kNN and K-Means on Font Recognition Data Set

This report builds off our last report on applying Principal Component Analysis and automatic classification algorithm kNN on the data. The dataset includes digitized images of characters that belong to a specific font and the features tell you the grey level intensity of each pixel. Today we are going to do a little bit of feature engineering to adjust the number of features in the dataset and seeing how this might impact our kNN classification algorithm. For reference on the steps in the preliminary treatment, standardization, and PCA on the dataset refer to the last report.

Question 1:

Figure 1 shows a graph of the eigenvalues that were calculated with the data. The larger the eigenvalue the more variance the associated vector accounts for. This means, if you divide the eigenvalue by the sum of the eigenvalues, you will get a ratio telling you how much of the original data the associated eigen vector accounts for.

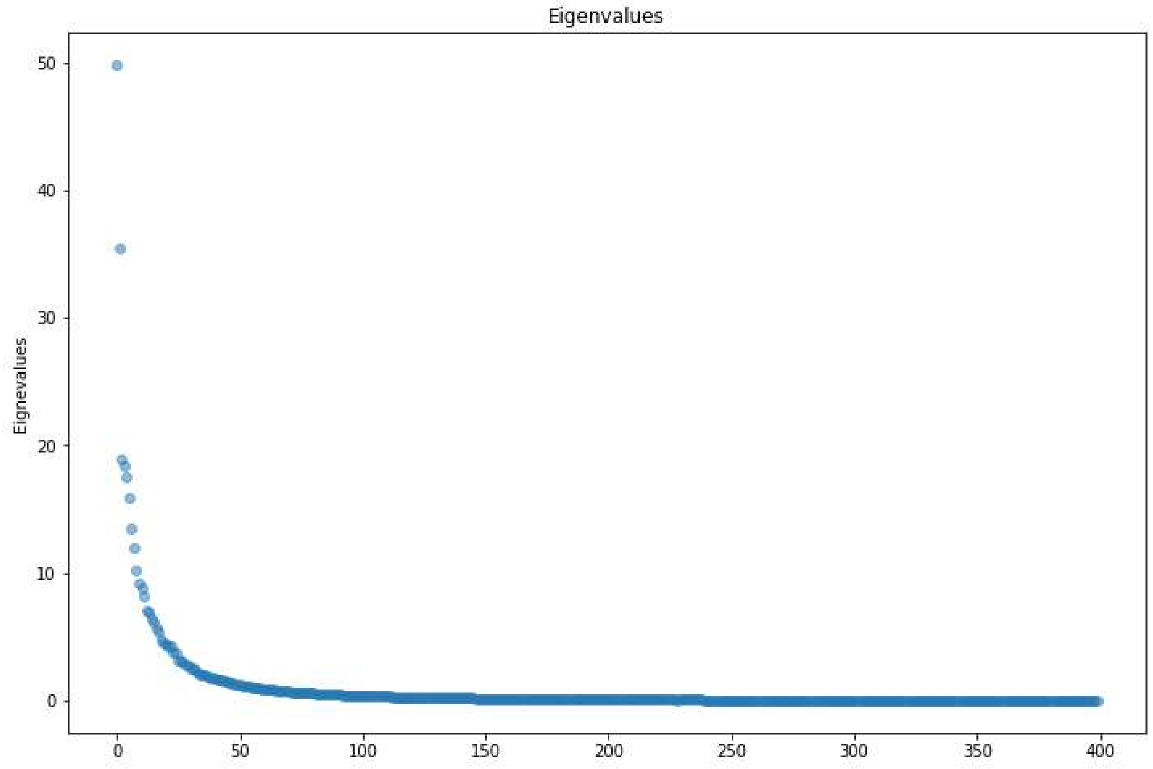

Figure 1: Eigenvalues

$$R_i = \frac{\lambda_i}{\sum_{i=1}^{N} \lambda_i}\,, \qquad \lambda_i = Eigenvalue, \qquad N = total\ Featurs$$

Equation 2: Cumulative Ratio

By doing a cumulative ratio, described in equation 1, you can select the number of eigenvalues you want to use that reach the percent of variance accountability desired. In this case we want to create new features for the standardized data that account for more than 35% and 60% of the original data. If you refer to Figure 2, you can see that the cumulative ratio has been plotted along with a line displaying the 35% and 60% variance accountability lines and the respective eigenvalues index, the points in green.

Table 2 below shows the eigenvalue index, eigenvalue, and ratio. Note: if using Python you have to remember your index starts at 0 not 1.

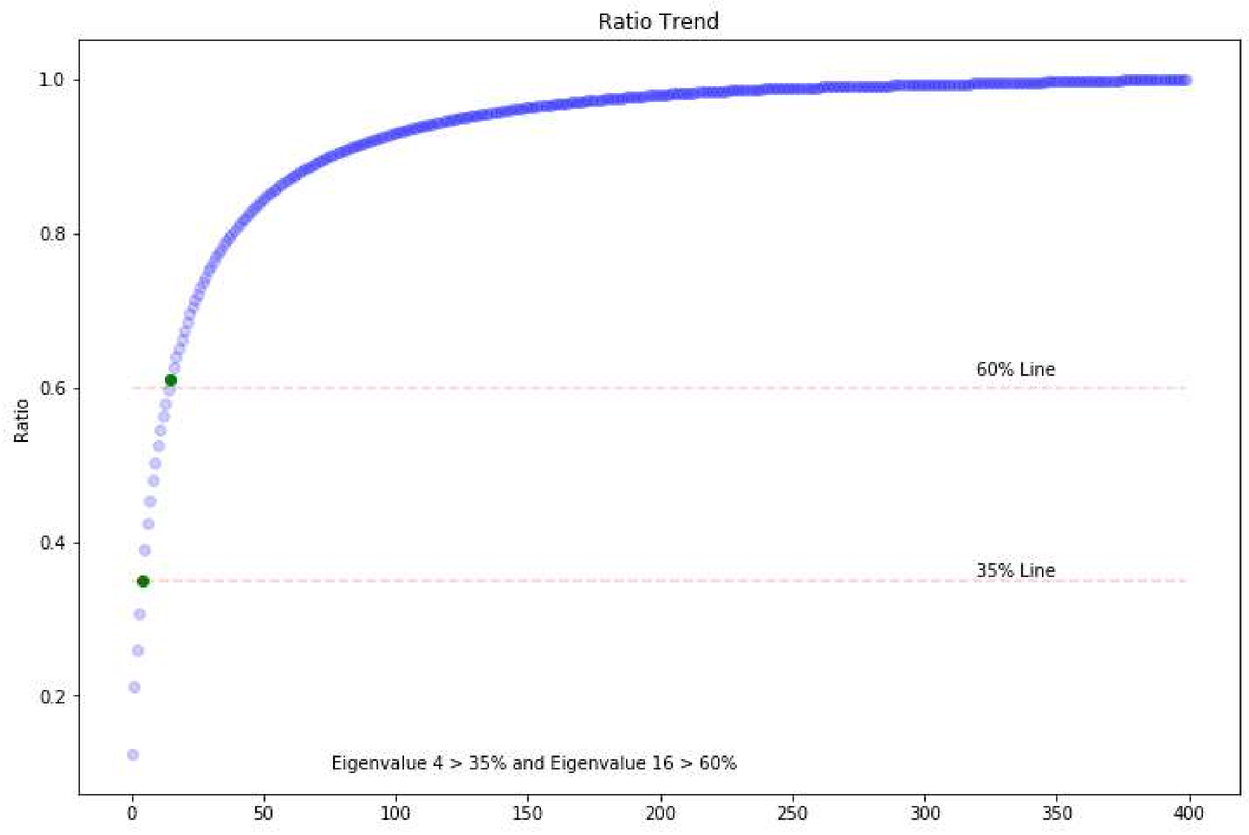

Figure 2: Cumulative Ratio Plot

Reference Name	Index	Eigenvalue	Ratio (%)
a	5	17.58	35.02%
b	16	6.16	61.12%

Table 2: Ratio a and b

Notice the reference name, we will be referring back to these when we start splitting data for the kNN algorithm and predictions.

After this we projected all of the original standardized data onto the eigenvectors to get a score for each dimension. This way when we start playing with different amounts of features we want to use, we can just pull that slice out of the data, split for test and train, and then do our analysis or run through the classification algorithm in a lesser dimension. The number of features directly determines the number of dimensions.

An area to take consideration of before moving onto the next step is splitting your data into test and train data. There is built in code through Python that will do this for. You want to make sure that if you are doing a 80% split for training data, that you use 80% of each category to equal 80% of all the data. You don't want to randomly select 80% of the values out of all the data and end up with 70% of CL1, 84% of CL2, and 86% of CL3. The way that we went about doing this was by separating the data into the original classes. Then shuffling each class and taking 80% for the train, leaving 20% of each for class for test. Then we recombined all of the classes to make up the train and test data.

Question 2:

Using the eigenvectors, given with calculation of eigenvalues, and our standardized data we are able to get a score for each observation in each dimension by using the following equation. In using this formula, you need to make sure that your eigenvectors are transposed. By doing the dot product of each observation with eigenvector (X_{λ_i}), you get a single value, called a score, that accounts for a ratio, R_i, of the variance in the data.

$$score_{i\lambda} = < E_i, X_{\lambda_i} >$$

$$<> = dot\ product, \qquad E_i = Standardized\ observation\ i, \qquad X_{\lambda_i} = Eigenvector\ for\ \lambda_i$$

Equation 2: Score

We will start by doing kNN with the first 5 eigenvectors, X_{λ_a}, that account for $R_a > 35\%$. By multiplying all of the observations by the first 5 eigenvectors, we get 5 scores for each observation reducing the dimension from 400 to 5.

Now that we have the standardized the data, split the data in to test/train groups, and projected the data onto the dimensions we want to account for, we can move onto our kNN algorithm and predictions. This works by comparing each test value to all of the training data and keeping the 5 closest neighbors, k. The closest neighbor is determined by length of the line segment between two points in the Euclidian 5-space, the 5 is the dimension of the space in our case, which is done by the Pythagorean formula shown in following equation.

$$d(\boldsymbol{q}, \boldsymbol{p}) = (\Sigma_{i=1}^{a}(q_i - p_i)^2)^{\frac{1}{2}}$$

Equation 3: distance in Euclidian space

Once the five closest neighbors are collected from the training data, you classify that test observation as the largest collection of classifications in the 5 neighbors. Then you move on and do this for the rest of the test observations. Once you have made it through all of the test data you have a collection of predictions that your algorithm has given and collection of actual classifications that you have from the test data. You now compare your predictions with the actual classifications and create a confusion matrix.

Using the training and test data of dimension 5 we got the following percent confusion matrix. An important aspect of this table is the percent correct prediction shown in $row_i\ vs.\ column_i\ , for\ i =$ 1,2,3. The mean of these three values gives you the overall percent correct for the predictions made from the test and train data with 5 features, or in dimension 5. The total percent correct is 79% in this case. The confusion matrix grows with the number of classifications, if we had 5 groups it would be a 5x5 square matrix.

N=2768
k=5
	Predicted Courier	**Predicted Calibri**	**Predicted Times**

Actual Courier (853)	80%	12%	8%
Actual Calibri (954)	11%	81%	8%
Actual Times (961)	12%	12%	76%

Table 3: Percent Confusion Matrix for dimension 5

In a previous report we did the kNN algorithm to predict off of the same test and train data and got the following percent confusion matrix, with a total percent correct prediction of 81%.

N=2768
k=5
	Predicted Courier	**Predicted Calibri**	**Predicted Times**

Actual Courier (853)	76%	12%	7%
Actual Calibri (954)	5%	88%	12%
Actual Times (961)	10%	10%	80%

Table 4: Percent Confusion Matrix for dimension 400

The thing to note here is that this prediction used all 400 of the features. Our predictions using the eigenvectors had a percent correct of about 2 percent less than the predictions from our previous report, but with 395 less features, or dimensions. That is really powerful when you are using a data set that is extremely large because of the amount of time it takes to run the kNN algorithm. Our test data, with a size of 2767, took 438 seconds to run the kNN algorithm on the 400 features from the previous report. It took less than a second to run the kNN algorithm on the test data with 5 features. That is a testament to the power of using PCA to reduce dimension and increase efficiency if you don't mind sacrificing some understanding between features and results and you are working with a lot of data.

Below are the confusion matrices on the test and train data with dimension 5. These read similar to the percent confusion matrices except they display the counts in the cells instead of percentages. The diagonal sums to the total correct predictions and each row sum up to the number of observations in that classification.

N=2768
k=5
	Predicted Courier	**Predicted Calibri**	**Predicted Times**

Actual Courier (853)	595	159	99
Actual Calibri (954)	133	685	136
Actual Times (961)	146	162	653

Table 5: Confusion Matrix for dimension 5 on Test Data

N=11067
k=5
	Predicted Courier	**Predicted Calibri**	**Predicted Times**

Actual Courier (3409)	2747	400	262
Actual Calibri (3814)	380	3138	296
Actual Times (3844)	426	459	2959

Table 6: Confusion Matrix for dimension 5 on Train Data

Question 3:

Let's use eigenvectors a through b, $X_{\lambda_{b-a}}$, to account for 26% of the data, $61\% > R_{b-a} > 35\%$ for our kNN algorithm input. Using eigenvectors $X_{\lambda_{b-a}}$ reduces our dimension from 400 to 12, which is more than X_{λ_a}, but the 12 features used account for less of the data than the 5 features used earlier. Below is the following percent confusion matrix for this prediction on the test data, giving us an overall percent correct for the kNN algorithm using $X_{\lambda_{b-a}}$ of about 79%.

N=2768
k=5
	Predicted Courier	**Predicted Calibri**	**Predicted Times**

Actual Courier (853)	79%	11%	10%
Actual Calibri (954)	11%	82%	7%
Actual Times (961)	14%	11%	75%

Table 7: Percent Confusion Matrix for dimension 12

That is interesting to note. The percent correct with 5 features, 35% of data accountability, and the percent correct with 12 features, 26% of data accountability, are roughly the same. This tells us that the number of dimensions used plays a pretty significant role in the kNN algorithm. The key is to get a balance between the amount of data that is accounted for, number of features, or dimensions, to use, and the amount of time it takes to run the algorithm. Below is a figure showing the accuracy of all 400 features, 5 (a) features, 12 (b-a) features, and 16 (b) features.

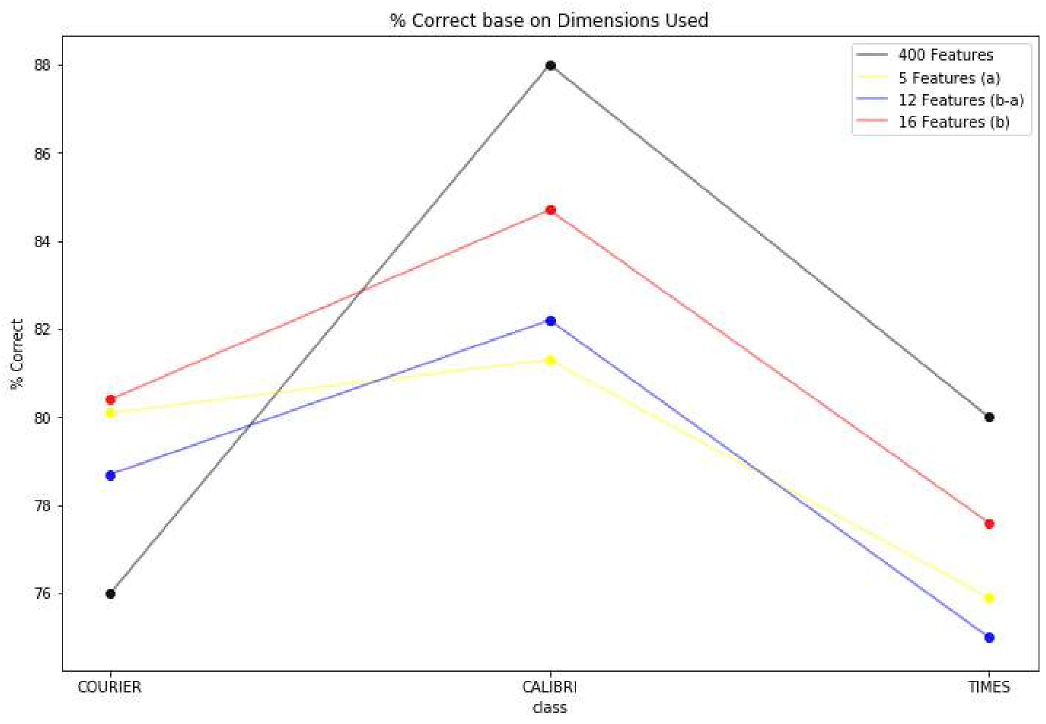

Figure 3: Percent correct based on dimensions used in kNN Algorithm

This graph shows the three different kNN algorithms ran versus the kNN algorithm ran on all 400 features. It looks like Calibri consistently gets a higher accuracy, making me think that the cluster for Calibri is fairly tight. Whereas the cluster for Times seems to be more spread out between the other two classifications. The amount of time it took to run each one is displayed below.

400 Features	5 Features (a)	12 Features (b-a)	16 Features (b)
≅438 Seconds	≅ .85 Seconds	≅ .84 Seconds	≅ .92 Seconds

Table 8: Time it took to run kNN Algorithm on number of Features

Question 4:

Another algorithm that can be used to try and classify this data is the kMeans algorithm. This algorithm is unsupervised, which just means that it assumes you do not have any results from the observations, you just have observations and want to create clusters based off of that data. The amount of cluster to be used is up to the person using the algorithm, in our case we want to use 3 because we have 3 classifications (Courier, Calibri, Times). We will run the kMeans algorithm on the training data in 5-dimensional space.

The kMean algorithm works by following these steps:

1) Select random centroids for the initialization of kMean's algorithm. You will have as many centroids as you have clusters, our case 3. Note that the dimension of the centroid has to be the dimension of your observations.
2) Calculating the distance between every observation in the data set and the current centroids.
3) Clustering every observation with the closest centroid.
4) Calculating a new centroid and the cost associated with each centroid.
5) Repeating steps 2 – 4 until the centroids converge, slow their adjustments

The distance between every observation is used by the Pythagorean formula, Equation 3. Once you know the distance from all of the cluster's centroids, you put each observation in the new cluster. Since the centroids are constantly shifting, an observation may be in one cluster one iteration, and in another cluster the next iteration. That is all part of the clusters finding their boundaries. Once all of the observations are re-clustered you calculate a new centroid for each cluster by taking the mean of each feature in the cluster, refer to following equation. In our case we have 5 means to calculate for each of the 3 cluster because each cluster is in 5-dimensional space.

$$\mu_f = \frac{\Sigma_{i=1}^{N} X_i}{N}$$
$$\mu_f = mean\ for\ a\ feature\ in\ a\ cluster$$
$$N = total\ number\ of\ observations\ for\ relative\ cluster$$

Equation 4: Mean

Once the observations are in the new clusters you calculate the dispersion, or cost, of each cluster, summing up to a total cost value. The goal with each iteration is reduce this cost value, but after so many iterations you will see this cost start to level out, signifying that the centroids are not really shifting any more. Refer to the following equation for calculating the cost. The smaller the cost value, the better because it is telling us that our cluster is tight and well grouped.

$$COST_{cluster} = \Sigma_{i=1}^{N} \left|\left|\boldsymbol{X_N} - \boldsymbol{Centroid_{cluster}}\right|\right|^2$$
$$X_N = Observation\ N \quad ; \quad Centroid_{cluster} = Centroid\ for\ cluster$$

$$COST_{total} = \Sigma_{i=1}^{k} COST_{cluster}$$
$$k = Number\ of\ clusters$$

Equation 5: Cost

If you refer to the figure 4, you can see how the $COST_{total}$ drops until it levels out. Now we have what we feel as the best set of centroids, because of our low terminal cost, for that initial centroid which we filled with random numbers. If we changed those random numbers, we would get another terminal cost. In order to make sure we get the best cost overall; we need to run these iterations for however many sets of random centroids we choose to use. We chose a set of random centroids by looking at the max and min values of the training data set for my bounds, which came out to between -16 and 16. Then we told Python to choose 5 random values within those bounds and set those values as our initial centroid. We chose random numbers for 10 different centroids that led to 10 different terminal costs, displayed in the following table.

Centroid 1	Centroid 2	Centroid 3	Centroid 4	Centroid 5	Centroid 6	Centroid 7	Centroid 8	Centroid 9	Centroid 10
1023631	1023594	1028495	1029013	1024044	1028327	1028493	1023589	1027950	1024925

Table 9: The cost for each randomly generated original centroid

Based off of this table, we can see that random centroid number 8 lead to the lowest terminal cost of 1,023,589. The centroids for each cluster relative to that terminal cost it displayed in the following table.

	Feature 1	Feature 2	Feature 3	Feature 4	Feature 5
Cluster 1 Centroid	9.86	-3.73	-1.17	1.08	-0.66
Cluster 2 Centroid	-4.24	-2.09	0.53	-0.14	-0.43
Cluster 3 Centroid	1.25	6.57	-0.16	-0.40	1.22

Table 10: Best Centroid for each cluster

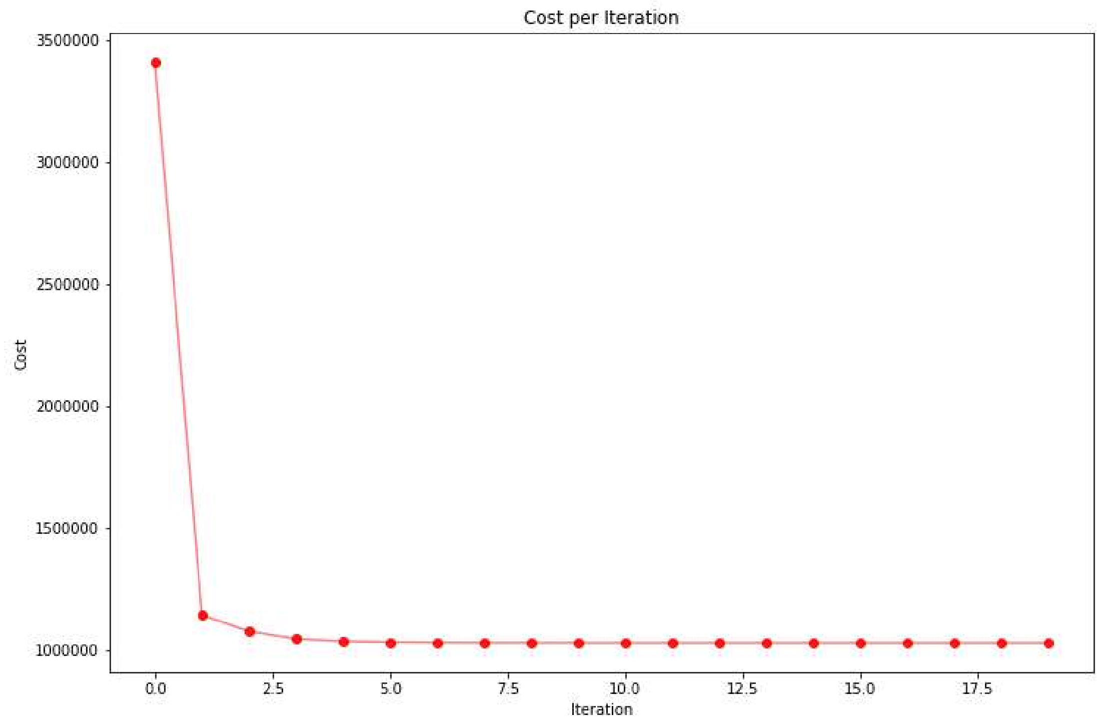

Figure 4: Cost per iteration in the reducing dispersion process

Question 5:

Since we did this on the training data, we have the observations along with their respective results. The first thing we can do is calculate the centroids of each actual cluster by splitting the data up into its original classifications and then computing the centroid for each class. Once we do this, we can look at the cost for each class, referred to CL, and compare that to the cost of each kMean cluster, referred to as H. Refer to the following table cost and the table after that with sizes.

	Cluster 1	Cluster 2	Cluster 3	Total
H1 Cost	395,882	304,689	326,449	1,027,019
CL Cost	396,186	554,844	548,482	1,499,512

Table 10: The cost for each cluster in Cl and H

	Cluster 1	Cluster 2	Cluster 3	Total
H1 Size	5,555	2,099	3,413	11,067
CL Size	3,409	3,814	3,844	11,067

Table 11: The count of observations for each cluster in Cl and H

Notice how many observations fall in the kMean Cluster 1 category, almost 50%. That makes me think that observations are dense around the centroid for H1, no matter what class they are actually in. If you look at the following matrix you will see what looks like a confusion matrix. This one is described well by the following formula.

$$\frac{size(H_i \cap CL_j)}{size(CL_j)}$$

$$where\ i = column\ and\ j = row$$

Equation 6

For instance, cell one displays how many (H_1 and CL_1) fall into CL_1. Each row sums up to 100 of that CL.

	H1	H2	H3
CL 1	67%	15%	18%
CL 2	41%	14%	45%
CL 3	49%	23%	28%

Table 11: Percent Confusion between H and CL

The next matrix shows us how many (H_1 and CL_1) fall into H_1, like displayed in the following formula.

$$\frac{size(H_i \cap CL_j)}{size(H_i)}$$

$$where\ i = column\ and\ j = row$$

Equation 7

	CL 1	CL 2	CL 3
H1	39%	27%	34%
H2	27%	28%	45%
H3	20%	50%	30%

Table 12: Percent Confusion between CL and H

It seems that these two matrices can give us some insight into the separation between the actual clusters. The larger the diagonal values for these, the denser and more pairwise disjoint we would suspect the clusters to be. If all three clusters were perfect you would expect to see the following two instances:

$$\frac{size(H_i \cap CL_j)}{size(H_i)} = \frac{size(H_i \cap CL_j)}{size(CL_i)} = 0\ \ for\ i \neq j$$

$$\frac{size(H_i \cap CL_j)}{size(H_i)} = \frac{size(H_i \cap CL_j)}{size(CL_i)} = 1\ \ for\ i = j$$

This is because if they were perfect then the ideal cluster, CL, and the kMean cluster, H, would be an identical match and there would be no overlap between other clusters because they would all be pairwise disjoint, or $H_i \cap CL_j = 0\ for\ i \neq j$.

Time:

Below is the following table that displays the time it took for each question in the journey. Notice how the kMean task took a little while because I did 20 iterations to minimize total cost with 10 different random centroids totaling to 200 iterations.

Time in Seconds						
Start	Preliminary	Question 1	Question 2	Question 3	Question 4	Question 5
0	10.33	10.54	11.63	13.28	39.21	39.29

Table 13: Time table for each task

```
@author: Raul Paz/Dustin/Stephen
"""

import pandas as pd
import numpy as np
from matplotlib import pyplot as plt
from mpl_toolkits import mplot3d
from sklearn.model_selection import train_test_split
from sklearn.neighbors import NeighborhoodComponentsAnalysis as NCA
from sklearn.neighbors import KNeighborsClassifier as KNC
from sklearn.metrics import classification_report, confusion_matrix, accuracy_score
import time
import copy

times = [time.time()]
fsz=(12,8)

def get_data(file):
    """Create a function for loading the csv and editing the file the way we want
    for this excel format
    """

    import pandas as pd

    #Import csv as a Data Frame, drop columns do now want, drop rows with value na
    data = pd.read_csv(file)
    data = data.drop(['fontVariant', 'm_label',  'orientation', 'm_top', 'm_left', 'originalH', 'originalW', 'h',
'w' ], axis = 1)
    data.dropna()
    print(data.shape)

    data = data.loc[(data.strength ==.4) & (data.italic == 0)]

    return data

#load the files and assign to class
file = r'C:\Users\Dustin\Desktop\Masters Program\Fall Semester\aaaaStatistical Learning and Data
mining\Homework and Readings\Homework\HW2\CALIBRI.csv'
cl2 = get_data(file)
cl2['font'] = 2

file = r'C:\Users\Dustin\Desktop\Masters Program\Fall Semester\aaaaStatistical Learning and Data
mining\Homework and Readings\Homework\HW2\COURIER.csv'
```

```
cl1 = get_data(file)
cl1['font'] = 1

file = r'C:\Users\Dustin\Desktop\Masters Program\Fall Semester\aaaaStatistical Learning and Data
mining\Homework and Readings\Homework\HW2\TIMES.csv'
cl3 = get_data(file)
cl3['font'] = 3

fonts = {1:'COURIER', 2:'CALIBRI', 3:'TIMES'}

del file

#Print the sizes
n_cl1 = cl1.shape; print( n_cl1)
n_cl2 = cl2.shape; print( n_cl2)
n_cl3 = cl3.shape; print( n_cl3)
v = 400

#combine the three classes for full set of Data
data=pd.concat([cl1,cl2,cl3],axis=0)
data.index = range(len(data))
n_data = data.shape; print(n_data)
if n_data[0] == (n_cl1[0]+n_cl2[0]+n_cl3[0]): #check
    print('\nLooks good')

## mean and standard deviation
m=data[data.columns[3:]].mean() #Mean
sd=data[data.columns[3:]].std() #Standard Deviation

## Centralize and stadardize the matrix
data_y = data[data.columns[0]] #Get your y's
data_s = (data[data.columns[3:]]-m)/sd #Centralizing and Standardizing the Data
#print(data_s.mean(), data_s.std())   #Check the values , m=0 sd=1
""" just for visualization of centralized data
#plot of centralized and standardized data
fig1, ax1 = plt.subplots(figsize=fsz)
ax1.set_title('Samples of Shaped Data')
ax1.boxplot([data_s[data_s.columns[3]],data_s[data_s.columns[4]],data_s[data_s.columns[5]]],
meanline = True, showmeans=True, labels = ['r0c0','r0c1','r0c2'])
"""

## correlation matrix
corr_m = data_s.corr()
eigs = np.linalg.eig(corr_m)
```

```
## eigen values
eig_value = eigs[0]
eig_vecto = eigs[1]
print(sum(eig_vecto[:,0]**2))
print('\n Sum of eig: ', sum(eig_value))

## Plots
#plot eig_values
fig2, ax2 = plt.subplots(figsize=fsz)
ax2.set_title('Eigenvalues')
ax2.set_ylabel('Eignevalues')
ax2.scatter(range(len(eig_value)), eig_value, alpha=0.5)

"""
Question 1
"""
print('\nQuestion 1\n')

times += [time.time()]

## Rj
ratio = np.cumsum(eig_value)/sum(eig_value)
# Where does Rj>.35 and Rj>60%
a_idx = np.where(ratio>.35)[0][0] #gets the index
b_idx = np.where(ratio>.60)[0][0] #gets the index
a = ratio[a_idx] #gets the value at index
b = ratio[b_idx] #gets the value at index
print('At eigenvalue', (a_idx+1), format(eig_value[a_idx], '.2f'), 'we get a ratio of', format(a, '.2%'),
    'and at', (b_idx+1), format(eig_value[b_idx], '.2f'), 'we get a ratio of', format(b, '.2%'))

#Plot Rj
fig3, ax3 = plt.subplots(figsize=fsz)
ax3.set_title('Ratio Trend')
ax3.set_ylabel('Ratio')
ax3.scatter(range(len(ratio)), ratio, c='b', alpha=.2)
ax3.scatter(a_idx, a, c='green', alpha=1)
ax3.scatter(b_idx, b, c='green', alpha=1)
ax3.plot(range(len(ratio)), [.35]*400, 'r--', alpha=.2)
ax3.plot(range(len(ratio)), [.60]*400, 'r--', alpha=.2)
ax3.text(350,a,'35% Line', horizontalalignment = 'right', verticalalignment = 'bottom')
ax3.text(350,b,'60% Line', horizontalalignment = 'right', verticalalignment = 'bottom')
ax3.text(a_idx,.1,'Eigenvalue 4 > 35% and Eigenvalue 16 > 60%', verticalalignment = 'bottom')

#split the data into test/train
def train_test_split(x,y,train, classification):
```

```
"""
Inputs: All of your features as x, all of your results as y, your train %,
        a list of your classifications
Outputs:train_data, test_data
This function will take your data set and shuffle it, then split it by
classification evenly to make up your train and test data set
"""
import time
start = time.time()
from sklearn.utils import shuffle

y = pd.DataFrame(y)
x = pd.DataFrame(x)
xy = pd.concat([y,x], axis=1)

#seperate the groups (class 1,2,...)
cl = list()
for i in range(len(classification)):
    group = xy['font'].where(xy['font'] == classification[i])
    cl += [group.dropna()]
#shuffle the groups (seperately)
for i in range(len(cl)):
    cl[i] = shuffle(cl[i])
    cl[i] = cl[i].reset_index()
#split each group into test/train
test_cl = []
train_cl = []
check = []
for i in range(len(cl)):
    num = int(train*len(cl[i])) #needed to make into integer
    train_cl += [cl[i][0:num]]
    test_cl += [cl[i][num:]]
    check += [round(len(train_cl[i])/len(cl[i]),3)]
    print('train', classification[i], 'has a size of',
          format(check[i], '.2%'), 'compared to data.')

#combine back to complete data frames
for i in range(len(classification)):
    if i == 0: #start the combining of the data frame with class 1
        train_data = xy.iloc[train_cl[i]['index']]
        test_data = xy.iloc[test_cl[i]['index']]
    else: #keep adding to the dataframe with class 2,3,....
        train_data = pd.concat([train_data,xy.iloc[train_cl[i]['index']]], axis = 0)
        test_data = pd.concat([test_data,xy.iloc[test_cl[i]['index']]], axis = 0)
end = time.time()
```

```
    #reset index

    print('The total ratio for train data is', len(train_data)/len(xy), 'of all data')
    print('This function took', end-start)
    return train_data, test_data, check

train = .80 #ratio you want to split
train_data, test_data, check = train_test_split(data_s,data_y,train, list(fonts.keys()))

#fig4, ax4 = plt.subplots(figsize=fsz)
#ax4.set_title('% of Data is Train')
#ax4.set_ylabel('%')
#ax4.set_xlabel('class')
#ax4.scatter(list(fonts.keys()), check, alpha=0.5, color = 'green')
#ax4.plot(list(fonts.keys()), check, alpha=0.5, color = 'green')
#ax4.scatter(list(fonts.keys()), [round((1-check[i]),2) for i in range(len(check))], alpha=0.5, color = 'red')
#ax4.plot(list(fonts.keys()), [round((1-check[i]),2) for i in range(len(check))], alpha=0.5, color = 'red')

"""
Question 2
"""
print('\nQuestion 2\n')

times += [time.time()]

#project your data

def project(train, test, eig_vector, index):
    """
    This projects whatever is input into the definition onto the eigenvectors
    specified. Make sure indexes are correct!
    """
    #reset Index for merging with project values
    train = train.reset_index(drop = True)
    test = test.reset_index(drop = True)
    #project the data
    trainx = pd.DataFrame(np.dot(train[train.columns[1:]],eig_vector))
    testx = pd.DataFrame(np.dot(test[test.columns[1:]],eig_vector))
    #concat the data
    trainy = train[train.columns[0]]
    testy = test[test.columns[0]]
    #get up to that index
    trainx = trainx[trainx.columns[index[0]:index[1]]]
    testx= testx[testx.columns[index[0]:index[1]]]
    return trainx, testx, trainy, testy

index = (0,a_idx+1) #index of Rj > 30%
X_train1, X_test1, y_train1, y_test1 = project(train_data,test_data, eig_vecto, index)

nbs1=[5] #neighbors

def KNN(X_train,y_train,X_test, nbs):
    """

    """
    pred=[]
    for i in range(len(nbs)):
        neigh = KNC(n_neighbors=nbs[i]) #Built in function that chooses the best method
        neigh.fit(X_train, y_train)
        preda=list(neigh.predict(X_test))
        pred += preda #code with all of the predictions

    #print(pred)
    return pred

## get to where this works for any values
def percentage(pred, classes, y_test):
    cat = list(classes.keys())
    totals = []
    for i in range(len(cat)):
        l = sum(y_test == cat[i])
        totals += [l]
    #Check that totals of each group sum to length of total
    print('Lengths are:', sum(totals)==len(y_test))

    correct = 0
    wrong = 0
    totals_wrong=[0] * len(totals) #have to fill it with zeros because just adding to index
    for j in range(len(pred)):
        if (pred[j] == y_test[j]) == True:
            correct += 1
        else:
            wrong += 1
            check = (pred[j] == cat) #which category is pred[j] in
            for i in range(len(check)): #finding the index of the correct value
                if check[i] == 1: #True =1, False = 0
                    totals_wrong[i] += 1
    #check that counts are correct
    print('Count for wrong + correct is:', len(y_test) == correct+wrong)
    tot_perc = correct/len(y_test)
    perc_correct = []
    perc_wrong = []
    for i in range(len(totals)):
        perc_wrong += [round((totals_wrong[i]/totals[i])*100,2)]
        perc_correct += [round((1-(totals_wrong[i]/totals[i]))*100,2)]
    return perc_correct, perc_wrong, tot_perc

#Confusion Matrix
def confusion_table(y_test, predictions):
    """
    """
    conf_m = confusion_matrix(y_test, predictions)
    #totals into a list
    total = []
    for i in range(len(conf_m[0])):
        total += [sum(conf_m[i])]
    #Get into percentages

    conf_per =  []
    i=0
    for i in range(len(total)):
        line = []
        for j in range(len(total)):
            val = round((conf_m[i][j] / total[i])*100, 1)
            line += [val]
            #print(conf_m[i][j], total[i])
        conf_per += [line]
    conf_per = np.array(conf_per)
    print(sum(total))
    print(round(sum(np.diag(conf_per)/3),1))


    #class_report = classification_report(y_test, pred[0])
    return conf_m, conf_per

ta = time.time()

pred1=KNN(X_train1, y_train1, X_test1, nbs1) #call the function
perc_correct, perc_wrong, tot_perc = percentage(pred1, fonts, y_test1)
print('Dimension is:', (index[1]-index[0])) #dimensions of data
print('Percent Correct: \n{0} \nMean Percent Correct: {1:.2f}'.format(perc_correct, tot_perc))
conf_m, conf_per = confusion_table(y_test1, pred1)
print('Confusion Matrix: \n{} \n\nConfusion Percentage: \n{}'.format(conf_m, conf_per))
```

```
pred1_train = KNN(X_train1, y_train1, X_train1, nbs1) #call the function
perc_correct_train, perc_wrong_train, tot_perc_train = percentage(pred1_train, fonts, y_train1)
print('Dimension is:', (index[1]-index[0])) #dimensions of data
print('Percent Correct: \n{0} \nMean Percent Correct: {1:.2f}'.format(perc_correct, tot_perc))
conf_m, conf_per = confusion_table(y_train1, pred1_train)
print('Confusion Matrix: \n{} \n\nConfusion Percentage: \n{}'.format(conf_m, conf_per))

print('{} Dimensions took {} seconds!'.format(a_idx+1, (time.time()-ta)))
"""
Question 3
"""
print('\nQuestion 3\n')

times += [time.time()]
ta = time.time()

#Get new test train data
index2 = (a_idx,b_idx+1)
X_train2, X_test2, y_train2, y_test2 = project(train_data,test_data, eig_vecto, index2)

#Run kNN
nbs=[5] #neighbors
pred2=KNN(X_train2, y_train2, X_test2, nbs) #call the funciton

#Get Accuracy
perc_correct2, perc_wrong2, tot_perc2 = percentage(pred2, fonts, y_test2)
print('Dimension is:', (index2[1]-index2[0])) #dimensions of data
print('Percent Correct: \n{0} \nMean Percent Correct: {1:.2f}'.format(perc_correct2, tot_perc2))

#Get confusion Matrix
conf_m2, conf_per2 = confusion_table(y_test2, pred2)
print(conf_m2, '\n', conf_per2)

print('{} Dimensions took {} seconds!'.format(b_idx+1-a_idx, (time.time()-ta)))

""" 3B - just b dimension """
print('\nQuestion 3b\n')
ta= time.time()

#Get new test train data
index3 = (0,b_idx+1)
X_train3, X_test3, y_train3, y_test3 = project(train_data,test_data, eig_vecto, index3)

#Run kNN
nbs=[5] #neighbors
```

```
pred3=KNN(X_train3, y_train3, X_test3, nbs) #call the funciton

#Get Accuracy
perc_correct3, perc_wrong3, tot_perc3 = percentage(pred3, fonts, y_test3)
print('Dimension is:', (index3[1]-index3[0])) #dimensions of data
print('Percent Correct: \n{0} \nMean Percent Correct: {1:.2f}'.format(perc_correct3, tot_perc3))

#Get confusion Matrix
conf_m3, conf_per3 = confusion_table(y_test3, pred3)
print(conf_m3, '\n', conf_per3)

print('{} Dimensions took {} seconds!'.format(b_idx+1, (time.time()-ta)))

#plot
fig5, ax5 = plt.subplots(figsize=fsz)
ax5.set_title('% Correct base on Dimensions Used')
ax5.set_ylabel('% Correct')
ax5.set_xlabel('class')
ax5.scatter(list(fonts.values()), [76, 88, 80], alpha=0.9, color = 'black')
ax5.plot(list(fonts.values()), [76, 88, 80], alpha=0.5, color = 'black')
ax5.scatter(list(fonts.values()), list(np.diag(conf_per)), alpha=0.9, color = 'yellow')
ax5.plot(list(fonts.values()), list(np.diag(conf_per)), alpha=0.5, color = 'yellow')
ax5.scatter(list(fonts.values()), list(np.diag(conf_per2)), alpha=0.9, color = 'blue')
ax5.plot(list(fonts.values()), list(np.diag(conf_per2)), alpha=0.5, color = 'blue')
ax5.scatter(list(fonts.values()), list(np.diag(conf_per3)), alpha=0.9, color = 'red')
ax5.plot(list(fonts.values()), list(np.diag(conf_per3)), alpha=0.5, color = 'red')
ax5.legend(['400 Features', '5 Features (a)', '12 Features (b-a)', '16 Features (b)'])

"""
Question 4
"""
print('\nQuestion 4\n')

times += [time.time()]

# centroids[i] = [x, y]
def get_centroids(k,ftrs):

    import numpy as np
    #starting with random values for centroids
    centroids = {
            i+1: [np.random.randint(-15.4, 16) for j in range(len(ftrs))]
            for i in range(k)
            }
    #copy the original centroids
    old_centroids = copy.deepcopy(centroids)

    return centroids, old_centroids

## ASSIGNMENT
def assignment(df, centroids, lst):
    """
    centroids = new closest centroids
    df = data frame
    lst = features
    """
    for i in centroids.keys():
        # sqrt((x1 - x2)^2 - (y1 - y2)^2)
        # A column for the distance from that point to each centroid
        df['distance_from_{}'.format(i)] = (
            np.sqrt( sum( [ (df[df.columns[j]]-centroids[i][j])**2
            for j in range(len(lst)) ] ) )
        )
    centroid_distance_cols = ['distance_from_{}'.format(i) for i in centroids.keys()]
    #the smallest distance gets their index put into the column
    df['closest'] = df.loc[:, centroid_distance_cols].idxmin(axis=1)
    #df['fonts'] = (int(i[-1]) for i in X_train.loc[:,'closest'])
    return df

## UPDATE
#update the new centroids
def update(centroids, df, lst):
    """
    centroids = new closest centroids
    df = data frame
    lst = features
    """
    #get the cost
    cst=[]
    for i in centroids.keys(): #for every classification
        c=0
        for j in range(len(lst)): #for every feature
            #Get cost before change centroids: magnitude of (x-center)^2
            ct = np.linalg.norm(df[df['closest'] == 'distance_from_{}'.format(i)][lst[j]]-centroids[i][j])
            c += ct**2
        cst += [c]

    #update the centroid
    for i in centroids.keys(): #for every classification
        for j in range(len(lst)): #for every feature
```

```
            #Get cost before change centroids: magnitude of (x-center)^2
            #Getting the mean of all the values that are closest to that specific class
            #[lst[j]] is calling the column/feature to get the mean of
            centroids[i][j] = np.mean( df[df['closest'] == 'distance_from_{}'.format(i)][lst[j]] )

    cost = round(sum(cst),0)
    #print(cost)
    return centroids, cost, cst

# CONTINUE UNTIL RAN 10 TIMES
X_train_km1 = copy.deepcopy(X_train1)
ftrs1 = X_train1.columns.tolist() # get features as a list
k = 3

tracker1 = {}
cost_dic1 = {}
cost_dic_red1 = {}
for i in range(10):
    centroids1, old_centroids1 = get_centroids(k, ftrs1) #pulls a random #
    cost_dic_red1[i] = [] #to keep track of cost for each iteration
    for j in range(20): #tries to reduce cost for each random # 10 times
        X_train_km1 = assignment(X_train_km1, centroids1, ftrs1)
        closest_centroids1 = X_train_km1['closest'].copy(deep=True)
        centroids1, cost1, cost_group1 = update(centroids1, X_train_km1, ftrs1)
        cost_dic_red1[i] += [cost1] #add it to current one for that iteration
    #print(cost)
    tracker1[cost1] = centroids1
    cost_dic1[cost1] = cost_group1

terminal_costs1 = list(tracker1.keys()) #list of all of the costs
best_cost1 = min(terminal_costs1)
best_centroids1 = tracker1[best_cost1] #pulls the best cost
best_ind_costs1 = cost_dic1[best_cost1] #pulls the individual best costs
X_train_km1 = assignment(X_train_km1, best_centroids1, ftrs1) #make your X_train of your best
centroids

#plot
fig6, ax6 = plt.subplots(figsize=fsz)
ax6.set_title('Cost per Iteration')
ax6.set_ylabel('Cost')
ax6.set_xlabel('Iteration')
ax6.scatter(list(range(len(cost_dic_red1[3]))), (cost_dic_red1[3]), alpha=0.9, color = 'red')
ax6.plot(list(range(len(cost_dic_red1[3]))), (cost_dic_red1[3]), alpha=0.5, color = 'red')

c = [] #get column of y values for comparison added to cluster
```

```
for i in X_train_km1.loc[:,'closest']:
    c += [int(i[-1])]
y = pd.DataFrame(c)
y.rename(columns={0:'font'}, inplace=True) #Had to convert to Panda DF and rename column
X_train_km1=pd.concat([X_train_km1,y],axis=1)

print('\nTerminal Cost for Ra: \n{}'.format(terminal_costs1))
print('\nBest Centroids for Ra: \n{} \n\nBest Cost: \n{}'.format(best_centroids1, best_cost1))

"""
Question 5
"""
print('\nQuestion 5\n')

times += [time.time()]

#Get a centroid into each group by splitting and getting mean of each ftr
def known_centroids(df, classification, key):
    """
    df = df of known features and y value (y value must be integer)
    classification: the different classes
    key: column name to compare
    """
    centroids = {}
    cl = {}
    for i in list(classification.keys()):
        group = df.where(df[key] == i) #split into group
        group = group.dropna()
        cl[i] = group[group.columns[1:]]
        group = list(group.mean(axis=0))
        centroids[i] = group[1:] #put that group into dictionary with group as key

    return centroids, cl

#Cost for each cluster from 'ideal'
def get_cost(dictionary, centroids, lst):
    """
    This gets the cost between two dictionary's
    """
    #get the cost

    cst=[]
    for i in centroids.keys(): #for every classification
        c=0
        for j in range(len(lst)): #for every feature
```

```
            #Get cost before change centroids: (magnitude of (x-center))^2
            ct = np.linalg.norm(dictionary[i][lst[j]]-centroids[i][j])
            c += ct**2
        cst += [round(c,0)]

    return cst

XY_train1 = pd.concat([y_train1,X_train1], axis=1) #combine y and x trains for splitting

cl_centroids1, cl_a = known_centroids(XY_train1, fonts, 'font')

cl_cost1 = get_cost(cl_a, cl_centroids1, ftrs1)
cl_total_cost1 = sum(cl_cost1)

#Count all of them that are grouped the same (Confusion Matrix)
km_conf_m1, km_conf_per1 = confusion_table(XY_train1['font'], X_train_km1['font'])
print('Confusion Matrix: \n{} \n\nConfusion Percentage: \n{}'.format(km_conf_m1, km_conf_per1))

km_conf_m12, km_conf_per12 = confusion_table(X_train_km1['font'], XY_train1['font'])
print('Confusion Matrix: \n{} \n\nConfusion Percentage: \n{}'.format(km_conf_m12, km_conf_per12))

times += [time.time()]

print('\nBest Centroids for ideal: \n{} \n\nBest Cost: \n{}'.format(cl_centroids1, cl_total_cost1))
```