PATTERN RECOGNITION

FACE RECOGNITION

PROBLEM STATEMENT

We intend to perform face recognition. Face recognition means that for a given image you can tell the subject id. This task is achieved by following the steps above…

1. DOWNLOAD DATASET & UNDERSTAND THE FORMAT

Our database of subject is very simple. There are ten different images of each of 40 distinct subjects. For some subjects, the images were taken at different times, varying the lighting, facial expressions (open / closed eyes, smiling / not smiling) and facial details (glasses / no glasses). All the images were taken against a dark homogeneous background with the subjects in an upright, frontal position (with tolerance for some side movement).
The files are in PGM format and can conveniently be viewed on UNIX (TM) systems using the 'xv' program. The size of each image is 92x112 pixels, with 256 grey levels per pixel. The images are organized in 40 directories (one for each subject), which have names of the form sX, where X indicates the subject number (between 1 and 40). In each of these directories, there are ten different images of that subject, which have names of the form Y.pgm, where Y is the image number for that subject (between 1 and 10).
Here is an examples of download faces dataset…

Also, as an additional work **(Bonus part)**, we performed a solution to a similar problem: Faces VS Non-Faces. It's achieved by downloading another random image of non-faces with same size 92x112 pixels .

2. GENERATE DATA MATRIX & LABEL VECTOR

The following steps must be done:

- Read every PGM image, convert it into a vector of 10304 values corresponding to the image size.
- Stack the 400 vectors into a single Data Matrix D and generate the label vector y accordingly.
- The labels are integers from 1:40 corresponding to the subject id.

Here is the implementation code …

```
##### Read pgm file ,converts it to a single row vector
def read_pgm(filename, byteorder='>'):

    with open(filename, 'rb') as f:
        buffer = f.read()
    try:
        header, width, height, maxval = re.search(
            b"(^P5\s(?:\s*#.*[\r\n])*"
            b"(\d+)\s(?:\s*#.*[\r\n])*"
            b"(\d+)\s(?:\s*#.*[\r\n])*"
            b"(\d+)\s(?:\s*#.*[\r\n]\s)*)", buffer).groups()
    except AttributeError:
        raise ValueError("Not a raw PGM file: '%s'" % filename)

    img = np.frombuffer(buffer,
                            dtype='u1' if int(maxval) < 256 else byteorder+'u2',
                            count=int(width)*int(height),
                            offset=len(header)
                            ).reshape((int(height), int(width)))

    return img.reshape(1,int(height)*int(width))


  ##### read all image test cases to prepare Data matrix D and Label Vector Y
def generate_dataMatrix_labelVector():

    #generate data matrix and vector label
    dataMatrix =  np.array([])
    yVector =  np.array([])

    #fill D matrix and Y vector
    src = os.getcwd()+"\orl_faces"
    for d in os.listdir(src):
        path = "\\"+ os.path.basename(d)
        label = int(os.path.basename(d)[1:])
       # print(label)
        for f in os.listdir(src+path):
            #print (os.path.basename(f))
            img = read_pgm(src+path+"\\"+os.path.basename(f), byteorder='<')
            dataMatrix = np.append(dataMatrix,img)
            #plot_pgm(img)
            yVector = np.append(yVector,label)

    dataMatrix = dataMatrix.reshape(400,10304)
    yVector = yVector.reshape(400,1)

    return dataMatrix,yVector
```

3. SPLIT DATASET INTO TRAINING & TESTING SETS

The following steps must be done:

- From the Data Matrix D 400x10304 keep the odd rows for training and the even rows.
- for testing. This will give you 5 instances per person for training and 5 instances
- per person for testing.
- Split the labels vector accordingly.

Here is the implementation code …

```
##### Read single vector vector image, converts plot it
def plot_pgm(imgVec):
    # fixed image dimensions: 92x112
    img = imgVec.reshape(112, 92)
    pyplot.imshow(img, pyplot.cm.gray)
    pyplot.show()


##### Spliting data into training and testing
def split_training_testing(dataMatrix, labelVector):
    # generate data matrix and vector label fro training and test
    dataTrain = np.array([])
    yTrain = np.array([])
    dataTest = np.array([])
    yTest = np.array([])

    # spliting: odd rows for training and even rows for testing
    for i in range(0, 400):
        if (i % 2 == 0):
            dataTest = np.append(dataTest, dataMatrix[i])
            yTest = np.append(yTest, labelVector[i])
        else:
            dataTrain = np.append(dataTrain, dataMatrix[i])
            yTrain = np.append(yTrain, labelVector[i])

    dataTrain = dataTrain.reshape(200, 10304)
    yTrain = yTrain.reshape(200, 1)
    dataTest = dataTest.reshape(200, 10304)
    yTest = yTest.reshape(200, 1)

    return dataTrain, yTrain, dataTest, yTest
```

4. CLASSIFICATION USING PCA + CLASSIFIER TUNING

The following steps must be done:

- Use the pseudo code below for computing the projection matrix U. Define the values of alpha ={0.8,0.85,0.9,0.95}

ALGORITHM 7.1. Principal Component Analysis

PCA $(\mathbf{D}, \alpha)$:

1 $\boldsymbol{\mu} = \frac{1}{n}\sum_{i=1}^{n} \mathbf{x}_i$ // compute mean
2 $\mathbf{Z} = \mathbf{D} - \mathbf{1} \cdot \boldsymbol{\mu}^T$ // center the data
3 $\boldsymbol{\Sigma} = \frac{1}{n}(\mathbf{Z}^T\mathbf{Z})$ // compute covariance matrix
4 $(\lambda_1, \lambda_2, \ldots, \lambda_d) = \text{eigenvalues}(\boldsymbol{\Sigma})$ // compute eigenvalues
5 $\mathbf{U} = \begin{pmatrix} \mathbf{u}_1 & \mathbf{u}_2 & \cdots & \mathbf{u}_d \end{pmatrix} = \text{eigenvectors}(\boldsymbol{\Sigma})$ // compute eigenvectors
6 $f(r) = \frac{\sum_{i=1}^{r} \lambda_i}{\sum_{i=1}^{d} \lambda_i}$, for all $r = 1, 2, \ldots, d$ // fraction of total variance
7 Choose smallest r so that $f(r) \geq \alpha$ // choose dimensionality
8 $\mathbf{U}_r = \begin{pmatrix} \mathbf{u}_1 & \mathbf{u}_2 & \cdots & \mathbf{u}_r \end{pmatrix}$ // reduced basis
9 $\mathbf{A} = \{\mathbf{a}_i \mid \mathbf{a}_i = \mathbf{U}_r^T \mathbf{x}_i, \text{for } i = 1, \ldots, n\}$ // reduced dimensionality data

- Project the training set and test sets **separately** using same projection matrix.
- Use a simple classifier (first Nearest Neighbor to determine the class labels).
- Report Accuracy for every value of alpha separately.
- Finding a relation between alpha and the classification accuracy:
  from retrieved results above , we can tabulate some interesting accuracy results…

PCA Test Accuracy Results for 50/50 Splits

	K = 1	K = 3	K = 5	K = 7
Alpha = 0.8	93%	85.5%	80.5%	78%
Alpha = 0.85	94%	85.5%	83%	77.5%
Alpha = 0.9	94.5%	85%	81.5%	75.5%
Alpha = 0.95	93.5%	84.5%	81.5%	74%

As we see, we find that the accuracy gets better when we increase Alpha value. But in contrast, the accuracy gets worst when increasing the K value for the same Alpha.

A best Accuracy Value: Alpha - 0.9 , k – 1.

Here is the implementation code …

```
##### Classification using PCA - returns Projection Matrix

    # We work on Training matrix to find projection matrix

    #feature scalling
    #scaler = StandardScaler().fit(dTrain)
    #dTrain = scaler.transform(dTrain)
    #dTest = scaler.transform(dTest)



    alpha = 0.8

    #calculate mean vector on D
    print("Mean")
    mean = np.mean(dTrain, axis=0).reshape(10304,1)
    print(mean)
    print(mean.shape)

    #Center data arround the mean
    print("Centered data")
    zTrain = dTrain - np.ones((200,1)).dot(mean.T)
    zTest  = dTest  - np.ones((200,1)).dot(mean.T)
    print("zTrain")
    print(zTrain)
    print(zTrain.shape)
    print("zTest")
    print(zTest)
    print(zTest.shape)

    #calculate covariance matrix
    print("Covariance matrix")
    #covarianceMatrix = np.cov(zTrain.T)
    covarianceMatrix = (1/len(zTrain)) * np.matmul(np.transpose(zTrain),zTrain)
    print(covarianceMatrix)

    #calculate eigen values &vectors from cov matrix
    eigenValues,eigenVectors = LA.eigh(covarianceMatrix)
    print("Eigen Values")
    print(eigenValues)
    print(eigenValues.shape)
    print("Eigen Vectors")
    print(eigenVectors)
    print(eigenVectors.shape)
```

```
#determine the number of reduced dimentionality for each alpha

 alpha  = [ 0.8, 0.85, 0.9, 0.95 ]

 kParam = [ 1, 3, 5, 7 ]

 for k in alpha:

     print("Alpha Value -> ",k)
     i = 10303
     EigenVals = 0
     projectionMatrix = []

     #projection matrix for each alpha value
     #print("Eigen values chosen:")
     count = 0

     totalEigVals = 0
     for t in range(0,len(eigenValues)):
           totalEigVals = totalEigVals + eigenValues[t]

     while ((EigenVals/(totalEigVals)) <= k):
             EigenVals = EigenVals + eigenValues[i]
             #print(eigenValues[i])
             projectionMatrix.append(np.array(eigenVectors[:,i]))
             i = i - 1
             count += 1


     print("Eigen values used:")
     print(count)

     #print("Projection Matrix")
     #print(projectionMatrix)
#     print(projectionMatrix.shape)


     #projection of training and test matrix
     print("Projection Data Train & Test")

 #Knn for each value of k
 for q in kParam:

   #use first nearest neighbour classifier
   knn = KNeighborsClassifier(n_neighbors=q)
   knn.fit(projectedDataTrain,yTrain)


   rowsValues_pred = knn.predict(projectedDataTest)
   rowsValues_pred1 = knn.predict(projectedDataTrain)

   print("k = ",q ,"Train Accuracy = ",accuracy_score(yTrain,rowsValues_pred1))
   print("k = ",q ,"Test  Accuracy = ",accuracy_score(yTest,rowsValues_pred))
```

```
Mean
[[85.12 ]
 [84.89 ]
 [85.165]
 ...
 [77.24 ]
 [74.335]
 [73.37 ]]
(10304, 1)
Covariance matrix
[[1318.8756   1299.2282   1304.6652   ... -235.7588   -126.0752
   -75.5394  ]
 [1299.2282   1295.5879   1293.30315  ... -218.1036   -112.70315
   -64.3043  ]
 [1304.6652   1293.30315  1306.107775 ... -216.9946    -99.325275
   -49.28605 ]
 ...
 [-235.7588   -218.1036   -216.9946   ... 2505.0524   1995.4546
  1882.7162  ]
 [-126.0752   -112.70315   -99.325275 ... 1995.4546   1972.392775
  1859.56105 ]
 [ -75.5394    -64.3043    -49.28605  ... 1882.7162   1859.56105
  1968.8831  ]]
Eigen Values
[-1.96486090e-09 -1.46217689e-09 -1.06714807e-09 ...  1.06569341e+06
  2.14179597e+06  2.76884465e+06]
(10304,)
Eigen Vectors
[[ 0.          0.          0.         ... -0.01891079 -0.01536171
   0.00124556]
 [-0.17094925 -0.50007679  0.13108778 ... -0.01911777 -0.01513146
   0.00126454]
 [ 0.47206151 -0.06540396 -0.10648273 ... -0.01903942 -0.01516195
   0.00156465]
 ...
 [-0.00566906  0.01751952 -0.00986042 ... -0.01287451  0.00961112
   0.00873725]
 [-0.00498918 -0.00574758 -0.00435695 ... -0.01377318  0.0077597
   0.00721705]
 [-0.00434485 -0.00878641 -0.02453822 ... -0.01438815  0.00692708
   0.0084008 ]]
(10304, 10304)
```

Here are the results …

```
Alpha Value ->  0.8
Eigen values used:
37
Projection Data Train & Test
(200, 37)
k =  1 Train Accuracy =  1.0
k =  1 Test  Accuracy =  0.93
k =  3 Train Accuracy =  0.945
k =  3 Test  Accuracy =  0.855
k =  5 Train Accuracy =  0.895
k =  5 Test  Accuracy =  0.805
k =  7 Train Accuracy =  0.815
k =  7 Test  Accuracy =  0.78
```

```
Alpha Value ->  0.85
Eigen values used:
53
Projection Data Train & Test
(200, 53)
k =  1 Train Accuracy =  1.0
k =  1 Test  Accuracy =  0.94
k =  3 Train Accuracy =  0.955
k =  3 Test  Accuracy =  0.855
k =  5 Train Accuracy =  0.895
k =  /usr/local/lib/python3.6/dis
```

```
5 Test  Accuracy =  0.83
k =  7 Train Accuracy =  0.815
k =  7 Test  Accuracy =  0.775
```

```
Alpha Value ->  0.9
Eigen values used:
77
Projection Data Train & Test
(200, 77)
k =  1 Train Accuracy =  1.0
k =  1 Test  Accuracy =  0.945
k =  3 Train Accuracy =  0.95
k =  3 Test  Accuracy =  0.85
k =  5 Train Accuracy =  0.88
k =  5 Test  Accuracy =  0.815
k =  7 Train Accuracy =  0.8
k =  7 Test  Accuracy =  0.755
Alpha Value ->  0.95
Eigen values used:
116
```

```
(200, 116)
k =  1 Train Accuracy =  1.0
k =  1 Test  Accuracy =  0.935
k =  3 Train Accuracy =  0.955
k =  3 Test  Accuracy =  0.845
k =  5 Train Accuracy =  0.87
k =  5 Test  Accuracy =  0.815
k =  7 Train Accuracy =  0.79
k =  7 Test  Accuracy =  0.74
```

```
Alpha Value ->  0.95
Eigen values used:
116
```

```
(200, 116)
k =  1 Train Accuracy =  1.0
k =  1 Test  Accuracy =  0.935
k =  3 Train Accuracy =  0.955
k =  3 Test  Accuracy =  0.845
k =  5 Train Accuracy =  0.87
k =  5 Test  Accuracy =  0.815
k =  7 Train Accuracy =  0.79
k =  7 Test  Accuracy =  0.74
```

5. CLASSIFICATION USING LDA + CLASSIFIER TUNING

The following steps must be done:

- Use the pseudo code below for LDA. We will modify few lines in pseudo code to handle multiclass LDA:
  - Calculate the mean vector for every class Mu1,Mu2..., Mu40
  - Replace B matrix by $\mathbf{S}_b = \sum_{k=1}^{m} n_k(\boldsymbol{\mu}_k - \boldsymbol{\mu})(\boldsymbol{\mu}_k - \boldsymbol{\mu})^T$
  - Here, m is the number of classes, is the overall sample mean, and
  - is the number of samples in the k-th class.
  - S matrix remains the same but it sums S1,S2,S3...S40.
  - Use 39 dominant eigenvectors instead of just one. You will have a projection matrix U 39x10304

ALGORITHM 20.1. Linear Discriminant Analysis

LINEARDISCRIMINANT $(\mathbf{D} = \{(\mathbf{x}_i, y_i)\}_{i=1}^n)$:

1 $\mathbf{D}_i \leftarrow \{\mathbf{x}_j \mid y_j = c_i, j = 1, \ldots, n\}, i = 1, 2$ // class-specific subsets
2 $\boldsymbol{\mu}_i \leftarrow \text{mean}(\mathbf{D}_i), i = 1, 2$ // class means
3 $\mathbf{B} \leftarrow (\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)(\boldsymbol{\mu}_1 - \boldsymbol{\mu}_2)^T$ // between-class scatter matrix
4 $\mathbf{Z}_i \leftarrow \mathbf{D}_i - \mathbf{1}_{n_i} \boldsymbol{\mu}_i^T, i = 1, 2$ // center class matrices
5 $\mathbf{S}_i \leftarrow \mathbf{Z}_i^T \mathbf{Z}_i, i = 1, 2$ // class scatter matrices
6 $\mathbf{S} \leftarrow \mathbf{S}_1 + \mathbf{S}_2$ // within-class scatter matrix
7 $\lambda_1, \mathbf{w} \leftarrow \text{eigen}(\mathbf{S}^{-1}\mathbf{B})$ // compute dominant eigenvector

- Project the training set and test sets **separately** using same projection matrix U. You will have 39 dimensions in the new space.
- Use a simple classifier (first Nearest Neighbor to determine the class labels + trying another value for K).
- Report Accuracy for the Multiclass LDA on the face recognition dataset:

LDA Test Accuracy Results for 50/50 Splits

K = 1	K = 3	K = 5	K = 7
92%	87.5%	82.5%	77%

As shown , the lesser value of K the better Accuracy.

Here is the implementation code …

```
###LDA
my_dict_for_data = {}
my_dict_for_mean = {}
my_dict_for_class_scatter_matrix = {}
total_mean = np.zeros((10304, 1))
between_class_scatter_matrix = np.zeros((10304, 10304))
mean_deviation = np.array([])
mean_deviation_transpose = np.array([])
class_scatter_matrix = np.zeros((10304, 10304))
within_class_scatter_matrix = np.zeros((10304, 10304))
D = np.array([])
mean = np.array([])
#total_mean = np.array([])

#peparing each data of a class in a matrix
for z in range(0,200,5):
    D = Dtrain[range(z,z+5), :]
    print(D.shape)
    #D.reshape(5,10304)
    my_dict_for_data[str(Ytrain[z])] = D
    #print(str(Ytrain[z]))
    #print(D)


#get the mean for each matrix
for key in my_dict_for_data:

    #print(key, my_dict_for_data[key])
    #print("\n Mean Vector")
    mean = np.mean(my_dict_for_data[key], axis=0)
    mean = mean.reshape(10304,1)
    print(mean.shape)
    my_dict_for_mean[key] = mean
    print(key, my_dict_for_mean[key])

# get the between-class scatter matrix (B)

# compute the total mean
for key in my_dict_for_mean:
    # print(my_dict_for_mean[key])
    # print(my_dict_for_mean[key].size)
    # print(total_mean)
    total_mean = total_mean + my_dict_for_mean[key]

#total_mean = total_mean.reshape(10304, 1)
print(total_mean.shape)
total_mean = total_mean / 40
```

```
#compute SB
i = 0
for key in my_dict_for_mean:
        i=i+1
        print(i)
        mean_deviation = my_dict_for_mean[key] - total_mean
        print(mean_deviation.shape)
        #mean_deviation = mean_deviation.reshape(10304, 1)
        mean_deviation_transpose = np.transpose(mean_deviation)
        print(mean_deviation_transpose.shape)
        #mean_deviation_transpose = mean_deviation_transpose.reshape(1, 10304)
        #mean_deviation_product = np.dot(mean_deviation, mean_deviation_transpose)
        #print(mean_deviation_product.shape)
        #mean_deviation_product = mean_deviation_product.reshape(10304, 10304)
        between_class_scatter_matrix = between_class_scatter_matrix + (5 * (np.dot(mean_deviation, mean_deviation_transpose)))
        print(between_class_scatter_matrix.shape)

#between_class_scatter_matrix = between_class_scatter_matrix.reshape(10304, 10304)
print("between_class_scatter_matrix")
print(between_class_scatter_matrix.shape)
print(between_class_scatter_matrix)
```

```
#compute class scatter matrix(s1,s2,s3,...)
temp1 = np.array([])
#temp2 = np.array([])
temp2_transpose = np.array([])
#deviation_point_from_its_mean = np.array([])
#deviation_point_from_its_mean_transpose = np.array([])
deviation_point_from_its_mean_product = np.array([])
i=0
for key in my_dict_for_data:
        i=i+1
        print(i)
        temp1 = my_dict_for_data[key]
        print("temp1",temp1.shape)
        print(temp1)
        for z in range(0, 5):
            temp2_transpose = np.transpose(temp1[z, :].reshape(1, 10304))

            deviation_point_from_its_mean_product = np.dot((temp2_transpose - my_dict_for_mean[key]),
                                                           (np.transpose(temp2_transpose - my_dict_for_mean[key])))
            class_scatter_matrix = class_scatter_matrix + deviation_point_from_its_mean_product

        within_class_scatter_matrix = within_class_scatter_matrix + class_scatter_matrix
```

```
inverse = np.linalg.inv(within_class_scatter_matrix)
print(inverse.shape)
w =np.dot(inverse,between_class_scatter_matrix)
print(w.shape)
eig_vals, eig_vecs = scipy.linalg.eigh(w , eigvals=((10304-40),(10304-1)))
print("eignvalue")
print(eig_vals)
print("eigenVectors")
print(eig_vecs)
```

```
#projection of training and test matrix
    print("Projection Data Train & Test")
    projectedDataTrain = Dtrain.dot(eig_vecs)
    projectedDataTest  = Dtest.dot(eig_vecs)

    print(projectedDataTrain.shape)
    print(projectedDataTest.shape)
```

```
import re
import pandas as pd
import pickle
from pprint import pprint
from sklearn.feature_extraction.text import CountVectorizer
from sklearn.metrics import confusion_matrix
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score

kParam = [ 1, 3, 5, 7 ]
#Knn for each value of k
for q in kParam:
      knn = KNeighborsClassifier(n_neighbors=q)
      knn.fit(projectedDataTrain,Ytrain)
      rowsValues_pred1 = knn.predict(projectedDataTrain)
      rowsValues_pred = knn.predict(projectedDataTest)
      print("k = ",q , "Accuracy = " ,accuracy_score(Ytrain,rowsValues_pred1))
      print("k = ",q , "Accuracy = " ,accuracy_score(Ytest,rowsValues_pred))
```

Here are the results …

```
 [[85.385]
  [85.345]
  [85.72 ]
  ...
  [78.32 ]
  [76.675]
  [74.45 ]]

between_class_scatter_matrix
(10304, 10304)
[[229130.555 229072.035 226469.96  ... -40194.44  -26944.175 -13324.45 ]
 [229072.035 229595.195 226526.92  ... -40723.08  -27963.775 -14604.25 ]
 [226469.96  226526.92  224267.12  ... -38156.08  -25089.4   -12146.2  ]
 ...
 [-40194.44  -40723.08  -38156.08  ... 345239.92  331249.6   308695.2  ]
 [-26944.175 -27963.775 -25089.4   ... 331249.6   328302.675 306029.05 ]
 [-13324.45  -14604.25  -12146.2   ... 308695.2   306029.05  293892.3  ]]
```

```
(10304, 10304)
(10304, 10304)
eignvalue
[2.91947520e+16 2.94729317e+16 3.01758590e+16 3.10701871e+16
 3.20595747e+16 3.25004838e+16 3.36386910e+16 3.46743638e+16
 3.52805433e+16 3.77075390e+16 3.84575391e+16 4.02723397e+16
 4.11710060e+16 4.13632135e+16 4.30898147e+16 4.52189873e+16
 4.53237678e+16 4.69336100e+16 5.09169170e+16 5.21909223e+16
 5.43777148e+16 5.78336427e+16 5.94519041e+16 6.27950643e+16
 6.75662181e+16 7.28830965e+16 7.78858621e+16 8.43356438e+16
 8.63643866e+16 9.52343342e+16 1.10585266e+17 1.16290102e+17
 1.32557022e+17 1.49750094e+17 1.55572070e+17 2.12348297e+17
 2.96348911e+17 4.05806369e+17 4.91127085e+17 2.28170335e+18]
eigenVectors
[[ 4.22433597e-03 -1.00139778e-02  1.42502904e-03 ...  1.68030090e-02
   1.18396764e-02  1.11582210e-03]
 [ 4.32357525e-03 -9.63585850e-03  1.90169132e-03 ...  1.68464430e-02
   1.22033694e-02  9.13675358e-04]
 [ 4.39049567e-03 -1.02514311e-02  1.64609264e-03 ...  1.67208399e-02
   1.17601259e-02  1.12726939e-03]
 ...
 [-7.09751746e-04 -1.45698851e-02 -1.91317468e-03 ...  3.22259362e-03
  -1.22260959e-02 -1.74764669e-02]
 [-1.26153691e-03  6.51589141e-06 -7.76849622e-03 ... -1.44790010e-02
  -9.47539781e-03  4.68031710e-03]
 [ 6.56301024e-03 -2.43617545e-02 -2.27863337e-03 ... -1.32605456e-02
  -6.73828697e-03  3.41092504e-03]]


Projection Data Train & Test
(200, 40)
(200, 40)


('k = ', 1, 'Accuracy = ', 1.0)
('k = ', 1, 'Accuracy = ', 0.92)
('k = ', 3, 'Accuracy = ', 0.94)
('k = ', 3, 'Accuracy = ', 0.875)
('k = ', 5, 'Accuracy = ', 0.9)
('k = ', 5, 'Accuracy = ', 0.825)
('k = ', 7, 'Accuracy = ', 0.865)
('k = ', 7, 'Accuracy = ', 0.77)
```

6. BONUS - 70/30% SPLITS

The following steps must be done:

- Use different Training and Test splits. Change the number of instances per subject to be 7 and keep 3 instances per subject for testing.

In order to achieve this split , we only need to change the Split function as show in the following code implementation…

```
def split_training_testing_Bonus(dataMatrix, labelVector):
    # generate data matrix and vector label fro training and test
    dataTrain = np.array([])
    yTrain = np.array([])
    dataTest = np.array([])
    yTest = np.array([])

    # spliting: 7 imgs for training and 3 imgs for testing
    for i in range(0,400,10):

        for j in range(i,i+3):
            dataTest = np.append(dataTest, dataMatrix[j])
            yTest = np.append(yTest, labelVector[j])

        i = i+3

        print (i)

        for j in range(i,i+7):
            dataTrain = np.append(dataTrain, dataMatrix[j])
            yTrain = np.append(yTrain, labelVector[j])

        i = i+7

    dataTrain = dataTrain.reshape(280, 10304)
    yTrain = yTrain.reshape(280, 1)
    dataTest = dataTest.reshape(120, 10304)
    yTest = yTest.reshape(120, 1)

    #print("yTrain")
    #print(yTrain)
    #print("yTest")
    #print(yTest)

    return dataTrain, yTrain, dataTest, yTest
```

Here are the results of the accuracy Test on our s new splits...

PCA Test Accuracy Results for 70/30 Splits

	K = 1	K = 3	K = 5	K = 7
Alpha = 0.8	97%	90%	85.8%	77.5%
Alpha = 0.85	98.3%	91.6%	85.8%	78.3%
Alpha = 0.9	98.3%	89.2%	85%	78.3%
Alpha = 0.95	98.3%	89.1%	86.6%	79.1%

LDA Test Accuracy Results for 70/30 Splits

K = 1	K = 3	K = 5	K = 7
97.5%	90%	85%	79.1%

7. BONUS – FACE VS NON-FACE CLASSIFICATION

I download 100 images of non-face image. And they have same size of face images(92*112).

And same format .pgm

We split the data 80% for training set and 20% for test set. (because number of data set examples is not big (200 examples)).

So, I also took 100 face images only to balance the data set.

i) failure and success cases:

- if the data set is not balance :

taking the same 400 examples of face image VS only 100 of the non-face image.

So, in this case el model is trained unfairly so the accuracy will not be good enough.

- Also, if the model is trained on the same person by many position but the test image is a different person. In this case the accuracy is not good enough.

  So, it must be in a sorted manner like we did in the code (4-1) or randomized manner .

ii) How many dominant eigenvectors will you use for the LDA solution?

Only one eignvector because we have 2 Classes .

And number of eignvectors = C -1.

```
eignvalue
[9.94896226e+18]
eigenVectors
[[-1.14514338e-02]
 [-1.16518707e-02]
 [-1.19124967e-02]
 ...
 [-1.97913240e-02]
 [ 1.67590317e-02]
 [-2.24685089e-06]]
```

iii) Criticize the accuracy measure : -

```
('k = ', 1, 'Accuracy Train = ', 1.0)
('k = ', 1, 'Accuracy Test= ', 0.775)
('k = ', 3, 'Accuracy Train = ', 0.86875)
('k = ', 3, 'Accuracy Test= ', 0.85)
('k = ', 5, 'Accuracy Train = ', 0.8375)
('k = ', 5, 'Accuracy Test= ', 0.775)
('k = ', 7, 'Accuracy Train = ', 0.8375)
('k = ', 7, 'Accuracy Test= ', 0.8)
```

Here are the results of the accuracy…

Face VS Non-Face PCA Accuracy Results for 80/20 Splits

	K = 1	K = 3	K = 5	K = 7
Alpha = 0.8	90%	82.5%	87.5%	87.5%
Alpha = 0.85	95%	85%	82.5%	85%
Alpha = 0.9	95%	82.5%	82.5%	82.5%
Alpha = 0.95	90%	85%	82.5%	80%

Face VS Non-Face LDA Test Accuracy Results 80/20 Splits

K = 1	K = 3	K = 5	K = 7
77.5%	85%	77.5%	80%

ACCURACY COMPARASION SUMMARY

PCA Test Accuracy Results for 50/50 Splits

	K = 1	K = 3	K = 5	K = 7
Alpha = 0.8	93%	85.5%	80.5%	78%
Alpha = 0.85	94%	85.5%	83%	77.5%
Alpha = 0.9	94.5%	85%	81.5%	75.5%
Alpha = 0.95	93.5%	84.5%	81.5%	74%

LDA Test Accuracy Results for 50/50 Splits

K = 1	K = 3	K = 5	K = 7
92%	87.5%	82.5%	77%

PCA Test Accuracy Results for 70/30 Splits

	K = 1	K = 3	K = 5	K = 7
Alpha = 0.8	97%	90%	85.8%	77.5%
Alpha = 0.85	98.3%	91.6%	85.8%	78.3%
Alpha = 0.9	98.3%	89.2%	85%	78.3%
Alpha = 0.95	98.3%	89.1%	86.6%	79.1%

LDA Test Accuracy Results for 70/30 Splits

K = 1	K = 3	K = 5	K = 7
97.5%	90%	85%	79.1%

ALL SOURCE CODES

PCA 50/50 SPLIT

https://colab.research.google.com/drive/1T2rdmZ00pRIVGwT7sWoB5F66exStefZu#scrollTo=SQThv4Lhvior

LDA 50/50 SPLIT

https://colab.research.google.com/drive/1nCrBsrYFtb7R9RaffJjJj9GydM90eOJ6?fbclid=IwAR2GMmvLRaXcvnOYqt1heiCcr-UXX5cZ9B3aF-Wo15sDM6n3FDKZzEyneIs#scrollTo=kErOZuLNjgr0&uniqifier=1

PCA 70/30 SPLIT

https://colab.research.google.com/drive/1C8xP956K0LKOzDZ8-Xts3P33PRTAi7Th

LDA 70/30 SPLIT

https://colab.research.google.com/drive/1tabXttkzYRsXMf7-cJ1QcxsQoYg_grC3?fbclid=IwAR2FLHLzdmfBJlvurJzHsaPifaltXEbmWWIcnwRCAVgNK8iCqxgcqrMuiQY#scrollTo=mFej-i7zAnXS

FACE VS NON-FACE – PCA

https://colab.research.google.com/drive/14OgDxFtAQt6aTXp7FcFth8A0pnH1668S#scrollTo=E7UI7q6xgxkA&uniqifier=1

FACE VS NON-FACE – LDA

https://colab.research.google.com/drive/1J2ZsPmdKSgntWrXY6bRJ4K6-d2u70H-W?fbclid=IwAR0mn62Ovr8iKmxQN18cMt7F-fOF4Ab3FgvRMQuuLVoV_2dI4o1vwnjo67w#scrollTo=5QwB_vgfQRY0

THANK YOU