MSIS 5223: Tutorial 6 – Data Reduction and Exploration in Python

Instructions

The purpose of this tutorial is to help you become familiar with using Python to narrow down data. While narrowing the scope of a project can in turn narrow the data, this often doesn't reduce the data enough. Additionally, an individual may not be familiar enough with a dataset to choose the correct variables for a given analysis.

This exercise will utilize several analyses to reduce the number of columns in a given dataset. Note, these techniques are not designed to reduce the number of rows in a dataset, necessarily. These are all analyses that fall under the umbrella term *multivariate statistics*, because they deal with multiple variables.

1. Principal Components Analysis (PCA)

For this example, the dataset datareduction.csv is used. This dataset resulted from an assessment of employees' adoption of a new system within an organization. Many different variables exist in the dataset, three of which will be used in this example:

- Perceived Usefulness – How useful an individual thinks the new technology is in performing job functions
- Perceived Ease of Use – How easy-to-use the new system appears to be
- Intention to Use – The extent to which an individual believes he/she will use the new system to perform his/her job

Each of these three variables are measured on a 7-point scale. A 1 represents strongly disagree while a 7 represents strongly agree. *Perceived Usefulness* (PU) and *Perceived Ease of Use* (PEOU) were measured with 6 different questions each while *Intention to Use* was measured with 3 different questions. Begin by separating out the columns of data that pertain to the analysis. This means we will need to create a new dataframe that only contains the measures of PU, PEOU, and Intention.

```
In [19]: reduc_data_pca = reduc_data[['peruse01', 'peruse02', 'peruse03', 'peruse04', 'peruse05',
   ....:                             'peruse06', 'pereou01', 'pereou02', 'pereou03', 'pereou04',
   ....:                             'pereou05','pereou06', 'intent01', 'intent02', 'intent03']]
```

Figure 1 Creating a New Dataframe

A new dataframe is created called reduc_data_pca. This only contains the columns of data related to PU, PEOU, and Intention. The next step is to run the PCA and assess the eigenvalues, or the variance calculated from the PCA.

```
In [21]: pca_result = pca(n_components=15).fit(reduc_data_pca)

In [22]: pca_result.explained_variance_
Out[22]:
array([ 9.19160849,  3.76000599,  2.24890063,  0.71822586,  0.53855461,
        0.49629962,  0.43223368,  0.37673153,  0.32293239,  0.29940556,
        0.27381155,  0.25994786,  0.17854935,  0.15814993,  0.13317753])
```

Figure 2 PCA Eigenvalues (i.e. Variances)

Look at Figure 2, the results of the PCA. Notice that the output has created 15 components. You will always have the same number of components as the number of columns you put into the analysis. Recall that PU has 6 items, PEOU has 6 items, and Intention has 3. This results in 15 columns of data used as input. You may be confused at this moment, because you were told that the purpose of PCA was to reduce the number of columns; the output has produced 15 components. It appears that nothing was removed.

This brings up the next step of the analysis. Notice that each component has an associated value; the first component has the greatest value, 3.032 for Component 1, and the last component has the smallest value, 0.365 for Component 15. This is where we start to reduce this dataset.

The way this PCA is assessed is any value greater than 1.0 is retained, while anything less than 1.0 is thrown out. The first three components all have values greater than 1.0. Starting with Component 4, however, the values are all smaller than 1.0. What this assessment tells us, is that in reality we are only dealing with 3 variables, not 15.

We can run a Scree Plot to confirm the findings from the PCA. Below is the R code for a Scree Plot along with the resulting plot itself.

```
In [18]: plt.figure(figsize=(7,5))
   ....: plt.plot([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15], pca_result.explained_variance_ratio_, '-o')
   ....: plt.ylabel('Proportion of Variance Explained')
   ....: plt.xlabel('Principal Component')
   ....: plt.xlim(0.75,4.25)
   ....: plt.ylim(0,1.05)
   ....: plt.xticks([1,2,3,4,5,6,7,8,9,10,11,12,13,14,15])
```

Figure 3 Code to Create Scree Plot

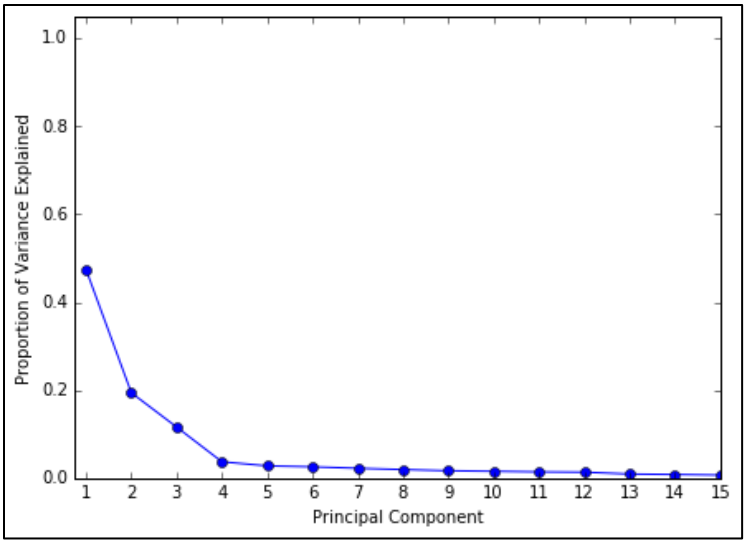

Figure 4 Scree Plot of Eigenvalues

Scree plots are more subjective than assessing eigenvalues. The purpose of using the scree plot is to determine how many components, or columns of data, you are truly dealing with. The way to read a scree plot is determining where the plot levels off and becomes flat; anything prior to that leveling off is a component that remains.

Look at Figure 4. You can see that the first 3 components are not flat. Starting with Component 4 you can see the plot is level and flat. This agrees with the eigenvalue results given on the previous page. Out of 15 variables, only 3 should be used for the analysis. This means that some combination of all 15 variables should yield just 3.

2. Cluster Analysis

Recall that cluster analysis attempts to maximize the between-cluster variance and minimizing within-cluster variance. K-means, k-medians, k-modes all are centroid-based algorithms. That is, each cluster is created based on a central point in space in which its data points surround. Hierarchical clustering, which includes agglomerative and divisive, are based on distance of objects to one another, not a centroid.

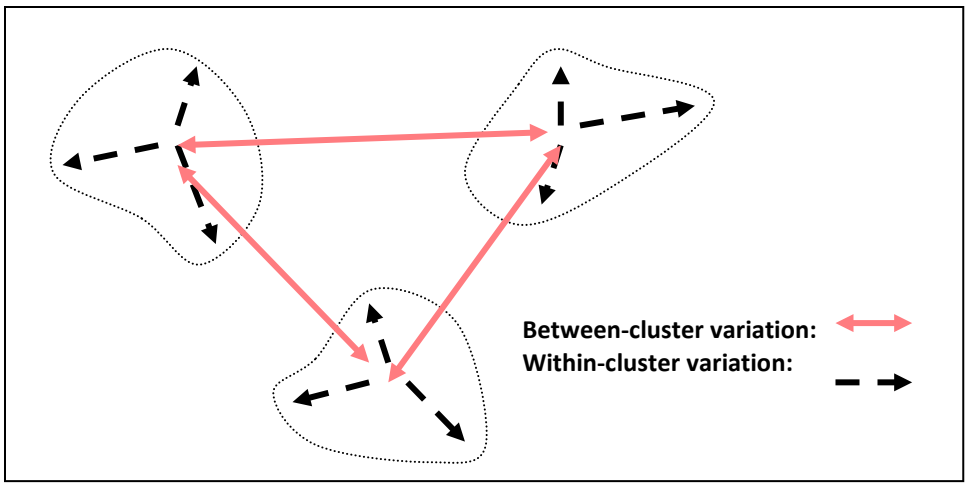

Figure 5 Variance for Cluster Analysis

The dataset kmeansdata.txt contains four variables, two of which are categorical: *group* and *grouping*. The variable *grouping* has values A, B, C, D, E, and F as shown in Figure 6. These represent natural groupings within the data. When using clustering, you would normally not have variables such as *grouping* or *group* because you are unaware of how the data group together.

```
In [65]: kmeans_data.grouping.unique()
Out[65]: array(['A', 'B', 'C', 'D', 'E', 'F'],
```

Figure 6 Grouping Variable Values

K-means is based on the idea that you choose the number of clusters *k* and the algorithm will determine, based on initial seeding, where those clusters are. In *Tutorial 05*, this same data was processed with 4 and 6 clusters; the same process is followed in Figure 7 below.

```
In [78]: km = cls.KMeans(n_clusters=6).fit(kmeans_data.loc[:,['x','y']])

In [79]: km2 = cls.KMeans(n_clusters=4).fit(kmeans_data.loc[:,['x','y']])

In [80]: km.labels_
Out[80]:
array([1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0, 2, 2, 3,
       3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
       3, 0, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 5, 1, 5, 5, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 2, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4, 4,
       4, 4, 4, 4, 4, 4, 4])

In [81]: km2.labels_
Out[81]:
array([2, 2, 2, 2, 2, 2, 0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 0, 1, 1, 1, 1, 1, 0, 1, 1, 1, 3,
       3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3,
       3, 1, 2, 2, 2, 2, 2, 2, 3, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0])
```

Figure 7 KMeans with 4 and 6 Clusters

The classification of the data for both cluster algorithms performs fairly well. The results of the 4-cluster result shows some consistency. The first group consists of 1s, the second group 2s, then 3s, 5s, 0s, and last 4s. When the clustering is switched to the 6-cluster group, the results appear to be improved.

Just like in R, you can assess the misclassification using a confusion matrix. The next two figures show two methods to accomplish this. The first is a text-based matrix that does not have labels (though, that is something you could easily change). The second figure presents a color-based gradient of the exact same data. Below is the code used to create both of these confusion matrices.

```
In [53]: print(cm)
[[ 0  0  0  0  0  0  0]
 [ 0  0  0  0  0 20  0]
 [ 0  0 24  1  0  0  0]
 [ 0 25  0  0  0  0  0]
 [18  0  0  1  0  1  0]
 [ 0  0  3 27  0  0  0]
 [ 0  0  0  0 25  0  0]]
```

Figure 8 Text-Based Confusion Matrix

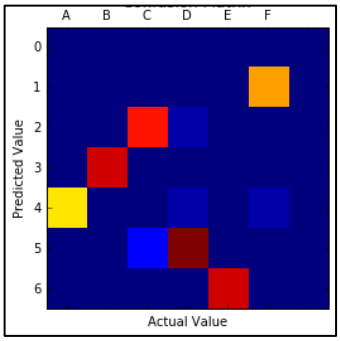

Figure 9 Color-Gradient Matrix

```
#Create a confusion matrix
cm = metrics.confusion_matrix(kmeans_data.group, km.labels_)
print(cm)       #Printed matrix

#Color-based chart
plt.matshow(cm)
plt.title('Confusion Matrix')
plt.xlabel('Actual Value')
plt.ylabel('Predicted Value')
plt.xticks([0,1,2,3,4,5,6], ['A','B','C','D','E','F'])
```

Figure 10 Creating a Confusion Matrix

The next example for this exercise is to illustrate that clustering may not always perform well. In fact, when a decision tree is used on the data, the categorization is better than with clustering. Use the taxon.txt data. It contains 120 observed plants from 4 different islands. Specific characteristics, seven variables, for taxonomy were measured for all plants.

```
In [15]: taxon_data.head()
Out[15]:
     Petals  Internode     Sepal      Bract    Petiole      Leaf     Fruit
1  5.621498  29.480596  2.462107  18.203409  11.279097  1.128033  7.876151
2  4.994617  28.360247  2.429321  17.652049  11.040838  1.197617  7.025416
3  4.767505  27.254318  2.570497  19.408385  10.490722  1.003808  7.817479
4  6.299446  25.924238  2.066051  18.379155  11.801823  1.614052  7.672492
5  6.489375  25.211308  2.901583  17.313047  10.121590  1.813333  7.758443
```

Figure 11 Taxon Data Variables

A k-means algorithm is run on the data for 4 clusters. The results are shown below in Figure 12. The data itself is sorted so that the first 30 observations belong to taxon I, the next 30 to taxon II, the next 30 to taxon III, etc. This cluster analysis was performed with 4 clusters; however, the results are not very good.

```
In [16]: km3 = cls.KMeans(n_clusters=4).fit(taxon_data)
   ....: km3.labels_
Out[16]:
array([3, 3, 3, 3, 0, 0, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 3, 1, 3, 3,
       3, 3, 1, 3, 3, 1, 3, 0, 0, 1, 1, 0, 0, 0, 0, 0, 0, 1, 1, 0, 0, 0, 1,
       0, 0, 3, 0, 0, 0, 0, 0, 1, 0, 1, 1, 1, 0, 1, 0, 0, 0, 1, 1, 0, 0, 0,
       1, 1, 0, 1, 0, 0, 0, 0, 1, 1, 1, 0, 3, 0, 0, 1, 1, 1, 0, 3, 1, 2, 1,
       2, 1, 2, 2, 1, 3, 2, 2, 1, 2, 2, 2, 1, 2, 2, 2, 2, 2, 2, 1, 1, 2, 2,
       2, 1, 2, 1, 1])
```

Figure 12 Cluster Assignment of Taxon Data

Look at the first 30 records; the observations are placed in clusters 0 and 3. All of these observations should be just one single cluster. The next group of 30 are worse: about half are in cluster 1 while the other half are in cluster 0. Obviously, clustering did not perform very well. If you run the clustering algorithm again with just 3 clusters, instead of 4, the results come out cleaner. This won't do, because we know this data contains 4 groups.

As an alternative, a hierarchical clustering technique can be performed on the data. Specifically, an agglomerative analysis. Agglomerative is an additive process. It starts out with each observation and slowly clusters them, based on Euclidean distance, until only one cluster remains. Go ahead and run the analysis and plot the results. Figure 13 presents the Python code to plot the clustering. Figure 10 is the plot created using the code.

```
#Create a plot to view the output
from scipy.cluster import hierarchy as hier
z = hier.linkage(taxon_data, 'single')
plt.figure()
dn = hier.dendrogram(z)
```

Figure 13 Running Agglomerative Analysis for Taxon Data

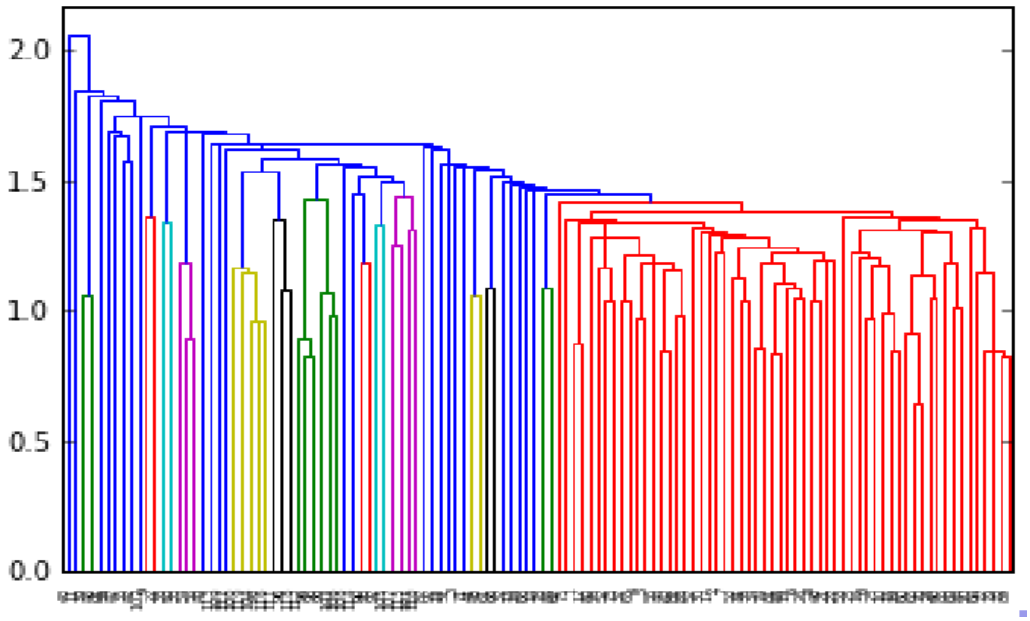

Figure 14 Plot of Cluster Results for Taxon Data

The results are not improved. No matter which break you look at, the clusters involve observations from more than one group. Clustering did not appear to perform very well with this data. This is an important lesson to learn about clustering. Clustering may not find a global optimum; in many situations it only finds the local optimum.

```
tre1 = tree.DecisionTreeClassifier('gini').fit(taxon_data.ix[:,1:8],taxon_data.Taxon)
col_names = list(taxon_data.columns.values)
classnames = list(taxon_data.Taxon.unique())
dot_data = StringIO()
tree.export_graphviz(tre1, out_file=dot_data,
                     feature_names=col_names[1:7],
                     class_names=classnames,
                     filled=True,
                     rounded=True,
                     special_characters=True)
graph = pydotplus.graph_from_dot_data(dot_data.getvalue())
Image(graph.create_png())
```

Figure 15 Classification Tree Code

To help further illustrate this, see Figure 16 below (Figure 15 above shows the code). This is the result of a decision tree on the taxon data. You can see that the tree was able to partition the data into four separate groups based on *sepal*, *leaf*, and *petiole*. This

is much cleaner than clustering. Note, if this is your first time running a classification tree in Python, you need to install Graphviz first prior to running this code (see Appendix at the end of this tutorial before progressing).

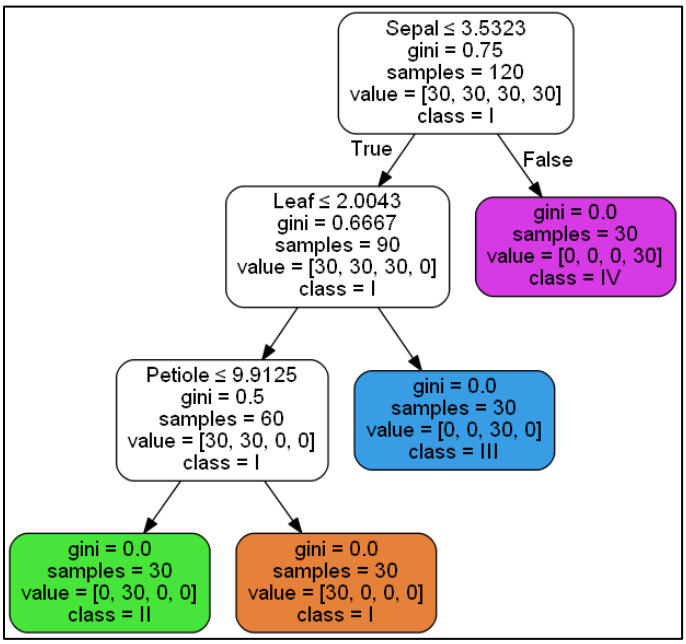

Figure 16 Decision Tree for Taxon Data

In various projects you engage in, it is often beneficial to attempt multiple types of statistical techniques. This process allows you to see where your analyses differ and agree. The temptation, typically, is to use a single process and then accept the results at face value. Don't just settle for using one type of analysis!

The last clustering example for this exercise covers hierarchical clustering. Specifically, agglomerative analysis. For this example, the data found in pgfull.txt is used. This data provides observations on plant growth for 54 plant species on 89 plots. Calculate the Euclidean distance for all of the rows and columns. Note, only do this for the first 54 columns; the other 5 (plot, lime, species, hay, pH) are not considered for this. Once calculated, the results can be plotted.

```
aggl = cls.AgglomerativeClustering(linkage='ward').fit(pg_data.ix[:,0:54])

#Create a plot to view the output
z = hier.linkage(pg_data, 'single')
plt.figure()
dn = hier.dendrogram(z)
```

Figure 17 Agglomerative Analysis Function

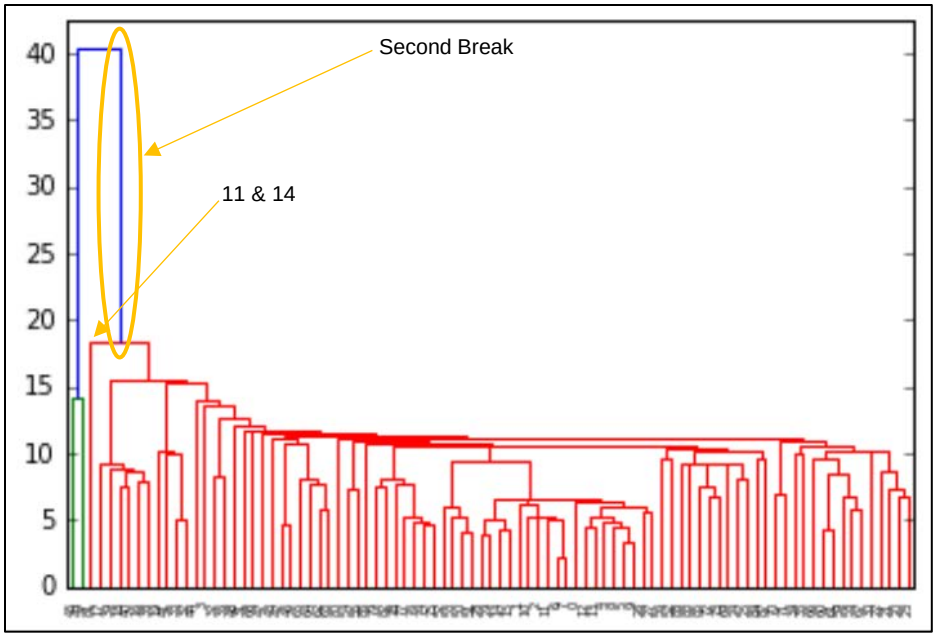

Figure 18 Hierarchical Clustering Results

Observe the second break in the clustering, the one circled in yellow. Notice that the plants on the left of the break belong to plots 11 and 14; those on the right are the rest. Plots 11 and 14 are high nitrogen plots receiving phosphorus.[1]

[1] See the following for more information on this data: Crawley et al. 2005. "Determinants of species richness in the Park Grass Experiment," *American Naturalist*, 165, pp. 348-362.

Appendix: Installing Graphviz

Graphviz is separate software that . To download the application, open the following link in a browser: http://www.graphviz.org/Download.php. Scroll down the page and select your version (see figure below). For Windows users, after clicking on the link you will be provided with two options, an .msi file and a .zip. Either is fine.

Figure 19 Available Packages of Graphviz

The rest of this process is for the Windows installation. Run the installation program and install. Once complete, the next step is to setup a path to Graphviz so Python can find the application. Press the Windows Key + Pause/Break. If you do not have a Windows Key on your keyboard, open up Control Panel, go to *System and Security*, and then *System*. On the left-hand side is a menu; click on *Advanced system settings*. Click on the *Advanced* tab. At the bottom is a button titled *Environmental Variables…*; click on this.

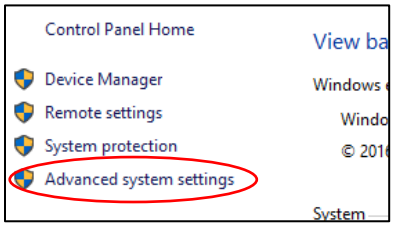

Figure 20 Windows System

You should now have an open window like that shown below. Ensure that PATH is highlighted and click the button *Edit*. You should now see a list of paths. Do not edit or delete any of these listed paths. Click on the button *New* to create a path to Graphviz.

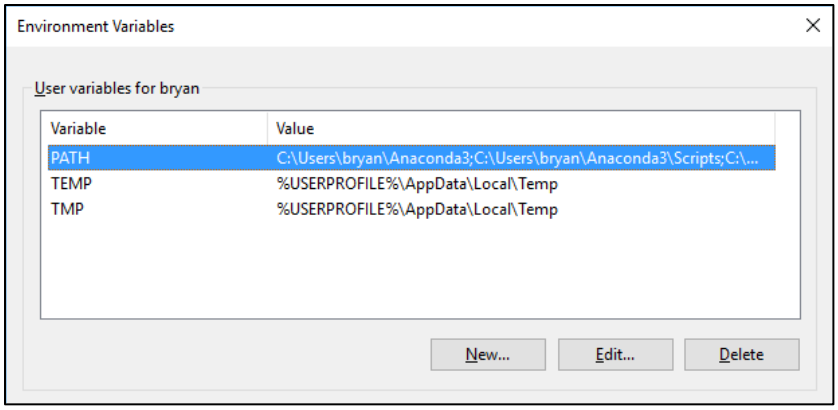

Figure 21 Environmental Variables.

Type in the path file to the bin folder for Graphviz. For example, the path for my installation of Graphviz is C:\Program Files (x86)\Graphviz2.38\bin. The bin folder is the location of the executable file used by Python. Once done, click OK on each subsequent window until you return to the System window. Close this down.

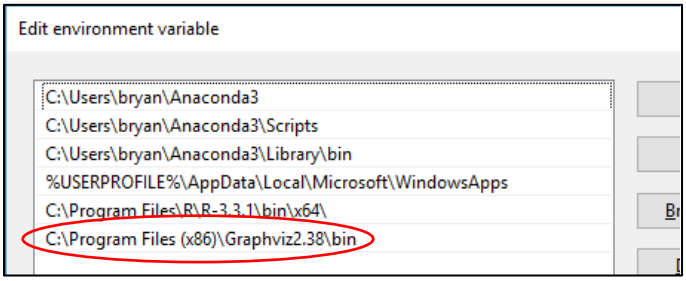

Figure 22 Environmental Variables or Paths

In addition to installing the Graphviz application, Python requires additional modules installed. For example, you will need pydotplus. If you do not already have these installed, please continue reading.

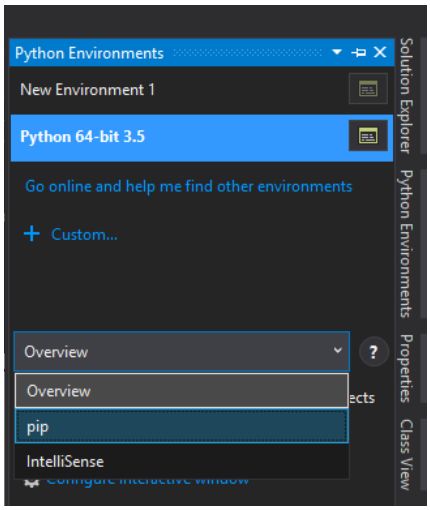

Figure 23 Using pip in Visual Studio

The best way to install new modules for Python is to use pip. Within Visual Studio, pip can be accessed by using the Python Environments window (see Figure to the right). Using the drop-down menu, select pip. Directly under the drop-down menu, a search box is presented. Inside the search box, type pydotplus. The first entry should be "pip install pydotplus" from PyPI. Click on this. You

should see a new screen within Visual Studio showing the install process. A message indicates if the installation was successful. Note, depending on your internet connection, the installation process may take longer than a few minutes.

Once completed, you will need to refresh the database for Python's modules. Within the Python Environments window, click on the drop-down menu and select IntelliSense. Click on the button to refresh the database. This may take anywhere from 2 minutes to 10. I suggest leaving your computer and coming back. Do not put your computer to sleep while the database is refreshing; this may pause the process.