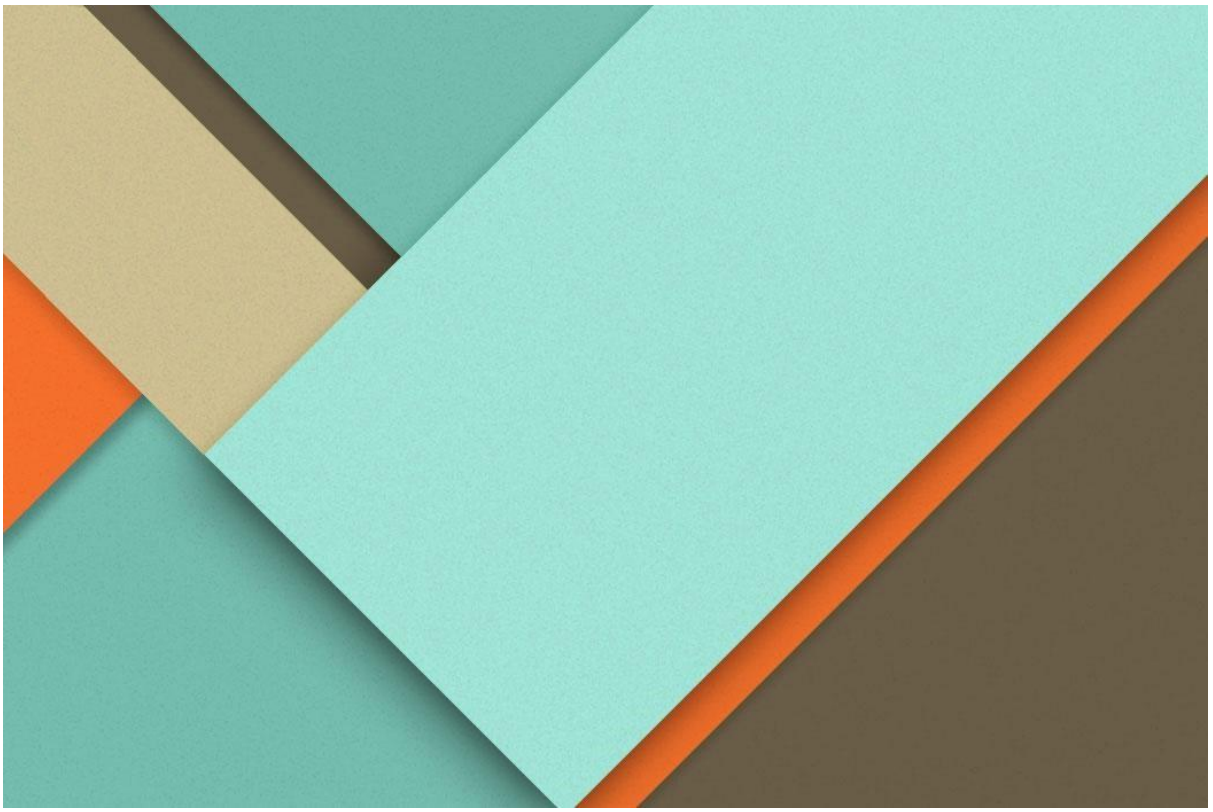


Face Recognition

–

Name	ID
Toka Ashraf Abo Alwafa	19015539
Rowan Nasser Edrees	19015686
Nada Mohamed Ibrahim	19016782

Problem Statement

The problem is to perform face recognition on the ORL dataset, which consists of 400 grayscale images of size 92x112 pixels belonging to 40 different individuals, with 10 images per person. The goal is to split the dataset into training and test sets, generate a data matrix and label vector, and perform classification using PCA and LDA. The performance of the classifiers will be evaluated using accuracy, and the effect of hyperparameters such as the number of neighbors in the KNN classifier will be explored. Additionally, the performance of the classifiers will be compared when applied to non-face images of the same size as the ORL dataset, and the accuracy measure will be criticized for large numbers of non-face images in the training data.

Code Flow

Imported Libraries

```
[1]  import zipfile
     import pandas as pd
     from skimage import io
     import numpy as np
     from PIL import Image
     import statistics as stat
     from sklearn import metrics
     from sklearn.neighbors import KNeighborsClassifier
     from sklearn.metrics import accuracy_score
     import matplotlib.pyplot as plt
     from numpy.linalg import eig
     import scipy
     import os
```

1. Download the Dataset and Understand the Format

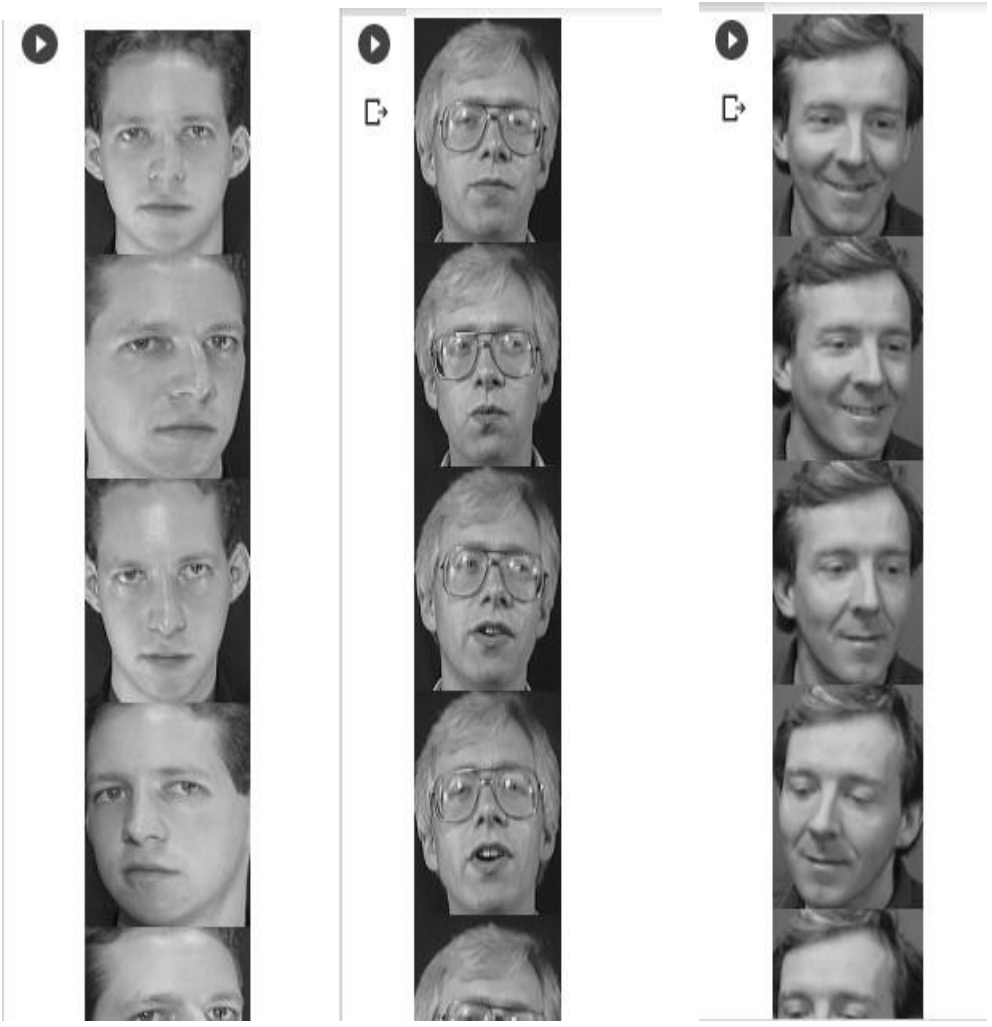
```
[2]  # !wget https://drive.google.com/file/d/1bIt96dpqtxrc7bHO1_uHeCSy0nqL3Ovj/view?usp=sharing
     with zipfile.ZipFile('/content/archive.zip', 'r') as zip_ref:
         zip_ref.extractall('/content')
```

You can download the dataset easily from the link provided in the first line of the cell. The following code snippet is responsible for extracting all the files in the archive.zip file using the ZipFile() Function which opens the file in the read mode.

2. Generate the Data Matrix and the Label vector

```
[3]  dataset = []
     y = []
     for i in range(40):
       for j in range(10):
         image = Image.open(f"/content/s{i + 1}/{j + 1}.pgm")
         image.show()
         image = io.imread(f"/content/s{i + 1}/{j + 1}.pgm", as_gray=True)
         vector = image.flatten()
         dataset.append(vector)
         y.append(i + 1)
     D = pd.DataFrame(data=dataset)  # Data Matrix
     print(D)
```

This code snippet loads a set of images in grayscale format from a dataset of 40 classes, each containing 10 images. It then converts each image into a 1D vector of pixel values and stores them in a data matrix D.



3. Split the Dataset into Training and Test Sets

```
[4]  test_x=[]
     test_y=[]
     train_x=[]
     train_y=[]
     def split():
      for i in range(400):
        if i%2 ==0:
          test_x.append(dataset[i])
          test_y.append(y[i])
        else:
          train_x.append(dataset[i])
          train_y.append(y[i])

     split()
```

This code snippet performs a train-test split on the dataset and their corresponding labels created by the previous code snippet.

4. Classification Using PCA

We followed the given algorithm of pca.

First we compute the mean, the centered matrix (z), then the covariance of transpose z and at the end we get the eigenvalues and eigenvectors in descending order.

```
[ ]  def compute_eigen(train_x):
       mean = np.mean(train_x,axis=0)
       Z = train_x - mean
       cov = np.cov(np.transpose(Z))
       eigenValues , eigenVectors = np.linalg.eigh(cov)

     # sorted index
       idx = eigenValues.argsort()[::-1]
       eigenValues = eigenValues[idx]
       eigenVectors = eigenVectors[:,idx]
       return eigenValues, eigenVectors
```

compute projection matrix

```
[ ]  def projection_matrix(eigenValues,eigenVectors,alpha):
       eigenValuesSum = eigenValues.sum()
       sum = 0
       r = 0
       for i in eigenValues :
           sum = sum + i
           r = r + 1
           if((sum/eigenValuesSum)>=alpha):
             break
       p = eigenVectors[:,:r]
       return p
```

knn classification function that takes train and test dataset to classify images

```
10] # Perform K-NN Classification
    def knn_classification(train_x, train_y, test_x, test_y, k):
      knn = KNeighborsClassifier(n_neighbors=k)
      train_y = np.ravel(train_y)
      knn.fit(train_x, train_y)
      y_pred = knn.predict(test_x)
      print_correction(y_pred, test_y)
      accuracy = accuracy_score(test_y, y_pred)
      return accuracy
```

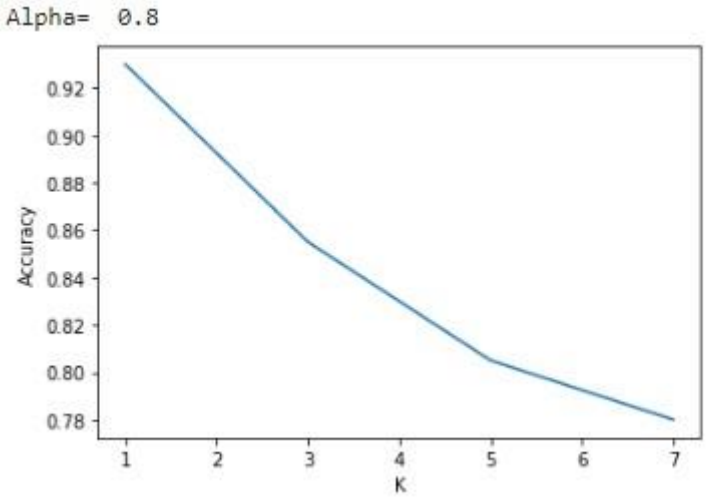
pca function that call the previous functions and calculate the accuracy of each alpha we have used for each neighbor(k) and plot the relation between the accuracy and k

```
[ ]  def pca(train_x,test_x,train_y,test_y):
       eigenValues,eigenVectors = compute_eigen(train_x)
       alpha =[0.8,0.85,0.9,0.95]
       k = [1, 3, 5, 7]
       x=[]
       for a in alpha:
         pca_accuracies =[]
         x=[]
         for i in k :
          p = projection_matrix(eigenValues,eigenVectors,a)
          reduced_train_x = np.dot(train_x,p)
          reduced_test_x = np.dot(test_x,p)
          acc =knn_classification(reduced_train_x, train_y, reduced_test_x, test_y, i)
          pca_accuracies.append(acc)
          x.append(i)

         print("Alpha= ",a)
         plot_accuracy(x,pca_accuracies, 'PCA')
```

for alpha = 0.8

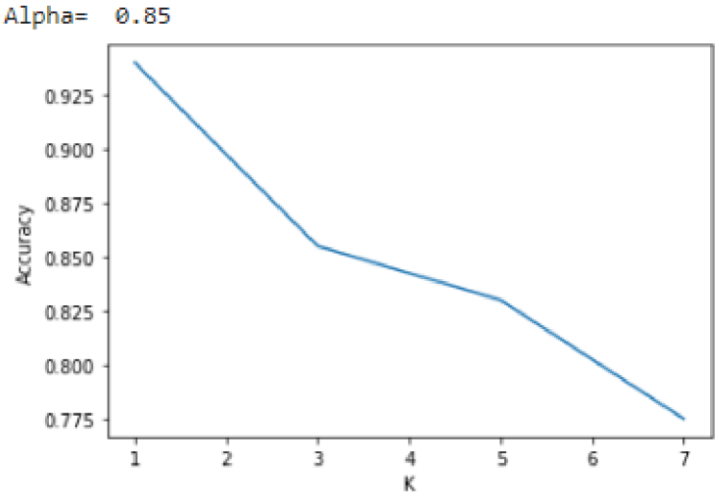
number of neighbors	accuracy
1	0.93

3	0.855
5	0.805
7	0.78



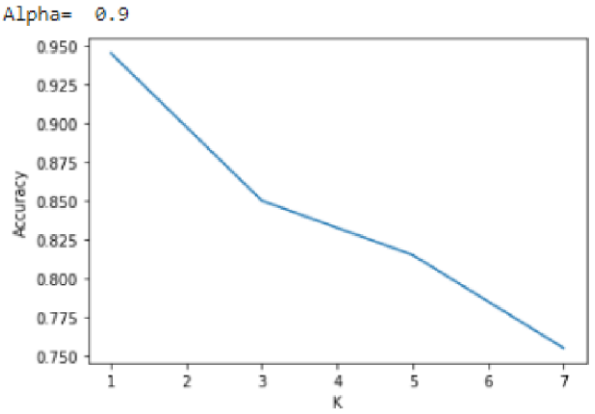
for alpha = 0.85

number of neighbors	accuracy
1	0.94
3	0.855
5	0.83
7	0.775



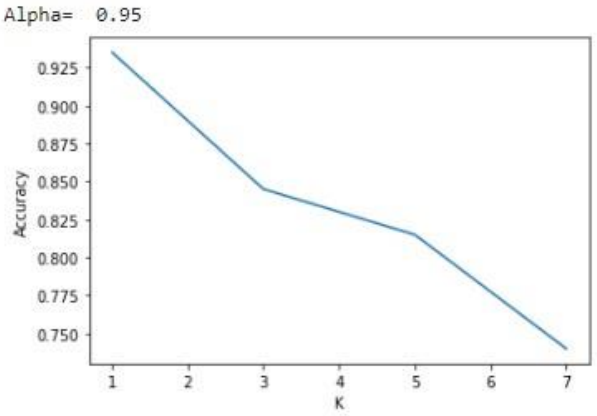
for alpha = 0.9

number of neighbors	accuracy
1	0.945
3	0.85
5	0.815
7	0.755



for alpha = 0.95

number of neighbors	accuracy
1	0.935
3	0.845
5	0.815
7	0.74



- **from these values we can see that when we increase alpha the accuracy decreases slightly.**

5. Classification Using LDA

```
def LDA(trainx,testx,no_of_samples_per_class):
  train_data=divide_train_data(trainx,no_of_samples_per_class)
  Mu=get_mean_vectors(train_data)
  Mu_overall_sample=get_overall_sample_mean(train_data)
  Sb=calculate_Sb(Mu,Mu_overall_sample,no_of_samples_per_class)

  z=center_data(train_data,Mu)
  S=calculate_s_matrix(z)
  eig_vec=compute_dominant_eigen(S,Sb)
  training_set,test_set=compute_train_test_projection(trainx,testx,np.real(eig_vec))
  return training_set,test_set
```

Illustration of the steps of LDA function

- Splitting the train of the dataset into an array of dimensions (40,5,10304) instead of (200,10304) using the number of samples per class(here we divide the samples into two equal halves and all samples per class is 10 so 5 for train data and 5 for test data) to enable us to get the mean of each class easily.

```
def divide_train_data(trainx, no_of_samples_per_class):
  train_data=[]
  length = int(len(trainx) / no_of_samples_per_class)
  for i in range(length):
    train_data.append([])
  j=-1
  for i in range(len(trainx)):
    if(i % no_of_samples_per_class == 0):
      j+=1
    train_data[j].append(np.asarray(trainx)[i])
  train_data=np.asarray(train_data)
  return train_data
```

- Calculating the mean for each class from the 40 classes using the function of mean from numpy library and set axis to 1 to calculate the mean for each of the 5 images of each subject and the resulting array of means will be in dimensions of (40, 10304).

```
#Calculate the mean vector for every class Mu1, Mu2, ..., Mu40.
def get_mean_vectors(train_data):
    Mu=np.mean(train_data , axis=1)
    return Mu
```

- Calculating the overall sample mean from the training data of the new dimension (40,5,10304) using function of mean but setting axis to zero to get the mean of each 40 subject to get the result as an array of dimension (5,10304)

```
def get_overall_sample_mean(train_data):
    Mu_overall=np.mean(train_data , axis=0)
    return Mu_overall
```

- Calculating the between class scatter matrix Sb using the law of

$$S_b = \sum_{k=1}^{m} n_k(\mu_k - \mu)(\mu_k - \mu)^T$$

  - Firstly, we calculate the difference between the mean and the overall sample mean by looping on the number of classes and subtracting the overall sample mean vector from the mean vector for every class.
  - Then we get the multiplication of the matrix of this difference and its transpose and multiply all with the number of samples per class.
  - Finally, add the result for each iteration together to get the between scatter matrix.

```
#calculate between class scatter matrix Sb
def calculate_Sb(Mu, Mu_overall , no_of_samples_per_class):
  Sb=np.zeros((len(Mu[0]),len(Mu[0])))  ## 10304*10304
  for i in range(len(Mu)):
    sub=(Mu[i]) - Mu_overall
    Sb =np.add(Sb , (no_of_samples_per_class * np.dot(sub.T , sub )) )

  return Sb
```

- Centralizing the train data by subtracting the mean for each class from the training data of that class.

```
##center class matrix   40x5x10304
def center_data(train_data,Mu):
  z=np.zeros(train_data.shape)
  for i in range(len(Mu)):
    z[i]= train_data[i] - Mu[i]
  return z
```

- Calculating the class scatter matrices S by multiplying the transpose of the z matrix(centralizing train data) by the z matrix itself for each class and then adding all together.

```
##class scatter matrices  S
def calculate_s_matrix(z_matrix):
  S=np.zeros((z_matrix.shape[2],z_matrix.shape[2])) ##10304*10304
  for i in range(z_matrix.shape[0]):
    S += np.dot(z_matrix[i].T , z_matrix[i])

  return S
```

- Computing the 39 dominant eigen vectors using the array of class scatter matrices S and the array of the between scatter matrix:
  - Firstly, we get the inverse of the S array using linalg.inv from numpy library and then multiply this inverse to the Sb matrix and from this product we will calculate the eigen values and eigen vectors using the function of linalg.eigh from numpy library
  - From the result eigen values, sorting will done from largest to smallest using argsort() function and then choose the first 39 largest eigen values to get their eigen vectors to obtain the dominant 39 eigen vector.

```
[ ]  ## compute dominant eigen vector
     # S^-1 * Sb
     def compute_dominant_eigen(S , Sb):
       s_inverse=np.linalg.inv(np.asarray(S))
       product_res=np.dot(s_inverse , Sb)
       eig_val, eig_vec = np.linalg.eigh(product_res)
         #get indexes of descendigly sorted eigen values array
       idx = eig_val.argsort()[::-1][:39]
         #get the 39 eigenvectors accordingly
       eig_vec = np.array(eig_vec[:,idx].real,dtype= np.float32)

       return eig_vec
```

- Projecting the train and test data using the dominant 39 eigen vector we got by multiplying them by the eigen vectors matrix.

```
##Project the training set, and test sets separately using the same
##projection matrix U. You will have 39 dimensions in the new space.
def compute_train_test_projection(trainx,testx,eigen_vectors):  ##eigen vectors >> 10304*39
  training_set=np.dot(np.asarray(trainx) , eigen_vectors)
  test_set=np.dot(np.asarray(testx) , eigen_vectors)

  return training_set,test_set
```

- Using a simple classifier (first Nearest Neighbor to determine the class labels): After applying the knn classifier when k=1 we get accuracy of 94.5 %.

```
reduced_train,reduced_test=LDA(train_x,test_x,5)
```

```
lda_accuracy = knn_classification(reduced_train, train_y, reduced_test, test_y, 1)
print("\nAccuracy is: " ,lda_accuracy)
```

```
Accuracy is:  0.945
```

- Comparing between PCA results and LDA results:

We noticed that the accuracy of PCA is less than LDA so LDA is better, this is because LDA uses the data label of the dataset.

6. Classifier Tuning

```
[18] # Perform K-NN Classification
     def knn_classification(train_x, train_y, test_x, test_y, k):
       knn = KNeighborsClassifier(n_neighbors=k)
       train_y = np.ravel(train_y)
       knn.fit(train_x, train_y)
       y_pred = knn.predict(test_x)
       print_correction(y_pred, test_y)
       accuracy = accuracy_score(test_y, y_pred)
       return accuracy
```

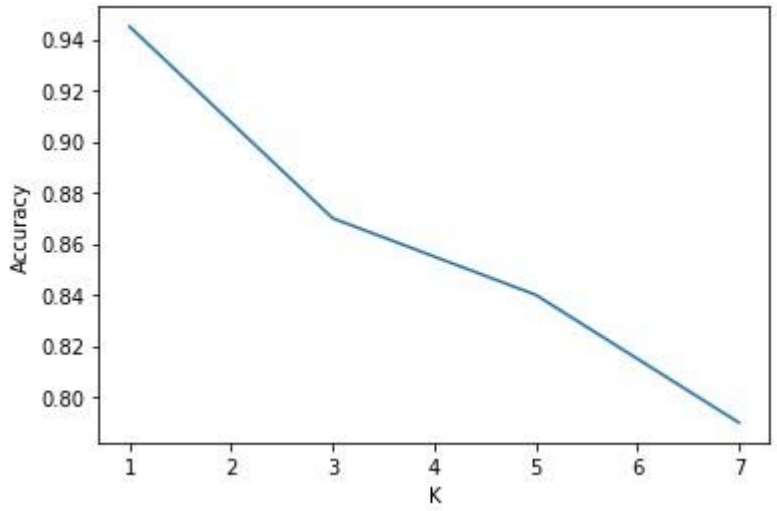
This code defines a function knn_classification that performs k-nearest neighbors (KNN) classification on a given dataset. The function takes the following arguments: train_x, train_y, test_x, test_y and k which is the number of nearest neighbors to consider for classification. The function first creates a KNN classifier object with k neighbors, and fits it to the training data and labels using the fit method. Then, it uses the trained classifier to predict the labels for the test data using the predict method. The predicted labels and the true test labels are then printed and the classification accuracy is calculated using the accuracy_score function from scikit-learn. Finally, the accuracy is returned as output from the function.

```
[19] def plot_accuracy(k, accuracy, title):
       plt.plot(k, accuracy, label=title)
       plt.xlabel('K')
       plt.ylabel('Accuracy')
       plt.show()
```

This function takes in the number of neighbors (k), the corresponding accuracy values, and a title for the plot. It then plots the accuracy values against the number of neighbors (k) on a line graph and labels the axes accordingly. Finally, it displays the graph using plt.show(). This function is likely used to visualize the performance of the k-NN classifier with different values of k.

```
def lda_accuracy_plot_knn(train_x, test_x):
  k = [1, 3, 5, 7]
  lda_accuracies = []
  lda_train_x, lda_test_x = LDA(train_x, test_x)
  for i in k:
    lda_accuracy = knn_classification(lda_train_x, train_y, lda_test_x, test_y, i)
    lda_accuracies.append(lda_accuracy)
  plot_accuracy(k, lda_accuracies, 'LDA')
  return lda_accuracies
```

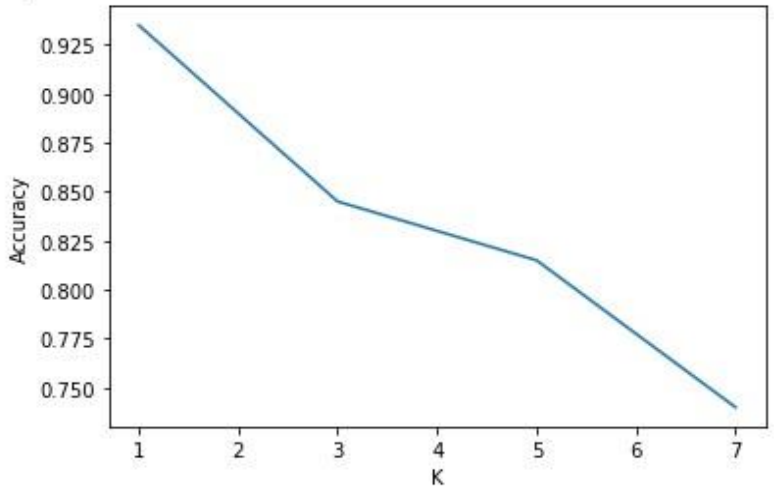
This code defines a function lda_accuracy_plot_knn that takes in the training and testing data for LDA projection, and plots the accuracy scores for different values of k (number of nearest neighbors) using KNN classification.



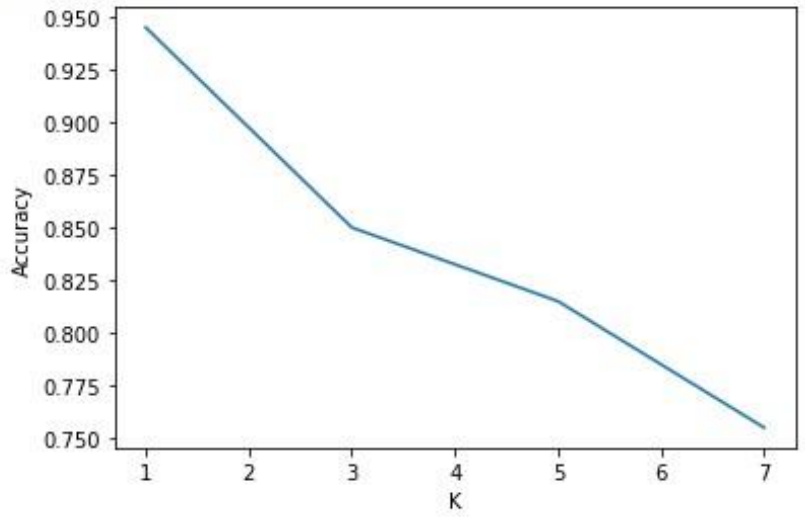

```
[8]  def pca(train_x,test_x,train_y,test_y):
       eigenValues,eigenVectors = compute_eigen(train_x)
       alpha =[0.8,0.85,0.9,0.95]
       k = [1, 3, 5, 7]
       x=[]
       for a in alpha:
         pca_accuracies =[]
         x=[]
         for i in k :
          p = projection_matrix(eigenValues,eigenVectors,a)
          reduced_train_x = np.dot(train_x,p)
          reduced_test_x = np.dot(test_x,p)
          acc =knn_classification(reduced_train_x, train_y, reduced_test_x, test_y, i)
          pca_accuracies.append(acc)
          x.append(i)

         print("Alpha= ",a)
         plot_accuracy(x,pca_accuracies, 'PCA')
```

This code performs Principal Component Analysis (PCA) on the training and test data, where the train_x and test_x are the input data matrices for training and testing, respectively. It computes the eigenvalues and eigenvectors of the covariance matrix of the training data using the compute_eigen function. The code then specifies a list of alpha values, which represent the proportion of the total variance that should be retained in the PCA projection. It also specifies a list of k values, which are the number of neighbors to consider in the k-NN algorithm. The projection_matrix function computes the projection matrix based on the specified alpha value and the computed eigenvectors and eigenvalues. It then uses the projection_matrix to project the training and test data into the reduced-dimensional space. For each alpha value and k value combination, the code calls the knn_classification function to perform k-NN classification on the projected data and compute the classification accuracy. The accuracies are stored in the pca_accuracies list. Finally, the plot_accuracy function is called to create a plot of the accuracy values for each alpha value and each k value. The plot shows how the accuracy changes as the number of neighbors changes and as the proportion of retained variance changes.

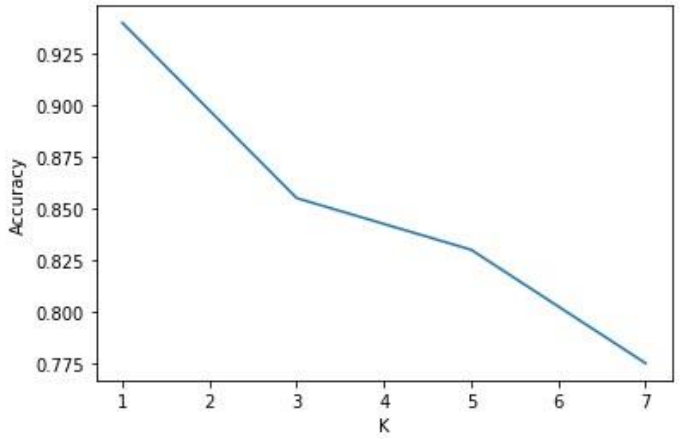
Alpha= 0.95



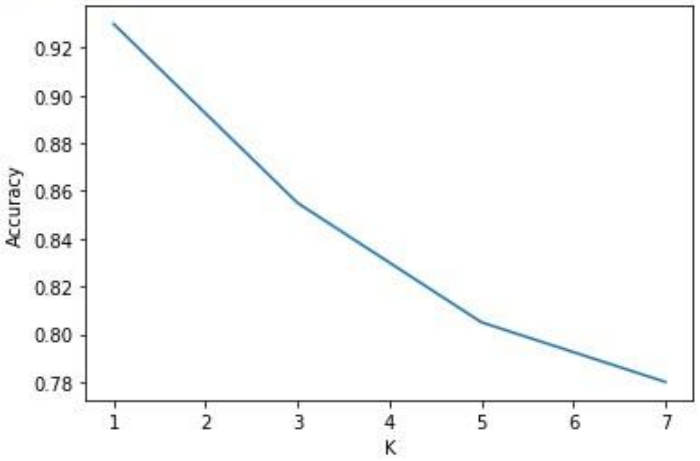
Alpha= 0.9



Alpha= 0.85



Alpha= 0.8



7. Compare vs Non-Face Images

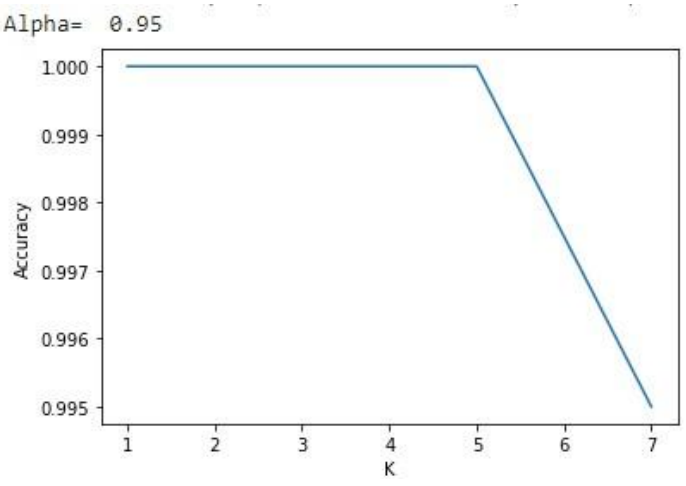
```
[24] # !wget https://drive.google.com/file/d/1Zhtf20Ylwk4RKANcczpzFZFJ2mZxC7ay/view?usp=sharing
     with zipfile.ZipFile('/content/non-face.zip', 'r') as zip_ref:
         zip_ref.extractall('/content/non-face')
```

This code snippet is for extracting the non faces dataset.

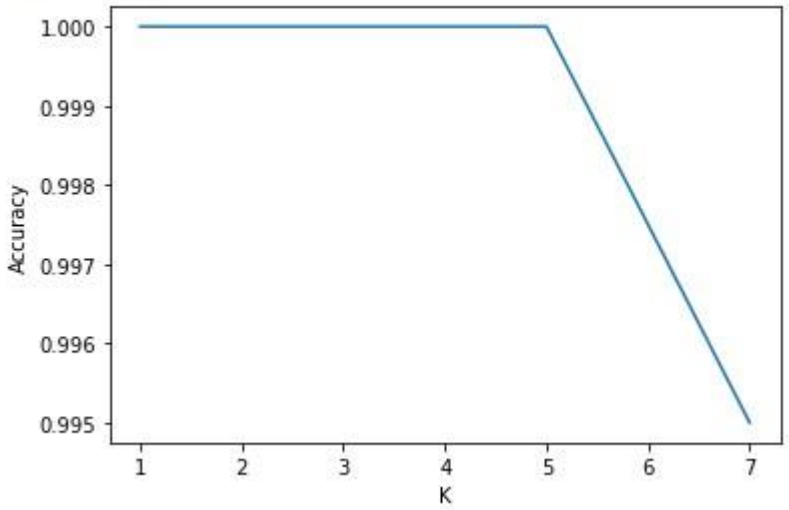
```
[25] # Extracting the Non Faces Dataset
     non_face_D = []
     non_face_y = []
     path = '/content/non-face'
     folders = os.listdir(path)
     for folder in folders:
         files = os.listdir(path + '/' + folder)
         for file in files:
           image = Image.open(path + '/' + folder + '/' + file)
           image.show()
           image_array = np.array(image)
           image_converted = np.resize(image_array, (10304))
           non_face_D.append(image_converted)
           non_face_y.append('non-face')
```

This code snippet extracts the Non Faces Dataset from the directory "/content/non-face". It first initializes two empty lists, non_face_D and non_face_y, to hold the image data and labels respectively. It then loops through each folder in the directory and for each folder, loops through each file. For each file, it opens the image file using the Pillow library's Image module, resizes the image to a 10304-dimensional vector, appends this vector to the non_face_D list, and appends the label 'non-face' to the non_face_y list. Finally, it displays each image as it is processed.

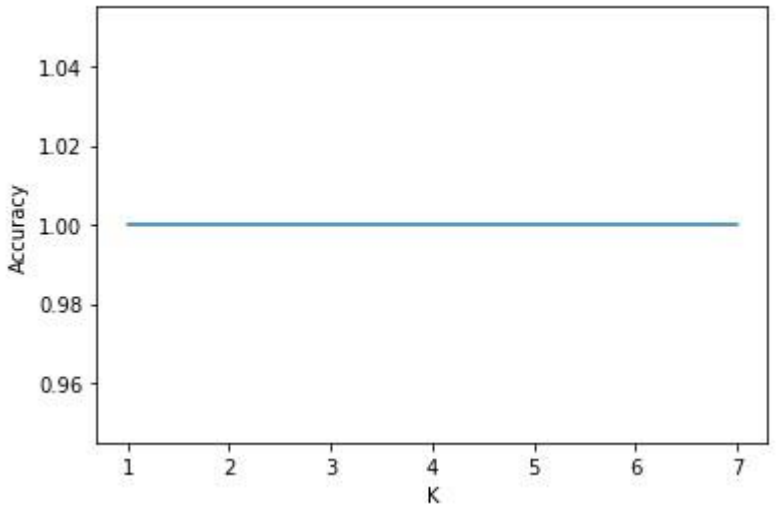
```
[27] # Computing The PCA For The Newly Combined Data
     combined_train_x = train_x[:100] + non_face_D[:100]
     train_y = new_y + non_face_y[:100]
     combined_test_x = test_x[:100] + non_face_D[100:200]
     test_y = new_y + non_face_y[100:200]
     pca(combined_train_x, combined_test_x, train_y, test_y)
```



Alpha= 0.9



Alpha= 0.85




```
[26] # Required Data
     accuracies = []
     no_non_faces = [100, 150, 200, 250, 300]
     new_y = ['face' for i in range(100)]

     for i in no_non_faces:
       combined_train_x = train_x[:100] + non_face_D[:i]
       train_y = new_y + non_face_y[:i]
       combined_test_x = test_x[:100] + non_face_D[i:i * 2]
       test_y = new_y + non_face_y[i:i * 2]
       accuracies.append(lda_accuracy_plot_knn(combined_train_x, combined_test_x))
     plt.plot(no_non_faces, accuracies, label='Number of Non Faces vs Accuracy')
     plt.xlabel('Number of Non Faces vs Accuracy')
     plt.ylabel('Accuracy')
     plt.show()
```

The code evaluates the accuracy of a k-Nearest Neighbors (k-NN) classifier with Linear Discriminant Analysis (LDA) on a dataset of face images. The dataset is split into a training set and a test set. The for loop iterates over the values in the no_non_faces list, which determines the number of non-face images to include in the training set. Inside the loop, a new training set and test set are constructed by concatenating face images from train_x and test_x respectively with a subset of non-face images from non_face_D and non_face_y. Then, the lda_accuracy_plot_knn function is called with the concatenated training set and test set as well as a parameter of 5 for the k value in k-NN. The resulting accuracy is appended to the accuracies list. Finally, a plot is generated using matplotlib, which shows the relationship between the number of non-face images in the training set and the accuracy of the k-NN classifier.

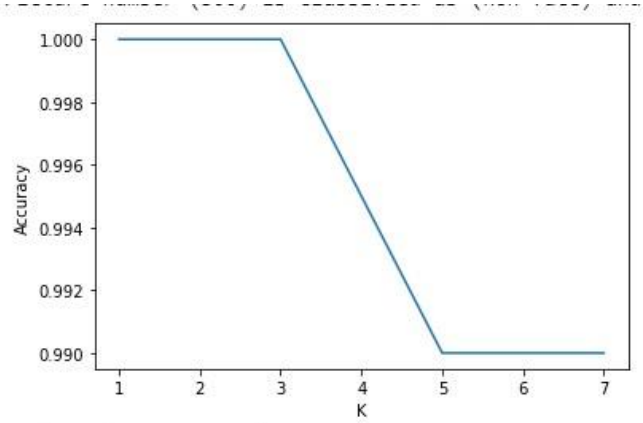
I. Show failure and success cases.

```
Picture number (199) is classified as (face) and is actually (face).
Picture number (200) is classified as (face) and is actually (face).
Picture number (201) is classified as (non-face) and is actually (non-face).
Picture number (202) is classified as (non-face) and is actually (non-face).
Picture number (203) is classified as (non-face) and is actually (non-face).
Picture number (204) is classified as (non-face) and is actually (non-face).
Picture number (205) is classified as (non-face) and is actually (non-face).
Picture number (206) is classified as (non-face) and is actually (non-face).
Picture number (207) is classified as (non-face) and is actually (non-face).
Picture number (208) is classified as (non-face) and is actually (non-face).
Picture number (209) is classified as (non-face) and is actually (non-face).
Picture number (210) is classified as (non-face) and is actually (non-face).
Picture number (211) is classified as (non-face) and is actually (non-face).
Picture number (212) is classified as (non-face) and is actually (non-face).
Picture number (213) is classified as (non-face) and is actually (non-face).
Picture number (214) is classified as (non-face) and is actually (non-face).
Picture number (215) is classified as (non-face) and is actually (non-face).
Picture number (216) is classified as (non-face) and is actually (non-face).
Picture number (217) is classified as (non-face) and is actually (non-face).
Picture number (218) is classified as (non-face) and is actually (non-face).
Picture number (219) is classified as (non-face) and is actually (non-face).
Picture number (220) is classified as (non-face) and is actually (non-face).
Picture number (221) is classified as (face) and is actually (non-face).
!! Classification error !!
Picture number (222) is classified as (non-face) and is actually (non-face).
Picture number (223) is classified as (non-face) and is actually (non-face).
Picture number (224) is classified as (non-face) and is actually (non-face).
Picture number (225) is classified as (non-face) and is actually (non-face).
Picture number (226) is classified as (non-face) and is actually (non-face).
```

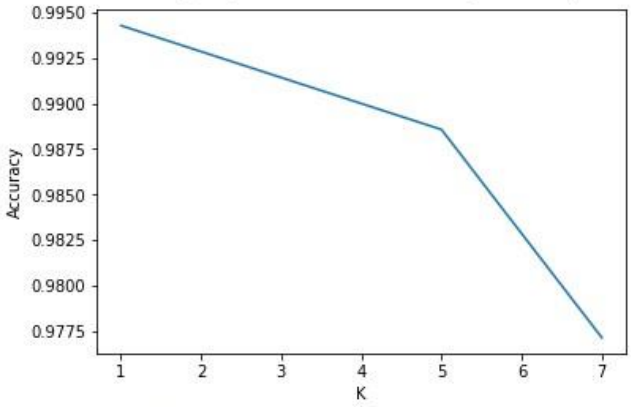
II. How many dominant eigenvectors will you use for the LDA solution?

The number of the dominant eigenvectors used is 39.

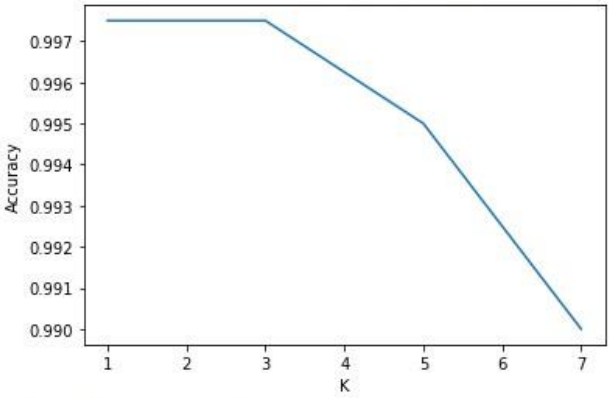
III. Plot the accuracy vs the number of non-faces images while fixing the number of face images.



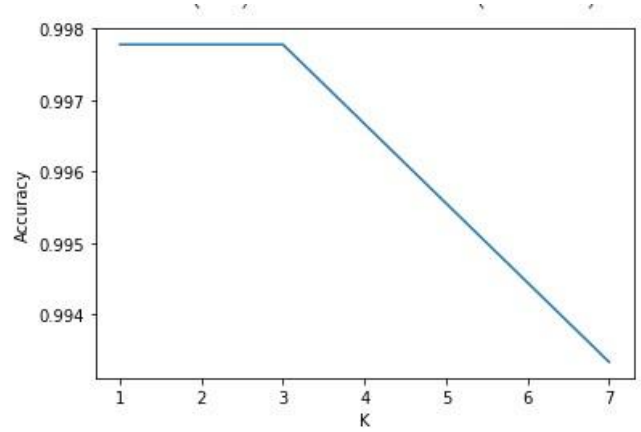
Number of Non Faces is 100



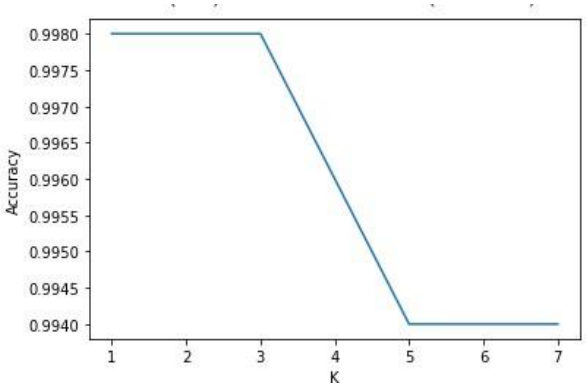
Number of Non Faces is 150

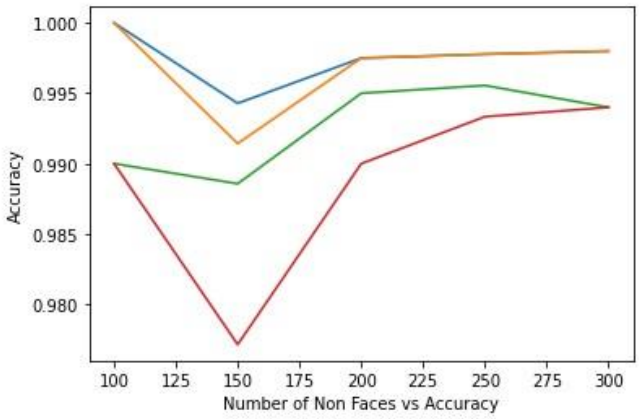


Number of Non Faces is 200



Number of Non Faces is 250





IV. Criticize the accuracy measure for large numbers non-faces images in the training data.

As the number of non faces images increases the accuracy measure increases.

8. Bonus:

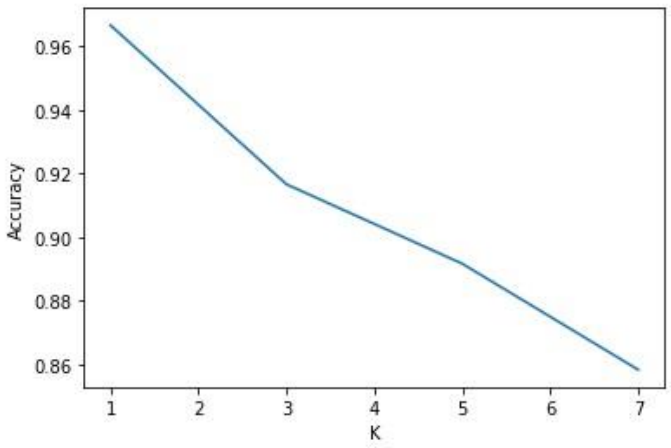
1- Use different Training and Test splits. Change the number of instances per subject to be 7 and keep 3 instances per subject for testing. compare the results you have with the ones you got earlier with 50% split.

```
[ ]  # Different Training and Test splits
     test_x=[]
     test_y=[]
     train_x=[]
     train_y=[]
     def split73():
      for i in range(400):
        if i % 3 == 0 and i % 10 != 0:
          test_x.append(dataset[i])
          test_y.append(y[i])
        else:
          train_x.append(dataset[i])
          train_y.append(y[i])
     split73()
```

```
[ ]  # Computing The PCA
     pca(train_x,test_x,train_y,test_y)
```

```
     # Computing The LDA
     lda_accuracy_plot_knn(train_x, test_x, 7)
```

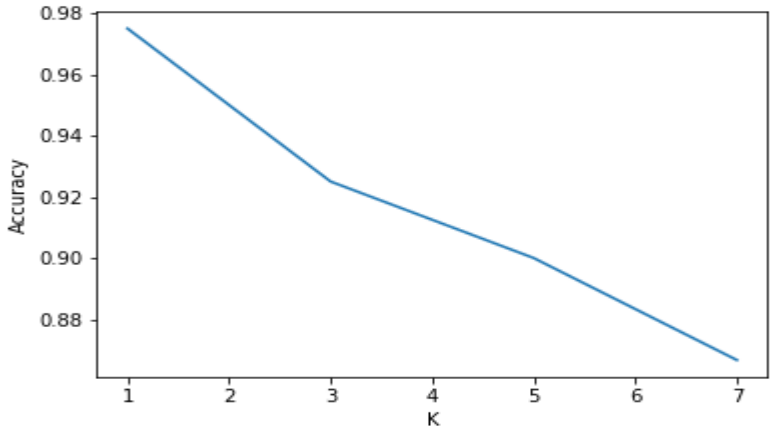
- After applying LDA algorithm with this different training testing we got:



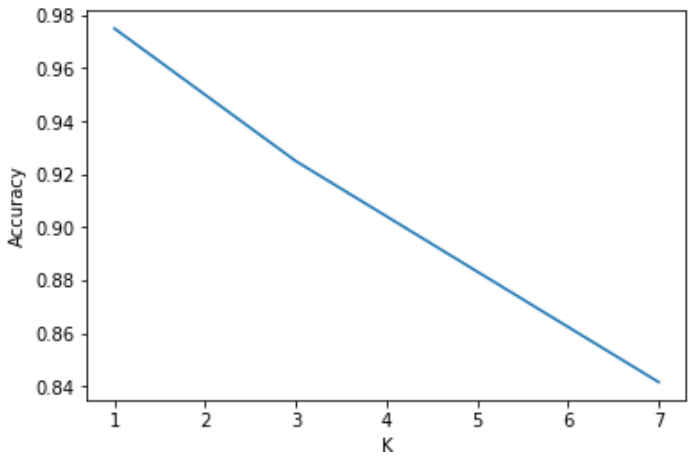
It is noticed that the accuracy of LDA in the new training is much better then 50% training and that because there is 70% of the data in the training set(much data to be classified) so it can be classified well.

- After applying PCA algorithm with this different training testing we got:

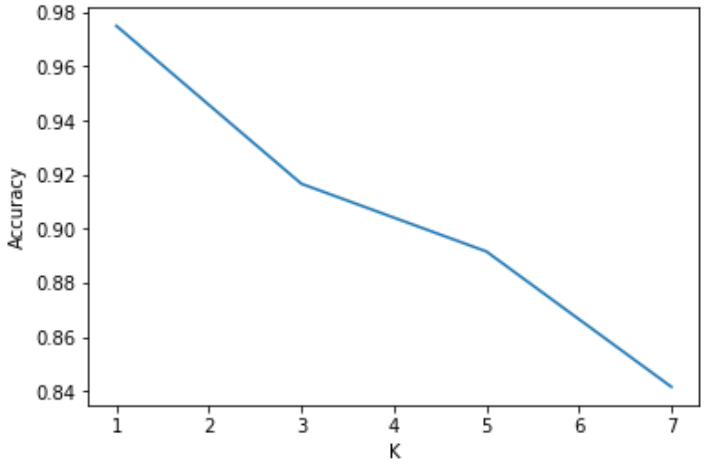
```
for k = 7 accuracy = 0.8666666666666667
```




```
for k = 7 accuracy = 0.8416666666666667
```




```
for k = 7 accuracy = 0.8416666666666667
```



The accuracy of new pca training increased from .75 to .84 in average.

It is noticed that the accuracy of PCA in the new training is much better then 50% training and that because there is 70% of the data in the training set(much data to be classified) so it can be classified well.