PR

Assignment#1
Face Recognition

Names	ID
Sally Kamel Soliman Armanyous	19015762
Salma Saeed Mahmoud Ghareb	19015779
Esraa Hassan Mokhtar Aboshady	19015407

Problem Statement:

We intend to perform face recognition. Face recognition means that for a given image you can tell the subject id. Our database of subject is very simple. It has 40 subjects. Below we will show the needed steps to achieve the goal of the assignment.

1. Download the Dataset and Understand the Format:

Face recognition dataset has been downloaded from Kaggle (40 classes each has 10 instances) and is taken from google drive.

Dataset has been splitted in 2 different ways:

- into two equal datasets one for training and the other one for testing
- Into 2 unequal datasets one for training with 70% of data and the other one for testing with 30% of data

Each image has been converted into contiguous flattened array that stores only the grayscale.

2. Generate the Data Matrix and the Label vector:

We read in image files from a directory, convert them to grayscale, flattens them into a 1D array, and appends them to a data matrix along with their corresponding labels. Here is a description of each point:

- Labels array: to hold the image labels.
- data_matrix: an empty array with 0 rows and 10304 columns, which is the number of pixels in each image (112 x 92), where we will append in it to get the overall data matrix.
- We will begin a loop through each directory in the specified data_path, sorted naturally.
- Then begins a nested loop through each subdirectory in the current directory, sorted naturally.
- Create the full path of the current subdirectory by joining the root path with the subdir name.
- Extract the integer label from the subdirectory name, which is assumed to start with the letter 's'.
- Then begin a loop through each file in the current subdirectory, sorted naturally and open the image file, converts it to grayscale, and assigns it to the img variable.
- We convert the 2D image array to a 1D array and assigns it to the image_vector variable.
- Appending the image_vector array to the bottom of the data_matrix array along the 0th axis (as rows). Appending the image_label to the labels list.
- Finally converting the data_matrix Numpy array into a Pandas DataFrame and returns it.

3. Split the Dataset into Training and Test sets:

The code defines the function of split_data that takes in the list of labels and the array of image data, and splits them into two sets (training set and a testing set)
Here is a breakdown of the code:

- faces_train and faces_test: Initializes empty arrays to hold the training and testing image data. Each array has 0 rows and 10304 columns, which is the number of pixels in each image (112 x 92).
- Begin a loop through each label in the labels list. Checks if the current label is even or odd based on a counter, if it is even, we append the label to the labels_test list, Otherwise, appends the label to the labels_train list.
- Then do the same thing for each image vector in the data_matrix array.
- And append the image vector to the faces_train array.
- Return the training and testing image data and labels as four separate objects.

4. Classification using PCA:

First we made a function “get_Eigens” to take the training face images and returns the eigenvalues and eigenvectors of the covariance matrix:

- Compute the mean of the face images by taking the average of each pixel across all images. This gives a single "average face" image that represents the center of the dataset.
- Subtract the mean face from each image to center the dataset around the origin.
- Then compute the covariance matrix of the centered dataset.
- Compute the eigenvalues and eigenvectors of the covariance matrix.
- Sort the eigenvalues in descending order and sort the corresponding eigenvectors accordingly.
- Return the sorted eigenvalues and eigenvectors.

Then we made a function “projection_Matrix”:

- It takes EigenValues, EigenVectors and alpha.
- Then compute the total sum of eigenvalues, which is the sum of all eigenvalues of the covariance matrix.
- Looping through the eigenvalues from largest to smallest and for each eigenvalue, increment sum by the eigenvalue and compute the ratio of sum to total sum of eigen values.
- If ratio is greater than or equal to alpha, exit the loop
- “r” is the number of eigenvalues required to retain at least alpha amount of variance, then select the first r eigenvectors from EigenVectors and store them in “U”.
- Return U

Then we made a function called “testing_pca” where we will Project the training set, andtest sets separately using the same projection matrix and report Accuracy for every value of alpha separately, it takes in : train_data, test_data, labels_train, and labels_test.

- Loop over each value in the alpha list.
- Call “projection_Matrix” function with the eigenvalues, eigenvectors, and the current value of alpha as arguments to calculate projection matrix.
- Then uses NumPy's dot function to project the train_data and test_data onto the projection matrix.
- Then apply the KNN_algorithm with the projected training data, training labels, projected test data, and test labels as arguments to perform the K-nearest neighbors classification algorithm on it.

- The function then prints out the best number of neighbors (optimal_k) and the maximum accuracy achieved (max_accuracy) by the K-nearest neighbors algorithm.
- Then repeat previous steps for each value in the alpha list.

```
testing_pca(faces_train,faces_test, labels_train, labels_test)
```

```
Alpha =  0.8
number of eigen values taken :  37
all accuracies               :  [0.93, 0.855, 0.805, 0.78]
the best accuracy among them :  0.93
the best k for the given k's :  1
Best n-neighbours:  1
K-NN Accuracy:  0.93

Alpha =  0.85
number of eigen values taken :  53
all accuracies               :  [0.94, 0.855, 0.83, 0.775]
the best accuracy among them :  0.94
the best k for the given k's :  1
Best n-neighbours:  1
K-NN Accuracy:  0.94

Alpha =  0.9
number of eigen values taken :  77
all accuracies               :  [0.945, 0.85, 0.815, 0.755]
the best accuracy among them :  0.945
the best k for the given k's :  1
Best n-neighbours:  1
K-NN Accuracy:  0.945

Alpha =  0.95
number of eigen values taken :  116
all accuracies               :  [0.935, 0.845, 0.815, 0.74]
the best accuracy among them :  0.935
the best k for the given k's :  1
Best n-neighbours:  1
K-NN Accuracy:  0.935
```

For 70% training and 30% testing

```
testing_pca(faces_train_set , faces_test_set , labels_train_set , labels_test_set)
```

```
Alpha =  0.8
number of eigen values taken :  39
all accuracies               :  [0.9583333333333334, 0.925, 0.9083333333333333, 0.8416666666666667]
the best accuracy among them :  0.9583333333333334
the best k for the given k's :  1
Best n-neighbours:  1
K-NN Accuracy:  0.9583333333333334

Alpha =  0.85
number of eigen values taken :  57
all accuracies               :  [0.9666666666666667, 0.9333333333333333, 0.9083333333333333, 0.8416666666666667]
the best accuracy among them :  0.9666666666666667
the best k for the given k's :  1
Best n-neighbours:  1
K-NN Accuracy:  0.9666666666666667

Alpha =  0.9
number of eigen values taken :  89
all accuracies               :  [0.9666666666666667, 0.9166666666666666, 0.8916666666666667, 0.85]
the best accuracy among them :  0.9666666666666667
the best k for the given k's :  1
Best n-neighbours:  1
K-NN Accuracy:  0.9666666666666667

Alpha =  0.95
number of eigen values taken :  145
all accuracies               :  [0.95, 0.9416666666666667, 0.9, 0.8]
the best accuracy among them :  0.95
the best k for the given k's :  1
Best n-neighbours:  1
K-NN Accuracy:  0.95
```

5. Classification Using LDA :

Function “LDA_Eigen” that takes in three arguments: number of classes (40), instances per class (10), and faces_train:

- First it calculates the total number of instances in the training set using the np.size function.
- Then calculates the mean vector of the overall sample by taking the mean of faces_train along the first axis (i.e., the row axis). The resulting vector is then reshaped to be a column vector with 10304 rows and 1 column.
- Then splits the training data into individual classes using a for loop.
- For each class, the function calculates the mean vector using the np.mean function, again along the first axis. The resulting vector is then reshaped to be a column vector with 10304 rows and 1 column.
- The function then calculates the between-class scatter matrix (Sb) using a for loop.

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

- Then calculates the within-class scatter matrix (s) using another for loop by subtracting the mean vector for each class from its instances, taking the dot product with its transpose, and summing the resulting matrices.
- The function computes the eigenvalues and eigenvectors of the product of the inverse of S and Sb. The resulting eigenvalues and eigenvectors are returned.

Getting train set and test set after dimensionality reduction for LDA using the function of “LDA_Projected”:

- It takes faces_train, faces_test, eigen_values, eigen_vectors, num_dominant_eigen_vectors (39).
- The function first sorts the eigenvalues in descending order and reorders the corresponding eigenvectors accordingly.
- A projection matrix is formed using the selected eigenvectors.
- The training and testing data are then projected onto the dominant eigenvectors using matrix multiplication with the projection matrix.
- The projected training and testing data are returned.

Getting the optimal k and max accuracy for 50% splitting:

```
max_accuracy, optimal_k, prediction = KNN_algorithm(projected_faces_train, labels_train, projected_faces_test, labels_test)
print("Best n-neighbours: ", optimal_k)
print("K-NN Accuracy: ", max_accuracy,"\n")

all accuracies              :  [0.945, 0.87, 0.84, 0.79]
the best accuracy among them :  0.945
the best k for the given k's :  1
Best n-neighbours:  1
K-NN Accuracy:  0.945
```

Getting the optimal k and max accuracy for 70-30 splitting:

```
[ ] max_accuracy_bonus, optimal_k_bonus, prediction_bonus = KNN_algorithm(projected_faces_train_set, labels_train_set, projected_faces_test_set, labels_test_set)
    print("Best n-neighbours: ", optimal_k_bonus)
    print("K-NN Accuracy: ", max_accuracy_bonus ,"\n")

    Best n-neighbours:  1
    K-NN Accuracy:  0.9416666666666667
```

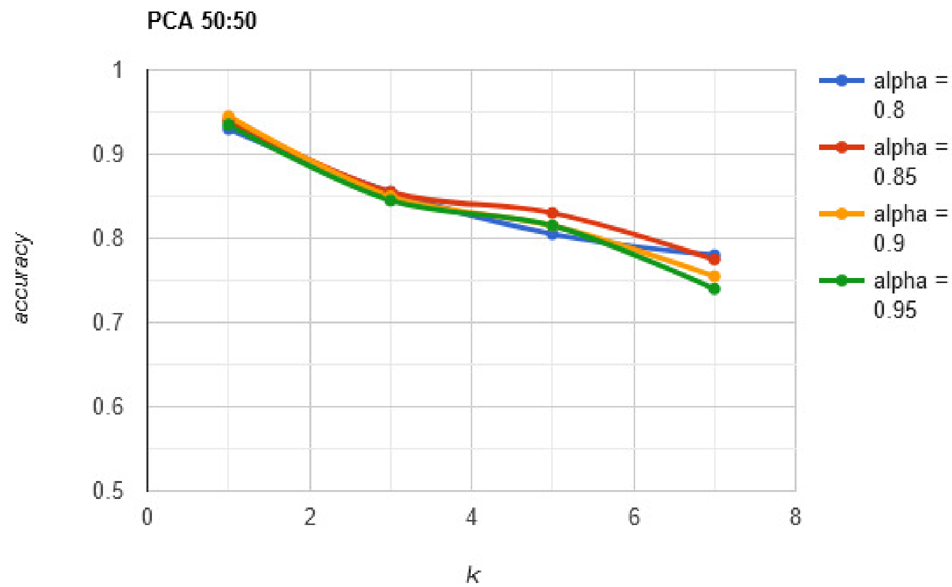
6. Classifier Tuning

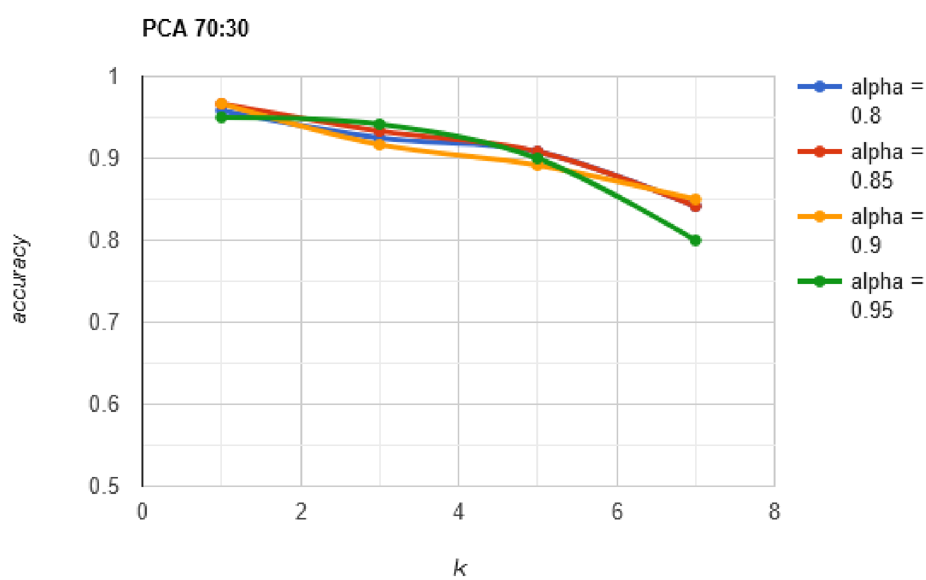
The KNN_algorithm that implements the k-nearest neighbors (KNN) algorithm for classification. It takes four parameters: faces_train_available, labels_train, faces_test_available, and labels_test. These parameters represent the training and test data and labels, respectively:

- We use a for loop to iterate over different values of k (1, 3, 5, and 7).
- For each value of k, the function creates a KNeighborsClassifier object with n_neighbors set to k, fits it to the training data using the fit method, and uses it to predict the labels of the test data using the predict method.
- Then calculating the accuracy of the prediction using the accuracy_score function and stores it in a list called all_accuracies.
- Then checking whether the current accuracy is better than the maximum accuracy seen so far, and if so, updates the maximum accuracy, the optimal value of k, and the optimal prediction.
- Finally, the function returns the maximum accuracy, the optimal value of k, and the optimal prediction.

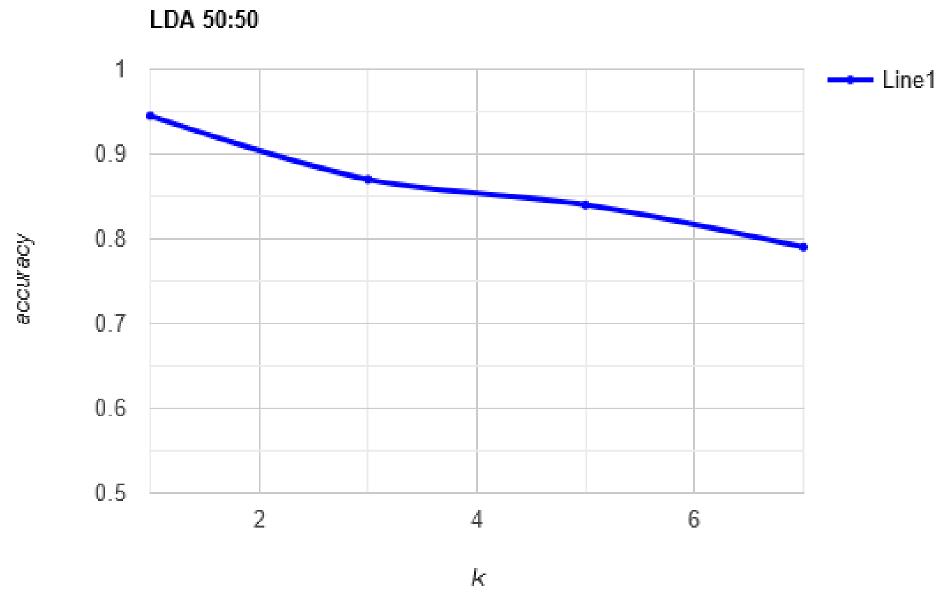
Plot the performance measure (accuracy) against the K value.

For PCA:

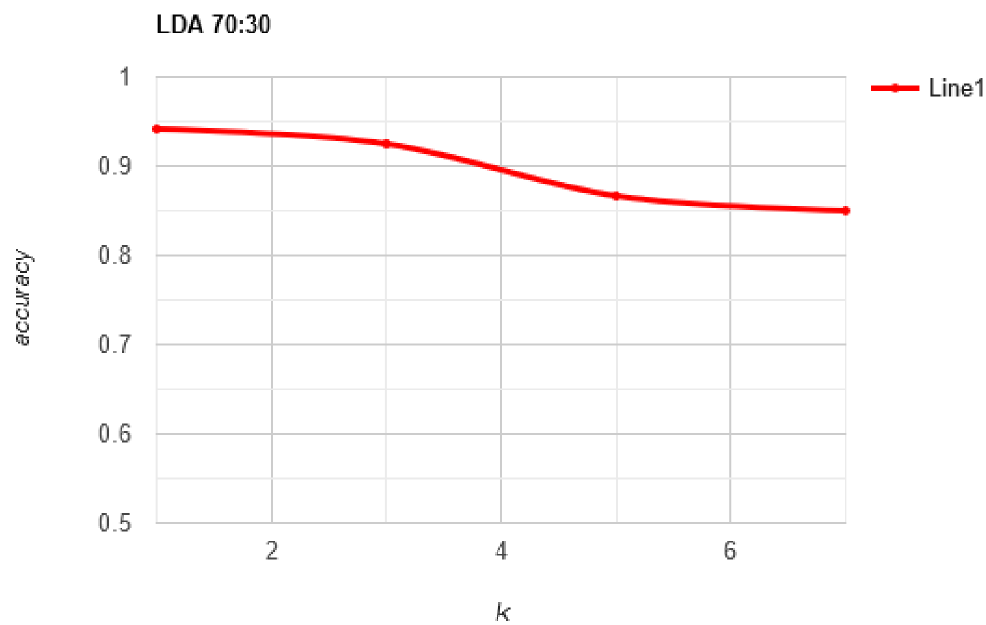




For LDA:



LDA 50:50
1
0.9
0.8
0.7
0.6
0.5
accuracy
2
4
6
k
Line1



LDA 70:30
1
0.9
0.8
0.7
0.6
0.5
accuracy
2
4
6
k
Line1

7. Compare vs Non-Face Images

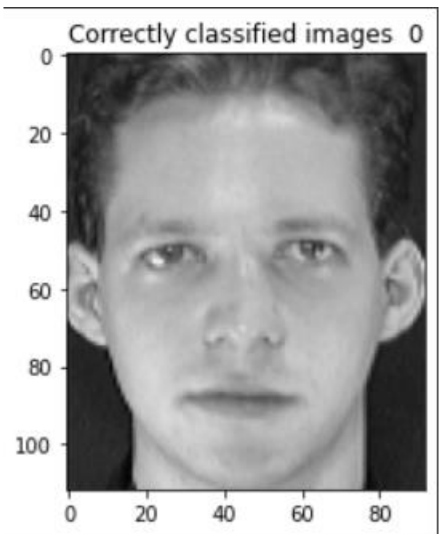
Natural images dataset has been downloaded from Kaggle as non-face dataset

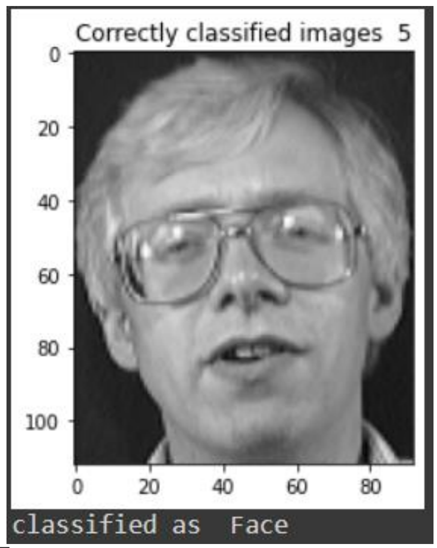
- Contains 4 classes of non faces images , each has 100 instances.
- persons images has been excluded
- Each image has been resized to 92 X 112
- It has been separated in the same way as the previous dataset

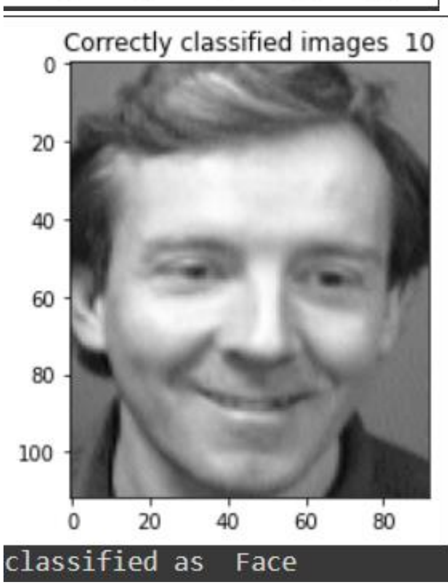
There's been 2 different type of classification

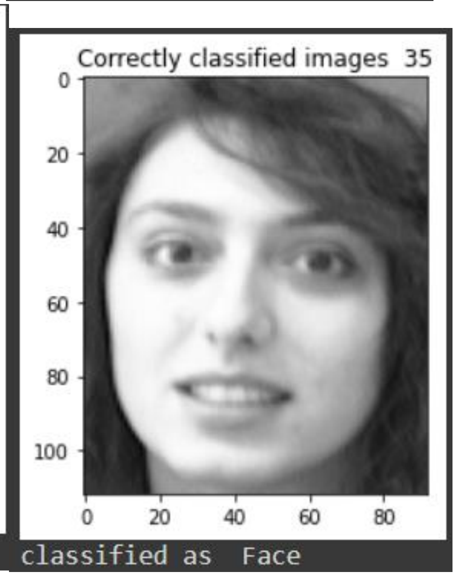
- Whether it's face or non-face images : Faces have got 1 and nonfaces have got 0 as labels.
- 70/30 and 50/50 splitting are both used
- If the image is for a face, it should be mapped into the face's label

Success cases:

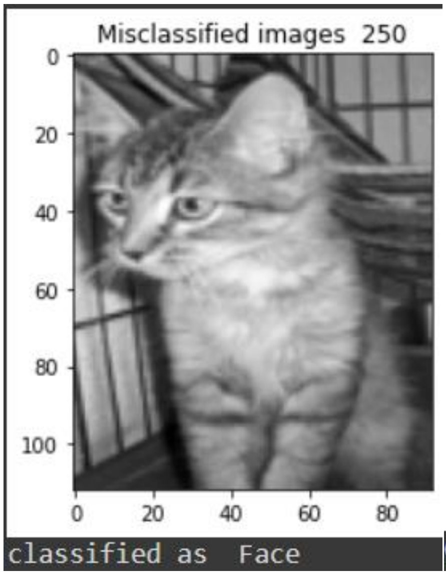


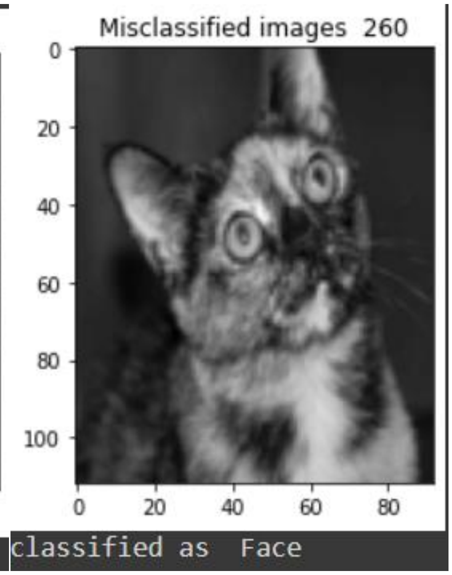


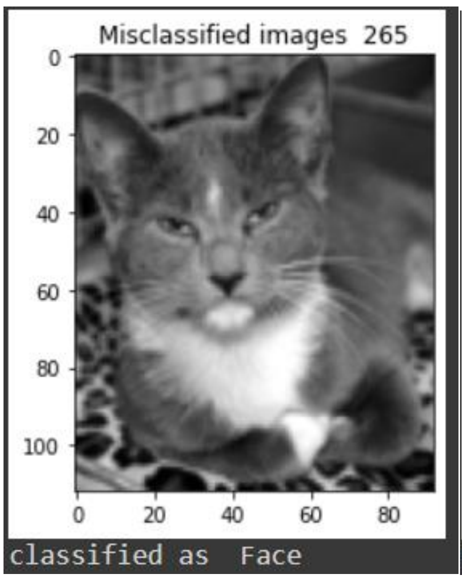


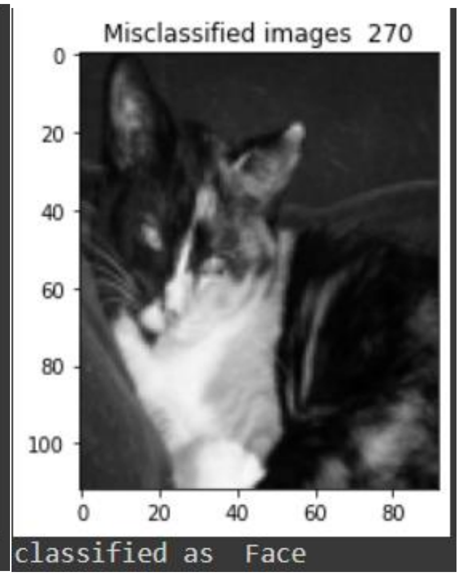


Failure cases:







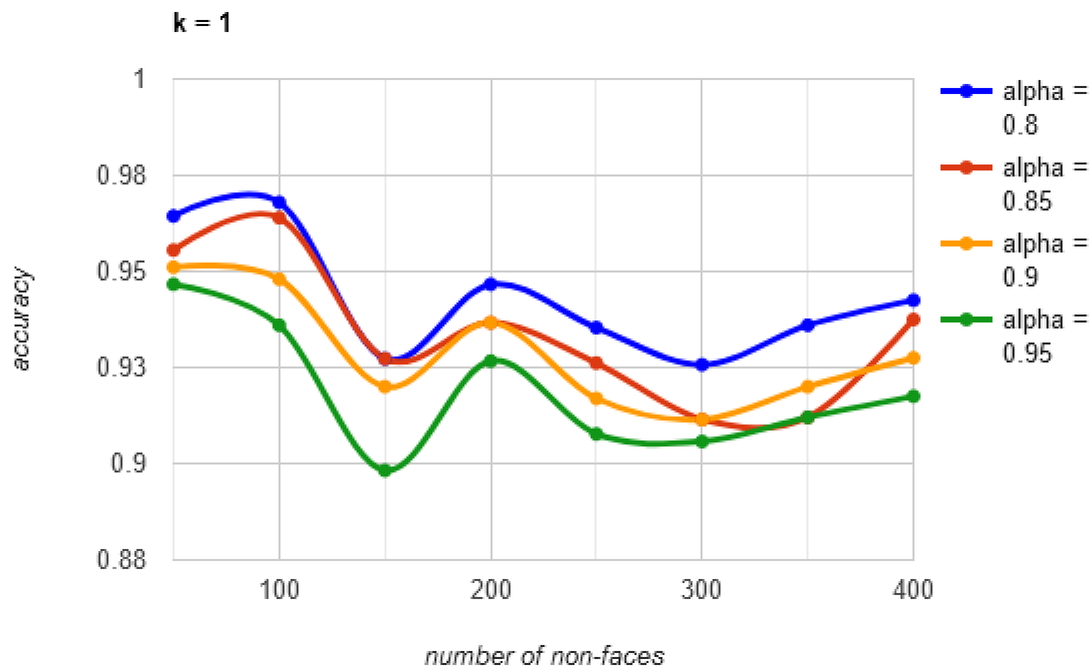


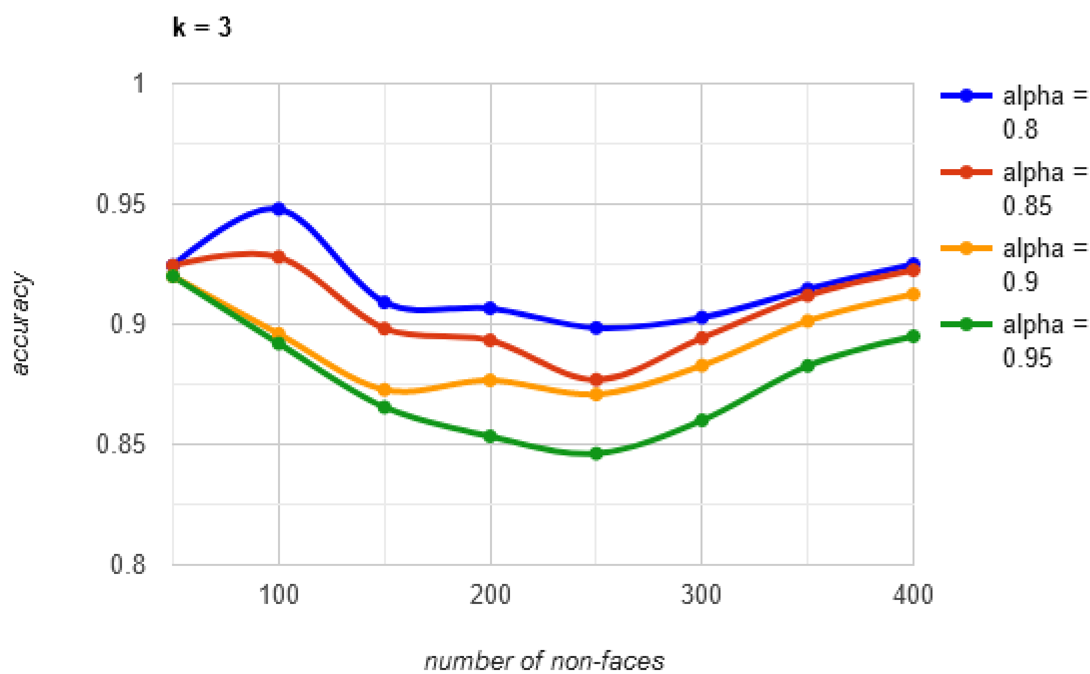
Dominant eigenvectors we will use for the LDA solution:

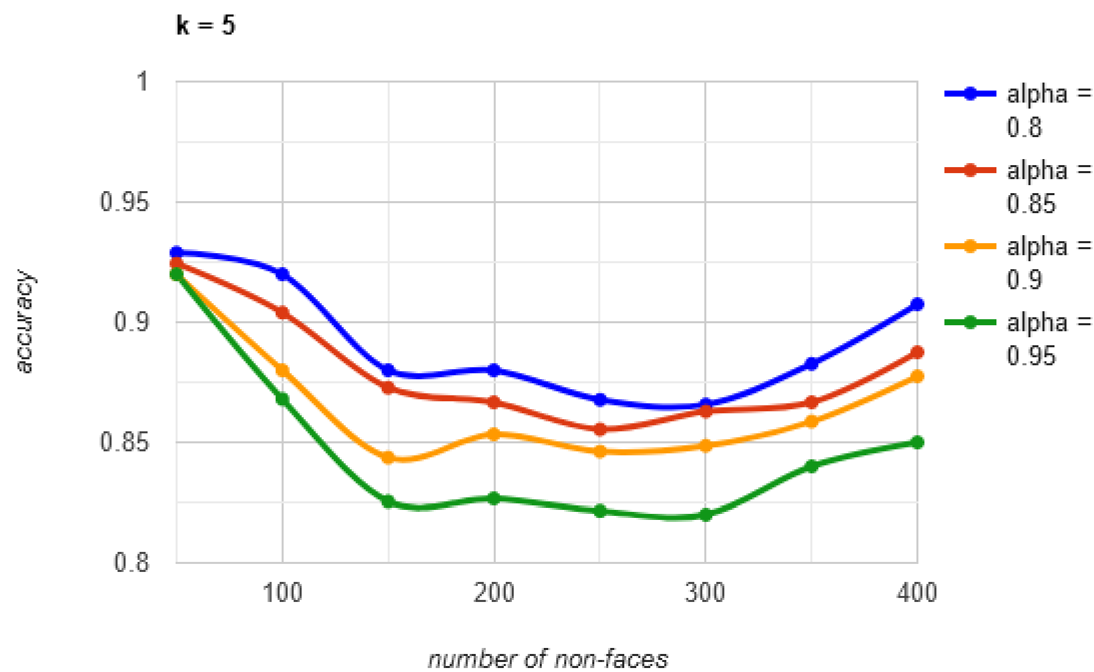
We will use 1 eigenvector as we have 2 classes and the number of eigenvectors the LDA uses is number of classes -1

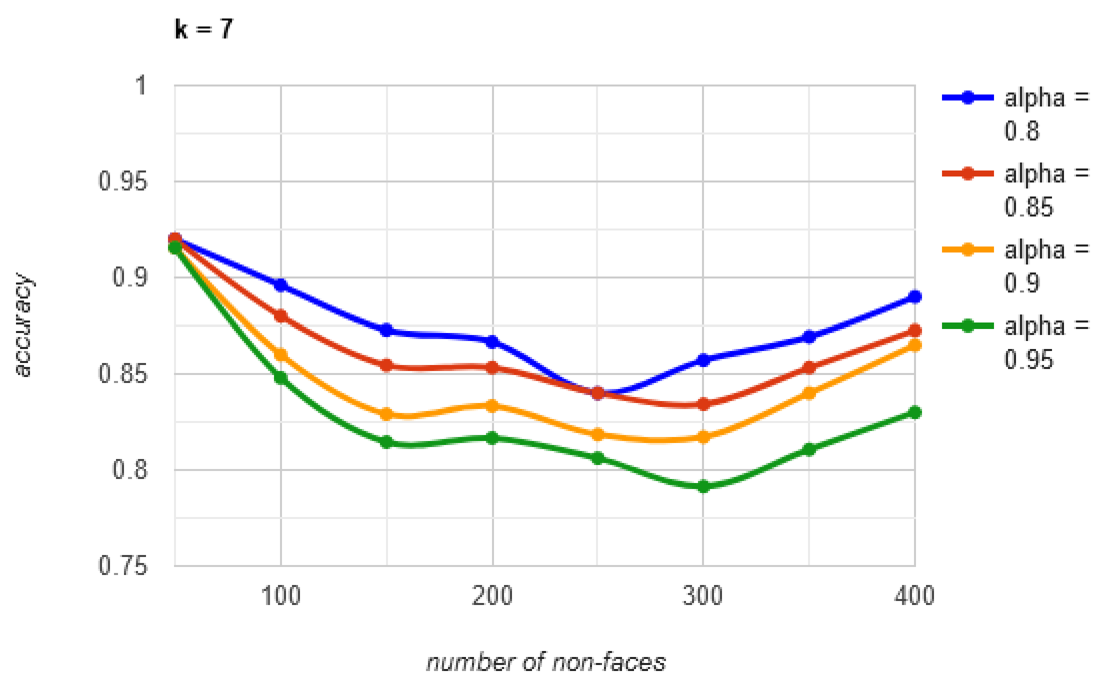
Plot the accuracy vs the number of non-faces images while fixing the number of face images:

- For PCA:






k = 5
1
0.95
0.9
0.85
0.8
accuracy
100
200
300
400
number of non-faces
alpha = 0.8
alpha = 0.85
alpha = 0.9
alpha = 0.95


k = 7
1
0.95
0.9
0.85
0.8
0.75
accuracy
100
200
300
400
number of non-faces
alpha = 0.8
alpha = 0.85
alpha = 0.9
alpha = 0.95

+ Code + Text

```
Accuracies for  50  Non-Faces :
Alpha =  0.8
number of eigen values taken :  33
all accuracies               :  [0.9644444444444444, 0.9244444444444444, 0.9288888888888889, 0.92]
the best accuracy among them :  0.9644444444444444
the best k for the given k's :  1
Best n-neighbours:  1
K-NN Accuracy:  0.9644444444444444

Alpha =  0.85
number of eigen values taken :  47
all accuracies               :  [0.9555555555555556, 0.9244444444444444, 0.9244444444444444, 0.92]
the best accuracy among them :  0.9555555555555556
the best k for the given k's :  1
Best n-neighbours:  1
K-NN Accuracy:  0.9555555555555556

Alpha =  0.9
number of eigen values taken :  70
all accuracies               :  [0.9511111111111111, 0.92, 0.92, 0.9155555555555556]
the best accuracy among them :  0.9511111111111111
the best k for the given k's :  1
Best n-neighbours:  1
K-NN Accuracy:  0.9511111111111111

Alpha =  0.95
number of eigen values taken :  113
all accuracies               :  [0.9466666666666667, 0.92, 0.92, 0.9155555555555556]
the best accuracy among them :  0.9466666666666667
the best k for the given k's :  1
Best n-neighbours:  1
K-NN Accuracy:  0.9466666666666667
```

+ Code + Text

```
Accuracies for  100  Non-Faces :
Alpha =  0.8
number of eigen values taken :  35
all accuracies               :  [0.968, 0.948, 0.92, 0.896]
the best accuracy among them :  0.968
the best k for the given k's :  1
Best n-neighbours:  1
K-NN Accuracy:  0.968

Alpha =  0.85
number of eigen values taken :  50
all accuracies               :  [0.964, 0.928, 0.904, 0.88]
the best accuracy among them :  0.964
the best k for the given k's :  1
Best n-neighbours:  1
K-NN Accuracy:  0.964

Alpha =  0.9
number of eigen values taken :  75
all accuracies               :  [0.948, 0.896, 0.88, 0.86]
the best accuracy among them :  0.948
the best k for the given k's :  1
Best n-neighbours:  1
K-NN Accuracy:  0.948

Alpha =  0.95
number of eigen values taken :  122
all accuracies               :  [0.936, 0.892, 0.868, 0.848]
the best accuracy among them :  0.936
the best k for the given k's :  1
Best n-neighbours:  1
K-NN Accuracy:  0.936
```

Executing (18m 5s) Cell > LDA_Projected()

+ Code + Text

42m

```
Accuracies for  150  Non-Faces :
Alpha =  0.8
number of eigen values taken :  35
all accuracies               :  [0.9272727272727272, 0.9090909090909091, 0.88, 0.8727272727272727]
the best accuracy among them :  0.9272727272727272
the best k for the given k's :  1
Best n-neighbours:  1
K-NN Accuracy:  0.9272727272727272

Alpha =  0.85
number of eigen values taken :  52
all accuracies               :  [0.9272727272727272, 0.8981818181818182, 0.8727272727272727, 0.8545454545454545]
the best accuracy among them :  0.9272727272727272
the best k for the given k's :  1
Best n-neighbours:  1
K-NN Accuracy:  0.9272727272727272

Alpha =  0.9
number of eigen values taken :  78
all accuracies               :  [0.92, 0.8727272727272727, 0.8436363636363636, 0.8290909090909091]
the best accuracy among them :  0.92
the best k for the given k's :  1
Best n-neighbours:  1
K-NN Accuracy:  0.92

Alpha =  0.95
number of eigen values taken :  128
all accuracies               :  [0.8981818181818182, 0.8654545454545455, 0.8254545454545454, 0.8145454545454546]
the best accuracy among them :  0.8981818181818182
the best k for the given k's :  1
Best n-neighbours:  1
K-NN Accuracy:  0.8981818181818182
```

+ Code + Text

2m

```
Accuracies for  200  Non-Faces :
Alpha =  0.8
number of eigen values taken :  35
all accuracies               :  [0.9466666666666667, 0.9066666666666666, 0.88, 0.8666666666666667]
the best accuracy among them :  0.9466666666666667
the best k for the given k's :  1
Best n-neighbours:  1
K-NN Accuracy:  0.9466666666666667

Alpha =  0.85
number of eigen values taken :  52
all accuracies               :  [0.9366666666666666, 0.8933333333333333, 0.8666666666666667, 0.8533333333333334]
the best accuracy among them :  0.9366666666666666
the best k for the given k's :  1
Best n-neighbours:  1
K-NN Accuracy:  0.9366666666666666

Alpha =  0.9
number of eigen values taken :  80
all accuracies               :  [0.9366666666666666, 0.8766666666666667, 0.8533333333333334, 0.8333333333333334]
the best accuracy among them :  0.9366666666666666
the best k for the given k's :  1
Best n-neighbours:  1
K-NN Accuracy:  0.9366666666666666

Alpha =  0.95
number of eigen values taken :  135
all accuracies               :  [0.9266666666666666, 0.8533333333333334, 0.8266666666666667, 0.8166666666666667]
the best accuracy among them :  0.9266666666666666
the best k for the given k's :  1
Best n-neighbours:  1
K-NN Accuracy:  0.9266666666666666
```

Executing (20m 29s) Cell > LDA_Eigen()

+ Code + Text

```
Accuracies for  250  Non-Faces :
Alpha =  0.8
number of eigen values taken :  38
all accuracies               :  [0.9353846153846154, 0.8984615384615384, 0.8676923076923077, 0.84]
the best accuracy among them :  0.9353846153846154
the best k for the given k's :  1
Best n-neighbours:  1
K-NN Accuracy:  0.9353846153846154

Alpha =  0.85
number of eigen values taken :  56
all accuracies               :  [0.9261538461538461, 0.8769230769230769, 0.8553846153846154, 0.84]
the best accuracy among them :  0.9261538461538461
the best k for the given k's :  1
Best n-neighbours:  1
K-NN Accuracy:  0.9261538461538461

Alpha =  0.9
number of eigen values taken :  86
all accuracies               :  [0.916923076923077, 0.8707692307692307, 0.8461538461538461, 0.8184615384615385]
the best accuracy among them :  0.916923076923077
the best k for the given k's :  1
Best n-neighbours:  1
K-NN Accuracy:  0.916923076923077

Alpha =  0.95
number of eigen values taken :  145
all accuracies               :  [0.9076923076923077, 0.8461538461538461, 0.8215384615384616, 0.8061538461538461]
the best accuracy among them :  0.9076923076923077
the best k for the given k's :  1
Best n-neighbours:  1
K-NN Accuracy:  0.9076923076923077
```

Executing (20m 50s) Cell > LDA_Eigen()

+ Code + Text

```
Accuracies for  300  Non-Faces :
Alpha =  0.8
number of eigen values taken :  41
all accuracies               :  [0.9257142857142857, 0.9028571428571428, 0.8657142857142858, 0.8571428571428571]
the best accuracy among them :  0.9257142857142857
the best k for the given k's :  1
Best n-neighbours:  1
K-NN Accuracy:  0.9257142857142857

Alpha =  0.85
number of eigen values taken :  61
all accuracies               :  [0.9114285714285715, 0.8942857142857142, 0.8628571428571429, 0.8342857142857143]
the best accuracy among them :  0.9114285714285715
the best k for the given k's :  1
Best n-neighbours:  1
K-NN Accuracy:  0.9114285714285715

Alpha =  0.9
number of eigen values taken :  94
all accuracies               :  [0.9114285714285715, 0.8828571428571429, 0.8485714285714285, 0.8171428571428572]
the best accuracy among them :  0.9114285714285715
the best k for the given k's :  1
Best n-neighbours:  1
K-NN Accuracy:  0.9114285714285715

Alpha =  0.95
number of eigen values taken :  156
all accuracies               :  [0.9057142857142857, 0.86, 0.82, 0.7914285714285715]
the best accuracy among them :  0.9057142857142857
the best k for the given k's :  1
Best n-neighbours:  1
K-NN Accuracy:  0.9057142857142857
```

+ Code + Text

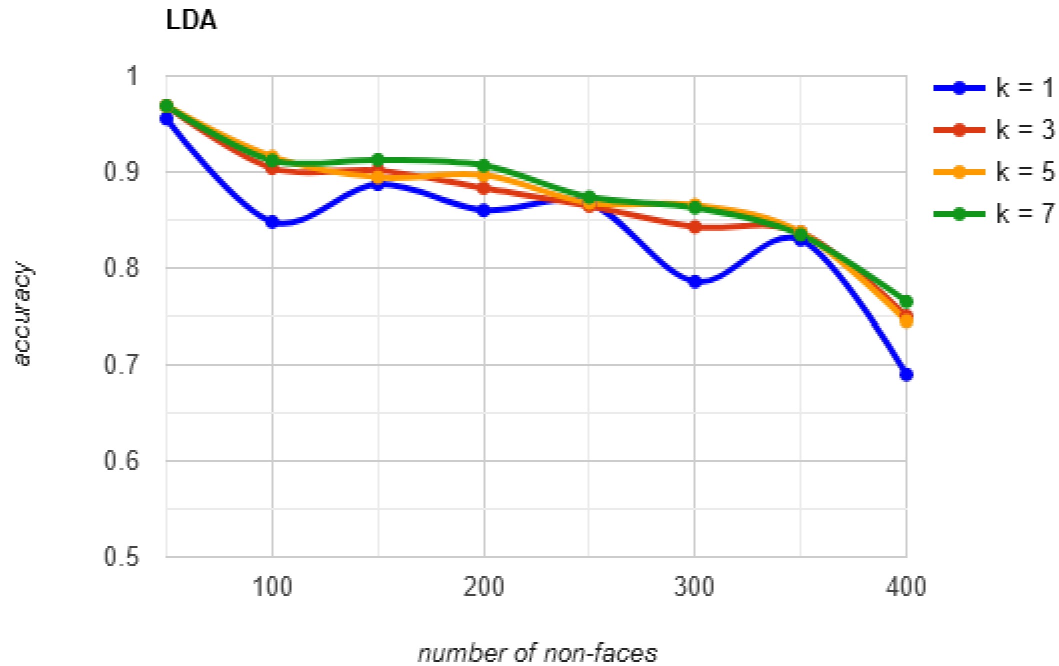
42m

```
Accuracies for  350  Non-Faces :
Alpha =  0.8
number of eigen values taken :  33
all accuracies               :  [0.936, 0.9146666666666666, 0.8826666666666667, 0.8693333333333333]
the best accuracy among them :  0.936
the best k for the given k's :  1
Best n-neighbours:  1
K-NN Accuracy:  0.936

Alpha =  0.85
number of eigen values taken :  51
all accuracies               :  [0.9333333333333333, 0.912, 0.8666666666666667, 0.8533333333333334]
the best accuracy among them :  0.9333333333333333
the best k for the given k's :  1
Best n-neighbours:  1
K-NN Accuracy:  0.9333333333333333

Alpha =  0.9
number of eigen values taken :  83
all accuracies               :  [0.92, 0.9013333333333333, 0.8586666666666667, 0.84]
the best accuracy among them :  0.92
the best k for the given k's :  1
Best n-neighbours:  1
K-NN Accuracy:  0.92

Alpha =  0.95
number of eigen values taken :  148
all accuracies               :  [0.912, 0.8826666666666667, 0.84, 0.8106666666666666]
the best accuracy among them :  0.912
the best k for the given k's :  1
Best n-neighbours:  1
K-NN Accuracy:  0.912
```

+ Code + Text

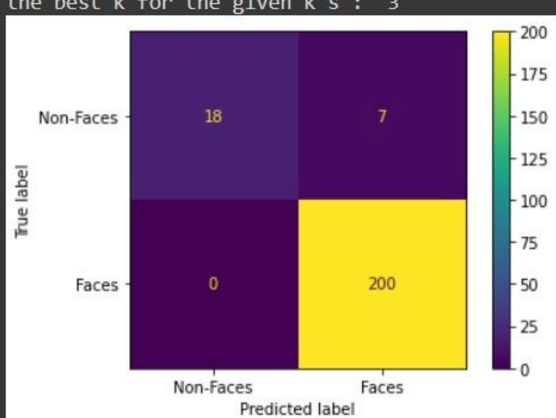
```
Accuracies for  400  Non-Faces :
Alpha =  0.8
number of eigen values taken :  27
all accuracies               :  [0.9425, 0.925, 0.9075, 0.89]
the best accuracy among them :  0.9425
the best k for the given k's :  1
Best n-neighbours:  1
K-NN Accuracy:  0.9425

Alpha =  0.85
number of eigen values taken :  44
all accuracies               :  [0.9375, 0.9225, 0.8875, 0.8725]
the best accuracy among them :  0.9375
the best k for the given k's :  1
Best n-neighbours:  1
K-NN Accuracy:  0.9375

Alpha =  0.9
number of eigen values taken :  75
all accuracies               :  [0.9275, 0.9125, 0.8775, 0.865]
the best accuracy among them :  0.9275
the best k for the given k's :  1
Best n-neighbours:  1
K-NN Accuracy:  0.9275

Alpha =  0.95
number of eigen values taken :  142
all accuracies               :  [0.9175, 0.895, 0.85, 0.83]
the best accuracy among them :  0.9175
the best k for the given k's :  1
Best n-neighbours:  1
K-NN Accuracy:  0.9175
```

- For LDA:



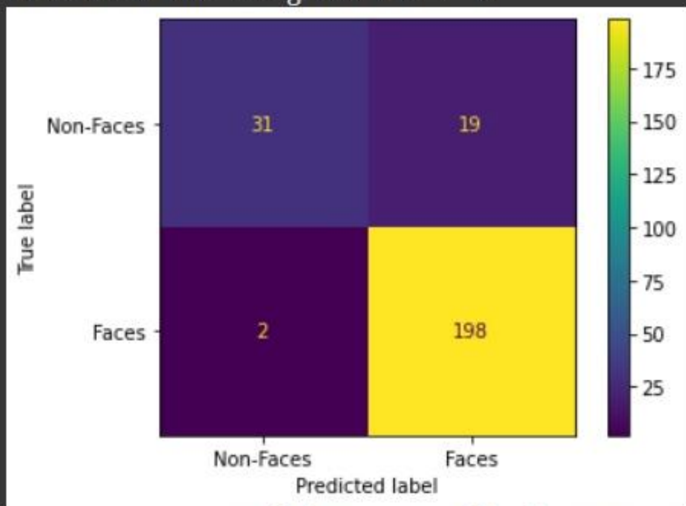
Number of faces vs Accuracy:

```
Non faces :  50
10304
all accuracies             :  [0.9555555555555556, 0.9688888888888889, 0.9688888888888889, 0.9688888888888889]
the best accuracy among them :  0.9688888888888889
the best k for the given k's :  3
```



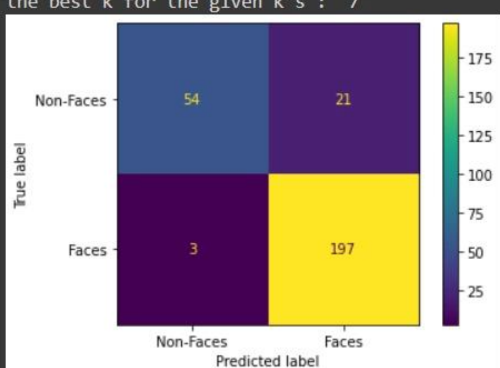

```
              precision    recall  f1-score   support

           0       1.00      0.72      0.84        25
           1       0.97      1.00      0.98       200

    accuracy                           0.97       225
   macro avg       0.98      0.86      0.91       225
weighted avg       0.97      0.97      0.97       225
```

```
Non faces :  100
10304
all accuracies               :  [0.848, 0.904, 0.916, 0.912]
the best accuracy among them :  0.916
the best k for the given k's :  5
```



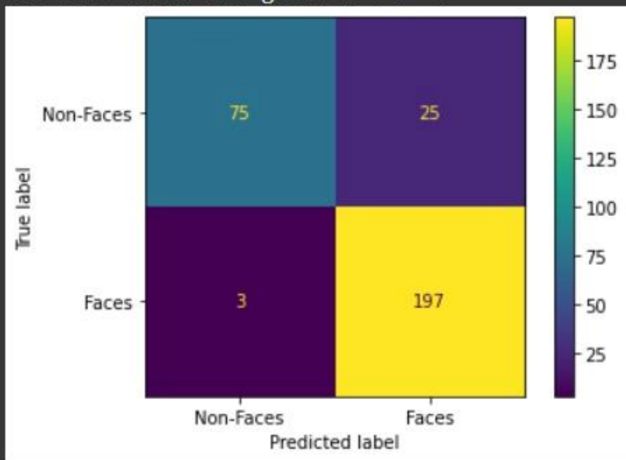

```
              precision    recall  f1-score   support

           0       0.94      0.62      0.75        50
           1       0.91      0.99      0.95       200

    accuracy                           0.92       250
   macro avg       0.93      0.80      0.85       250
weighted avg       0.92      0.92      0.91       250
```

```
Non faces :  150
10304
all accuracies               :  [0.8872727272727273, 0.9018181818181819, 0.8945454545454545, 0.9127272727272727]
the best accuracy among them :  0.9127272727272727
the best k for the given k's :  7
```



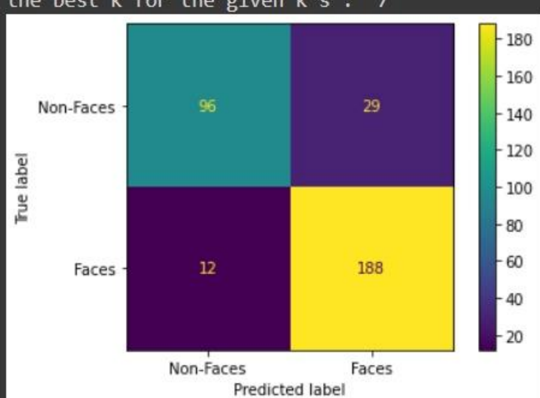

```
              precision    recall  f1-score   support

           0       0.95      0.72      0.82        75
           1       0.90      0.98      0.94       200

    accuracy                           0.91       275
   macro avg       0.93      0.85      0.88       275
weighted avg       0.92      0.91      0.91       275
```

```
Non faces :  200
10304
all accuracies                : [0.86, 0.8833333333333333, 0.8966666666666666, 0.9066666666666666]
the best accuracy among them : 0.9066666666666666
the best k for the given k's :  7
```



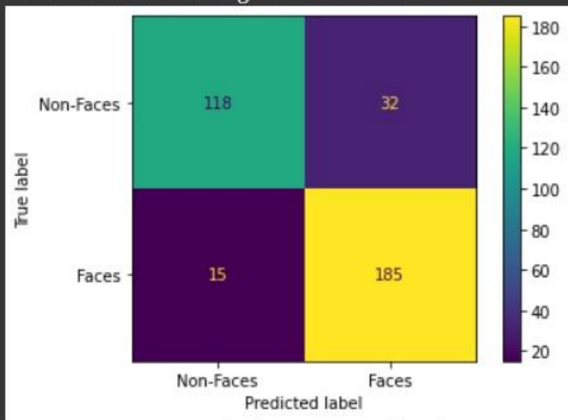

```
              precision    recall  f1-score   support

           0       0.96      0.75      0.84       100
           1       0.89      0.98      0.93       200

    accuracy                           0.91       300
   macro avg       0.92      0.87      0.89       300
weighted avg       0.91      0.91      0.90       300
```

```
Non faces :  250
10304
all accuracies                : [0.8676923076923077, 0.8646153846153846, 0.8676923076923077, 0.8738461538461538]
the best accuracy among them : 0.8738461538461538
the best k for the given k's :  7
```



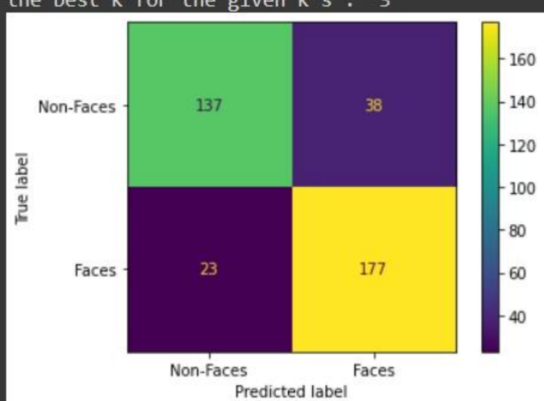

```
              precision    recall  f1-score   support

           0       0.89      0.77      0.82       125
           1       0.87      0.94      0.90       200

    accuracy                           0.87       325
   macro avg       0.88      0.85      0.86       325
weighted avg       0.88      0.87      0.87       325
```

```
Non faces :  300
10304
all accuracies              :  [0.7857142857142857, 0.8428571428571429, 0.8657142857142858, 0.8628571428571429]
the best accuracy among them :  0.8657142857142858
the best k for the given k's :  5
```



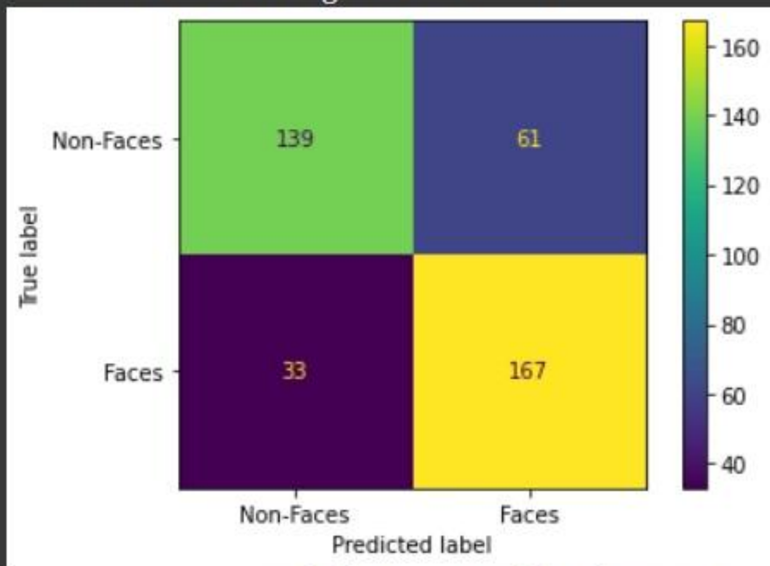

```
              precision    recall  f1-score   support

           0       0.89      0.79      0.83       150
           1       0.85      0.93      0.89       200

    accuracy                           0.87       350
   macro avg       0.87      0.86      0.86       350
weighted avg       0.87      0.87      0.86       350
```

```
Non faces :  350
10304
all accuracies              :  [0.8293333333333334, 0.8373333333333334, 0.8373333333333334, 0.8346666666666667]
the best accuracy among them :  0.8373333333333334
the best k for the given k's :  3
```




```
              precision    recall  f1-score   support

           0       0.86      0.78      0.82       175
           1       0.82      0.89      0.85       200

    accuracy                           0.84       375
   macro avg       0.84      0.83      0.84       375
weighted avg       0.84      0.84      0.84       375
```

```
[35]
Non faces :  400
10304
all accuracies              :  [0.69, 0.75, 0.745, 0.765]
the best accuracy among them :  0.765
the best k for the given k's :  7
```




```
              precision    recall  f1-score   support

           0       0.81      0.69      0.75       200
           1       0.73      0.83      0.78       200

    accuracy                           0.77       400
   macro avg       0.77      0.76      0.76       400
weighted avg       0.77      0.77      0.76       400
```

Criticize the accuracy measure for large numbers of non-faces images in the training data:

- On changing the number of non-face images while keeping rest of the settings constant, it is noticed that
  - **Accuracy is at its highest value** when the number of face images and non face images are **non balanced**.
- Accuracy is **not a good metric** for non-balanced data where it doesn't distinguish between numbers of correctly classified examples of different classes. So it may lead to wrong conclusions. High accuracy may be achieved only by predicting the majority class while most of the minority class may be predicted wrong.

  (Accuracy = correct classifications / number of classifications)

- Too many examples will result in good, but perhaps slightly lower than ideal test accuracy, perhaps because the dataset is over-representative of the problem.

8. Bonus:

Change the number of instances per subject to be 7 and keep 3 instances per subject for testing.

- For the first 7 labels, the code adds the label to the labels_train_set list.
- For the remaining labels (i.e., labels 8-10), the code adds the label to the labels_test_set list.
- This ensures that the training set contains 70% of the labels, and the testing set contains 30% of the labels.
- Similarly, the code loops through each image in the dataset. For the first 7 images, the code adds the image to the faces_train_set numpy array. For the remaining images (i.e., images 8-10), the code adds the image to the faces_test_set numpy array.
- This ensures that the training set contains 70% of the images, and the testing set contains 30% of the images.

9. Link colab:

https://colab.research.google.com/drive/1djfCQ78d18H15EmuIfYKXD713tAcVBpY#scrollTo=58h1qWAOcaBO