Final Project
Handwritten Digit Recognition with Principal Component Analysis and k-Nearest Neighbors
ENGR 396

Daniel Giacoio & Ryan Yee

May 1, 2020

Date Performed:	April 26, 2019
Partners:	Daniel Giacoio
	Ryan Yee
Instructor:	Dr. Kump

1 Objective

For this project, we will be comparing the accuracy's of Mini Project 5 Linear Discriminant Analysis on handwritten digits trained with American Census Bureau and tested on American High School Students hand written digits. Further, the algorithm will reduce the size of our original data, classify it and make a prediction.

2 Data

The data we used is 10,000 examples of American High school students hand written digits. This is the same test data used in Mini Project 5. The data has 784 features of 255-bit pixel value of zero through nine handwritten digits.

3 Converting Data

The data is downloaded as a 'csv' file. By using 'maritimeENGR428helper', we are able to read the data into the program as a list of strings. In order for us to run the data through the algorithm, we need the data to be in the form of an integer. Using 'np.array' allows us to turn the lists of data into an array of data which then can be converted into integers through the function 'int()'.

The 0^{th} row of the data is labels of the target and features. The 0^{th} column of our data is the target variables (0-9). The targets will be extracted by calling all the data but the first row , 'data-downloaded[1:,0]'. By getting the the rest of the data in the form we need it, we will have to loop through each row and column. First we will iterate through each row in the data that is not including the labels and the target.We will then iterate through each column of the rows and then take the integer from each column and put it into an array 'np.array([[int(column)for column in row] for row in data-downloaded[1:,1:]])'. Now we have a 10,000 x 784 integer matrix.

```
9  #get data
10 data_downloaded = myCSVreader('mnist_test.csv')
11 data_downloaded = np.array(data_downloaded)
12 data_targets = np.array([int(num_str) for num_str in \
13                            data_downloaded[1:,0]])
14 data = np.array([[int(num_str) for num_str in inner] \
15                   for inner in data_downloaded[1:,1:]] )
16
```

Figure 1: Code for Changing Data into Integers.

4 Procedure

4.1 Principal Component Analysis

Before we can do anything with our data we first have to fit the data so we perform principal component analysis. We first used the the *PCA* function from *sklearn.decompostion* library to perform our principal component analysis. In line 20, we called the PCA function to use on our data. The parameter *n-components* gives you the option to chose the number of features you want the PCA function to reduce the size of the data down to. We first called for the function to use PCA without any feature reduction, making *n-components = 784*. The reason to not reducing our feature with the PCA function is to observe every principal component in the data and with the information, make a decision to what size we should reduce down to. In lines 23 and 24, we calculated the variance in each principal component using the function *explained-variance-ratio*. The percent variance in each principal component is going to tell us how much information of the original data set is being retained in the given principal component.

	0		
0	10.05	16	61.73
1	17.59	17	63.01
2	23.73	18	64.19
3	29.16	19	65.34
4	34.19	20	66.41
5	38.44	21	67.44
6	41.75	22	68.42
7	44.7	23	69.32
8	47.43	24	70.21
9	49.71	25	71.04
10	51.84	26	71.84
11	53.94	27	72.61
12	55.67	28	73.33
13	57.36	29	74.02
14	58.94		

Figure 2: Accumulation of Variance in Each Principal Component.

In line 24 we accumulated the variance in each principal component, this gives how much percent variance the corresponding principal component holds up to. In Figure 2, each row is the corresponding principal component with its percent variance. The image only shows up to thirty principal components, there is 784 principal components in total because we did not

reduce our data's features yet. With this information we can now decide on to what dimension we want to reduce our data down to. To help our decision process we will plot the principal components vs its corresponding variance.

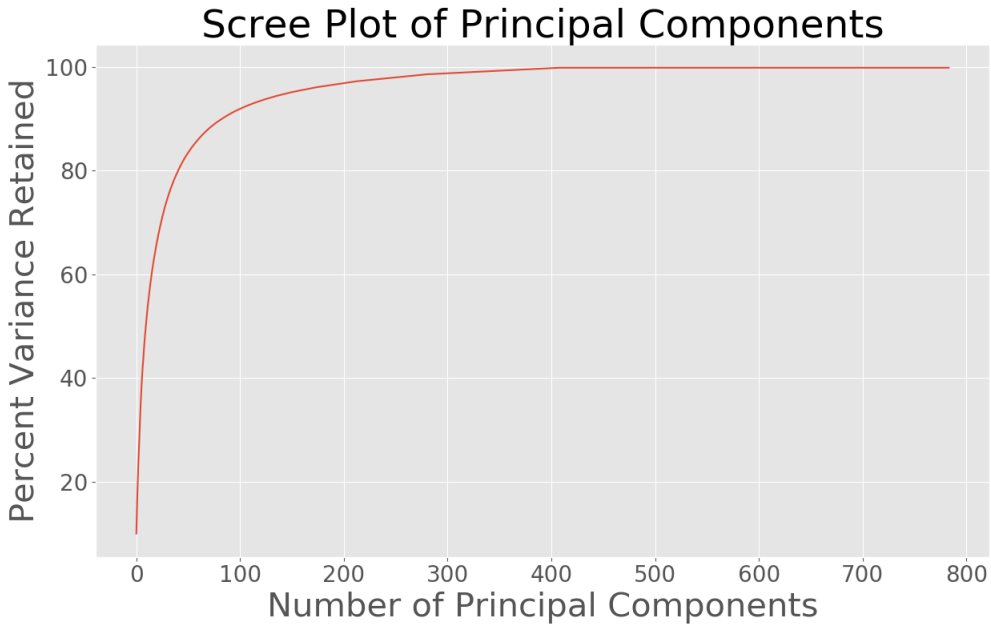

Figure 3: Scree Plot of All Principal Components.

Figure 3 is called a scree plot, the scree plot shows the variance in the principal components. To determine which number of principal components we should keep, observe the elbow of the graph. At this point of the graph is where we have enough information from the original data. The rest of the principal components are useless to us if we have a good amount of information from the original data. For this project we chose to use *36 principal components.* Now that we have decided on our new size to reduce to, we will use the PCA function again and set our *n-components = 36.* Each principal component is an eigenvector corresponding to its eigenvalue. What these eigen vectors are giving us is a new axis for our data. We now need to project our data onto these new dimensions to officially reduce the size of our data. By doing so, this projection matrix is called the transformed matrix. To compute our transformed matrix we used the function *fit-transform(data)* on line 32.
The next assignment is to see what would happen if we transformed the data into 2 dimensions. We will compare both transformed matrices of 36 dimensions and 2 dimensions.

4.2 Transformed Images

In order to plot our transformed 2-D data, we first need to assign the data examples to its respective class using *np.where.* We will then scatter all nine classes with its first principal component on the x-axis and the second principal component on its y-axis. The goal here is to show what this data would look like in 2-D and how it would compare to our 36 dimension reduction.

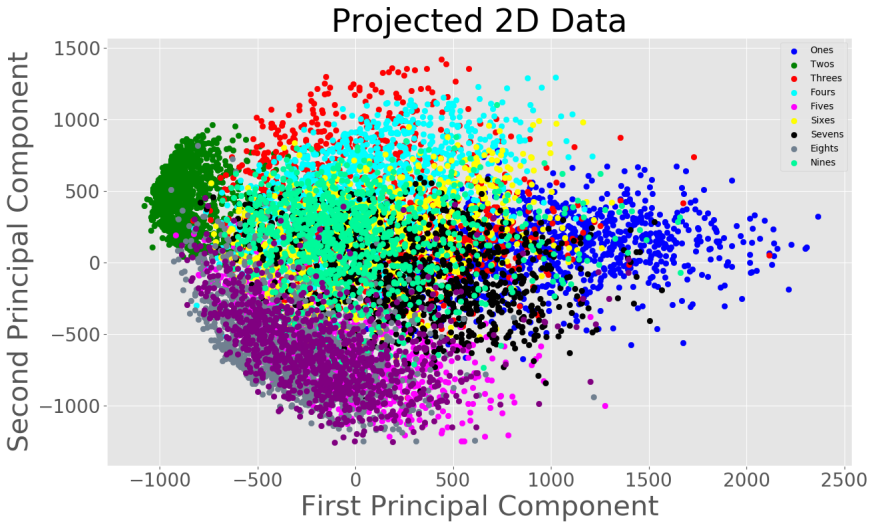

Figure 4: 2-D Scatter plot of The First Two Principal Components.

In Figure 4, each class has its respective color which shows the classes clustered together. Although we can get an idea of where each class belongs, but in 2-D these classes are stacked on top of each other. Being in 2 dimensions would not give any classifying algorithm enough space to decipher what class a test target would belong to.

We cant display what the data would look like in 36 dimensions but we can plot the image in a 6x6 grey scale pixel image consisting of a total of 36 pixels which is our total number of features.

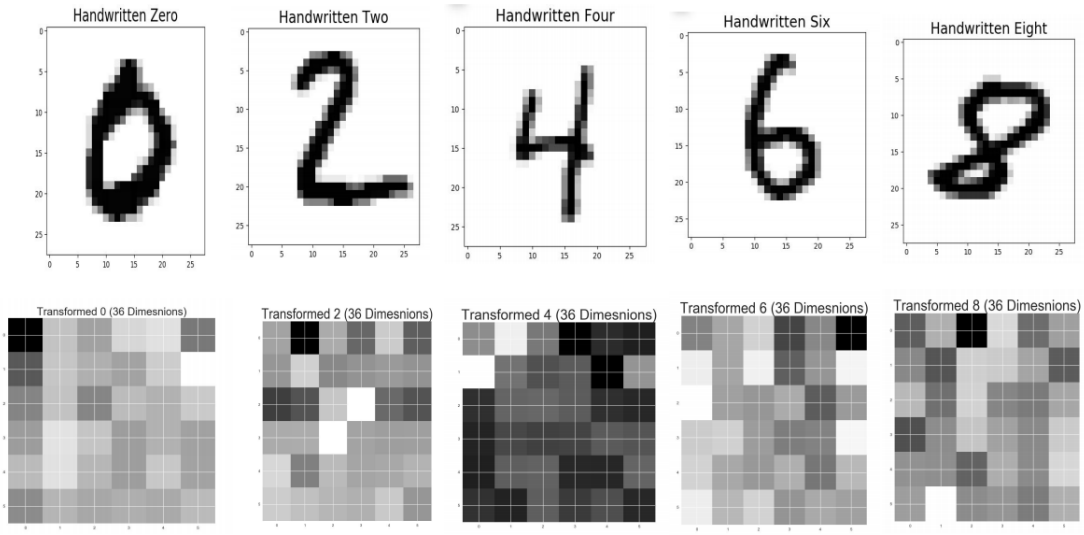

Figure 5: Examples from Original Data Set vs Its Transformed 36 Dimension Version.

Numerically our transformed data is not the same as our original data, these numbers are scaled in accordance to the transformed data but still represent the same thing. Figure 5 has examples from the original data set compared to its transformed 36 dimensional image. These images do not mean anything to the human eye, rather these images have enough variance or information from the original data to show the proper number we transformed to. Now we want to teach the computer to classify these transformed images into knowing which numerical value each handwritten digit is.

4.3 k-Nearest Neighbors

The data is now transformed to the selected size of features to retain. Now we need to classify the transformed data to then make predictions. What k-Nearest Neighbors(kNN) will do is input a test target to the transformed data and calculate its distance from its 'k' number of neighbors. kNN then calculates probability of the classes in the selected distance of the test target. The class of the highest probability will then be assigned to that test target.The downside to this classifier is the demand of memory space it needs. kNN loads all the data into memory and test one at a time, each test target and saves that class it predicts. If we refer back to Figure 4, trying kNN will be difficult for the classifier to make a correct prediction. The classes are all overlapping each other so the distance of its neighbors will be mixed in with other classes. By adding more dimensions, this will reduce the classes from overlapping on each other and will improve our accuracy. We will use kNN for both transformed data of 36 dimensions and 2 dimensions.

The next objective is to find the best number of neighbors to select to give us the highest accuracy for our predictions. We will first loop through a range of one neighbor to ten neighbors, after looping through the range of neighbors the data will be broken up into a 5 fold Cross Validation(CV). This 5 fold CV will test each fold one at a time with respect to the rest of the 4 folds. This will give us the predicted classes. To test if our predictions are correct, we will compare it to the true target values from *data-targets* in a confusion matrix.

```
accur = []
neighbors = []
for i in range(1,10):
    knn =  KNeighborsClassifier(n_neighbors = i)
    prediction = cross_val_predict(knn,T_new,data_targets,cv = 5)
    C = confusion_matrix(data_targets, prediction)
    accur = np.append(accur,np.trace(C)/np.sum(C))
    neighbors = np.append(neighbors,i)
maximum = np.where(accur == np.max(accur))
maximum = np.array(maximum)
maximum_neighbors = neighbors[maximum]
print('----------------------------')
print('The number of neighbors that gives the highest \
      accuracy (36 Dimensions):',int(maximum_neighbors))
print('Accuracy(36 Dimensions):' \
      ,accur[int(maximum_neighbors -1)]*100,'%')
```

Figure 6: 36 Dimension k-Nearest Neighbors Code.

```
Accur = []
Neighbors = []
for j in range(1,10):
    KNN =  KNeighborsClassifier(n_neighbors = j)
    Prediction = cross_val_predict(KNN,T_two,data_targets ,cv = 5)
    c = confusion_matrix(data_targets, Prediction)
    Accur = np.append(Accur,np.trace(c)/np.sum(c))
    Neighbors = np.append(Neighbors,j)
Maximum = np.where(Accur == np.max(Accur))
Maximum = np.array(Maximum)
Maximum_neighbors = Neighbors[Maximum]
print('----------------------------')
print('The number of neighbors that gives the highest accuracy \
      (2 Dimensions):',int(Maximum_neighbors))
print('Accuracy (2 Dimensions):' \
      ,Accur[int(Maximum_neighbors -1)]*100,'%')
```

Figure 7: 2 Dimension k-Nearest Neighbors Code.

In lines 148-155, *accur* and *neighbors* create a list so we can eventually append our values from the loop into it. In line 151, we use the *KNeighborsClassifier* function from *sklearn.neighbors* library to do our classification. We then use parameter *n-neighbors* to gives us the option to select the number of neighbors we want. We will then iterate through *n-neighbors* to give us the best accuracy. For testing the kNN classifier, we use the function *cross-val-predict* in the *sklearn.model-selection* library to make our predictions. *Cross-val-predict* calls for our classifier function, the data we are breaking into folds, the true target values and the number of selected folds. We then will use *np.append* to append the accuracy of our prediction for each neighbor in line 169 into *accur*. To calculate the accuracy of our predictions we will add the numbers in the diagonal together dividing by the sum of the total confusion matrix. The function *np.trace* allows us to add the numbers along the diagonal of the matrix, the numbers in the diagonal represent the number of times the predictor classified the corresponding number correctly. If we divide this by the sum of the confusion matrix we can see the percent of correct numbers classified.

For lines 165-172, we did the exact process that we did for lines 148-155 just this is in 2 dimensions and lines 148-155 was 36 dimensions.

	0	1	2	3	4	5	6	7	8	9
0	966	1	1	0	0	3	8	1	0	0
1	0	1129	2	2	0	0	2	0	0	0
2	15	10	970	4	0	1	5	20	6	1
3	0	2	7	952	0	21	0	9	14	5
4	0	10	1	0	913	0	10	1	2	45
5	4	5	2	28	6	833	8	2	1	3
6	6	6	0	0	3	3	939	0	1	0
7	0	34	8	0	2	0	1	962	1	20
8	4	4	5	26	6	18	5	5	891	10
9	3	8	4	10	16	5	2	16	8	937

Figure 8: Confusion Matrix of 36 Dimensions.

	0	1	2	3	4	5	6	7	8	9
0	722	0	61	14	2	44	120	0	15	2
1	0	1055	21	15	0	9	5	6	21	3
2	146	28	249	238	21	131	103	7	103	6
3	26	24	174	500	9	88	32	8	142	7
4	1	10	16	3	444	29	45	170	12	252
5	80	20	160	140	56	166	104	16	140	10
6	170	10	104	39	52	122	337	8	100	16
7	0	37	13	10	284	35	17	384	17	231
8	84	21	158	162	37	160	101	11	229	11
9	10	20	12	1	385	17	31	276	12	245

Figure 9: Confusion Matrix of 2 Dimensions.

5 Results/Conclusion

For each value of neighbors tested in the range of one to ten, the result for the 36 dimension transformed data came out to be **3** neighbors. For 2 dimensions, **9** neighbors was the best number of neighbors to use. With 3 neighbors in 36 dimensions, we were able to achieve an accuracy rate of **95.55** percent. With 9 neighbors in 2 dimensions, we were able to achieve an accuracy rate of **43.33** percent. As you could imagine from viewing Figure 4, the classifier would have a hard time classifying the test target to the correct class due to the overlapping of the classes. 43.33 percent for 2 dimensions makes sense, this accuracy is not ideal compared to 36 dimensions.

Bringing this back to our initial goal, we wanted to compare our outputs to the output of Mini Project 5. Both projects have similarities and differences, but the steps to feature reduction, classification and prediction are the same.

	0	1	2	3	4	5	6	7	8	9
0	940	0	1	4	2	13	9	1	9	1
1	0	1096	4	3	2	2	3	0	25	0
2	15	32	816	34	21	5	37	9	57	6
3	5	5	25	883	4	25	3	16	29	15
4	0	12	6	0	888	4	7	2	10	53
5	8	8	4	44	12	735	15	10	38	18
6	12	8	11	0	25	29	857	0	16	0
7	2	30	15	9	22	2	0	864	4	80
8	7	27	8	27	20	53	10	6	790	26
9	9	7	1	13	63	6	0	37	12	861

Figure 10: Confusion Matrix of Linear Discriminant Analysis from Mini Project 5.

Figure 10 is the confusion matrix output from Mini Project 5 it was able to predict an accuracy of **87.3**. With linear discriminant analysis(LDA), it can only reduce the data down to the size of its class's minus one. So in this situation we have ten classes, linear discriminant analysis will reduce our 784 features down to 9 features and be able to achieve a accuracy of 87.3 percent. Both LDA and PCA are feature reduction methods, in our project we were able to choose what size we wanted to reduce down to. For LDA you are restricted to what size you reduce down to. Although LDA is a feature reduction and classification method, its more efficient, with respect to computing memory, than using PCA and kNN. As we observed in this project we see the trade offs with using PCA and kNN to LDA.

6 Code

```
from maritimeENGR428helper import myCSVreader
from sklearn.model_selection import cross_val_predict
from sklearn.neighbors import KNeighborsClassifier
import numpy as np
from sklearn.metrics import confusion_matrix
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA

#get data
data_downloaded = myCSVreader('mnist_test.csv')
data_downloaded = np.array(data_downloaded)
data_targets = np.array([int(num_str) for num_str in \
                         data_downloaded[1:,0]])
data = np.array([[int(num_str) for num_str in inner] \
                 for inner in data_downloaded[1:,1:]] )


#each feature

pca = PCA(n_components = 784)
fit = pca.fit(data,data_targets)

var = fit.explained_variance_ratio_
var1 = np.cumsum(np.round(var, decimals=4)*100)
T = fit
W = fit.components_


pca_new = PCA(n_components = 36 )
T_new = pca_new.fit_transform(data)
pca_two = PCA(n_components = 2 )
T_two = pca_two.fit_transform(data)

```

Figure 11: Part 1 of Code.

```
T_two = pca_two.fit_transform(data)


zero = np.where(data_targets == 0)
one = np.where(data_targets  == 1)
two = np.where(data_targets  == 2)
three = np.where(data_targets  == 3)
four = np.where(data_targets  == 4)
five = np.where(data_targets  == 5)
six = np.where(data_targets  == 6)
seven = np.where(data_targets  == 7)
eight = np.where(data_targets  == 8)
nine = np.where(data_targets  == 9)
#---------------------------
t1 = T_two[:,0]
t2 = T_two[:,1]

#class zero
D1_0 = t1[zero]
D2_0 = t2[zero]
#class one
D1_1 = t1[one]
D2_1 = t2[one]
#class two
D1_2 = t1[two]
D2_2 = t2[two]
#class three
D1_3 = t1[three]
D2_3 = t2[three]
#class four
D1_4 = t1[four]
D2_4 = t2[four]
#class five
```

Figure 12: Part 2 of Code.

```
D2_4 = t2[four]
#class five
D1_5 = t1[five]
D2_5 = t2[five]
#class six
D1_6 = t1[six]
D2_6 = t2[six]
#class seven
D1_7 = t1[seven]
D2_7 = t2[seven]
#class eight
D1_8= t1[eight]
D2_8 = t2[eight]
#class nine
D1_9= t1[nine]
D2_9 = t2[nine]

blue = plt.scatter(D1_0,D2_0,color='blue')
green = plt.scatter(D1_1,D2_1,color='green')
red = plt.scatter(D1_2,D2_2,color='red')
cyan = plt.scatter(D1_3,D2_3,color='cyan')
magenta = plt.scatter(D1_4,D2_4,color='magenta')
yellow = plt.scatter(D1_5,D2_5,color='yellow')
black = plt.scatter(D1_6,D2_6,color='black')
slategray = plt.scatter(D1_7,D2_7,color='slategray')
mediumspringgreen = plt.scatter(D1_8,D2_8\
,color='mediumspringgreen')
purple = plt.scatter(D1_9,D2_9,color='purple')

plt.title('Projected 2D Data',fontsize  = 25)
plt.tick_params(labelsize=12)
plt.xlabel('First Principal Component',fontsize = 20)
```

Figure 13: Part 3 of Code.

```
plt.xlabel('First Principal Component',fontsize = 20)
plt.ylabel('Second Principal Component',fontsize = 20)
plt.legend((blue,green,red,cyan,magenta,yellow,black\
,slategray,mediumspringgreen,purple),('Ones','Twos'\
,'Threes','Fours','Fives','Sixes','Sevens','Eights'\
,'Nines'), loc=0,fontsize = 12)
plt.show()




#-------Plotting the scree of variance in the pc
plt.figure()
plt.plot(var1)

plt.title('Scree Plot of Principal Components', fontsize  = 25)
plt.tick_params(labelsize=12)
plt.xlabel('Number of Principal Components',fontsize  = 20)
plt.ylabel('Percent Variance Retained', fontsize  = 20)
plt.show()
print('----------------------------')

#-------Images of regular and transformed
queryIm = T_new[3,:]
queryIm.shape=(6,6)
plt.figure()
plt.imshow(255-queryIm,cmap='gray')
plt.title('Transformed 0 (36 Dimesnions)',fontsize  = 25)
plt.show()

queryIm = data[3,:]
queryIm.shape=(28,28)
```

Figure 14: Part 4 of Code.

```
queryIm = data[3,:]
queryIm.shape=(28,28)
plt.figure()
plt.imshow(255-queryIm,cmap='gray')
plt.title('Regular 0 (784 Dimesnions)',fontsize  = 25)
plt.show()
print('----------------------------')
#-------------------------------------------------
queryIm = T_new[1,:]
queryIm.shape=(6,6)
plt.figure()
plt.imshow(255-queryIm,cmap='gray')
plt.title('Transformed 2 (36 Dimesnions)',fontsize  = 35)
plt.show()

queryIm = data[1,:]
queryIm.shape=(28,28)
plt.figure()
plt.imshow(255-queryIm,cmap='gray')
plt.title('Regular 2 (784 Dimensions)',fontsize  = 35)
plt.show()
print('----------------------------')
#------------36 Dimensions----------------------------
accur = []
neighbors = []
for i in range(1,10):
    knn =  KNeighborsClassifier(n_neighbors = i)
    prediction = cross_val_predict(knn,T_new,data_targets,cv = 5)
    C = confusion_matrix(data_targets, prediction)
    accur = np.append(accur,np.trace(C)/np.sum(C))
    neighbors = np.append(neighbors,i)
maximum = np.where(accur == np.max(accur))
```

Figure 15: Part 5 of Code.