Pattern Recognition

Face Recognition System

Prepared by

Mennatullah Ibrahim Mahmoud **6221**

Abdelrahman Salem **6309**

Reem Abdelhaleem **6114**

Supervised by **Dr. Marwan Torki**

Overview:

The first part of this project aims to train the model to identify a person out of a selection of 40 people. The dataset is a collection of 400 black and white images for 40 people, 10 photos for each person. The dimensions of each image are 92x112.

The dataset is split into two sets, a training set, and a test set. PCA and LDA algorithms are used to project the dataset and a K-NN classifier is used to predict the test set.

The second part of this project solves another classification problem faces vs non-faces. Other images of non-faces are added to the existing dataset and the same steps are repeated on two classes instead of 40 classes.

1. Reading the dataset on Colab:

Reading from Google Drive

```
[ ]  # Code to read file into Colaboratory:
     ! pip install -U -q PyDrive
     from pydrive.auth import GoogleAuth
     from pydrive.drive import GoogleDrive
     from google.colab import auth
     from oauth2client.client import GoogleCredentials
     # Authenticate and create the PyDrive client.
     auth.authenticate_user()
     gauth = GoogleAuth()
     gauth.credentials = GoogleCredentials.get_application_default()
     drive = GoogleDrive(gauth)
```

Upload the dataset

```
[ ]  # upload CSV file from google drive
     link = 'https://drive.google.com/file/d/1Mb0WeDv0s1Y3K7f1-3Qp7iFUfFw6K_GF/view?usp=sharing' # The shareable link
     # to get the id part of the file
     id = link.split("/")[-2]
     downloaded = drive.CreateFile({'id':id})
     downloaded.GetContentFile('archive.zip')
```

The zip file was uploaded on google drive and a connection between colab notebook and google drive is set and the zip file is downloaded.

The verification is used to connect:

4/1AX4XfWhS4_1SD7V1XXyhBkIOA7pr_a3SLzZwuB4Y0devlZwQQRZfUWJPNl0

Unzipping

```
[ ]  ! unzip archive.zip -d faces
```

The file content is unzipped into folder names “faces” which will be used later to load the dataset into a numpy array.

The necessary imports are then made which will be used throughout the project.

Imports

```
[4]  import numpy as np
     import cv2 as cv
     import os
     #from google.colab.patches import cv2_imshow # for image display
     #from skimage import io
     #from PIL import Image
     import matplotlib.pylab as plt
     import matplotlib.pyplot as plt1
     import math
     from sklearn.neighbors import KNeighborsClassifier
     from sklearn.metrics import accuracy_score
     from tabulate import tabulate
     from prettytable import PrettyTable
```

Vectoring the Images

```
[ ]  def vectorizing(imagename):
       img=cv.imread(imagename)
       result = img[:, :, 0]
       img_np = np.asarray(result)
       img_np= img_np.flatten()
       return img_np
```

A function is defined to transform the image to a vector. First step is removing the 3rd dimension from the array which was read using ***cv.imread*** function. The image is returned from the first line as an array of dimensions (92,112,3). The 3rd dimension represents the RGB channel values but since the images are black and white, so in this case, it represents redundant data. The second line in this function removes the third dimension which makes the dimensions of the image now (92,112) and then this array is transformed into a ***numpy*** array which will help in flattening the image into a vector in one step using ***flatten*** function. Now the image vector is returned ready for further processing.

Read the dataset

```
[ ]  directory = os.fsencode("/content/faces/")
     dataset = []
     for file in os.listdir(directory):
           filename = os.fsdecode(file)
           filename= "/content/faces/"+ filename +"/"
           filenameV= os.fsencode(filename)
           for image in os.listdir(filenameV):
             imagename = os.fsdecode(image)
             imagename= filename+imagename
             dataset.append(vectorizing(imagename))
```

In this step, the images are read from the directory that the unzipped file went in. The os library is used here to encode the directory path name and decode it to use it to read all the images in one loop. The images from the files are read sequentially. Every 10 images in one folder are read and vectorized then appended to the dataset.

Generating the labels vector

```
[ ]  labels = []
     for i in range(1,41):
       for j in range(10):
         labels.append(i)
```

Since the images are read sequentially, the label vector is simply generated by repeating each number from 1 to 40 ten times (number of samples per class).

To make the processing easier every array used so far is then changed to a numpy array.

Convert to NumPy

```
[ ]  dataset = np.array(dataset)
     labels = np.array(labels)
     print(f"Length of dataset {len(dataset)} and type {type(dataset)}")
     print(f"Length of labels {len(labels)} and type {type(labels)}")
```

```
Length of dataset 400 and type <class 'numpy.ndarray'>
Length of labels 400 and type <class 'numpy.ndarray'>
```

2. Splitting the data:

Since the data is read sequentially, the 50%-50% splitting is easily done using even and odd indices.

Dataset Splitting

```
[ ]  train_features = []
     train_labels = []

     test_features = []
     test_labels = []

     # even indices for train and odd for test
     for i in range(0,400,2):
       train_features.append(dataset[i])
       train_labels.append(labels[i])
       test_features.append(dataset[i+1])
       test_labels.append(labels[i+1])
```

And as mentioned before, to make life easier the train and test set are converted to numpy arrays as well.

Convert Train and Test sets to NumPy

```
[ ]  train_features = np.array(train_features)
     train_labels = np.array(train_labels)
     test_features = np.array(test_features)
     test_labels = np.array(test_labels)

     print(train_features.shape)
     print("-------------------------------------------------")
     print(test_features.shape)
```

```
(200, 10304)
-------------------------------------------------
(200, 10304)
```

3. PCA:

The next part is reducing the dimensions of the data using PCA algorithm. Two approaches were followed. The first approach depended on the absolute values of the eigen values as the negative sign here only represents the direction of the vector in the new space.

PCA

```
[ ]  def pca(dataset,alpha):
       mean=dataset.mean(axis=0)
       Z=dataset-mean
       cov=np.cov(Z,rowvar=False,bias=False)
       eigen_values,eigen_vectors=np.linalg.eigh(cov)
       r=0
       sum=0.0
       n=len(eigen_values)
       eigen_values=np.abs(eigen_values)
       eigen_values_sum=math.fsum(eigen_values)
       ind=(eigen_values.argsort())
       sorted_eigenvectors=eigen_vectors[:,ind]
       sorted_eigenvalues=eigen_values[ind]
       for i in reversed(range(n)):
         sum=sum+sorted_eigenvalues[i]
         tolerance=float(sum/eigen_values_sum)
         if tolerance>=alpha:
           r=i
           break
       P=sorted_eigenvectors[:,r:n]
       return P
```

The algorithm first receives the train set of the data and the alpha which represents the tolerance specified. In the dimensionality reduction, some data is lost and the tolerance represents how much we care about the accuracy of the reduced data. The first step is computing the mean to center the dataset by subtracting the mean from each attribute. The covariance matrix is then computed using ***cov*** function for efficiency. However, we can also compute it using the dot product between the transpose centered dataset and itself and dividing the product by n-1 where n is the number of samples. The n-1 is the form followed here but in some other versions, the product is divided by n. This is related to statistics which we will not focus on in this part. This part is

denoted by the attribute ***Bias*** in the function ***cov***, if the bias is true, it resembles dividing by n and if it is false, it resembles dividing by n-1.

Moving on to the next part, the eigen values and vectors are then calculated by using ***np.linalg.eigh*** which returns the eigen values ascendingly given a symmetric matrix as input and the eigen vectors corresponding to the eigen values in the same order.

As mentioned before, in this approach we focused only on the magnitude of the eigen values and not its sign. To achieve this, the absolute function is performed on the eigen values array and the indices of sorting them has been returned to a variable which will help us sort both the eigen values and eigen vectors. The sum of the eigen values is also calculated and we start with the largest eigen value which lies in the end of the array now. The largest eigen values corresponds to the most dominant eigen vectors, therefore we only need the most dominant r eigen vectors where the sum of their corresponding r eigen values over the sum of all the eigen values are at least equal or higher than the tolerance specified by the alpha. Once we reach this criterion, we break out of the loop and return the r eigen vectors to use this later as the P matrix for projecting the data.

We made a very simple function for projecting the data which only takes the dataset and the P matrix. The dataset is centered and then a dot product between the dataset and the P matrix simply returns the projected matrix.

Projection

```
[ ]  def projection(features,P):
       features=features-features.mean(axis=0)
       projected_features=np.dot(features,P)
       return projected_features
```

A simple classifier KNN classifier is used here as well to predict the dataset and calculate the accuracy. The sklearn KNN classifier is used here for efficiency.

K-NN Classifier

```
[ ]  def knn(train_features,train_labels,test_features,test_labels,k):
       clf = KNeighborsClassifier(n_neighbors=k)
       clf.fit(train_features,train_labels)
       predicition  = clf.predict(test_features)
       acc = accuracy_score(predicition,test_labels)
       return acc
```

A function for classifier tuning is also made to make the tuning easier for each alpha used.

The function uses the given alpha along with the projected data to calculate the accuracy at K=1,3,5,7 and add an entry to the accuracy table which we will plot later.

```
[ ]  def classifierTuningPCA (alpha,projected_train_features,train_labels,projected_test_features,test_labels,accuracyTable):
       accuracies=[]
       for k in range (1,8,2):
         accuracies.append(knn(projected_train_features,train_labels,projected_test_features,test_labels,k))
       accuracyTable.add_row([alpha,accuracies[0],accuracies[1],accuracies[2],accuracies[3]])
       return accuracies
```

The K here represents the number of nearest neighbors which will help in determining the class of the unknown sample.

To understand this better, the KNN algorithm will be explained briefly in the following part.

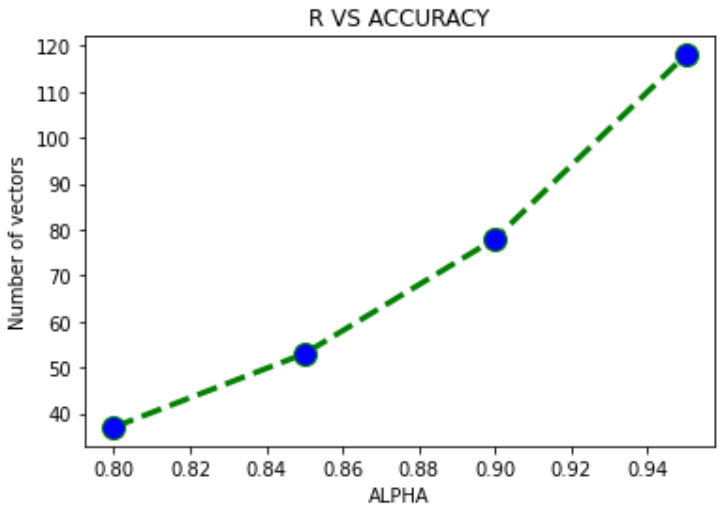
The KNN classifier simply fits the projected train data in a graph along with the class label for each sample. Then it simply takes the unknown sample and adds it to the graph. It then looks for the nearest K samples in this plane. It determines that using the Euclidean distance in our case. Then after this, it determines the class label of this sample using voting from the K samples nearest to it. In case of a tie, it breaks the tie automatically using the ordering of the data. This method is implemented differently in other algorithms, but they mostly depend on some random criteria and tests which results in better accuracy. The K is usually taken as an odd number, and it is preferred to not pick the K as a multiple of the classes number to avoid relying on the tie breaker algorithm.

Performing PCA on the dataset

```
[ ]  alphas = [0.8, 0.85, 0.9, 0.95]
     accuracies=[]
     r=[]
     accuracyTable = PrettyTable(["alpha\KNN", "1", "3", "5", "7"])
     accuracies_tuning=[]
     for i in range(4):
       projection_matrix = pca(train_features,alphas[i])
       projected_train_features = projection(train_features,projection_matrix)
       projected_test_features = projection(test_features,projection_matrix)
       accuracies.append(knn(projected_train_features,train_labels,projected_test_features,test_labels,1))
       accuracies_tuning.append(classifierTuningPCA(alphas[i],projected_train_features,train_labels,projected_test_features,test_labels,accuracyTable))
       r.append(projection_matrix.shape[1])
```

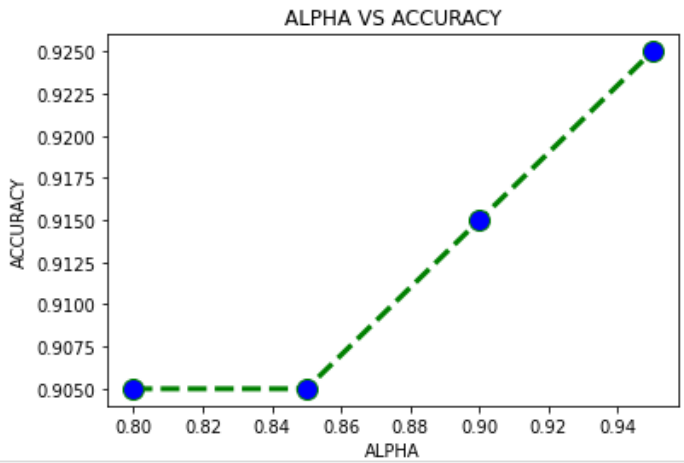
In this part, we simply use the function defined earlier to calculate the projection matrix of the train data and use the projection function to calculate the projected train and test data. After this, we calculate the accuracy at different alphas and different K for analysis.

```
plt.plot(alphas, r,color='green', linestyle='dashed', linewidth = 3,
         marker='o', markerfacecolor='blue', markersize=12)
plt.xlabel('ALPHA')
plt.ylabel('Number of vectors')
plt.title('R VS ACCURACY')
plt.show()
```



Plotting Relation between Alpha and Accuracy

```
[ ]  plt.plot(alphas, accuracies,color='green', linestyle='dashed', linewidth = 3,
             marker='o', markerfacecolor='blue', markersize=12)
     plt.xlabel('ALPHA')
     plt.ylabel('ACCURACY')
     plt.title('ALPHA VS ACCURACY')
     plt.show()
```



As expected, as the alpha increases, the number of principle components increase which in process increases the accuracy of the system.

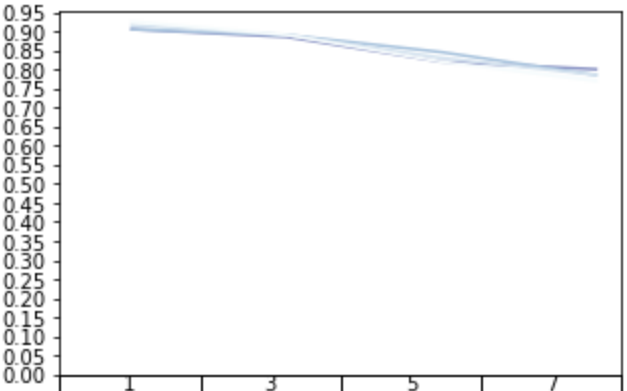
Classifier Tuning with K=1,3,5,7

```
[ ]  print(accuracyTable)
```

```
+-----------+-------+-------+-------+-------+
| alpha\KNN |   1   |   3   |   5   |   7   |
+-----------+-------+-------+-------+-------+
|    0.8    | 0.905 | 0.885 |  0.82 |  0.8  |
|    0.85   | 0.905 |  0.89 | 0.845 | 0.785 |
|    0.9    | 0.915 |  0.89 |  0.83 | 0.785 |
|    0.95   | 0.925 |  0.89 |  0.82 |  0.77 |
+-----------+-------+-------+-------+-------+
```

Plotting relation between K and accuracy at each alpha

```
[ ]  data = accuracies_tuning
     columns = ('1', '3', '5', '7')
     rows = ['0.8', '0.85', '0.9', '0.95']
     values = np.arange(0, 1, 0.05)
     # Get some pastel shades for the colors
     colors = plt.cm.BuPu(np.linspace(0, 0.5, len(rows)))
     colors = colors[::-1]
     index = np.arange(len(columns)) + 0.3
     bar_width = 0.4
     n_rows = len(data)

     for row in range(n_rows):
         plt.plot(index, data[row], bar_width, color=colors[row])

     plt.table(cellText=data, rowLabels=rows, rowColours=colors, colLabels=columns, loc='bottom')

     plt1.subplots_adjust(left=0.25, bottom=0.25)
     plt1.ylabel("")
     plt1.yticks(values)
     plt1.xticks([])
     plt1.title('')
```



	1	3	5	7
0.8	0.905	0.885	0.82	0.8
0.85	0.905	0.89	0.845	0.785
0.9	0.915	0.89	0.83	0.785
0.95	0.925	0.89	0.82	0.77

As seen here, as the K increases, the accuracy decreases. This is expected since the number of samples for each class is only 5. The more samples we consider, the less accuracy we will get as we are sure that we are casting a vote that samples out of the correct class will participate in. There is a way to solve this using ununiform weights in the voting process. This means that the closer the sample is to the unknown sample, the higher weight it will have in the voting process.

```
[ ]  def pca2(dataset,alpha):
       mean=dataset.mean(axis=0)
       Z=dataset-mean
       cov=np.cov(Z,rowvar=False,bias=False)
       eigen_values,eigen_vectors=np.linalg.eigh(cov)
       r=0
       sum=0.0
       n=len(eigen_values)
       eigen_values_sum=math.fsum(eigen_values)
       for i in reversed(range(n)):
         sum=sum+eigen_values[i]
         tolerance=float(sum/eigen_values_sum)
         if tolerance>=alpha:
           r=i
           break
       P=eigen_vectors[:,r:n]
       return P
```

The second approach takes the largest r eigen vectors, but it does not ignore the negative sign of the eigen values. It is not preferred as it is against the geometric meaning of the eigen vectors in space. However, it resulted in the same accuracy as the first approach in this case.

4. **LDA**

LDA

```
[ ]  def lda(train_features,samples):
       Stotal=np.zeros(shape=(train_features.shape[1],train_features.shape[1]))
       overallmean=train_features.mean(axis=0)
       sum=np.zeros(shape=(train_features.shape[1],train_features.shape[1]))
       for i in range(40):
         c=train_features[samples*i:samples*i+samples]
         mean=c.mean(axis=0)
         c=c-mean
         Stotal+=np.dot(c.transpose(),c)
         meandiff=(mean-overallmean).reshape(train_features.shape[1], 1)
         sum+=samples*np.dot(meandiff,meandiff.T)
       eigen_values, eigen_vectors = np.linalg.eigh(np.dot(np.linalg.inv(Stotal),sum))
       ind=(np.abs(eigen_values).argsort())
       sorted_eigenvectors=eigen_vectors[:,ind]
       sorted_eigenvalues=eigen_values[ind]
       U=sorted_eigenvectors[:,-39:]
       return U
```

The LDA function receives the train set and the samples number. In a more general case, the train labels set is sent instead of the samples number to determine the samples number of each class. However, since the number of samples is fixed here and the samples of each class in the dataset is sequential, the samples number is sufficient here. The algorithm first calculates the overall mean of the dataset which will be used to calculate the B matrix which is denoted here by sum variable. The sum and Stotal which represents the S matrix is first initialized by zeros. Then a simple for loop is used to calculate the mean of each class, centering the class, calculating the $nk(u_c\text{-overall mean})(u_c\text{-overall mean})^T$ term and adding to the sum

variable and also calculating the S matrix of this class using the dot product between the transpose of the centered data of this class and itself and adding it to the Stotal matrix. The reshaping of the mean and overall mean is an important part to compute the B matrix correctly. If this step is not performed, the result of the dot product will be a scalar value instead of a matrix.

Finally, $Stotal^{-1}$.sum is calculated and the eigen values and vectors are computed.

Similarly, we can follow one of the two approaches here, either depend on the absolute value of the eigen values or depend on the ordering done by the eigh function. In this approach, we follow what we believe is more accurate and sort the eigen values according to their magnitude.

After this, the most dominant 39 vectors are taken as the projection matrix U. We use 39 separators as we have 40 classes, so we need n-1 (39) vectors to differentiate between classes.

We use the U matrix to project the datasets and in a similar way to the PCA, we use the KNN function to predict the test set and calculate the accuracy. We also preform the classifier tuning using K=1,3,5,7 and plot the results.

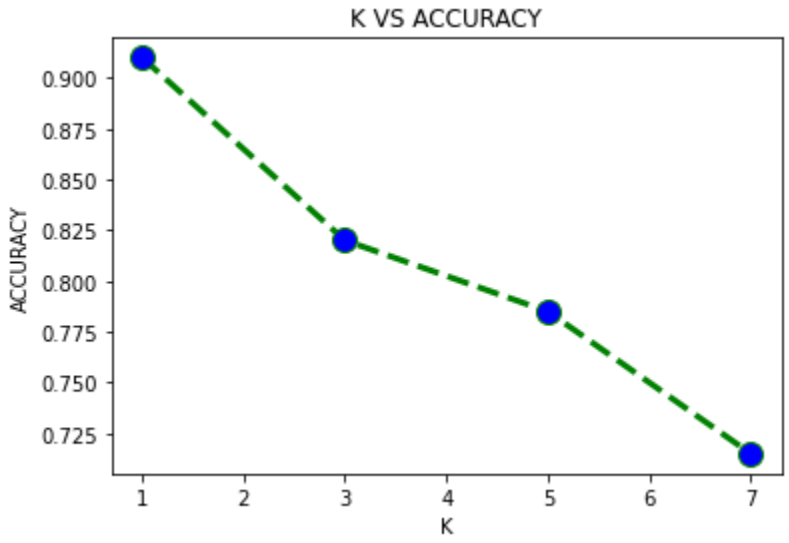
Performing LDA on the dataset

```
[ ]  U=lda(train_features,5)
     projected_train_features_lda=projection(train_features,U)
     projected_test_features_lda=projection(test_features,U)
     accuracy=knn(projected_train_features_lda,train_labels,projected_test_features_lda,test_labels,1)
     print(accuracy)
     print(U.shape)
```

```
0.91
(10304, 39)
```

Classifier tuning for k=1,3,5 and 7

```
[ ] accuracy=[]
    k=[1,3,5,7]
    for K in k:
      accuracy.append(knn(projected_train_features_lda,train_labels,projected_test_features_lda,test_labels,K))
    plt.plot(k, accuracy,color='green', linestyle='dashed', linewidth = 3,
             marker='o', markerfacecolor='blue', markersize=12)
    plt.xlabel('K')
    plt.ylabel('ACCURACY')
    plt.title('K VS ACCURACY')
    plt.show()
```



As expected, the accuracy also decreased as we increased the number of samples. As explained before, the K=1 here is the best option since the number of samples is limited (only 5) per class. The worst accuracy will be at K=200, since the algorithm will have a tie since 5 samples of each class (the whole data set) will cast 5 votes per class so the whole prediction will be dependent on the tie breaker algorithm.

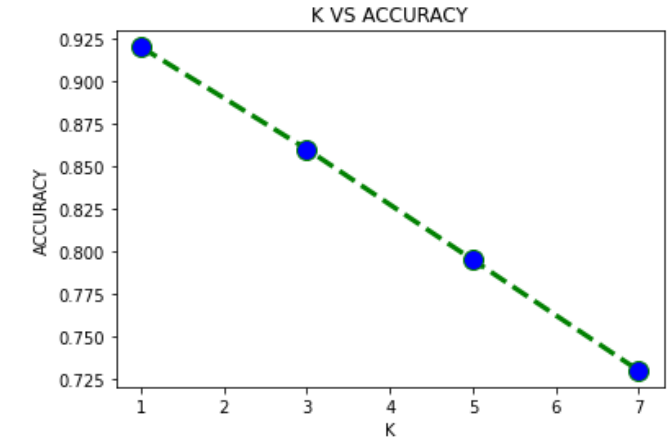
We can also follow the other approach regarding the calculation of the eigen values which surprisingly gave a higher accuracy.

```
[ ]  def lda2(train_features,samples):
       Stotal=np.zeros(shape=(train_features.shape[1],train_features.shape[1]))
       overallmean=train_features.mean(axis=0)
       sum=np.zeros(shape=(train_features.shape[1],train_features.shape[1]))
       for i in range(40):
         c=train_features[samples*i:samples*i+samples]
         mean=c.mean(axis=0)
         c=c-mean
         Stotal+=np.dot(c.transpose(),c)
         meandiff=(mean-overallmean).reshape(train_features.shape[1], 1)
         sum+=samples*np.dot(meandiff,meandiff.T)
       eigen_values, eigen_vectors = np.linalg.eigh(np.dot(np.linalg.inv(Stotal),sum))
       U=eigen_vectors[:,-39:]
       return U
```

Performing LDA on the dataset

```
[ ]  U=lda2(train_features,5)
     projected_train_features_lda=projection(train_features,U)
     projected_test_features_lda=projection(test_features,U)
     accuracy=knn(projected_train_features_lda,train_labels,projected_test_features_lda,test_labels,1)
     print(accuracy)
     print(U.shape)
```

```
0.92
(10304, 39)
```



In general, LDA was expected to perform better than PCA as this is a classification problem. It resulted in slightly better accuracy, but it was very close which reflects how well the eigen faces using the PCA works.

In the end, we changed the splitting ratio to 70% train set and 30% test set which is more realistic. Often it is not preferred to neglect 50% of the data as test set. This depends on the size of the dataset, but the test set is usually between 20% and 30%

It's expected to see better results here after taking into consideration an extra 20% of the data which indeed improved our model performance as expected.

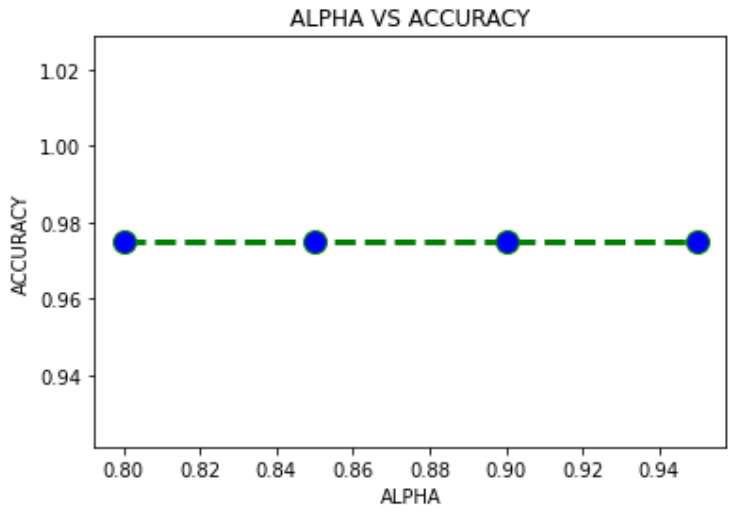
Changing Splitting Ratio from 50%-50% to 70%-30%

```
[ ]  train_features2 = []
     train_labels2 = []

     test_features2 = []
     test_labels2 = []


     for i in range(0,40):
       for j in range(0,7):
         train_features2.append(dataset[j+10*i])
         train_labels2.append(labels[j+10*i])
       for j in range(7,10):
         test_features2.append(dataset[j+10*i])
         test_labels2.append(labels[j+10*i])

     train_features2 = np.array(train_features2)
     train_labels2 = np.array(train_labels2)
     test_features2 = np.array(test_features2)
     test_labels2 = np.array(test_labels2)
```

PCA Results on the new data:

Plotting Relation between Alpha and Accuracy

```
[ ]  plt.plot(alphas, accuracies,color='green', linestyle='dashed', linewidth = 3,
             marker='o', markerfacecolor='blue', markersize=12)
     plt.xlabel('ALPHA')
     plt.ylabel('ACCURACY')
     plt.title('ALPHA VS ACCURACY')
     plt.show()
```




```
[ ]  print(r)
```

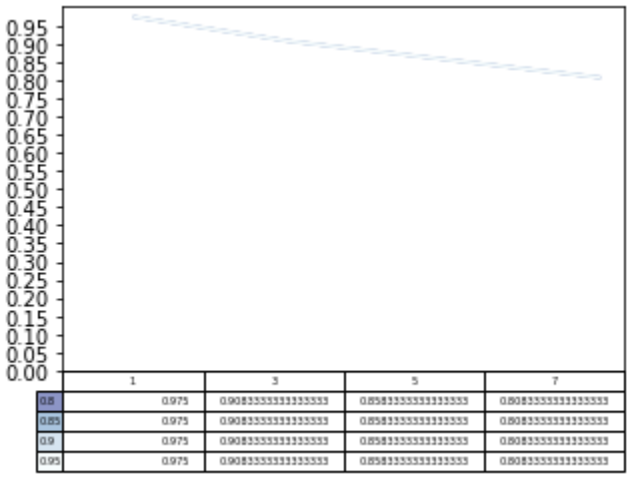
```
[118, 118, 118, 118]
```

It is worth mentioning that the accuracy in the PCA didn't change by changing the alpha and the speculation here is that it's due to the truncation error, the ratio of the sum of 118 eigen values to the total sum of eigen values was very close so it was approximated to 1.

Classifier Tuning

```
[ ]  print(accuracyTable)
```

```
+-----------+-------+--------------------+--------------------+--------------------+
| alpha\KNN |   1   |         3          |         5          |         7          |
+-----------+-------+--------------------+--------------------+--------------------+
|    0.8    | 0.975 | 0.9083333333333333 | 0.8583333333333333 | 0.8083333333333333 |
|    0.85   | 0.975 | 0.9083333333333333 | 0.8583333333333333 | 0.8083333333333333 |
|    0.9    | 0.975 | 0.9083333333333333 | 0.8583333333333333 | 0.8083333333333333 |
|    0.95   | 0.975 | 0.9083333333333333 | 0.8583333333333333 | 0.8083333333333333 |
+-----------+-------+--------------------+--------------------+--------------------+
```

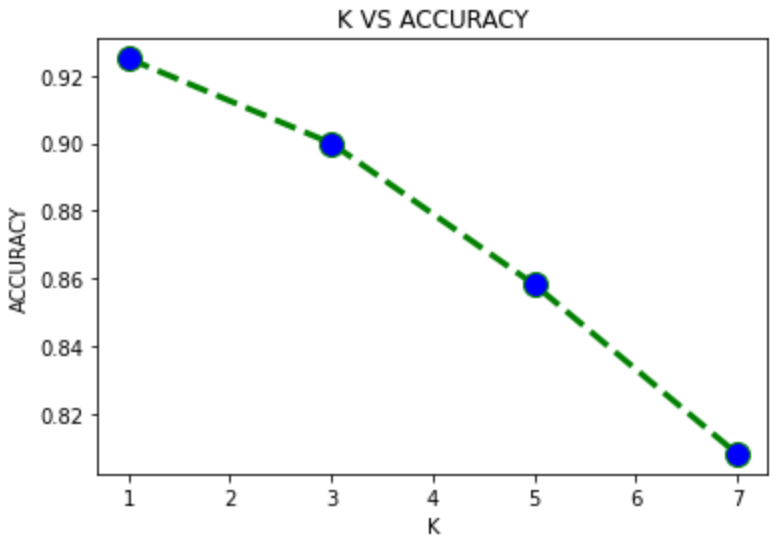


LDA performance on the new data:

Performing LDA on the new data

```
[ ]  U=lda(train_features2,7)
     projected_train_features_lda2=projection(train_features2,U)
     projected_test_features_lda2=projection(test_features2,U)
     accuracy=knn(projected_train_features_lda2,train_labels2,projected_test_features_lda2,test_labels2,1)
     print(accuracy)
     print(U.shape)
```

```
0.925
(10304, 39)
```



In the following part, we will discuss the second problem which is the faces vs non-faces problem.

Compare vs Non-Face Images

1. Reading the dataset on Colab:

```
# Code to read file into Colaboratory:
! pip install -U -q PyDrive
from pydrive.auth import GoogleAuth
from pydrive.drive import GoogleDrive
from google.colab import auth
from oauth2client.client import GoogleCredentials
# Authenticate and create the PyDrive client.
auth.authenticate_user()
gauth = GoogleAuth()
gauth.credentials = GoogleCredentials.get_application_default()
drive = GoogleDrive(gauth)
```

Upload non-faces dataset

```
link = 'https://drive.google.com/file/d/1csdpO4I9u7FGUD7CNxwEkQpIAqhiD5c_/view?usp=sharing' # The shareable link
# to get the id part of the file
id = link.split("/")[-2]
downloaded = drive.CreateFile({'id':id})
downloaded.GetContentFile('non-face.zip')
```

Upload faces dataset

```
[3] # upload CSV file from google drive
link = 'https://drive.google.com/file/d/1Mb0WeDv0s1Y3K7f1-3Qp7iFUfFw6K_GF/view?usp=sharing' # The shareable link
# to get the id part of the file
id = link.split("/")[-2]
downloaded = drive.CreateFile({'id':id})
downloaded.GetContentFile('archive.zip')
```

Both zip file was uploaded on google drive and a connection between Colab notebook and google drive is set and the zip file is downloaded.

The verification is used to connect:

Zipping the datasets

```
[4]  ! unzip non-face.zip -d non-faces
     ! unzip archive.zip -d faces
```

The content of non-faces.zip is unzipped into folder names “non-faces”, The content of archive.zip is unzipped into folder names “faces”.

Both will be used later to load the dataset into a numpy array,

The necessary imports are then made which will be used throughout the project.

Imports

```
import numpy as np
import cv2 as cv
import os
#from google.colab.patches import cv2_imshow # for image display
#from skimage import io
#from PIL import Image
import matplotlib.pylab as plt
import matplotlib.pyplot as plt1
import math
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
from tabulate import tabulate
from prettytable import PrettyTable
from PIL import Image as im
from sklearn.model_selection import train_test_split
```

Convert image to vector function

```
[ ]  def vectorizing(imagename):
       img=cv.imread(imagename)
       result = img[:, :, 0]
       img_np = np.asarray(result)
       img_np= img_np.flatten()
       return img_np
```

Apply vectorizing on the datasets

```
[7]  directory = os.fsencode("/content/faces/")
     faces_dataset = []
     for file in os.listdir(directory):
           filename = os.fsdecode(file)
           filename= "/content/faces/"+ filename +"/"
           filenameV= os.fsencode(filename)
           for image in os.listdir(filenameV):
             imagename = os.fsdecode(image)
             imagename= filename+imagename
             faces_dataset.append(vectorizing(imagename))
```

```
directory = os.fsencode("/content/non-faces/")
non_faces_dataset = []
for file in os.listdir(directory):
      filename = os.fsdecode(file)
      filename= "/content/non-faces/"+ filename +"/"
      filenameV= os.fsencode(filename)
      for image in os.listdir(filenameV):
        imagename = os.fsdecode(image)
        imagename= filename+imagename
        non_faces_dataset.append(vectorizing(imagename))
```

Generate labels vector

```
# 0--> non-face and 1-->face
labels = []
for i in range(850):
  if i < 400:
    labels.append(1)
  else:
    labels.append(0)
```

Our problem now is the classify the faces and non-faces samples in the dataset so we have only two labels to be generated, 0 for non-face sample and 1 for face sample and since we uploaded total 400 faces a 450 non faces samples in our project, 850 labels is generated.

Concatenate into one dataset

```
dataset = np.concatenate((faces_dataset,non_faces_dataset))
print(dataset.shape)
```

```
(850, 10304)
```

Then concatenate the two sets together

Splitting Function

```
def split(dataset,labels,ratio=0.3):
  train_features, test_features, train_labels, test_labels = train_test_split(dataset, labels, test_size=ratio, random_state=42,stratify=labels)
  return train_features, test_features, train_labels, test_labels
```

We have used Sklearn built in function for splitting with balanced classes as possible and equally distributed classes upon train and test sets with ratio 70% for train and 30% for test in all the upcoming operations

To record the accuracies and R satisfying alpha=0.95

```
acc_list = []
r_list = []
```

Those two sets used to store the results of accuracy and R (number of dominant eigen vectors for specific alpha) we fixed the alpha=0.95 as convention and fixed number of faces samples in the dataset and change the number of non-faces in the dataset and apply PCA then use the projected train sets to fit the KNN classifier and record the accuracy on the test set, the following snips taken for the results and titled with the number of non faces samples taken

With 400 faces and 100 non-faces samples

```
# slice non-faces dataset
d1 = np.concatenate((faces_dataset,non_faces_dataset[0:100]))
l1 = labels[0:500]
# split into train and test sets
train_features, test_features, train_labels, test_labels = split(d1,l1)

# get the projection matrix computed using PCA
projection_matrix = pca(train_features,0.95)

# project the train and test sets on the projection matrix
projected_train_features = projection(train_features,projection_matrix)
projected_test_features = projection(test_features,projection_matrix)

# use knn classifier
clf = knn(projected_train_features,train_labels,1)

# store the results
acc_list.append(get_accuracy(clf,projected_test_features,test_labels))
r_list.append(projected_train_features.shape[1])

print(acc_list[-1])
print(r_list[-1])
```

```
0.9533333333333334
153
```

With 400 faces and 150 non-faces samples

```
# slice non-faces dataset
d2 = np.concatenate((faces_dataset,non_faces_dataset[0:150]))
l2 = labels[0:550]
# split into train and test sets
train_features, test_features, train_labels, test_labels = split(d2,l2)

# get the projection matrix computed using PCA
projection_matrix = pca(train_features,0.95)

# project the train and test sets on the projection matrix
projected_train_features = projection(train_features,projection_matrix)
projected_test_features = projection(test_features,projection_matrix)

# use knn classifier
clf = knn(projected_train_features,train_labels,1)

# store the results
acc_list.append(get_accuracy(clf,projected_test_features,test_labels))
r_list.append(projected_train_features.shape[1])

print(acc_list[-1])
print(r_list[-1])
```

```
0.9272727272727272
149
```

With 400 faces and 200 non-faces samples

```
# slice non-faces dataset
d3 = np.concatenate((faces_dataset,non_faces_dataset[0:200]))
l3 = labels[0:600]
# split into train and test sets
train_features, test_features, train_labels, test_labels = split(d3,l3)

# get the projection matrix computed using PCA
projection_matrix = pca(train_features,0.95)

# project the train and test sets on the projection matrix
projected_train_features = projection(train_features,projection_matrix)
projected_test_features = projection(test_features,projection_matrix)

# use knn classifier
clf = knn(projected_train_features,train_labels,1)

# store the results
acc_list.append(get_accuracy(clf,projected_test_features,test_labels))
r_list.append(projected_train_features.shape[1])

print(acc_list[-1])
print(r_list[-1])
```

```
0.9388888888888889
162
```

With 400 faces and 250 non-faces samples

```
[24] # slice non-faces dataset
     d4 = np.concatenate((faces_dataset,non_faces_dataset[0:250]))
     l4 = labels[0:650]
     # split into train and test sets
     train_features, test_features, train_labels, test_labels = split(d4,l4)

     # get the projection matrix computed using PCA
     projection_matrix = pca(train_features,0.95)

     # project the train and test sets on the projection matrix
     projected_train_features = projection(train_features,projection_matrix)
     projected_test_features = projection(test_features,projection_matrix)

     # use knn classifier
     clf = knn(projected_train_features,train_labels,1)

     # store the results
     acc_list.append(get_accuracy(clf,projected_test_features,test_labels))
     r_list.append(projected_train_features.shape[1])

     print(acc_list[-1])
     print(r_list[-1])
```

```
0.958974358974359
168
```

With 400 faces and 300 non-faces samples

```
[25] # slice non-faces dataset
     d5 = np.concatenate((faces_dataset,non_faces_dataset[0:300]))
     l5 = labels[0:700]
     # split into train and test sets
     train_features, test_features, train_labels, test_labels = split(d5,l5)

     # get the projection matrix computed using PCA
     projection_matrix = pca(train_features,0.95)

     # project the train and test sets on the projection matrix
     projected_train_features = projection(train_features,projection_matrix)
     projected_test_features = projection(test_features,projection_matrix)

     # use knn classifier
     clf = knn(projected_train_features,train_labels,1)

     # store the results
     acc_list.append(get_accuracy(clf,projected_test_features,test_labels))
     r_list.append(projected_train_features.shape[1])

     print(acc_list[-1])
     print(r_list[-1])
```

```
0.9142857142857143
177
```

With 400 faces and 350 non-faces samples

```
[26] # slice non-faces dataset
     d6 = np.concatenate((faces_dataset,non_faces_dataset[0:350]))
     l6 = labels[0:750]
     # split into train and test sets
     train_features, test_features, train_labels, test_labels = split(d6,l6)

     # get the projection matrix computed using PCA
     projection_matrix = pca(train_features,0.95)

     # project the train and test sets on the projection matrix
     projected_train_features = projection(train_features,projection_matrix)
     projected_test_features = projection(test_features,projection_matrix)

     # use knn classifier
     clf = knn(projected_train_features,train_labels,1)

     # store the results
     acc_list.append(get_accuracy(clf,projected_test_features,test_labels))
     r_list.append(projected_train_features.shape[1])

     print(acc_list[-1])
     print(r_list[-1])
```

```
0.9422222222222222
181
```

With 400 faces and 400 non-faces samples

```
# slice non-faces dataset
d7 = np.concatenate((faces_dataset,non_faces_dataset[0:400]))
l7 = labels[0:800]
# split into train and test sets
train_features, test_features, train_labels, test_labels = split(d7,l7)

# get the projection matrix computed using PCA
projection_matrix = pca(train_features,0.95)

# project the train and test sets on the projection matrix
projected_train_features = projection(train_features,projection_matrix)
projected_test_features = projection(test_features,projection_matrix)

# use knn classifier
clf = knn(projected_train_features,train_labels,1)

# store the results
acc_list.append(get_accuracy(clf,projected_test_features,test_labels))
r_list.append(projected_train_features.shape[1])
```

```
print(acc_list[-1])
print(r_list[-1])
```

```
0.9333333333333333
190
```

With 400 faces and 450 non-faces samples

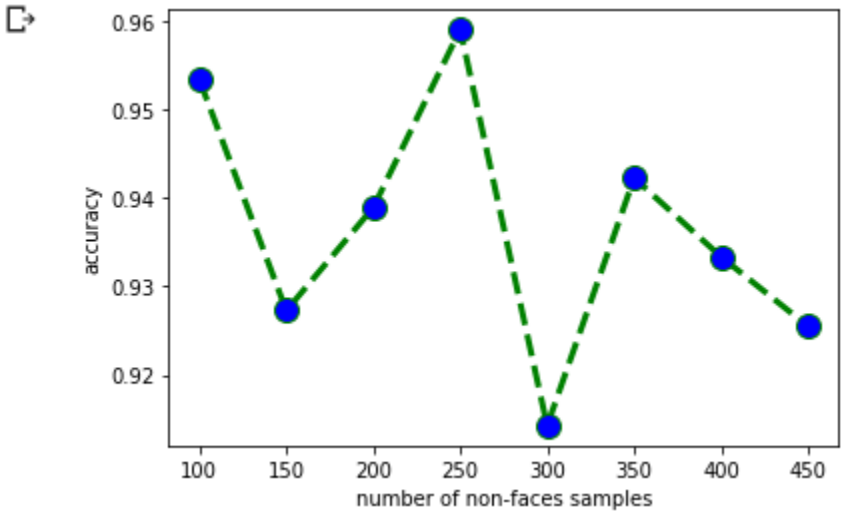
```
# slice non-faces dataset
d8 = np.concatenate((faces_dataset,non_faces_dataset[0:450]))
l8 = labels[0:850]
# split into train and test sets
train_features, test_features, train_labels, test_labels = split(d8,l8)

# get the projection matrix computed using PCA
projection_matrix = pca(train_features,0.95)

# project the train and test sets on the projection matrix
projected_train_features = projection(train_features,projection_matrix)
projected_test_features = projection(test_features,projection_matrix)

# use knn classifier
clf = knn(projected_train_features,train_labels,1)

# store the results
acc_list.append(get_accuracy(clf,projected_test_features,test_labels))
r_list.append(projected_train_features.shape[1])
```

```
print(acc_list[-1])
print(r_list[-1])
```

```
0.9254901960784314
200
```

Plot Accuracy vs number of non-faces samples in the dataset

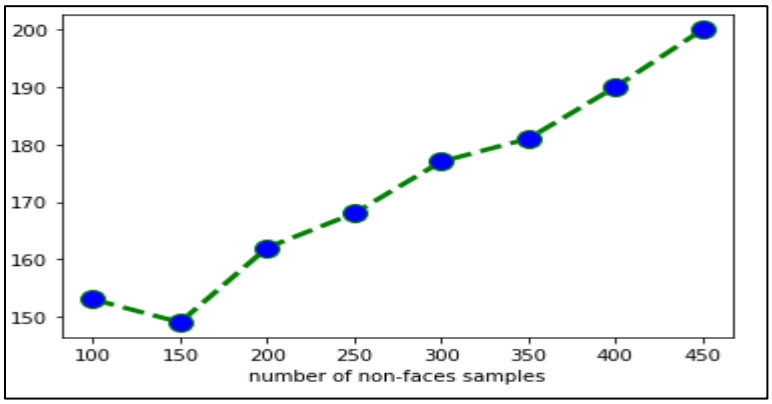
```
[29] ranges = [100,150,200,250,300,350,400,450]
     plt.plot(ranges, acc_list,color='green', linestyle='dashed', linewidth = 3,marker='o', markerfacecolor='blue', markersize=12)
     plt.xlabel('number of non-faces samples')
     plt.ylabel('accuracy')
     plt.show()
```



Plotting accuracy vs number of non-faces samples in the dataset we actually didn't find something special as it depends on dataset itself and using KNN classifier is non the best classifier but it is simple and fast learner. However, it was expected to find the accuracy decreasing as the number of non-faces exceeded, the number of faces by a higher ratio.

Plot R vs number of non-faces samples in the dataset

```
plt.plot(ranges, r_list,color='green', linestyle='dashed', linewidth = 3,marker='o', markerfacecolor='blue', markersize=12)
plt.xlabel('number of non-faces samples')
plt.ylabel('accuracy')
plt.show()
```



Plotting number of dominant eigenvectors selected for alpha=0.95 is increasing by increasing number of non – faces samples in the dataset as the variance increased in the dataset and need more eigenvectors to represent the variance in the dataset with 5% error only

LDA

```
def lda_2(train_features,labels):
  n1,n2 = get_length(labels)
  Stotal=np.zeros(shape=(train_features.shape[1],train_features.shape[1]))
  mean_1 = np.mean(train_features[0:n1],axis=0)          # (10304,0)
  mean_2 = np.mean(train_features[n1:n1+n2],axis=0)        # (10304,0)
  Z1 = train_features[0:n1] - mean_1
  Z2 = train_features[n1:n1+n2] - mean_2
  m = mean_1 - mean_2
  m = np.reshape(m, (train_features.shape[1], 1))
  B = np.dot(m,m.transpose())                             #
  S1 = np.dot(Z1.transpose(),Z1)
  S2 = np.dot(Z2.transpose(),Z2)
  Stotal = S1 + S2
  eigen_val,eigen_vec = np.linalg.eigh(np.linalg.inv(Stotal).dot(B))
  return eigen_vec[:,-1]
```

Same algorithm used for the faces problem but instead picking 39 dominant eigenvectors to classify 40 classes, this problem we got only two classes so we need the highest dominant eigenvector that gives highest mean distance between classes and lowest co-variance.

Appling same procedures by fixing number of face samples and changing number of non-faces samples in the dataset and apply the LDA function separately and record the accuracy.

Note: in this case the R=1 as it only classify between two classes only

Apply LDA on dataset with 400 faces and 100 non-faces samples

```
# slice non-faces dataset
d1 = np.concatenate((faces_dataset,non_faces_dataset[0:100]))
l1 = labels[0:500]

# split into train and test sets
train_features, test_features, train_labels, test_labels = split(d1,l1)

# get the projection matrix computed using LDA
projection_matrix = lda_2(train_features,train_labels)

# reshaping the projection matrix
projection_matrix=np.reshape(projection_matrix, (10304, 1))

# project the train and test sets on the projection matrix
projected_train_features = projection(train_features,projection_matrix)
projected_test_features = projection(test_features,projection_matrix)

# use knn classifier
clf = knn(projected_train_features,train_labels,1)
```

```
# store the results
acc_list.append(get_accuracy(clf,projected_test_features,test_labels))
r_list.append(projected_train_features.shape[1])

print(acc_list[-1])
print(r_list[-1])
```

```
0.82
1
```

Apply LDA on dataset with 400 faces and 150 non-faces samples

```
# slice non-faces dataset
d2 = np.concatenate((faces_dataset,non_faces_dataset[0:150]))
l2 = labels[0:550]

# split into train and test sets
train_features, test_features, train_labels, test_labels = split(d2,l2)

# get the projection matrix computed using LDA
projection_matrix = lda_2(train_features,train_labels)

# reshaping the projection matrix
projection_matrix=np.reshape(projection_matrix, (10304, 1))

# project the train and test sets on the projection matrix
projected_train_features = projection(train_features,projection_matrix)
projected_test_features = projection(test_features,projection_matrix)

# use knn classifier
clf = knn(projected_train_features,train_labels,1)

# store the results
acc_list.append(get_accuracy(clf,projected_test_features,test_labels))
```

```
r_list.append(projected_train_features.shape[1])

print(acc_list[-1])
print(r_list[-1])
```

```
0.8363636363636363
1
```

Apply LDA on dataset with 400 faces and 200 non-faces samples

```
[35] # slice non-faces dataset
     d3 = np.concatenate((faces_dataset,non_faces_dataset[0:200]))
     l3 = labels[0:600]

     # split into train and test sets
     train_features, test_features, train_labels, test_labels = split(d3,l3)

     # get the projection matrix computed using LDA
     projection_matrix = lda_2(train_features,train_labels)

     # reshaping the projection matrix
     projection_matrix=np.reshape(projection_matrix, (10304, 1))

     # project the train and test sets on the projection matrix
     projected_train_features = projection(train_features,projection_matrix)
     projected_test_features = projection(test_features,projection_matrix)

     # use knn classifier
     clf = knn(projected_train_features,train_labels,1)

     # store the results
     acc_list.append(get_accuracy(clf,projected_test_features,test_labels))

     print(acc_list[-1])

     0.7388888888888889
```

Apply LDA on dataset with 400 faces and 250 non-faces samples

```
# slice non-faces dataset
d4 = np.concatenate((faces_dataset,non_faces_dataset[0:250]))
l4 = labels[0:650]

# split into train and test sets
train_features, test_features, train_labels, test_labels = split(d4,l4)

# get the projection matrix computed using LDA
projection_matrix = lda_2(train_features,train_labels)

# reshaping the projection matrix
projection_matrix=np.reshape(projection_matrix, (10304, 1))

# project the train and test sets on the projection matrix
projected_train_features = projection(train_features,projection_matrix)
projected_test_features = projection(test_features,projection_matrix)

# use knn classifier
clf = knn(projected_train_features,train_labels,1)

# store the results
acc_list.append(get_accuracy(clf,projected_test_features,test_labels))

print(acc_list[-1])
```

```
0.6102564102564103
```

Apply LDA on dataset with 400 faces and 300 non-faces samples

```
# slice non-faces dataset
d5 = np.concatenate((faces_dataset,non_faces_dataset[0:300]))
l5 = labels[0:700]

# split into train and test sets
train_features, test_features, train_labels, test_labels = split(d5,l5)

# get the projection matrix computed using LDA
projection_matrix = lda_2(train_features,train_labels)

# reshaping the projection matrix
projection_matrix=np.reshape(projection_matrix, (10304, 1))

# project the train and test sets on the projection matrix
projected_train_features = projection(train_features,projection_matrix)
projected_test_features = projection(test_features,projection_matrix)

# use knn classifier
clf = knn(projected_train_features,train_labels,1)

# store the results
acc_list.append(get_accuracy(clf,projected_test_features,test_labels))

print(acc_list[-1])
```

```
0.7285714285714285
```

Apply LDA on dataset with 400 faces and 350 non-faces samples

```
# slice non-faces dataset
d6 = np.concatenate((faces_dataset,non_faces_dataset[0:350]))
l6 = labels[0:750]

# split into train and test sets
train_features, test_features, train_labels, test_labels = split(d6,l6)

# get the projection matrix computed using LDA
projection_matrix = lda_2(train_features,train_labels)

# reshaping the projection matrix
projection_matrix=np.reshape(projection_matrix, (10304, 1))

# project the train and test sets on the projection matrix
projected_train_features = projection(train_features,projection_matrix)
projected_test_features = projection(test_features,projection_matrix)

# use knn classifier
clf = knn(projected_train_features,train_labels,1)

# store the results
acc_list.append(get_accuracy(clf,projected_test_features,test_labels))

print(acc_list[-1])
```

```
0.48
```

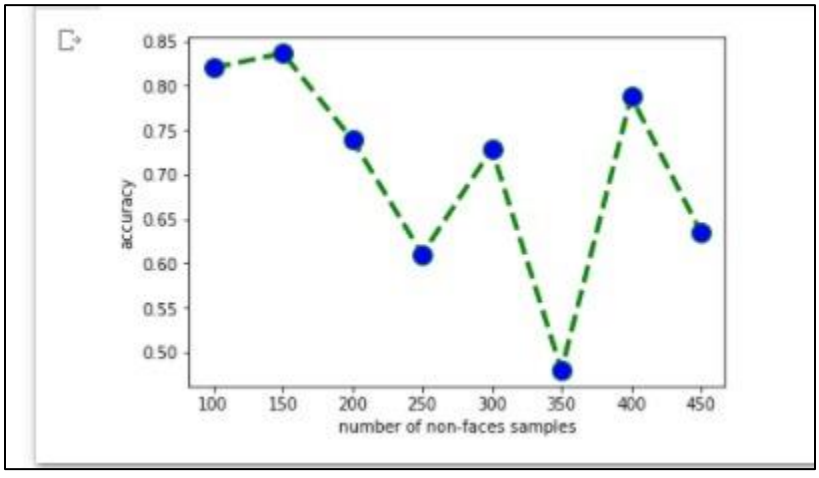
Apply LDA on dataset with 400 faces and 400 non-faces samples

```
# slice non-faces dataset
d7 = np.concatenate((faces_dataset,non_faces_dataset[0:400]))
l7 = labels[0:800]

# split into train and test sets
train_features, test_features, train_labels, test_labels = split(d7,l7)

# get the projection matrix computed using LDA
projection_matrix = lda_2(train_features,train_labels)

# reshaping the projection matrix
projection_matrix=np.reshape(projection_matrix, (10304, 1))

# project the train and test sets on the projection matrix
projected_train_features = projection(train_features,projection_matrix)
projected_test_features = projection(test_features,projection_matrix)

# use knn classifier
clf = knn(projected_train_features,train_labels,1)

# store the results
acc_list.append(get_accuracy(clf,projected_test_features,test_labels))

print(acc_list[-1])
```

```
0.7875
```

Apply LDA on dataset with 400 faces and 450 non-faces samples

```
[40] # slice non-faces dataset
     d8 = np.concatenate((faces_dataset,non_faces_dataset[0:450]))
     l8 = labels[0:850]

     # split into train and test sets
     train_features, test_features, train_labels, test_labels = split(d8,l8)

     # get the projection matrix computed using LDA
     projection_matrix = lda_2(train_features,train_labels)

     # reshaping the projection matrix
     projection_matrix=np.reshape(projection_matrix, (10304, 1))

     # project the train and test sets on the projection matrix
     projected_train_features = projection(train_features,projection_matrix)
     projected_test_features = projection(test_features,projection_matrix)

     # use knn classifier
     clf = knn(projected_train_features,train_labels,1)

     # store the results
     acc_list.append(get_accuracy(clf,projected_test_features,test_labels))

     print(acc_list[-1])

     0.6352941176470588
```



As it depends on the dataset of non-faces the accuracy for small non-faces samples is better as the dataset biased towards the face samples and same for test sets in addition to we chose K=1 for KNN classifier by default but it could be better it we tuned the K hyperparameter to get better results

Some test cases to show failure and success classification for our model

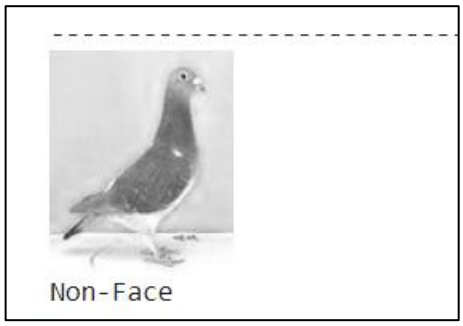
First, we picked 400 faces and 400 non-faces samples then apply our PCA to it with alpha=0.95 to get the dominant eigenvectors that contains 95% of the information needed, the we project our test and train sets using the resultant eigenvectors with specific R= 190, applying LDA to get the highest dominant eigenvector to project the data and produce best classification dimension, the following code shows our operation:

Apply PCA and LDA on 400 faces and 400 non-faces samples

```
[19] # slice non-faces dataset
     d = np.concatenate((faces_dataset,non_faces_dataset[0:400]))
     l = labels[0:800]

     # split into train and test sets
     train_features, test_features, train_labels, test_labels = split(d,l)

     # apply PCA with alpha=0.95
     projection_matrix_pca = pca(train_features,0.95)

     # project the train and test sets on the projection matrix
     projected_train_features_pca = projection(train_features,projection_matrix_pca)
     projected_test_features_pca = projection(test_features,projection_matrix_pca)

     # get the projection matrix computed using LDA
     projection_matrix_lda = lda_2(projected_train_features_pca,train_labels)

     #reshaping the projection matrix
     projection_matrix_lda=np.reshape(projection_matrix_lda, (projected_train_features_pca.shape[1], 1))

     # project the train and test sets on the projection matrix
     projected_train_features_lda = projection(projected_train_features_pca,projection_matrix_lda)
     projected_test_features_lda = projection(projected_test_features_pca,projection_matrix_lda)

     # use knn classifier
     clf = knn(projected_train_features_lda,train_labels,1)

     # store the results
     accuracy = get_accuracy(clf,projected_test_features_lda,test_labels)
     print(accuracy)
```

Examples for the test results:

0 for non-face prediction and 1 for face prediction



Face

Face

Non-Face

Show failure and success cases

```
[20] def test_image(image_vector,clf):
       pred = clf.predict([image_vector])
       image_vector = np.reshape(image_vector, (112, 92))
       img = im.fromarray(image_vector)
       return img, pred
```

```
# you may find corrupted images due to resize
classifier = knn(train_features,train_labels,1)
test = non_faces_dataset[400:450]
for x in test:
  image, pred = test_image(x,classifier)
  display(image)
  if pred == [0]:
    print("Non-Face")
  else:
    print("Face")
  print("----------------------------------------------------")
```

Links for colab notebooks:

https://colab.research.google.com/drive/1_QBh1rwmnblmSGaCKChsfJIfITRZGWJO

https://colab.research.google.com/drive/1II3hMaHaHAQrOSB7rr5mEZntyCdw84RW#scrollTo=-wnWGLt_H4av