Clustering

1. DBSCAN

Using DBSCAN iterate (for-loop) through different values of `min_samples` (1 to 10) and `epsilon` (.05 to .5, in steps of .01) to find clusters in the road-data used in the Lesson and calculate the Silohouette Coeff for `min_samples` and `epsilon`. Plot ***one*** line plot with the multiple lines generated from the min_samples and epsilon values. Use a 2D array to store the SilCoeff values, one dimension represents `min_samples`, the other represents epsilon.

Expecting a plot of `epsilon` vs `sil_score`.

```
In [1]: import pandas as pd
        import numpy as np
        %matplotlib notebook
        import matplotlib.pyplot as plt
        import seaborn
        # from mpl_toolkits.mplot3d import Axes3D
        plt.rcParams['font.size'] = 14
        from sklearn.cluster import DBSCAN
        from sklearn import metrics
        from sklearn.preprocessing import StandardScaler
```

```
In [2]: # Read Data in
        Q1X = pd.read_csv('../data/3D_spatial_network.txt.gz', header=None, names=['osm', 'lat','lon','alt'])
        Q1X = Q1X.drop(['osm'], axis=1).sample(10000)
        Q1X.head()
```

Out[2]:

	lat	lon	alt
96133	8.992851	56.806220	6.667074
184854	10.301893	57.030648	5.680403
424293	9.796924	56.631642	41.530143
129724	9.138709	56.701715	36.073703
327651	10.036135	57.582191	16.018632

```
In [3]: # Scale Data
        scaler = StandardScaler()
        Q1X_scaled = scaler.fit_transform(Q1X)
```

```
In [4]: Q1X_scaled
```

```
Out[4]: array([[-1.18305522, -0.95108985, -0.84315631],
               [ 0.92270456, -0.16924913, -0.89627039],
               [ 0.11039833, -1.55926897,  1.03357883],
               ...,
               [ 0.93495667,  0.3982034 ,  2.44251092],
               [ 0.32851064, -0.22065049,  1.03188215],
               [-0.1843514 ,  0.67253115,  0.19013874]])
```

```
In [5]: # convert scaled data to dataframe
        Q1X_scaled = pd.DataFrame(Q1X_scaled, columns=['lat', 'lon', 'alt'])
        Q1X_scaled
```

Out[5]:

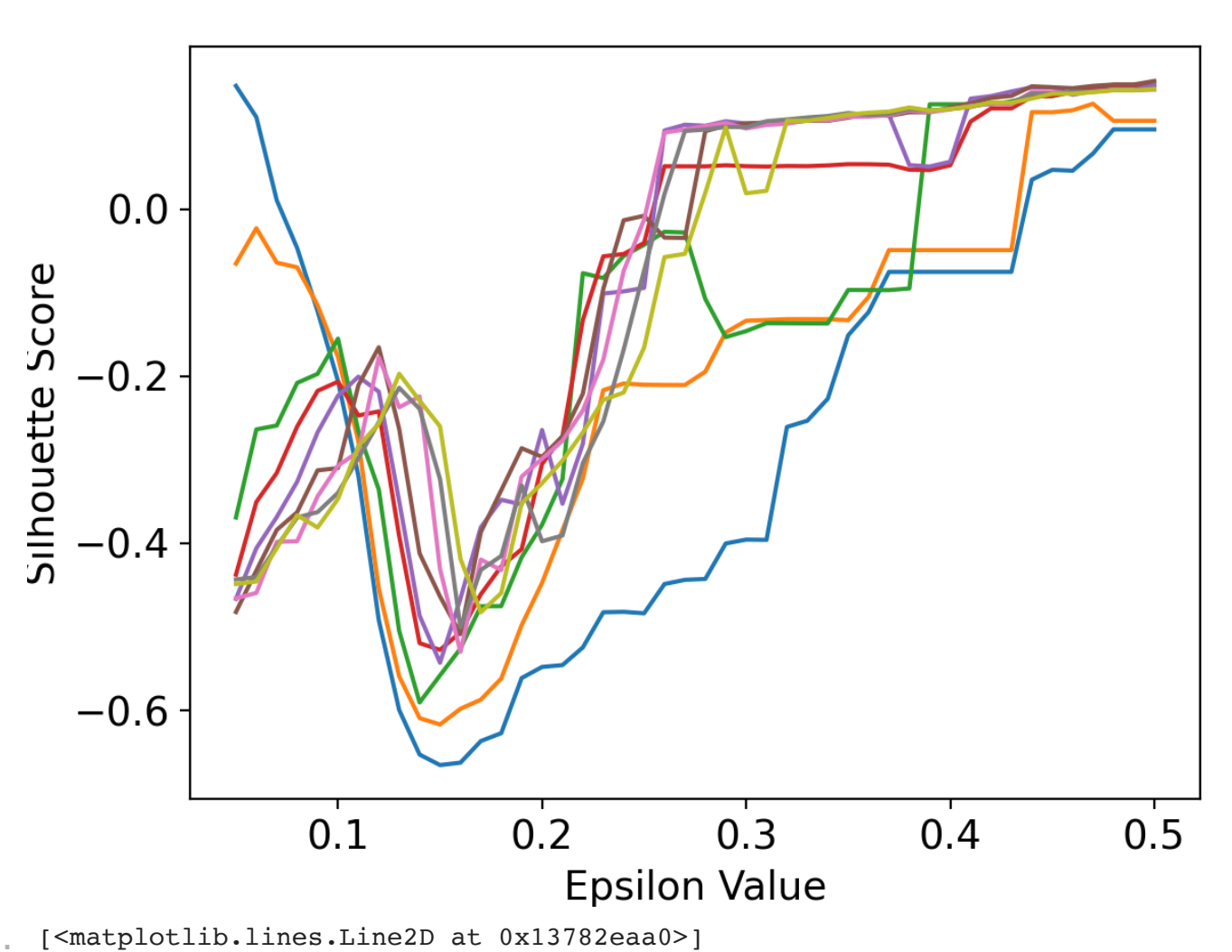
	lat	lon	alt
0	-1.183055	-0.951090	-0.843156
1	0.922705	-0.169249	-0.896270
2	0.110398	-1.559269	1.033579
3	-0.948424	-1.315153	0.739850
4	0.495200	1.752164	-0.339747
...	...	...	...
9995	0.665449	-0.159764	-1.118352
9996	0.323551	-0.306497	0.288464
9997	0.934957	0.398203	2.442511
9998	0.328511	-0.220650	1.031882
9999	-0.184351	0.672531	0.190139

10000 rows × 3 columns

Please note, the next cell will take a few minutes to fully execute

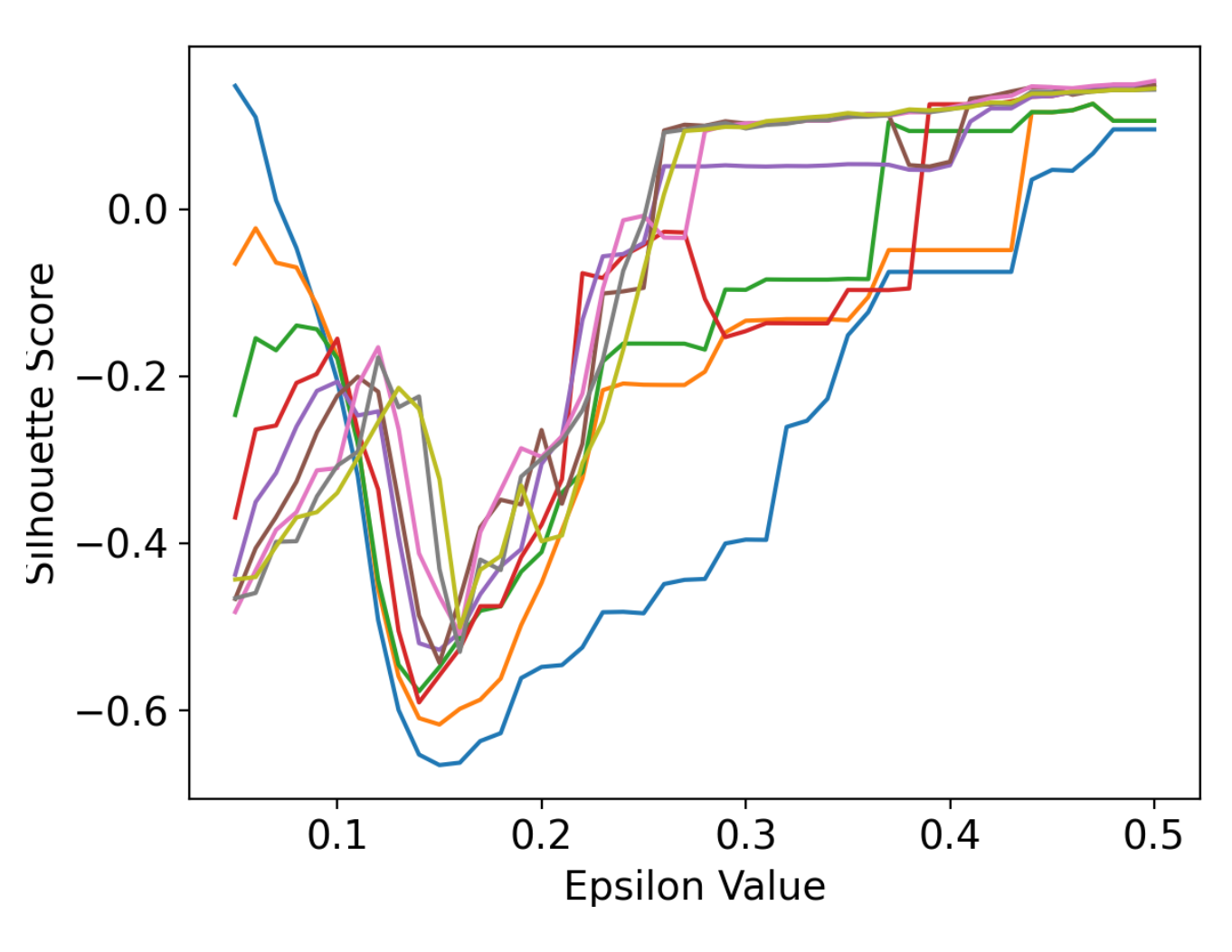
```
In [6]: min_samples = np.arange(1, 11)
        epsilons = np.arange(0.05,0.51,0.01)

        all_scores = []
        all_eps_values = []
        for min_sample in min_samples:
            scores = []
            eps_values = []
            for epsilon in epsilons:
                dbscanQ1 = DBSCAN(eps=epsilon, min_samples=min_sample).fit(Q1X_scaled[['lat','lon', 'alt']])
                score = metrics.silhouette_score(Q1X_scaled[['lon', 'lat', 'alt']], dbscanQ1.labels_)
                epsvalue = epsilon
                scores.append(score)
                eps_values.append(epsvalue)

            all_scores.append(scores)
            all_eps_values.append(eps_values)
```

```
In [7]: plt.figure()
        plt.xlabel('Epsilon Value')
        plt.ylabel('Silhouette Score')
        plt.plot(all_eps_values[0], all_scores[0])
        plt.plot(all_eps_values[1], all_scores[1])
        plt.plot(all_eps_values[3], all_scores[3])
        plt.plot(all_eps_values[4], all_scores[4])
        plt.plot(all_eps_values[5], all_scores[5])
        plt.plot(all_eps_values[6], all_scores[6])
        plt.plot(all_eps_values[7], all_scores[7])
        plt.plot(all_eps_values[8], all_scores[8])
        plt.plot(all_eps_values[9], all_scores[9])
```



```
Out[7]: [<matplotlib.lines.Line2D at 0x13782eaa0>]
```

```
In [8]: # Another way of plotting all the lines
        graph_loop_val = np.arange(0, 9)
        plt.figure()
        plt.xlabel('Epsilon Value')
        plt.ylabel('Silhouette Score')
        for i in graph_loop_val:
            plt.plot(all_eps_values[i],all_scores[i])
```



2. Clustering your own data

Using your own data, find relevant clusters/groups within your data (repeat the above). If your data is labeled with a class that you are attempting to predict, be sure to not use it in training and clustering.

You may use the labels to compare with predictions to show how well the clustering performed using one of the clustering metrics (http://scikit-learn.org/stable/modules/clustering.html#clustering-performance-evaluation).

If you don't have labels, use the silhouette coefficient to show performance. Find the optimal fit for your data but you don't need to be as exhaustive as above.

Additionally, show the clusters in 2D or 3D plots.

As a bonus, try using PCA first to condense your data from N columns to less than N.

Two items are expected:

- Metric Evaluation Plot (like in 1.)
- Plots of the clustered data

```
In [9]: Q2X = pd.read_csv("marketing_campaign.csv", sep=';')
        pd.set_option('display.max_columns', 50)
        Q2X.shape
```

```
Out[9]: (2240, 29)
```

Quick Exploration of dataset

```
In [10]: Q2X.describe(include='all')
```

Out[10]:

	ID	Year_Birth	Education	Marital_Status	Income	Kidhome	Teenhome	Dt_Customer	Recency	MntWines	MntFruits	MntMeatProducts	MntFishProducts	M
count	2240.000000	2240.000000	2240	2240	2216.000000	2240.000000	2240.000000	2240	2240.000000	2240.000000	2240.000000	2240.000000	2240.000000	
unique	NaN	NaN	5	8	NaN	NaN	NaN	663	NaN	NaN	NaN	NaN	NaN	
top	NaN	NaN	Graduation	Married	NaN	NaN	NaN	2012-08-31	NaN	NaN	NaN	NaN	NaN	
freq	NaN	NaN	1127	864	NaN	NaN	NaN	12	NaN	NaN	NaN	NaN	NaN	
mean	5592.159821	1968.805804	NaN	NaN	52247.251354	0.444196	0.506250	NaN	49.109375	303.935714	26.302232	166.950000	37.525446	
std	3246.662198	11.984069	NaN	NaN	25173.076661	0.538398	0.544538	NaN	28.962453	336.597393	39.773434	225.715373	54.628979	
min	0.000000	1893.000000	NaN	NaN	1730.000000	0.000000	0.000000	NaN	0.000000	0.000000	0.000000	0.000000	0.000000	
25%	2828.250000	1959.000000	NaN	NaN	35303.000000	0.000000	0.000000	NaN	24.000000	23.750000	1.000000	16.000000	3.000000	
50%	5458.500000	1970.000000	NaN	NaN	51381.500000	0.000000	0.000000	NaN	49.000000	173.500000	8.000000	67.000000	12.000000	
75%	8427.750000	1977.000000	NaN	NaN	68522.000000	1.000000	1.000000	NaN	74.000000	504.250000	33.000000	232.000000	50.000000	
max	11191.000000	1996.000000	NaN	NaN	666666.000000	2.000000	2.000000	NaN	99.000000	1493.000000	199.000000	1725.000000	259.000000	

Finding NaN values and dropping them

It looks like NaN values are only in the income column

```
In [11]: print(Q2X.isna().sum().sum())
         print(len(Q2X))
         print(Q2X.columns[Q2X.isna().any()].tolist())

         24
         2240
         ['Income']
```

```
In [12]: Q2X = Q2X.dropna()
         Q2X.shape
```

```
Out[12]: (2216, 29)
```

Data Wrangling to prepare for model input

```
In [13]: from sklearn import preprocessing
```

```
In [14]: drop_cols = ['ID','Dt_Customer', 'Z_Revenue','Z_CostContact','Complain']
         transform_cols = ['Education','Marital_Status']
         classes = ['AcceptedCmp3','AcceptedCmp4','AcceptedCmp5', 'AcceptedCmp1','AcceptedCmp2','Response']
```

```
In [15]: Q2X = Q2X.drop(drop_cols, axis=1)
```

```
In [16]: Q2Xt = Q2X.drop(classes, axis=1)
         Q2classes = Q2X[classes]

         enc = preprocessing.OrdinalEncoder()
```

```
In [17]: Q2Xt[transform_cols] = enc.fit_transform(Q2Xt[transform_cols])
```

```
In [18]: Q2Xt.describe()
```

Out[18]:

	Year_Birth	Education	Marital_Status	Income	Kidhome	Teenhome	Recency	MntWines	MntFruits	MntMeatProducts	MntFishProducts	MntSweetProducts	MntGoldPr
count	2216.000000	2216.000000	2216.000000	2216.000000	2216.000000	2216.000000	2216.000000	2216.000000	2216.000000	2216.000000	2216.000000	2216.000000	2216.000
mean	1968.820397	2.393953	3.726083	52247.251354	0.441787	0.505415	49.012635	305.091606	26.356047	166.995939	37.637635	27.028881	43.965
std	11.985554	1.124141	1.077731	25173.076661	0.536896	0.544181	28.948352	337.327920	39.793917	224.283273	54.752082	41.072046	51.815
min	1893.000000	0.000000	0.000000	1730.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000
25%	1959.000000	2.000000	3.000000	35303.000000	0.000000	0.000000	24.000000	24.000000	2.000000	16.000000	3.000000	1.000000	9.000
50%	1970.000000	2.000000	4.000000	51381.500000	0.000000	0.000000	49.000000	174.500000	8.000000	68.000000	12.000000	8.000000	24.500
75%	1977.000000	3.000000	5.000000	68522.000000	1.000000	1.000000	74.000000	505.000000	33.000000	232.250000	50.000000	33.000000	56.000
max	1996.000000	4.000000	7.000000	666666.000000	2.000000	2.000000	99.000000	1493.000000	199.000000	1725.000000	259.000000	262.000000	321.000

```
In [19]: Q2Xt.shape
```

```
Out[19]: (2216, 18)
```

Scale dataset

```
In [20]: Q2Xt_scaled = preprocessing.scale(Q2Xt)
```

```
In [21]: Q2Xt_scaled = pd.DataFrame(Q2Xt_scaled, columns=Q2Xt.columns)
```

PCA

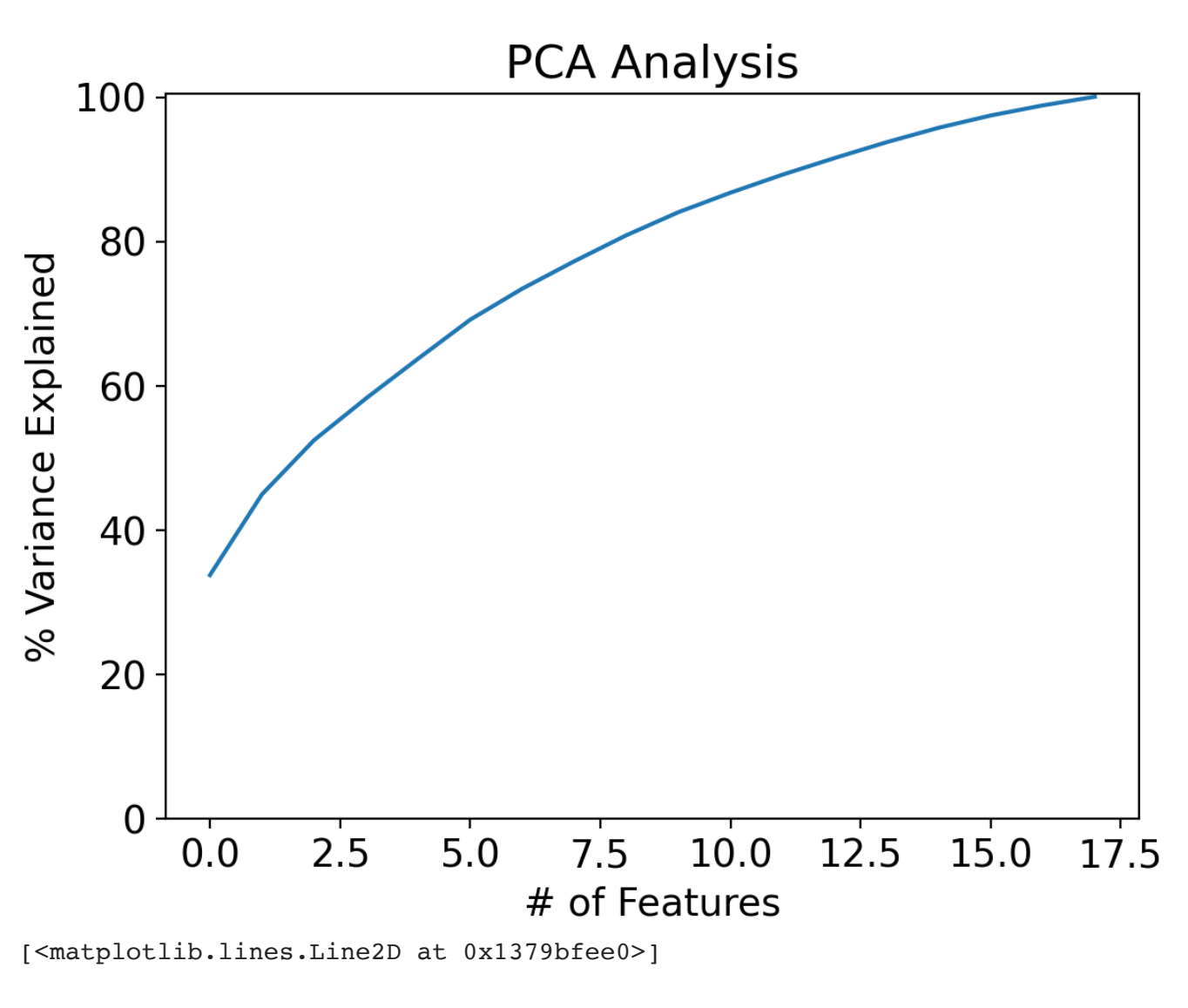
```
In [22]: from sklearn import decomposition
```

```
In [23]: pca = decomposition.PCA(n_components=18)
         pca.fit(Q2Xt_scaled)
```

```
Out[23]: ▾         PCA
         PCA(n_components=18)
```

```
In [24]: variance = pca.explained_variance_ratio_
         var = np.cumsum(np.round(variance, 3)*100)
```

```
In [25]: plt.figure()
         plt.ylabel('% Variance Explained')
         plt.xlabel('# of Features')
         plt.title('PCA Analysis')
         plt.ylim(0,100.5)


         plt.plot(var)
```



```
Out[25]: [<matplotlib.lines.Line2D at 0x1379bfee0>]
```

```
In [26]: var
```

```
Out[26]: array([ 33.8,  45. ,  52.5,  58.3,  63.8,  69.2,  73.5,  77.3,  80.9,
                 84.1,  86.8,  89.3,  91.6,  93.8,  95.8,  97.5,  98.9, 100.1])
```

A little over 50 percent of the variance was explained from PC1,PC2, and PC3. Will use 3 PCs for the model.

```
In [27]: pca = decomposition.PCA(n_components=3)
         pca.fit(Q2Xt_scaled)
         pca_3 = pca.transform(Q2Xt_scaled)

         pca_df = pd.DataFrame(pca_3, columns=['pc1', 'pc2', 'pc3'])
         print(pca.explained_variance_ratio_)

         [0.33834694 0.11186552 0.07505233]
```

```
In [28]: pca_df
```

Out[28]:

	pc1	pc2	pc3
0	3.821341	-0.263557	1.301352
1	-2.298688	0.211997	-1.076478
2	1.897460	-0.274380	-0.115207
3	-2.466214	-1.439636	0.322325
4	-0.270008	0.016467	0.591359
...	...	...	...
2211	2.680224	1.061813	1.816778
2212	-1.509791	3.436739	0.045068
2213	1.288094	-0.730501	0.152661
2214	2.074264	1.206376	-1.089310
2215	-1.864647	1.451817	-1.026216

2216 rows × 3 columns

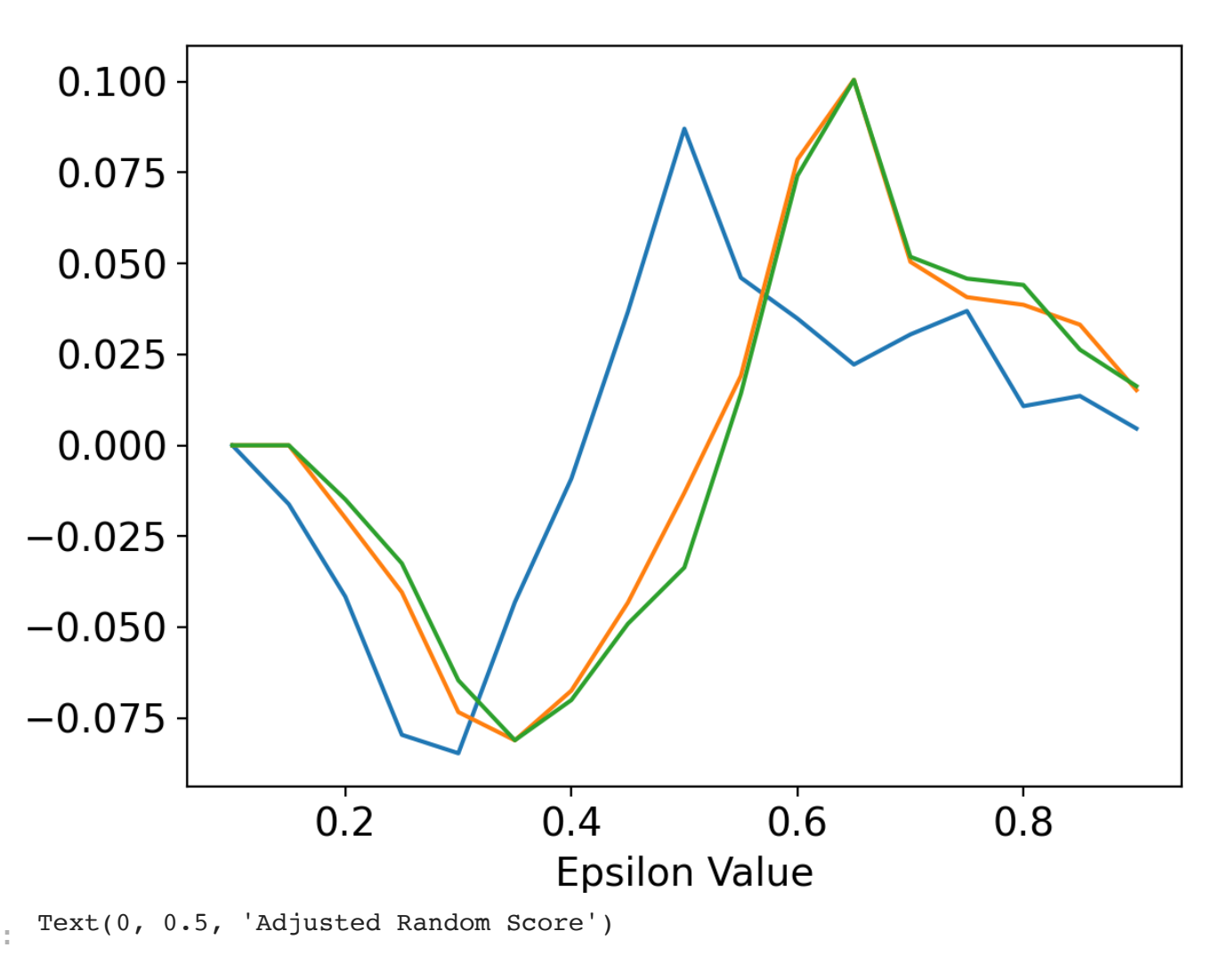
Feeding PCs into DB Scan loop to find ideal epsilon number.

I chose the Adjusted Rand Index as the metric to use for my metric evaluation plot, below.

The true clusters I used were from the AcceptedCmp5 variable. 1 indicates it belongs to this cluster and 0 indicates it does not.

```
In [29]: min_samples = np.arange(1, 21)
         epsilons = np.arange(0.1,0.95,0.05)

         all_scores = []
         all_eps_values = []
         for min_sample in min_samples:
             scores = []
             eps_values = []
             for epsilon in epsilons:
                 dbscanQ2 = DBSCAN(eps=epsilon, min_samples=min_sample).fit(pca_df)
                 score = metrics.adjusted_rand_score(Q2classes['AcceptedCmp5'], dbscanQ2.labels_)
                 epsvalue = epsilon
                 scores.append(score)
                 eps_values.append(epsvalue)

             all_scores.append(scores)
             all_eps_values.append(eps_values)
```

```
In [31]: plt.figure()
         plt.xlabel('Epsilon Value')
         plt.ylabel('Adjusted Random Score')
         #plt.plot(all_eps_values[0], all_scores[0])
         #plt.plot(all_eps_values[1], all_scores[1])
         #plt.plot(all_eps_values[3], all_scores[3])
         #plt.plot(all_eps_values[4], all_scores[4])
         #plt.plot(all_eps_values[5], all_scores[5])
         #plt.plot(all_eps_values[6], all_scores[6])
         #plt.plot(all_eps_values[7], all_scores[7])
         #plt.plot(all_eps_values[8], all_scores[8])
         plt.plot(all_eps_values[9], all_scores[9])
         plt.plot(all_eps_values[18], all_scores[18])
         #plt.plot(all_eps_values[17], all_scores[17])
         #plt.plot(all_eps_values[16], all_scores[16])
         plt.plot(all_eps_values[19], all_scores[19])
```



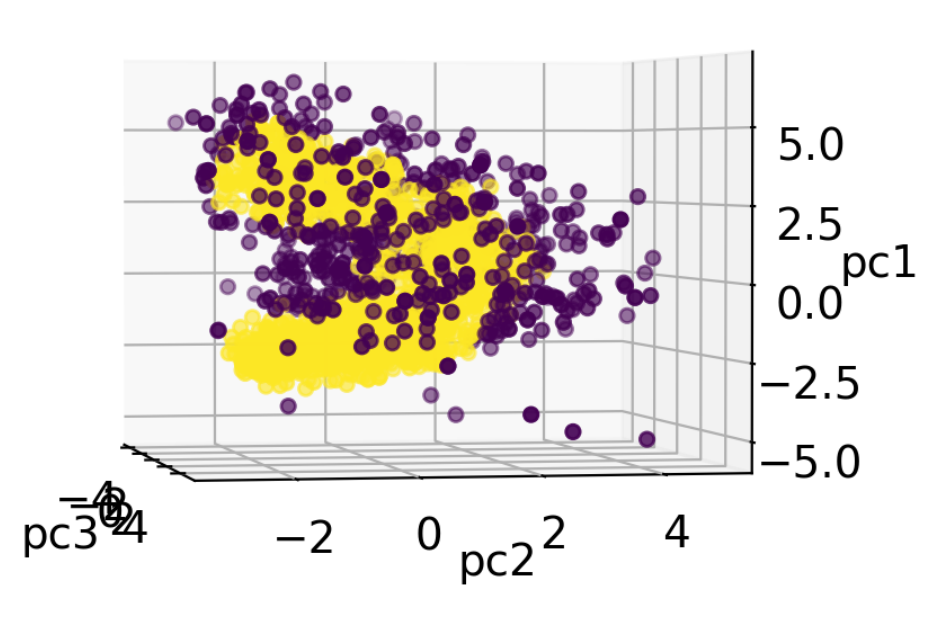
```
Out[31]: Text(0, 0.5, 'Adjusted Random Score')
```

The plot above shows that the ideal epsilon value is is arond 0.65 (since the adjusted random index (ARI)) was closest to 1. The min sample value was set to around 18 for the highest ARIs as well.

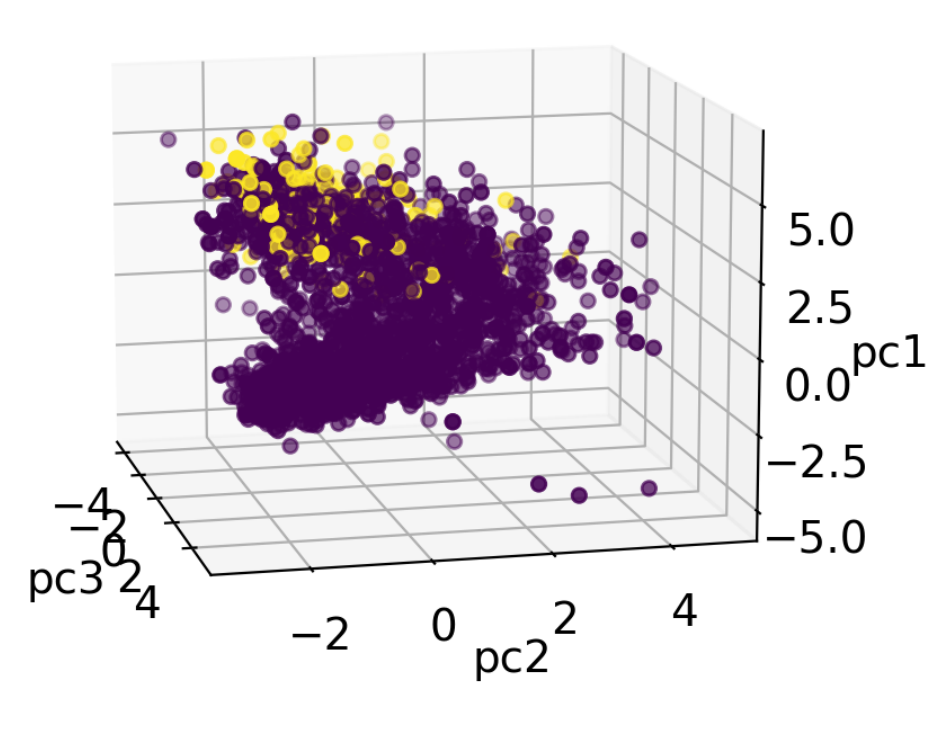
Below, I use those values for a final model

```
In [33]: from mpl_toolkits.mplot3d import Axes3D

         dbscanQ2 = DBSCAN(eps=0.65, min_samples=18)
         pca_df['cluster'] = dbscanQ2.fit_predict(pca_df)
```

```
In [34]: fig = plt.figure()
         plt.clf()
         ax = fig.add_subplot(111, projection="3d")

         ax.scatter(pca_df['pc3'],pca_df['pc2'],pca_df['pc1'], c=pca_df['cluster'])
         ax.set_xlabel('pc3')
         ax.set_ylabel('pc2')
         ax.set_zlabel('pc1')
         plt.show()
```



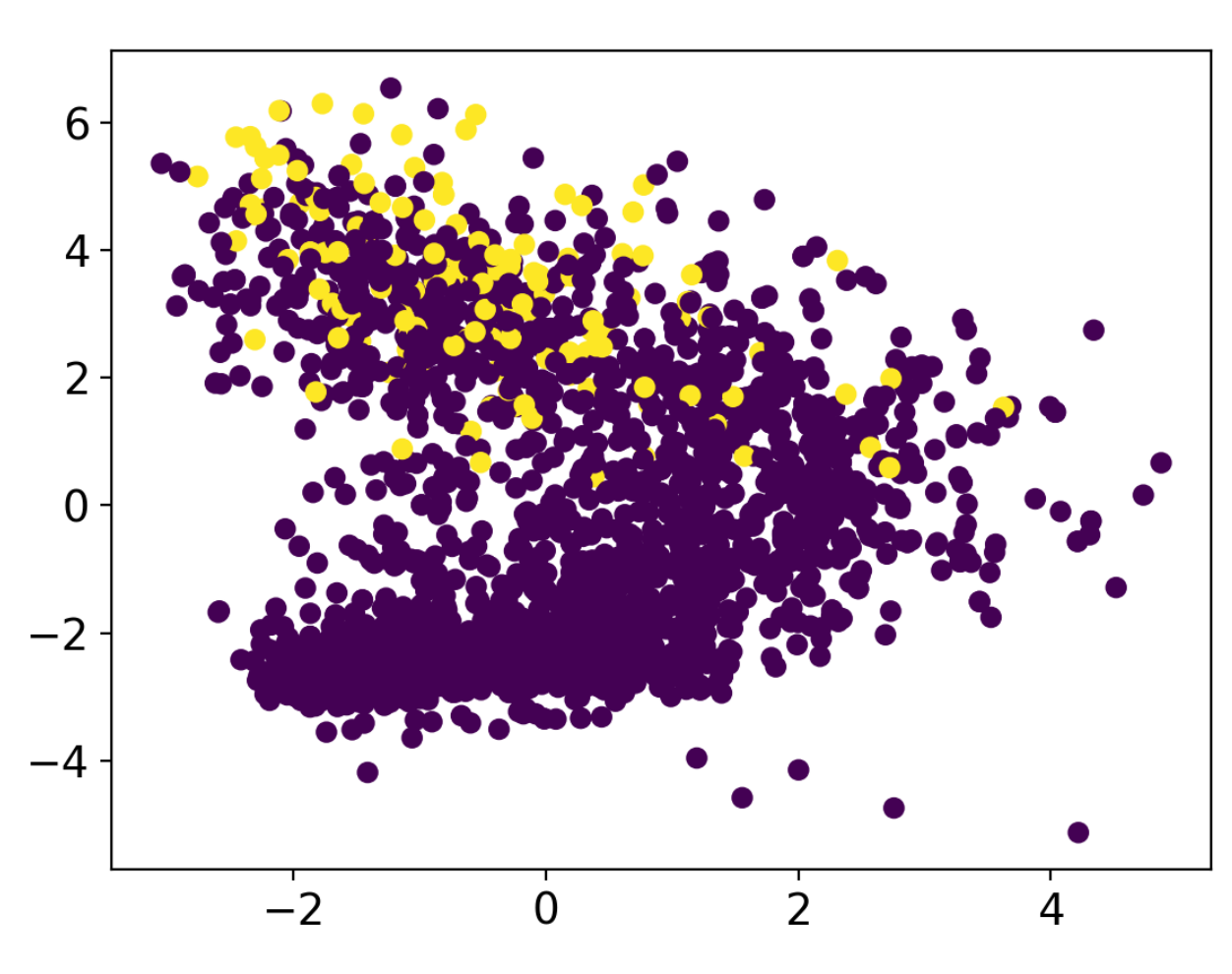
```
In [35]: fig = plt.figure()
         plt.clf()
         ax = fig.add_subplot(111, projection="3d")

         ax.scatter(pca_df['pc3'],pca_df['pc2'],pca_df['pc1'], c=Q2classes['AcceptedCmp5'])
         ax.set_xlabel('pc3')
         ax.set_ylabel('pc2')
         ax.set_zlabel('pc1')
         plt.show()
```



The first 3d plot (fig5) shows the predicted clusters, while the second 3d plot (fig6) shows the true clusters.

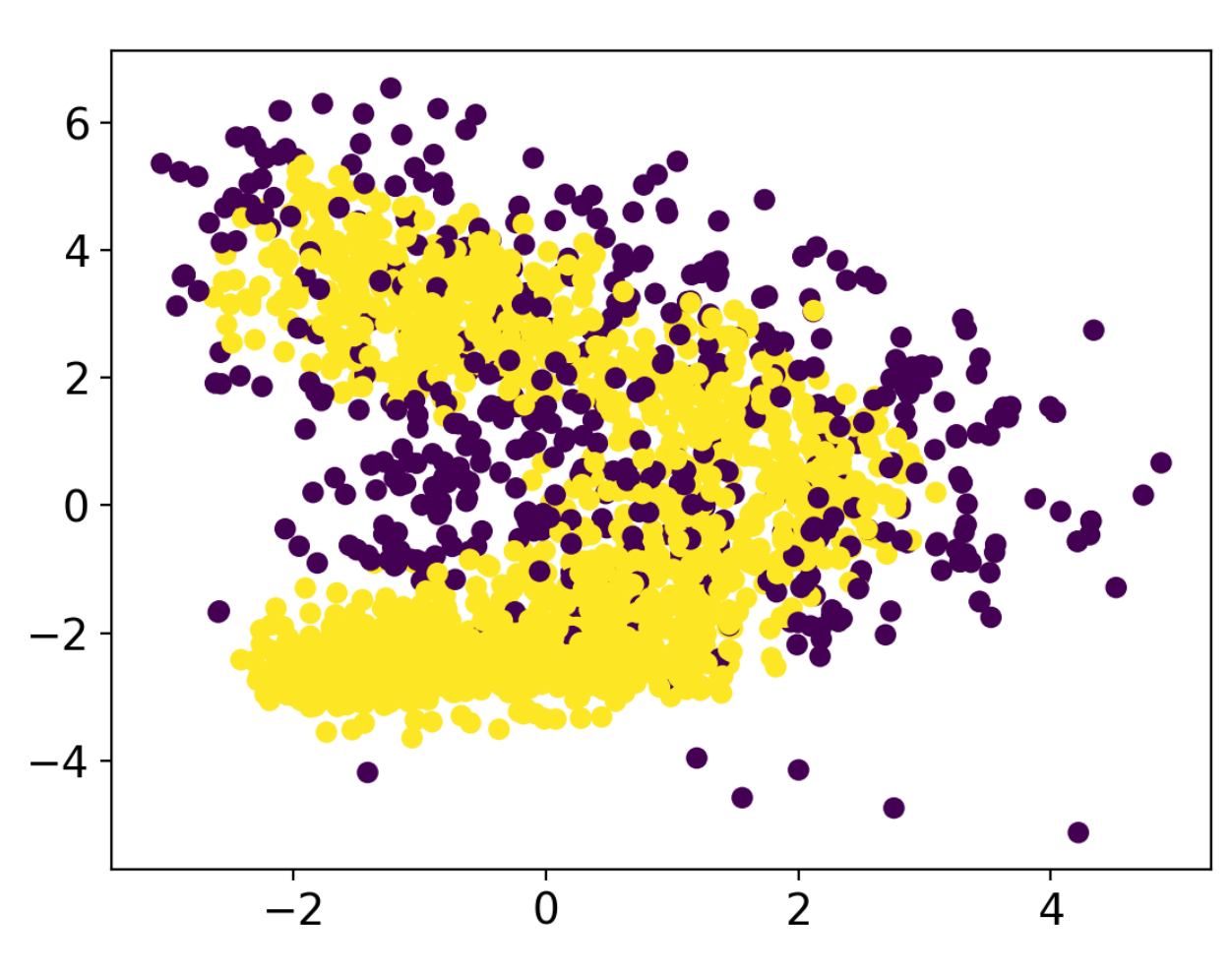
The 3d plots were a bit messy, but it looks like it might be better to reduce it by 1 dimension and just check out PC1 and 2, since it looks like they may tell more of the story

```
In [36]: plt.figure()
         plt.scatter(pca_df['pc2'],pca_df['pc1'], c=Q2classes['AcceptedCmp5'])
```



```
Out[36]: <matplotlib.collections.PathCollection at 0x143b0e680>
```

```
In [37]: plt.figure()
         plt.scatter(pca_df['pc2'],pca_df['pc1'], c=pca_df['cluster'])
```



```
Out[37]: <matplotlib.collections.PathCollection at 0x143b53f70>
```

When viewed with just two of the PCs, the predicted clustering (the second 2d plot, fig8) and the true clustering (the first 2d plot, fig7) can be more easily discerned

```
In [ ]:
```