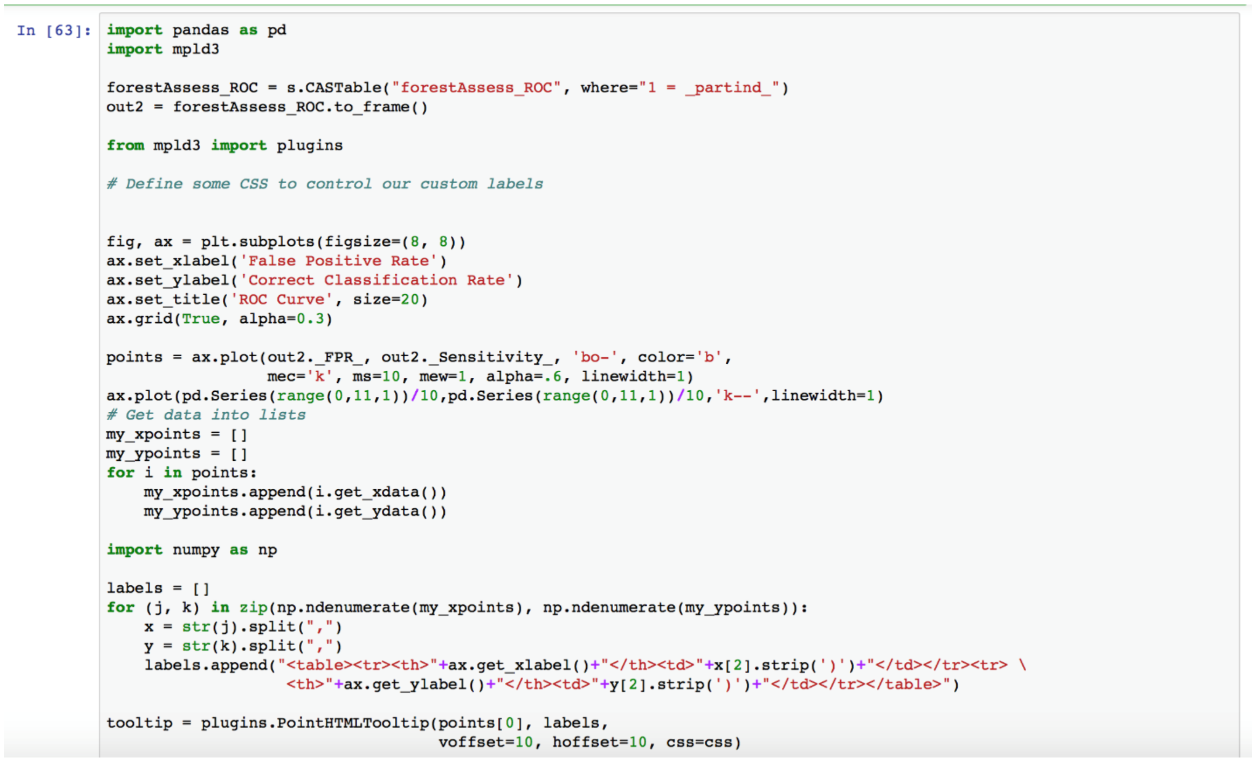
	Variable	Importance	Std
0	sex	43.636961	20.875796
1	deck	10.966477	6.054770
2	pclass	9.365861	7.859291
3	home.dest	7.420146	3.124172
4	fare	4.423036	3.636955
5	sibsp	3.093761	1.775985
6	age	3.051349	1.929421
7	parch	2.304506	1.842525
8	embarked	1.364805	2.157820

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.

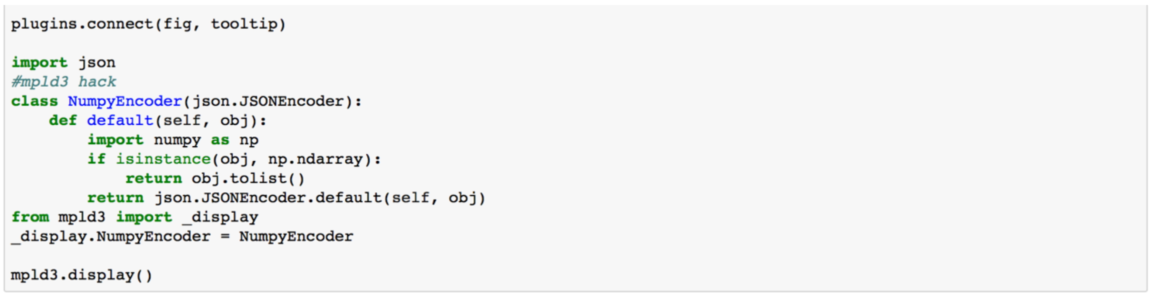
MATPLOTLIB AND D3

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 mpld3

         forestAssess_ROC = s.CASTable("forestAssess_ROC", where="1 = _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='b',
                          mec='k', ms=10, mew=1, alpha=.6, linewidth=1)
         ax.plot(pd.Series(range(0,11,1))/10,pd.Series(range(0,11,1))/10,'k--',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) in zip(np.ndenumerate(my_xpoints), np.ndenumerate(my_ypoints)):
             x = str(j).split(",")
             y = str(k).split(",")
             labels.append("<table><tr><th>"+ax.get_xlabel()+"</th><td>"+x[2].strip(')')+"</td></tr><tr> \
                           <th>"+ax.get_ylabel()+"</th><td>"+y[2].strip(')')+"</td></tr></table>")

         tooltip = plugins.PointHTMLTooltip(points[0], labels,
                                            voffset=10, hoffset=10, css=css)
```

Figure 10. Applying mpld3



```
plugins.connect(fig, tooltip)

import json
#mpld3 hack
class NumpyEncoder(json.JSONEncoder):
    def default(self, obj):
        import numpy as np
        if isinstance(obj, np.ndarray):
            return obj.tolist()
        return json.JSONEncoder.default(self, obj)
from mpld3 import _display
_display.NumpyEncoder = NumpyEncoder

mpld3.display()
```

Figure 11. Displaying the Data

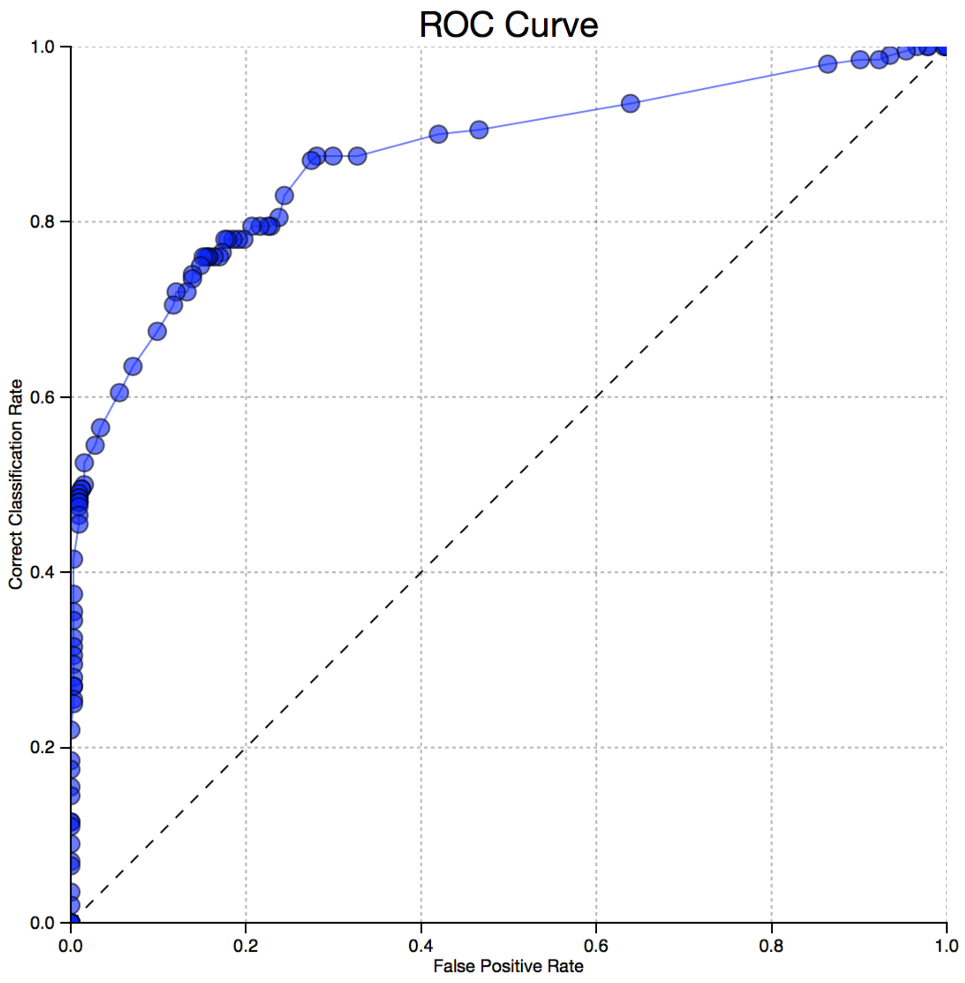
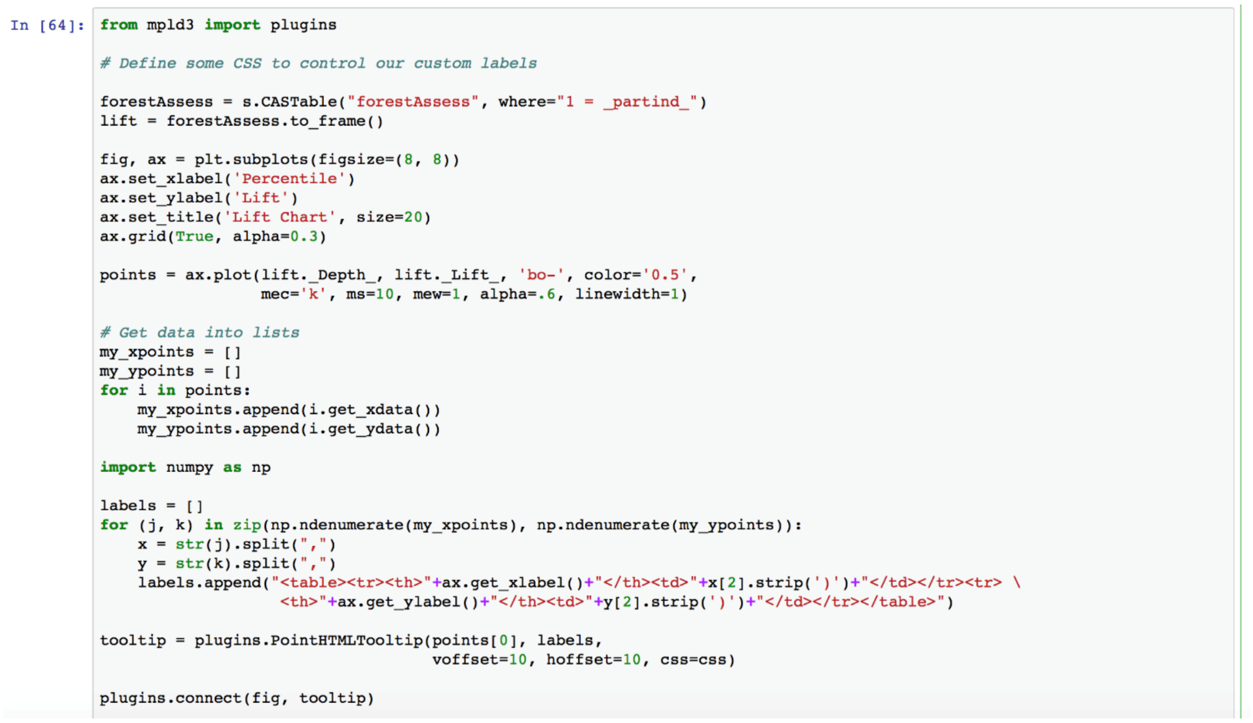


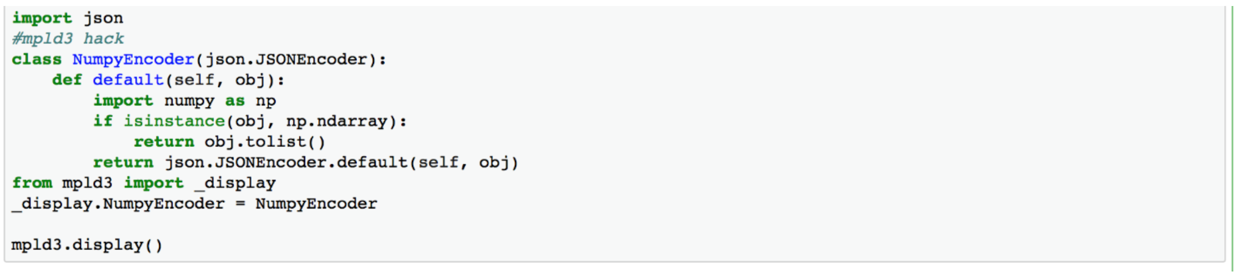
Figure 12. D3 Visualization

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.



```
In [64]: from mpld3 import plugins

         # Define some CSS to control our custom labels

         forestAssess = s.CASTable("forestAssess", where="1 = _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=.6, 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) in zip(np.ndenumerate(my_xpoints), np.ndenumerate(my_ypoints)):
             x = str(j).split(",")
             y = str(k).split(",")
             labels.append("<table><tr><th>"+ax.get_xlabel()+"</th><td>"+x[2].strip(')')+"</td></tr><tr> \
                           <th>"+ax.get_ylabel()+"</th><td>"+y[2].strip(')')+"</td></tr></table>")

         tooltip = plugins.PointHTMLTooltip(points[0], 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):
        import numpy as np
        if isinstance(obj, np.ndarray):
            return obj.tolist()
        return json.JSONEncoder.default(self, obj)
from mpld3 import _display
_display.NumpyEncoder = NumpyEncoder

mpld3.display()
```

Figure 14. Displaying the Conditionally Formatted Chart

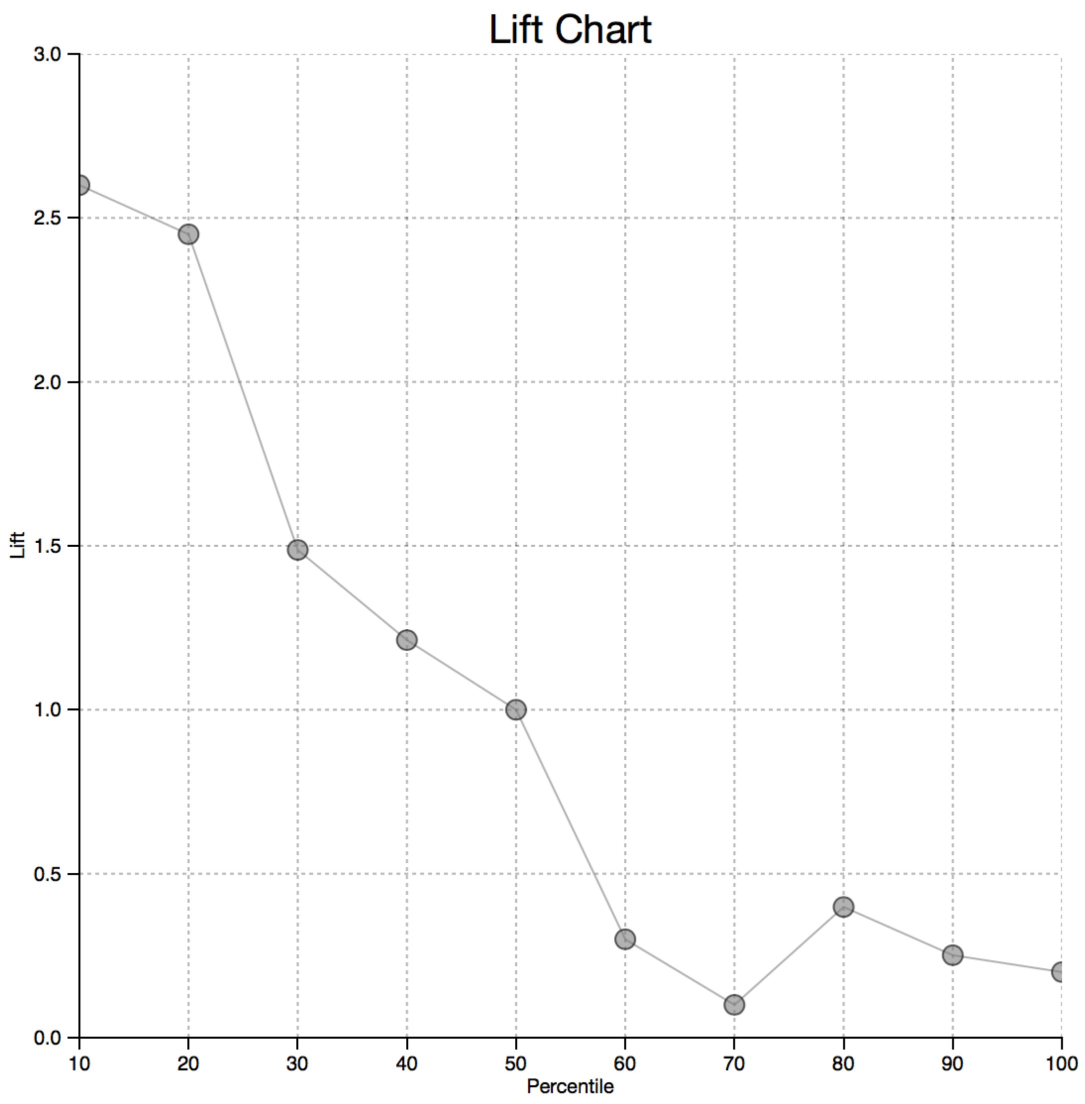


Figure 15. D3 Visualization of Conditionally Formatted Data

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

SAS Institute Inc. 2019. SAS Software / python-swat. Accessed April 10, 2019. Available: https://github.com/sassoftware/python-swat.

SAS Institute Inc. 2018. "Programming Considerations for Data-Driven Visualizations." In *SAS Visual Analytics 8.2: Reference.* Cary, NC: SAS Institute Inc. Available https://documentation.sas.com/?cdcId=vacdc&cdcVersion=8.2&docsetId=varef&docsetTarget=n109mqtyl6quiun1mwfgtcn2s68b.htm&locale=en

SAS Institute Inc. 2018. "API Reference." In *SAS Scripting Wrapper for Analytics Transfer*. Cary NC; SAS Institute Inc. Accessed April 10, 2019. Available: https://developer.sas.com/apis/swat/python/v1.4.0/api.html

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