Pattern recognition

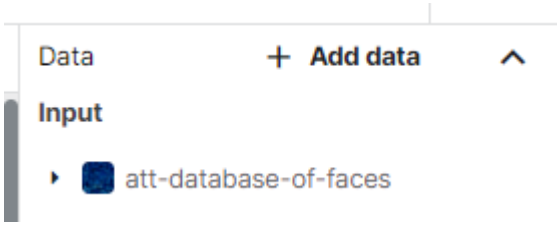
Assignment 1

Face recognition

ID	Name
18011777	مصطفى أحمد كمال أحمد مصباح
18010717	زياد بديع السعيد البنا
18011169	عمرو ابراهيم احمد

Download the dataset and understand the format:

- Importing the dataset




```
data = np.asarray(load_images('../input/att-database-of-faces'))
```

Generate the data matrix and the label vectors:

- Converting every image into a vector of 10304 values corresponding to the image size and then stacking the 400 vectors into a single Data Matrix D and generate the label vector y.

```
def load_images(path):
    folders = os.listdir(path)
    data = []
    for folder in folders:
        if(folder != "README"):
            images = os.listdir(os.path.join(path, folder))
            for image in images:
                i = Image.open(os.path.join(path, os.path.join(folder, image)))
                i = i.convert("L")
                i = np.asarray(i, dtype = np.uint8)
                i = np.append(i, int(folder[1:]))
                data.append(i.tolist())
    return data
```

Split the dataset into training and test sets:

- Splitting the data matrix (odd rows for training and even rows for testing).

```
training = data[1:][::2] # odd
testing = data[0:][::2] # even
```

- Splitting the labels vector accordingly.

```
xTraining = training[:,:-1]
print(xTraining.shape)
yTraining = training[:,-1]
print(yTraining.shape)

xTesting = testing[:,:-1]
print(xTesting.shape)
yTesting = testing[:,-1]
print(yTesting.shape)
```

Classification using PCA:

The goal is to reduce the dimensionality of the dataset from 10304 dimensions to lower dimensions .

This reduces the learning complexity of the machine and helps visualizing the data.

It's done using the Principal Component Analysis(PCA).

- Compute the mean for train data matrix and test data matrix
- Compute centre the train data matrix, test data matrix
- Compute covariance matrix
- Compute eigen values and vector
- Sort eigen values and associated vector in descending order

```
def getEginValueAndVector(xTraining,xTesting):
  images_train=np.array(xTraining)
  images_test=np.array(xTesting)
  # get mean vecgtor of image train
  mean_train = images_train.mean(axis=0)
  # get mean vecgtor of image test
  mean_test=images_test.mean(axis=0)
  Z_train = images_train - mean_train
  Z_test = images_test - mean_test
  cov = (1/len(images_train)) * np.dot(np.transpose(Z_train), Z_train)
  value, vector = np.linalg.eigh(cov)
  idx=np.argsort(value)[::-1]
  print(idx)
  value=value[idx]
  vector=vector[:,idx]
  return value,vector,Z_train,Z_test
```

- Iterate through eigen values until we reach specific alpha
- Get the projection matrix
- Compute the new train and test data matrix

```
def PCA(value,vector,Z_train,Z_test ,alpha):
        i=0
        valueDim=0
        sumValue=np.sum(value)
        ratio=0;
        for v in value:
            valueDim+=v
            ratio=valueDim/sumValue
            i+=1
            if(ratio>=alpha):
                break
        projection_matrix=vector[: , 0:i]
        new_dataSet_train=np.transpose(np.dot(np.transpose(projection_matrix),np.transpose(Z_train)))
        new_dataSet_test=np.transpose(np.dot(np.transpose(projection_matrix),np.transpose(Z_test)))
        return new_dataSet_train,new_dataSet_test
```

- Different values of alpha (0.8,0.85,0.9,0.95) are used to get different values for dimensionality reduction.

```
#train pca over face dataset
alpha = [0.8,0.85,0.9,0.95]
nn=[1,3,5,7]
xTraining,yTraining,xTesting,yTesting=getFacesData()
value,vector,Z_train,Z_test=getEginValueAndVector(xTraining,xTesting)
print("N              Alpha         Accuracy")
for n in nn:
    #Create KNN Classifier
    for a in alpha:
        x_train,x_test=PCA(value,vector,Z_train,Z_test,a)
        knn = KNeighborsClassifier(n_neighbors=n)
        #Train the model using the training sets
        knn.fit(x_train,yTraining)
        #Predict the response for test dataset
        y_pred = knn.predict(x_test)
        accuary=metrics.accuracy_score(yTesting, y_pred)
        print(f"{n}            {a}            {accuary}\n")
```

- To test our algorithm we use the knn model to train our data, using parameter tuning and trying k values of 1 , 3 ,5 and 7. it was noticed that the higher K value the lower the accuracy of the algorithm gets and the best values for alpha are 0.8, 0.85

N	Alpha	Accuracy
1	0.8	0.94
1	0.85	0.945
1	0.9	0.935
1	0.95	0.935
3	0.8	0.82
3	0.85	0.845
3	0.9	0.845
3	0.95	0.84
5	0.8	0.805
5	0.85	0.8
5	0.9	0.79
5	0.95	0.79
7	0.8	0.775
7	0.85	0.765
7	0.9	0.755
7	0.95	0.75

Classification using LDA:

The goal is to reduce the dimensionality of the dataset from 10304 dimensions to 39.

This reduces the learning complexity of the machine and helps visualizing the data.

It's done using the linear discriminant analysis algorithm (LDA).

- Get number of features (dimensions) and number of unique classes (40 classes in our case):

```
def LDA(x,y):
    numberOfFeatures = x.shape[1]
    labels = np.unique(y)
```

- Compute overall mean:

```
Mu = np.mean(x, axis=0)
```

- Initialize Sb(between classes scatter matrix) and S (within class scatter matrix)

```
Sb = np.zeros((numberOfFeatures,numberOfFeatures))
S = np.zeros((numberOfFeatures,numberOfFeatures))
```

- For each class, get Di(class i's data) , calculate the mean of Di (Mui)

Sb = ∑ni (Mui − Mu)(Mui− Mu) T

Centralize data

Zi = Di – (Mui)T

Si = (Zi)T.Z

S = ∑Si

```
for c in labels:
    Di = x[y == c]
    Mui = np.mean(Di, axis=0)
    # Sb = sigma(ni*(Mui-M)*(Mui-M)Transpose)
    MeanDifference = (Mui - Mu).reshape(numberOfFeatures,1)
    Sb += Di.shape[0]*np.dot( MeanDifference , np.transpose(MeanDifference))
    #Sb += Di.shape[0]*np.dot( (Mui - Mu) , np.transpose(Mui-Mu))
    Zi = Di - np.transpose(Mui)
    Si = np.dot(np.transpose(Zi), Zi)
    S += Si
```

- Compute dominant eigen vectors from S-1.Sb (in our case we take the most 39 dominant vectors)

```
A = np.dot(np.linalg.inv(S),Sb)
# get eigen vectors and values
# what is the diffrence between eigh and eig ?
eigenValues , eigenVectors = np.linalg.eigh(A)
#eigenValues , eigenVectors = np.linalg.eig(A)

eigenVectors = np.transpose(eigenVectors)
# sort in decreasing order
idxs = np.argsort(abs(eigenValues))[::-1]
eigenValues = eigenValues[idxs]
eigenVectors = eigenVectors[idxs]
# return the first n eigenVectors in our case we need the 39th dominant eigenVectors
return eigenVectors[0 : dominantEigenVecotrsTaken]
```

- The return of this function represents our projection matrix, we project our desired data to reduce its dimensionality from 10304 to 39

```
def Project(DataMatrix , ProjectionMatrix):
        return np.dot(DataMatrix , np.transpose(ProjectionMatrix))
ProjectionMatrix = LDA(xTraining , yTraining)
ProjectionMatrix.shape
```

Data shape before LDA => (400,10304)

Data shape after LDA => (400,39)

- Apply the LDA algorithm on the training data and then use the same projection matrix for both training and testing data

```
xProjectedTrain = Project(xTraining, ProjectionMatrix)
xProjectedTest = Project(xTesting, ProjectionMatrix)
xProjectedTrain.shape
```

- To test our algorithm we use the knn model to train our data, using parameter tuning and trying k values of 1 , 3 ,5 and 7. it was noticed that the higher K value the lower the accuracy of the algorithm gets

```
from sklearn.model_selection import GridSearchCV
from sklearn.neighbors import KNeighborsClassifier
from sklearn import metrics

def analyze_model(trained_model, prediction,x_test, y_test , k):
    print(f'k={k}: accuracy={metrics.accuracy_score(y_test, prediction )} ')

def knn_classifier (x_train, y_train, x_test, y_test):
    kArray = [1,3,5,7]
    for i in kArray:
        #classifier = classifier = GridSearchCV(KNeighborsClassifier(), {
        #    'n_neighbors': (1,9,2)
        #})
        classifier = KNeighborsClassifier(n_neighbors = i)
        # fit the data to train the model
        trained = classifier.fit(x_train, y_train)
        # find y_predicted.
        predicted = classifier.predict(x_test)
        #print(classifier.best_params_)
        # report analysis and draw the confusion matrix
        analyze_model(trained, predicted, x_test, y_test , i)
knn_classifier(xProjectedTrain , yTraining , xProjectedTest , yTesting)
```

```
k=1: accuracy=0.95
k=3: accuracy=0.86
k=5: accuracy=0.835
k=7: accuracy=0.815
```

Faces VS non-Faces:

Using the same techniques to reduce the data dimensionality, we train another model to differentiate between faces and non faces.

We used the natural-images dataset along with att-dataset-of-faces to train our model

Data Splitting:

Read all folders from dataset and resize the images to 91X112 before converting it

```
from random import sample
def faceVSNonFace(path , label):
    folders = sorted(os.listdir(path))
    data = []
    for folder in folders:
        if(folder != "README" and folder != "person"):
            images = sorted(os.listdir(os.path.join(path, folder)))
            for image in images:
                i = Image.open(os.path.join(path, os.path.join(folder, image)))
                i = i.resize((92,112))
                i = i.convert("L")
                i = np.asarray(i, dtype = np.uint8)
                i = np.append(i, label)
                data.append(i.tolist())
    return data
```

Take a random sample of the non-face dataset and split all data into training and testing sets (50-50)

```
from sklearn.model_selection import train_test_split
def splitData(nOfRows):
    facesData = np.asarray(faceVSNonFace('../input/att-database-of-faces' , 1))
    nonFacesData = sample(faceVSNonFace('../input/natural-images/natural_images' , 0) , nOfRows)
    #nonFacesData = np.asarray(faceVSNonFace('../input/natural-images/natural_images' , 0))
    nonFacesData = np.asarray(nonFacesData)
    print(facesData.shape)
    print(nonFacesData.shape)

    # split both into train and test
    SEED = 0
    #faces
    xFace = facesData[:,:-1]
    yFace = facesData[:,-1]
    #non faces
    xNonFace = nonFacesData[:,:-1]
    yNonFace = nonFacesData[:,-1]

    xFaceTrain , xFaceTest, yFaceTrain , yFaceTest = train_test_split(xFace, yFace, test_size = 0.5, random_state = SEED)
    xNonFaceTrain , xNonFaceTest , yNonFaceTrain , yNonFaceTest = train_test_split(xNonFace, yNonFace, test_size = 0.5, random_state = SEED)

    xTrainData = np.concatenate((xFaceTrain, xNonFaceTrain), axis=0)
    yTrainData = np.concatenate((yFaceTrain, yNonFaceTrain), axis=0)

    xTestData = np.concatenate((xFaceTest, xNonFaceTest), axis=0)
    yTestData = np.concatenate((yFaceTest, yNonFaceTest), axis=0)

    # shuffle train data
    randomize = np.arange(len(xTrainData))
    np.random.shuffle(randomize)
    xTrainData = xTrainData[randomize]
    yTrainData = yTrainData[randomize]
    return xTrainData , yTrainData , xTestData , yTestData
```

Same as before, we pass our data into both algorithms to reduce their dimensions before training our model

PCA:

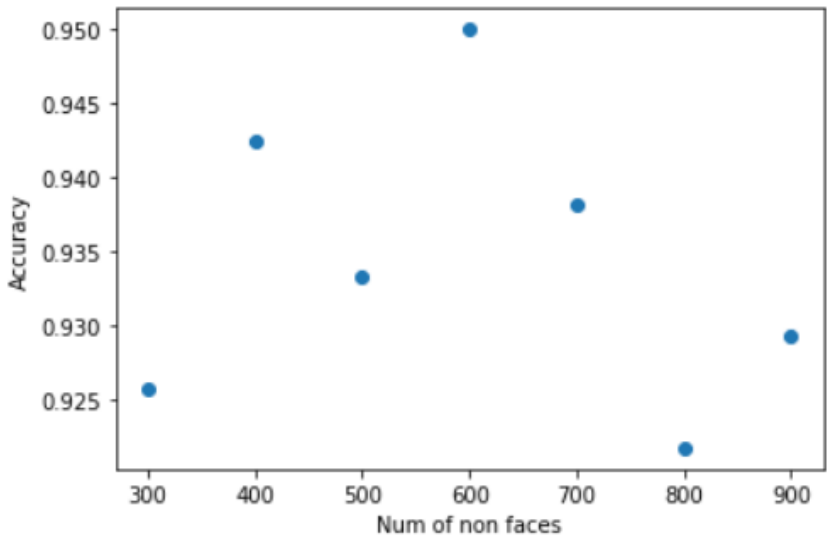
We fix the number of face images and try different number of non-faces images

```
#train pca over face VS nonfaces dataset
alpha = [0.8,0.85,0.9,0.95]
nn=[1,3,5,7]
numofRows=[300,400,500,600,700]
print("NumOfNonHuman             N              Alpha           Accuracy")
for num in numofRows:
    xTraining,yTraining,xTesting,yTesting=splitData(num)
    value,vector,Z_train,Z_test=getEginValueAndVector(xTraining,xTesting)
    for n in nn:
        #Create KNN Classifier
        for a in alpha:
            x_train,x_test=PCA(value,vector,Z_train,Z_test,a)
            knn = KNeighborsClassifier(n_neighbors=n)
            #Train the model using the training sets
            knn.fit(x_train,yTraining)
            #Predict the response for test dataset
            y_pred = knn.predict(x_test)
            accuary=metrics.accuracy_score(yTesting, y_pred)
            print(f"{num}           {n}              {a}             {accuary}\n")
```

The training accuracies to the number of rows taken were as follows:

NumOfNonHuman	N	Alpha	Accuracy
300	1	0.8	0.8942857142857142
300	1	0.85	0.8857142857142857
300	1	0.9	0.8742857142857143
300	1	0.95	0.86
300	3	0.8	0.8714285714285714
300	3	0.85	0.86
300	3	0.9	0.84
300	3	0.95	0.8257142857142857
300	5	0.8	0.8457142857142858
300	5	0.85	0.84
300	5	0.9	0.82
300	5	0.95	0.7971428571428572
300	7	0.8	0.8428571428571429
300	7	0.85	0.8228571428571428
400	1	0.8	0.94
400	1	0.85	0.935
400	1	0.9	0.9175
400	1	0.95	0.9025
400	3	0.8	0.915
400	3	0.85	0.91
400	3	0.9	0.8975
400	3	0.95	0.8875
400	5	0.8	0.9025
400	5	0.85	0.895
400	5	0.9	0.875
400	5	0.95	0.865
400	7	0.8	0.895
400	7	0.85	0.885
400	7	0.9	0.8675
400	7	0.95	0.85

600	3	0.9	0.862
600	3	0.95	0.836
600	5	0.8	0.9
600	5	0.85	0.866
600	5	0.9	0.84
600	5	0.95	0.814
600	7	0.8	0.89
600	7	0.85	0.844
600	7	0.9	0.826
600	7	0.95	0.794
700	1	0.8	0.9381818181818182
700	1	0.85	0.9272727272727272
700	1	0.9	0.9272727272727272
700	1	0.95	0.9181818181818182
700	3	0.8	0.9290909090909091
700	3	0.85	0.92
700	3	0.9	0.9090909090909091
700	3	0.95	0.8963636363636364



LDA:

We fix the number of face images (400 instances) and try different number of non-faces images

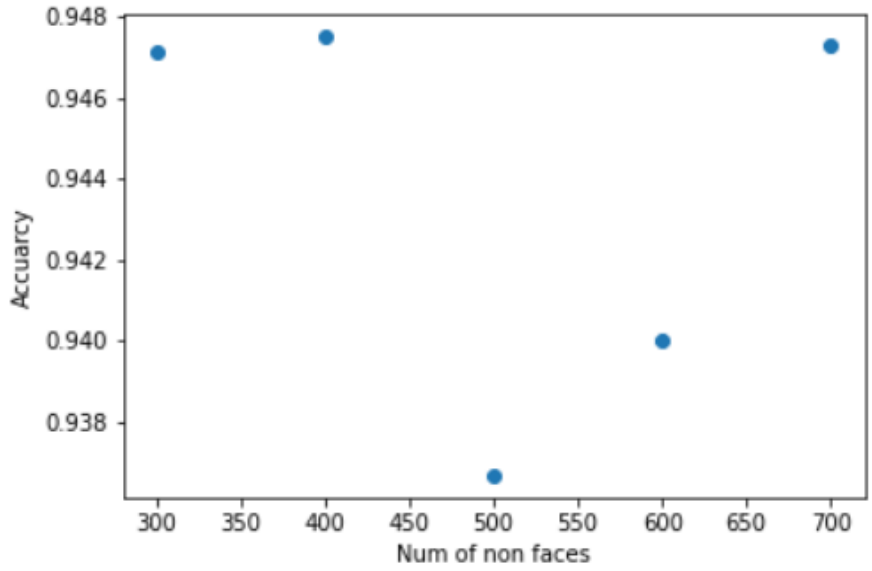
```
numofRows=[300,400,500,600,700]
for n in numofRows:
    xTraining,yTraining,xTesting,yTesting=splitData(n)
    ProjectionMatrix = LDA(xTraining , yTraining)
    ProjectionMatrix.shape
    xProjectedTrain = Project(xTraining, ProjectionMatrix)
    xProjectedTest = Project(xTesting, ProjectionMatrix)
    xProjectedTrain.shape
    print(f'for n = {n}')
    knn_classifier(xProjectedTrain , yTraining , xProjectedTest , yTesting)
```

How many eigenvectors shall we take?

Different number of eigenvectors where tried. It was predicted that **1 eigenvector** could have done the job as we only have 2 classes in the face VS non-Face problem but after analysing other results it was found that **12 eigenvectors** was the best value that gives high accuracies.

The training accuracies to the number of rows taken were as follows:

```
for n = 300
k=1: accuracy=0.9571428571428572
k=3: accuracy=0.9485714285714286
k=5: accuracy=0.9371428571428572
k=7: accuracy=0.9314285714285714
[0 1]
for n = 400
k=1: accuracy=0.9425
k=3: accuracy=0.9475
k=5: accuracy=0.935
k=7: accuracy=0.9325
[0 1]
for n = 500
k=1: accuracy=0.9266666666666666
k=3: accuracy=0.92
k=5: accuracy=0.9111111111111111
k=7: accuracy=0.9088888888888889
[0 1]
for n = 600
k=1: accuracy=0.94
k=3: accuracy=0.94
k=5: accuracy=0.932
k=7: accuracy=0.922
[0 1]
for n = 700
k=1: accuracy=0.9472727272727273
k=3: accuracy=0.9381818181818182
k=5: accuracy=0.9345454545454546
k=7: accuracy=0.9254545454545454
```



Conclusion:

LDA reduced the data's dimensions in a good way that helped the model get high accuracies

the accuracy of the model is affected by 2 factors:

1. **The balancing of the data**, this means that if the data is balanced (not biased to a specific class) the model accuracy will increase.
2. **The number of records**, the more we train our model the better it gets.

Failure and success cases:

1. **Failure cases:**

- 

- 
- 
- 

2. **Success cases:**

- 
- 
- 

Bonus:

Splitting the data 70-30 instead of 50-50:

```
def Bonus():
    data = np.asarray(load_images('../input/att-database-of-faces'))
    data.shape
    training = []
    testing = []
    i=0
    while i<len(data):
        for j in range(7):
            training.append(data[i])
            i+=1
        for j in range(3):
            testing.append(data[i])
            i+=1
    training = np.asarray(training)
    testing = np.asarray(testing)
    print(training.shape)
    print(testing.shape)
    print(testing[:7,:])
    xTraining = training[:,:-1]
    print(xTraining.shape)
    yTraining = training[:,-1]
    print(yTraining.shape)
    xTesting = testing[:,:-1]
    print(xTesting.shape)
    yTesting = testing[:,-1]
    print(yTesting.shape)
    return xTraining,yTraining,xTesting,yTesting
```

Using the resulting splits to train our model after reducing their dimensionality using both **PCA & LDA** algorithms:

PCA:

N	Alpha	Accuracy
1	0.8	0.9583333333333334
1	0.85	0.9416666666666667
1	0.9	0.9333333333333333
1	0.95	0.925
3	0.8	0.9083333333333333
3	0.85	0.925
3	0.9	0.925
3	0.95	0.9
5	0.8	0.8833333333333333
5	0.85	0.9
5	0.9	0.8916666666666667
5	0.95	0.8583333333333333
7	0.8	0.825
7	0.85	0.8083333333333333
7	0.9	0.7916666666666666
7	0.95	0.8

LDA:

LDA results were as follows:

```
k=1: accuracy=0.9666666666666667
k=3: accuracy=0.8833333333333333
k=5: accuracy=0.85
k=7: accuracy=0.7916666666666666
```

Conclusion:

The model results in the 70-30 split were better than the original split, this is due giving the model more data to train with and also giving the LDA algorithm more points to give better dimensionality reduction vectors