PATTERN RECOGNITION

ASSIGNMENT#1

FACE RECOGNITION

TEAM MEMBERS

Name	ID
Ramy Ahmed El Sayed	19015649
Zyad Samy Ramadan	19015720
Abdelmoniem Hany	

CODE EXPLANATION

LIBRARIES USED

```
import os
import numpy as np
import scipy as sc
import pandas as pd
from PIL import Image
from math import floor, ceil
from sklearn.neighbors import KNeighborsClassifier
from sklearn.metrics import accuracy_score
import matplotlib.pyplot as plt
from sklearn.decomposition import PCA as RPCA
from sklearn.discriminant_analysis import QuadraticDiscriminantAnalysis as QDA
```

The main libraries used are:

Numpy: for data manipulation and matrix simulation

Scipy: for optimized mathematical operations.

Pillow (PIL): for image manipulation

Math: for common mathematical implementations.

Sklearn: for KNN classifier, accuracy measure, and PCA/LDA variants.

Matplotlib: for plotting.

Loading The Data

Unzipping The Kaggle Archive

```
!unzip archive.zip
```

Unzips the archive in the colab's file explorer.

Note: Data used for training/testing the models can be found here.

Reading The Data

```
def readFile(path):
  image = Image.open(path)
  imageVector = np.asarray(image)
  imageVector = imageVector.flatten()
  return imageVector
```

Reads the given face images (applicable only for the given gray scale images) and flattens them into a vector to make them of size 400x10304.

Generating Images and Labels

```
[4] def loadImages():
      images = []
      labels = []
      for id in range(1,41):
        for img in range(1,11):
          path = f's{id}/{img}.pgm'
          img = readFile(path)
          images.append(img)
          labels.append(id)
      npimages = np.array(images, dtype=np.uint8)
      nplabels = np.array(labels, dtype=np.uint8)

      return npimages, nplabels
```

Reads the images in each labeled folder and assigns a label for them depending on the folder's ID which generates an image matrix with a corresponding label vector.

Splitting The Data

```
[35] def splitdataEvenly(dataMatrix, labelVector):
       print(dataMatrix.dtype, labelVector.dtype)
       trainingMatrix = np.zeros((int(dataMatrix.shape[0]/2), dataMatrix.shape[1]), dtype=np.uint8)
       trainingVector = np.zeros(int(labelVector.shape[0]/2), dtype=np.uint8)
       testingMatrix = np.zeros((int(dataMatrix.shape[0]/2), dataMatrix.shape[1]), dtype=np.uint8)
       testingVector = np.zeros(int(labelVector.shape[0]/2), dtype=np.uint8)
       for i in range(dataMatrix.shape[0]):
         if (i % 2 == 0):
           trainingMatrix[int(i/2)] = dataMatrix[i]
           trainingVector[int(i/2)] = labelVector[i]
         else:
           testingMatrix[floor(i/2)] = dataMatrix[i]
           testingVector[floor(i/2)] = labelVector[i]

       return trainingMatrix, trainingVector, testingMatrix, testingVector
```

We initialize our training/testing data matrices and labels, then we simply add the even indexed data in our training datasets and the odd indexed data in our testing datasets.

Main Algorithms

PCA

```
[9]  def PCA(data, alpha):
       meanVector = np.mean(data, axis = 0)
       z = data - meanVector
       cov = np.cov(z.transpose(), bias = True)

       eigenValues, eigenVectors = sc.linalg.eigh(cov)
       eigenValues = np.real(eigenValues)
       eigenVectors = np.real(eigenVectors)
       eigenValues = np.abs(eigenValues)
       sort_perm = eigenValues.argsort()
       eigenValues = np.flip(eigenValues[sort_perm])
       eigenVectors = np.flip(eigenVectors[:, sort_perm], axis = 1)

       localAlpha = 0;
       i = 0
       eigenValuesSum = np.sum(eigenValues)

       while (localAlpha < alpha):
         localAlpha += eigenValues[i] / eigenValuesSum
         i += 1


       projectionMatrix = eigenVectors[:, :i]
       projectedData = np.matmul(z, projectionMatrix)
       return projectedData, projectionMatrix
```

The code is clear and mimics the algorithm mentioned in the assignment report.

Testing The PCA Model

```
[ ]  alpha = [0.8,0.85,0.9,0.95]

     for a in alpha:
       projectedData, projectionMatrix = PCA(trainingMatrix, a)

       projectedTestData = np.matmul((testingMatrix - np.mean(testingMatrix, axis = 0)), projectionMatrix)

       knc = KNeighborsClassifier(n_neighbors=1)
       knc.fit(projectedData, trainingVector)

       predictedLabels = knc.predict(projectedTestData)

       accuracy = accuracy_score(testingVector, predictedLabels)
       print(f'for {a}: accuracy = {accuracy}')
```

All the alphas are iterated over and tested by training our PCA model and projecting the testing matrix in our new component/s.

The KNeighborsClassifier is then used to classify the testing data based on our model and then measure our accuracy.

Measured Accuracy

Alpha	Accuracy
0.8	0.95
0.85	0.95
0.9	0.935
0.95	0.94

It can be noticed that as we add more components to our model, the accuracy starts lowering as adding additional components that doesn't contribute much information wise could skew the models values.

LDA

Calculating Class Means

```
[10] def classMeans(data, labels, labelsCount):
       means = dict()
       for i in range(1, labelsCount + 1):
         vectorsPos = np.where(labels == i)
         vectors = np.take(data, vectorsPos, axis = 0)
         means[i] = np.mean(vectors, axis = 1)
       return means
```

Calculating Overall Mean

```
[11] def overallMean(data):
       return np.mean(data, axis = 0)
```

Calculating Scatter Matrices

```
[12] def calculateScatterMatrices(data, labels, labelsCount):
       means = classMeans(data, labels, labelsCount)
       overallmean = overallMean(data)
       S = np.zeros((len(data[0]),len(data[0])))
       Sb = np.zeros((len(data[0]), len(data[0])))
       for i in range(1, labelsCount):
         vectorsPos = np.where(labels == i)
         Z = np.take(data, vectorsPos, axis = 0)[0] - means[i][0]
         S += np.dot(Z.T, Z)

         nk = len(np.where(labels == i)[0])
         currentMean = means[i][0].reshape(len(data[0]),1) # make column vector
         overallm = overallmean.reshape(len(data[0]),1) # make column vector
         Sb += nk * (currentMean - overallm).dot((currentMean - overallm).T)
       return S, Sb
```

Main Module

```
[13] def LDA(n, data, labels):
       S, Sb = calculateScatterMatrices(data, labels, len(np.unique(labels)))
       S_inv = sc.linalg.pinv(S)
       W = np.dot(S_inv, Sb)
       print("The within-class matrix Sw = \n",S,"\n")
       print("The inverse within-class matrix Sw = \n",S_inv,"\n")
       print("The between-class matrix Sb = \n",Sb,"\n")
       eigenValues, eigenVectors = sc.linalg.eig(W)
       print("Eigenvectors before any operation = \n",eigenValues, eigenVectors,"\n")
       eigenValues = np.real(eigenValues)
       eigenVectors = np.real(eigenVectors)
       eigenValues = np.abs(eigenValues)
       sort_perm = eigenValues.argsort()
       eigenValues = np.flip(eigenValues[sort_perm])
       eigenVectors = np.flip(eigenVectors[:, sort_perm], axis = 1)

       projectionMatrix = eigenVectors[:, :n]

       print("Projection Matrix = \n", projectionMatrix, "\n")

       return projectionMatrix
```

Notes:

- Pinv (Pseudo Inverse) is used as the normal inverse function would output unexpected results if a submatrix is singular.
- Scipy was used instead of numpy as it proved to be a bit faster.
- Sorting is based on the absolute value of the eigen values.

Testing The LDA Model

```
[ ]  projectionMatrix = LDA(39, trainingMatrix, trainingVector)

     projectedTrain = np.matmul(trainingMatrix - np.mean(trainingMatrix, axis = 0), projectionMatrix)
     projectedTest = np.matmul(testingMatrix- np.mean(testingMatrix, axis = 0), projectionMatrix)

     knc = KNeighborsClassifier(n_neighbors=1)
     knc.fit(projectedTrain, trainingVector)

     predictedLabels = knc.predict(projectedTest)

     accuracy = accuracy_score(testingVector, predictedLabels)
     print(f'LDA accuracy = {accuracy}')
```

Measured Accuracy

```
LDA accuracy = 0.945
```

The overall accuracy is higher than PCA's average accuracy as LDA is better for classifying than PCA is.

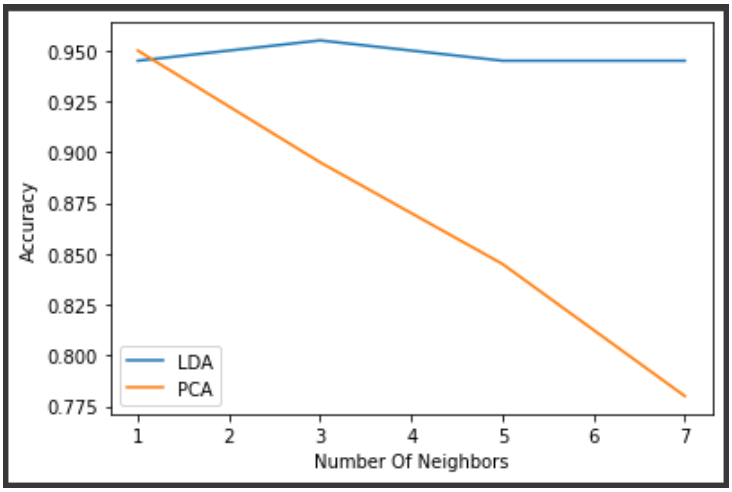
Classifier Tuning

We tested both our PCA/LDA models using the KNN classifier for the values 1, 3, 5, and 7.

Note: For PCA, we assumed a fixed alpha of 0.85 for simplicity's sake.

Measured Accuracy

Number of Neighbors (K)	PCA (Alpha = 0.85)	LDA
1	0.95	0.945
3	0.895	0.955
5	0.845	0.945
7	0.78	0.945



Non-Face vs Face Classification

Loading Non-Face Images

```
[14] def loadNonFace():
       images = []
       for id in range(1,682):
         path = f'drive/MyDrive/Pattern Lab 1 Data/Non Face Images/nonface/d ({id}).jpg'
         image = Image.open(path)
         image = image.resize((92, 112)).convert('L')
         imageVector = np.asarray(image)
         imageVector = imageVector.flatten()
         images.append(imageVector)
       npimages = np.array(images, dtype=np.uint8)

       return npimages
```

Loading the non-face images is equivalent to loading the faces images but with only one difference:

- Resizing the data dimensions to be equal to the face's dimensions.
- Converting the image to grayscale to avoid the unnecessary color dimensions.

Around 680 non face picture mostly containing vehicles, plants, and fruits were used for this part.

Splitting the Non-Face and Face Images

```
[40] def get_faces_nonfaces_data(faces, nonfaces, k):
       # k: number of nonfaces in training
       # labels: face => 1, nonface => 0
       nonfacesTesting = nonfaces[:200]
       nonfacesTraining = nonfaces[200:200+k]
       facesTraining,_, facesTesting, _ = splitdataEvenly(faces, np.array([1]*400))

       hybridTraining = np.concatenate((nonfacesTraining, facesTraining), axis=0)
       hybridTrainingLabels = np.asarray([0]*len(nonfacesTraining) + [1]*len(facesTraining))

       hybridTesting = np.concatenate((nonfacesTesting, facesTesting), axis = 0)
       hybridTestingLabels = np.asarray([0]*len(nonfacesTesting) + [1]*len(facesTesting))

       return hybridTraining, hybridTrainingLabels, hybridTesting, hybridTestingLabels
```

We split each dataset individually then concatenate them and their labels:

- Faces' Label: 1.
- Non-Faces' Label: 0.

Subsidiary Function

```
def number_success_fail(predicted_labels, actual_labels):
  success = [0,0]
  fail = [0,0]
  for i in range(len(predicted_labels)):
    if predicted_labels[i] == actual_labels[i]:
      success[actual_labels[i]] += 1
    else:
      fail[actual_labels[i]] += 1
  return success, fail
```

This function simply counts the number of successes/failures to be viewed after every test.

Testing The PCA Model on The Hybrid Dataset

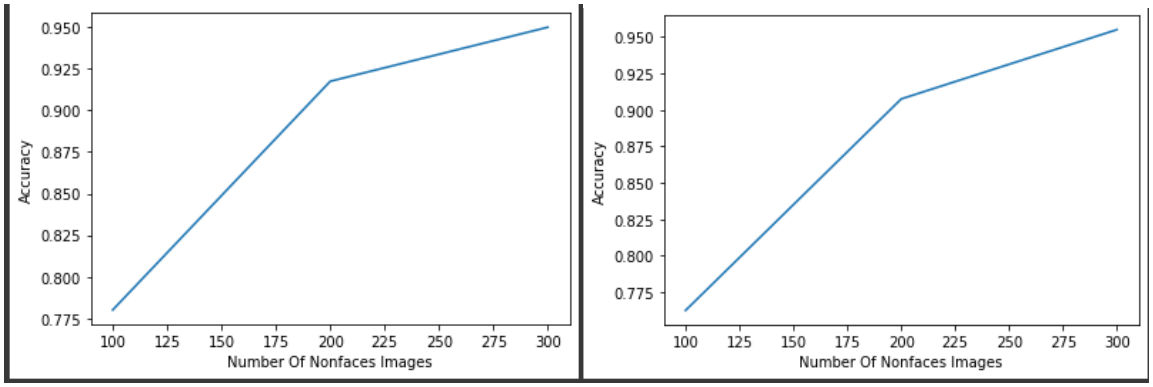
```
[41] nonfaces = loadNonFace()
     faces = images
     training, _, testing, testingLabels = get_faces_nonfaces_data(faces, nonfaces, 100)

     for a in [0.8, 0.9]:
       accuracies = []
       n_nonfaces_images = [100,200,300]
       for k in n_nonfaces_images:
         training, trainingLabels, testing, testingLabels = get_faces_nonfaces_data(faces, nonfaces, k)

         projectedData, projectionMatrix = PCA(training, a)
         projectedTestData = np.matmul((testing - np.mean(testing, axis=0)), projectionMatrix)

         knc = KNeighborsClassifier(n_neighbors=1)
         knc.fit(projectedData, trainingLabels)

         predictedLabels = knc.predict(projectedTestData)
         n_success, n_fail = number_success_fail(predictedLabels, testingLabels)
         accuracy = accuracy_score(testingLabels, predictedLabels)
         accuracies.append(accuracy)
         print(f'for {a}, {k}:')
         print(f'\tfaces #success: {n_success[1]}, #fail: {n_fail[1]} ')
         print(f'\tnonfaces #success: {n_success[0]}, #fail: {n_fail[0]} ')
         print(f'\tAccuarcy: {accuracy*100}%')

       plt.plot([100,200,300], accuracies)
       plt.xlabel('Number Of Nonfaces Images')
       plt.ylabel('Accuracy')
       plt.show()
```

For this dataset, we applied our test on alpha = 0.8 and 0.9.

We train our model on 200 faces images and 200 nonface images, and consequently train it on a dataset containing 300, 400, and 500 pictures containing face and non-face images.

MEASURED ACCURACY

Alpha	No. Of Face Images	No. Of Non-Face Images	Accuracy	Successes/Fails Face Images	Successes/Fails Non-Face Images
0.8	200	100	0.78	200/0	112/88
0.8	200	200	0.9175	199/1	168/32
0.8	200	300	0.95	198/2	182/18
0.9	200	100	0.7625	200/0	105/95
0.9	200	200	0.9075	200/0	163/37
0.9	200	300	0.95.5	200/0	182/18



Alpha = 0.8 **Alpha = 0.9**

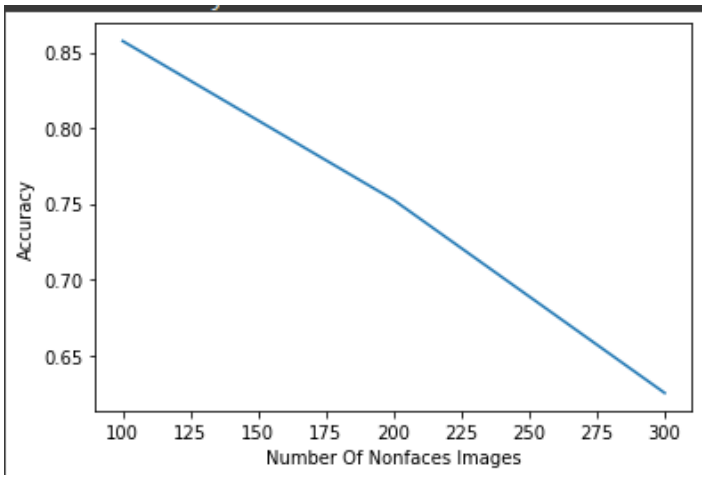
The accuracy of the model increases the more nonface images we add.

Testing The LDA Model on The Hybrid Dataset

```
print("LDA:")

accuracies = []
n_nonfaces_images = [100,200,300]
for k in n_nonfaces_images:
  training, trainingLabels, testing, testingLabels = get_faces_nonfaces_data(faces, nonfaces, k)

  projectionMatrix = LDA(1, training, trainingLabels)
  projectedData = np.matmul((training - np.mean(training, axis=0)), projectionMatrix)
  projectedTestData = np.matmul((testing - np.mean(testing, axis=0)), projectionMatrix)

  knc = KNeighborsClassifier(n_neighbors=1)
  knc.fit(projectedData, trainingLabels)

  predictedLabels = knc.predict(projectedTestData)
  n_success, n_fail = number_success_fail(predictedLabels, testingLabels)
  accuracy = accuracy_score(testingLabels, predictedLabels)
  accuracies.append(accuracy)
  print(f'for {a}, {k}:')
  print(f'\tfaces #success: {n_success[1]}, #fail: {n_fail[1]} ')
  print(f'\tnonfaces #success: {n_success[0]}, #fail: {n_fail[0]} ')
  print(f'\tAccuarcy: {accuracy*100}%')

plt.plot([100,200,300], accuracies)
plt.xlabel('Number Of Nonfaces Images')
plt.ylabel('Accuracy')
plt.show()
```

Measured Accuracy

No. Of Face Images	No. Of Non-Face Images	Accuracy	Successes/Fails Face Images	Successes/Fails Non-Face Images
200	100	0.8575	188/12	155/45
200	200	0.7525	120/80	181/19
200	300	0.625	61/139	189/11



Unlike the PCA, the accuracy for face images decrease drastically while the nonface images accuracy increase the more nonface images we add.

BONUS 1

SPLITTING DATA 70/30

```python
def splitdata(dataMatrix, labelVector, noOfSamplesTrain, noOfSamplesTest):
  k = 0
  print(dataMatrix.dtype, labelVector.dtype)
  trainingMatrix = np.zeros((int(dataMatrix.shape[0]/2), dataMatrix.shape[1]), dtype=np.uint8)
  trainingVector = np.zeros(int(labelVector.shape[0]/2), dtype=np.uint8)
  testingMatrix = np.zeros((int(dataMatrix.shape[0]/2), dataMatrix.shape[1]), dtype=np.uint8)
  testingVector = np.zeros(int(labelVector.shape[0]/2), dtype=np.uint8)
  for i in range(dataMatrix.shape[0]):
    if (k >= 0 and k < noOfSamplesTrain):
      trainingMatrix[int(i/2)] = dataMatrix[i]
      trainingVector[int(i/2)] = labelVector[i]
    elif(k >= noOfSamplesTrain and k < noOfSamplesTest + noOfSamplesTrain):
      testingMatrix[floor(i/2)] = dataMatrix[i]
      testingVector[floor(i/2)] = labelVector[i]
    else:
      k = 0
    k += 1
  return trainingMatrix, trainingVector, testingMatrix, testingVector
```

LDA ACCURACY

```
for 1 neighbors: accuracy = 0.97
for 3 neighbors: accuracy = 0.945
for 5 neighbors: accuracy = 0.96
for 7 neighbors: accuracy = 0.95
```

The accuracy increases compared to the 50/50 LDA model, as we increased our training data which improves our accuracy but might cause a lot of overfitting if we try the model with different data.

BONUS 2

Random PCA chooses random projection which makes it fast but not well suited for high dimensional data. Time Complexity = O(nkd)

More info

```
[ ]  randPCA = RPCA(n_components=50, svd_solver='randomized')
     projectedData = randPCA.fit_transform(trainingMatrix)
     projectedTestData = randPCA.transform(testingMatrix)
     K = [1, 3, 5, 7]
     for k in K:
       knc = KNeighborsClassifier(n_neighbors=k)
       knc.fit(projectedData, trainingVector)
       predictedLabels = knc.predict(projectedTestData)
       accuracy = accuracy_score(testingVector, predictedLabels)
       print(f'for {k} neighbors: accuracy = {accuracy}')
```

```
for 1 neighbors: accuracy = 0.95
for 3 neighbors: accuracy = 0.895
for 5 neighbors: accuracy = 0.85
for 7 neighbors: accuracy = 0.78
```

Quadratic Discriminant Analysis is a generalization of LDA that can learn quadratic boundaries and assumes a gaussian distribution for classes.

More Info

```
qda = QDA()
qda.fit(trainingMatrix, trainingVector)
predictedLabels = qda.predict(testingMatrix)
accuracy = accuracy_score(testingVector, predictedLabels)
print(f'accuracy = {accuracy}')
```

```
/usr/local/lib/python3.9/dist-packages/sklearn/discriminant_analysis.py:926: UserWarning: Variables are collinear
  warnings.warn("Variables are collinear")
accuracy = 0.05
```