Assignment 1

Face Recognition

Team Members

Aya Khames Khairy	18010442
Basel Ayman Mohamed	18010458
Pancee Wahid Mohamed	18010467

Used Libraries

```
[1]  import numpy as np
     import pandas as pd
     from sklearn import neighbors, metrics
     from sklearn.metrics import classification_report
     import matplotlib.pyplot as plt
     import cv2
     import os
     from zipfile import ZipFile
```

Numpy → for computations on arrays
Pandas → for using and computations of datasets
sklearn → for KNN classifier (neighbors)
sklearn.metrics → for calculating accuracy
matplotlib.pyplot → for plotting graphs
cv2 (OpenCV) → for dealing with images (reading as array, resizing, grayscaling)
os → for reading files from folder
zipfile → for extracting files (unzipping zipped files to get the datasets)

Extracting the datasets

```
[2]  paths = ['/content/faces.zip', '/content/non_faces.zip']

     for path in paths:
       with ZipFile(path, 'r') as zip:
         zip.extractall()
```

Using Zipfile library, the zipped files of the images of both datasets, faces and non-faces, are extracted.

Generating the data matrix and label vector

```
[4]  # Generating Data Matrix D
     D = []
     for i in range(1,41):
       for j in range(1,11):
         a = cv2.imread('/content/s'+ str(i) + '/' + str(j) + '.pgm',
                        cv2.IMREAD_GRAYSCALE).flatten()
         D.append(np.asarray(a, dtype=np.uint16))
     dataset = pd.DataFrame(D)
     # display(dataset)

     # Generating label vector y
     y = np.empty((0))
     for i in range(1,41):
       x = np.arange(10)
       z = np.full_like(x, i)
       y = np.concatenate((y, z))
     labels = pd.DataFrame(y, columns=['id'])
     # display(labels)
```

Using OpenCV library, each image is read as a matrix (2d array) of dimensions 112x92. Then using flatten() it's converted to a 1d array of 10304 elements and added to the dataset.
Labels are generated by creating an array of 10 similar elements with the value of the id of each person then concatenating these 40 arrays forming the label vector of 400 elements corresponding to the 400 converted images.

Splitting the dataset into training and testing sets

```
[5]  training_features = dataset.iloc[lambda x: x.index % 2 == 1]
     testing_features = dataset.iloc[lambda x: x.index % 2 == 0]

     training_labels = labels.iloc[lambda x: x.index % 2 == 1]
     testing_labels = labels.iloc[lambda x: x.index % 2 == 0]

     display(training_features)
     display(testing_features)

     display(training_labels)
     display(testing_labels)
```

Splitting the dataset and labels into training and testing sets by appending the row with odd index (not divisible by 2) into training set and the row with even index into testing set.

Classification using PCA

Compute covariance matrix, eigenvalues and eigenvectors

```
[6]  def cov_eig(training):
       covariance = training.cov(ddof=0)

       # compute eigenvalues & compute eigenvectors -> projection matrix U (a)
       eigenvalues, eigenvectors = np.linalg.eigh(covariance)

       return covariance, eigenvalues, eigenvectors
```

Computing the covariance matrix of the training features using cov() from Pandas library.
Computing the eigenvalues and eigenvectors of the covariance matrix using eigh() from np.linalg.

PCA algorithm

```
[6]  def PCA(training, testing, a, eigenvalues, eigenvectors):
       # Fraction of total variance
       sum_of_eigenvalues = np.sum(eigenvalues)
       sum = 0
       k = 0
       for j in range(10303, -1, -1): # for r from 1 to d
         k += 1
         sum += eigenvalues[j]
         frac = sum / sum_of_eigenvalues
         if frac >= a:
           break

       # Get reduced basis
       Ur = eigenvectors[:, 10303] # eigenvector corrosponding to largest eigenvalue
       for x in range(10302, 10303-k, -1): # add column in each iteration
         Ur = np.hstack((Ur.reshape(10304,10303-x),
                         eigenvectors[:, x].reshape(10304,1))) # add column to Ur

       # Projection
       projected_training_data = np.dot(training, Ur)
       projected_testing_data = np.dot(testing, Ur)
       return projected_training_data, projected_testing_data
```

The function receives the value of alpha, the eigenvalues and the eigenvectors of the covariance matrix of the features. Then it applies the PCA algorithm to get the dataset (training & testing) after projection using the projection matrix U_r.

Method used to apply classification by K-nearest neighbors classifier

```
[9]  def classification(training, testing, training_labels, k):
       # determine class labels using KNN
       knn_classifier = neighbors.KNeighborsClassifier(n_neighbors=k)
       knn_classifier = knn_classifier.fit(training,
                                           np.asarray(training_labels).ravel())
       knn_predicted_class = knn_classifier.predict(testing)
       return knn_predicted_class
```

Applying a simple classifier which is k-nearest neighbors with k=1 to classify the samples in the testing set.

Method to calculate the classification accuracy

```
[10] def get_accuracy(actual, predicted):
       report = classification_report(actual , predicted, output_dict=True)
       return report.get('accuracy')
```

Calculate the classification accuracy using classification_report() from sklearn.metrics.

Method for applying PCA algorithm

```
[8]  def apply_PCA(training, testing, alpha, y_training, y_testing):
       covariance, eigenvalues, eigenvectors = cov_eig(training)
       projected_training_data, projected_testing_data = PCA(
           training, testing, alpha, eigenvalues, eigenvectors)
       predicted = classification(projected_training_data,
                                  projected_testing_data, y_training, 1)
       accuracy = get_accuracy(y_testing, predicted)
       return projected_training_data, projected_testing_data, predicted, accuracy
```

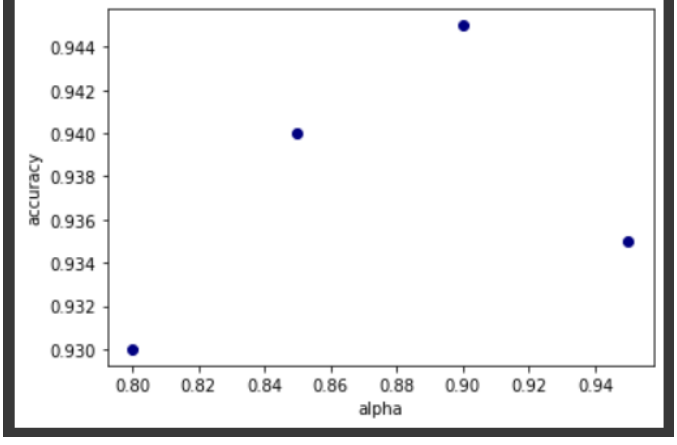
Get accuracy of PCA using the given values of alpha

This method returns the alpha with maximum accuracy.

```
[9]  def reporting_accuracy_PCA(training, testing, y_training, y_testing):
       alpha = [0.8, 0.85, 0.9, 0.95]
       covariance, eigenvalues, eigenvectors = cov_eig(training)
       acc = []
       for i in alpha:
         projected_training_data, projected_testing_data = PCA(
             training, testing, i, eigenvalues, eigenvectors)
         predicted_testing = classification(projected_training_data,
                                            projected_testing_data, y_training, 1)
         accuracy = get_accuracy(y_testing, predicted_testing)
         print('alpha = ', i, '\t accuracy = ', accuracy)
         acc.append(accuracy)
       # plotting
       plt.scatter(np.transpose(alpha), np.transpose(acc), color='navy')
       plt.xlabel('alpha')
       plt.ylabel('accuracy')
       return alpha[np.argmax(acc)], predicted_testing
```

Since alpha value and accuracy are not correlated, we can say that there is no relation between them.

```
alpha =  0.8     accuracy =  0.93
alpha =  0.85    accuracy =  0.94
alpha =  0.9     accuracy =  0.945
alpha =  0.95    accuracy =  0.935
```



Classification using LDA

Methods used in LDA algorithm

```
[14] def calculate_means(training):
       # class means
       mean_vectors = {}
       for i in range(0, 200, 5):
         mean_vectors[(i/5)+1] = (training.iloc[i:i+5 ,:].mean(axis='index')).to_numpy()

       # overall mean
       overall_mean = (training.mean(axis='index')).to_numpy()

       return mean_vectors, overall_mean

     def calculate_Sb(mean_vectors, overall_mean):
       Sb = np.zeros((10304,10304))
       nk = 5
       for i in range(1, 41):
         Muk_Mu = np.subtract(mean_vectors[i], overall_mean)
         x = nk * np.dot(Muk_Mu.reshape((10304,1)), Muk_Mu.reshape((1,10304)))
         Sb = np.add(Sb, x)
       return Sb

     def center_data(training, mean_vectors):
       Z = {}
       for i in range(0, 200, 5):
         Z[(i/5)+1] = training.iloc[i:i+5 ,:].subtract(mean_vectors[(i/5)+1] , axis=1)
       return Z

     def calculate_S(Z):
       S = pd.DataFrame(np.zeros((10304, 10304)))
       for i in range(1, 41):
         S = S.add((Z[i].transpose()).dot(Z[i]))
       return S
```

calculate_means(training) → returns the overall mean vector of training data that contains the mean of each feature values and returns the mean vectors for every class alone

calculate_Sb(mean_vectors, overall_mean) → returns the between-class scatter matrix S_B

center_data(training, mean_vectors) → returns centered data matrix which is the result of subtracting the mean vector of each class from each row of this class in the dataset

calculate_S(Z) → returns the within-class scatter matrix which is the summation of the product of multiplying Z^T by Z for each class

LDA algorithm

```
[19] def LDA(training, testing, i):
       # calculate class means and overall mean
       mean_vectors, overall_mean = calculate_means(training)

       # between-class scatter matrix
       Sb = calculate_Sb(mean_vectors, overall_mean)

       # center class matrices
       Z = center_data(training, mean_vectors)

       # within_class scatter matrix
       S = calculate_S(Z)

       # compute dominant eigenvector
       eigenvalues, eigenvectors = np.linalg.eigh(np.dot(
           np.linalg.inv(S.to_numpy()), Sb))

       # get projection matrix
       # i = 10265, 10303
       U = np.flip(eigenvectors[:, i:10304], 1)

       # project training and testing
       projected_training_data = np.dot(training, U)
       projected_testing_data = np.dot(testing, U)

       return projected_training_data, projected_testing_data
```

Method for applying LDA algorithm

This method returns the dataset (training & testing) after projection and the accuracy gained by applying LDA on the data which is then used for classification using KNN with k = 1

```
[20] def apply_LDA(training, testing, y_training, y_testing, i):
       projected_training_data, projected_testing_data = LDA(training, testing, i)
       predicted = classification(projected_training_data,
                                  projected_testing_data, y_training, 1)
       accuracy = get_accuracy(y_testing, predicted)
       print(accuracy)
       return projected_training_data, projected_testing_data, predicted, accuracy
```

Running LDA algorithm

```
proj_train_LDA, proj_test_LDA, acc_LDA = apply_LDA(
    training_features, testing_features)
```

```
0.945
```

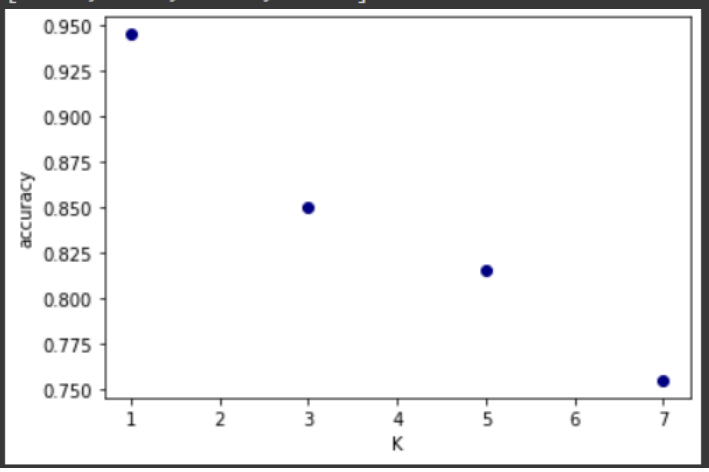
Classifier Tuning

```
[16] def tuning(projected_training_data, projected_testing_data, y_training, y_testing):
       K = [1, 3, 5, 7]
       a = []
       for i in K:
         predicted = classification(projected_training_data,
                                    projected_testing_data, y_training, i)
         accuracy = get_accuracy(y_testing, predicted)
         a.append(accuracy)

       print(a)
       # plotting
       plt.scatter(K, a, color='navy')
       plt.xlabel('K')
       plt.ylabel('accuracy')
```

This method applies classification with KNN using given values of k which are 1, 3, 5 & 7 and plot a graph for the values of K versus the accuracy.

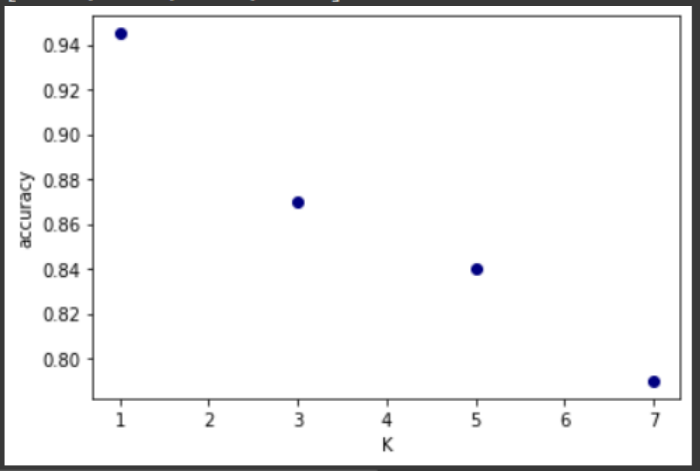
```
print('PCA')
tuning(proj_train_PCA, proj_test_PCA, training_labels, testing_labels)
```

```
PCA
[0.945, 0.85, 0.815, 0.755]
```




```
print('LDA')
tuning(proj_train_LDA, proj_test_LDA, training_labels, testing_labels)
```

```
LDA
[0.945, 0.87, 0.84, 0.79]
```



Compare faces vs non-faces images

Read non-faces images

```
[45] # Change the directory
     path = '/content/non_faces'
     os.chdir(path)

     # Iterate over all the files in the directory
     non_faces = []
     for file in os.listdir():
       file_path =f"{path}/{file}"
       # read image
       img = cv2.imread(file_path, cv2.IMREAD_UNCHANGED)
       # resize image
       resized = cv2.resize(img, (112, 92), interpolation = cv2.INTER_AREA)
       # grayscale image
       grayscaled = cv2.cvtColor(resized, cv2.COLOR_BGR2GRAY).flatten()
       non_faces.append(np.asarray(grayscaled, dtype=np.uint16))

     non_faces_ds = pd.DataFrame(non_faces)
     display(non_faces_ds)

     # Generating label vector y_faces
     y_faces = np.ones((400,1))
     y_faces_ds = pd.DataFrame(y_faces, columns=['id'])

     y_faces_training = y_faces_ds.sample(frac = 0.5)
     y_faces_testing =  y_faces_ds.drop(y_faces_training.index)
```

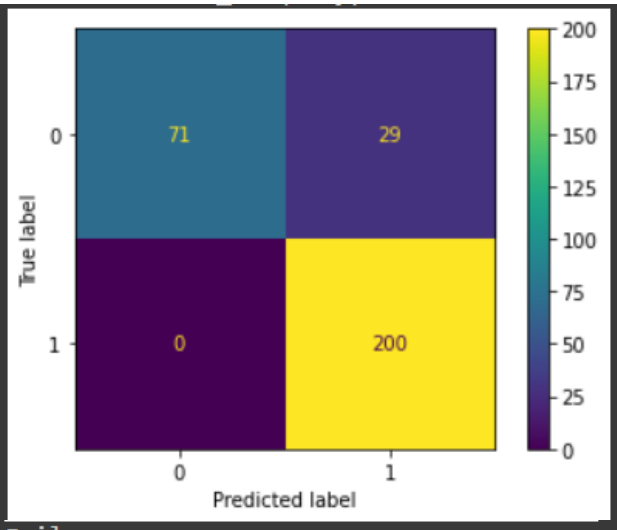
```
[48] def faces_non_faces(n, train_percentage):
       # Generating label vector y_non_faces
       y_non_faces = np.zeros((n,1))
       y_non_faces_ds = pd.DataFrame(y_non_faces, columns=['id'])

       non_faces = non_faces_ds.head(n)
       y_non_faces = y_non_faces_ds.head(n)

       non_faces_training = non_faces.sample(frac = train_percentage)
       non_faces_testing = non_faces.drop(non_faces_training.index)

       y_non_faces_training = y_non_faces_ds.sample(frac = train_percentage)
       y_non_faces_testing =  y_non_faces_ds.drop(y_non_faces_training.index)

       training_mixed = pd.concat([training_features, non_faces_training],
                                  axis=0, ignore_index=True)
       y_training_mixed = pd.concat([y_faces_training, y_non_faces_training],
                                    axis=0, ignore_index=True)

       testing_mixed = pd.concat([testing_features, non_faces_testing],
                                 axis=0, ignore_index=True)
       y_testing_mixed = pd.concat([y_faces_testing, y_non_faces_testing],
                                   axis=0, ignore_index=True)

       return training_mixed, testing_mixed, y_training_mixed, y_testing_mixed
```

```
[21] def faces_vs_non_faces_PCA(training, testing, y_training, y_testing, alpha):
       proj_training, proj_testing, predicted, accuracy = apply_PCA(
           training, testing, alpha, y_training, y_testing)
       cm = metrics.confusion_matrix(y_testing, predicted)
       disp = metrics.ConfusionMatrixDisplay(confusion_matrix=cm,
                                             display_labels=['0','1'])
       disp.plot()
       plt.show()
       testing_array = np.asarray(y_testing)
       print('Failure')
       for i in range(0, 300):
         if predicted[i] != testing_array[i]:
           img = np.reshape(np.asarray(testing.iloc[i]), (112, 92))
           plt.imshow(img, cmap='gray')
           plt.show()
           break
       print('Success')
       for i in range(0, 300):
         if predicted[i] == testing_array[i]:
           img = np.reshape(np.asarray(testing.iloc[i]), (112, 92))
           plt.imshow(img, cmap='gray')
           plt.show()
           break
       return accuracy
```

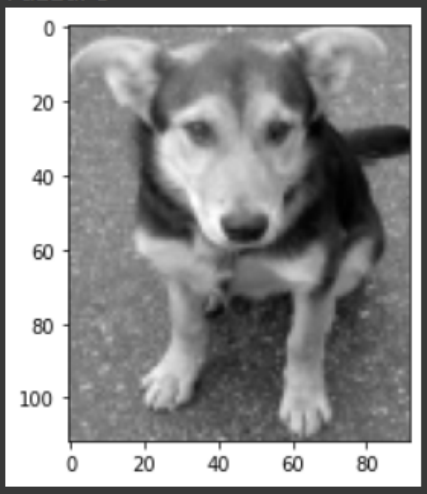
```
[23] def faces_vs_non_faces_LDA(training, testing, y_training, y_testing):
       proj_training, proj_testing, predicted, accuracy = apply_LDA(
           training, testing, y_training, y_testing, 10303)
       cm = metrics.confusion_matrix(y_testing, predicted)
       disp = metrics.ConfusionMatrixDisplay(confusion_matrix=cm,
                                             display_labels=['0','1'])
       disp.plot()
       plt.show()
       testing_array = np.asarray(y_testing)
       print('Failure')
       for i in range(0, 300):
         if predicted[i] != testing_array[i]:
           img = np.reshape(np.asarray(testing.iloc[i]), (112, 92))
           plt.imshow(img, cmap='gray')
           plt.show()
           break
       print('Success')
       for i in range(0, 300):
         if predicted[i] == testing_array[i]:
           img = np.reshape(np.asarray(testing.iloc[i]), (112, 92))
           plt.imshow(img, cmap='gray')
           plt.show()
           break
       return accuracy
```

Trying PCA with 200 non-faces images and 400 faces images.

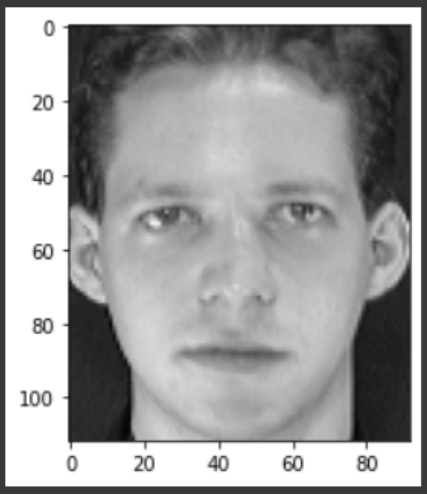
```
[19] acc_200 = faces_vs_non_faces_PCA(faces_non_faces_200_training_features,
                                      faces_non_faces_200_testing_features,
                                      faces_non_faces_200_training_labels,
                                      faces_non_faces_200_testing_labels,
                                      0.9)
```



Failure



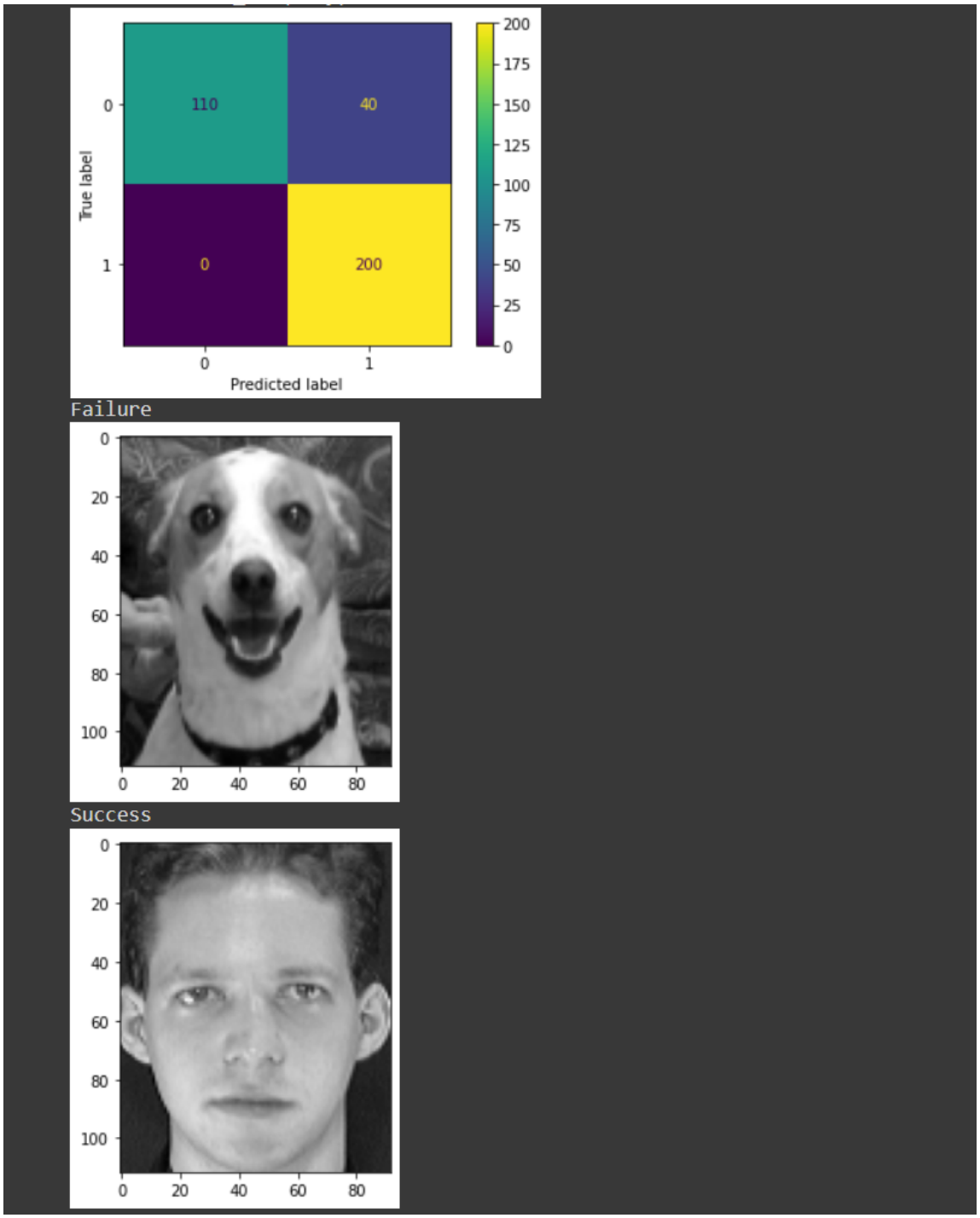
Success



acc_200 = 90.33%

Trying PCA with 300 non-faces images and 400 faces images.

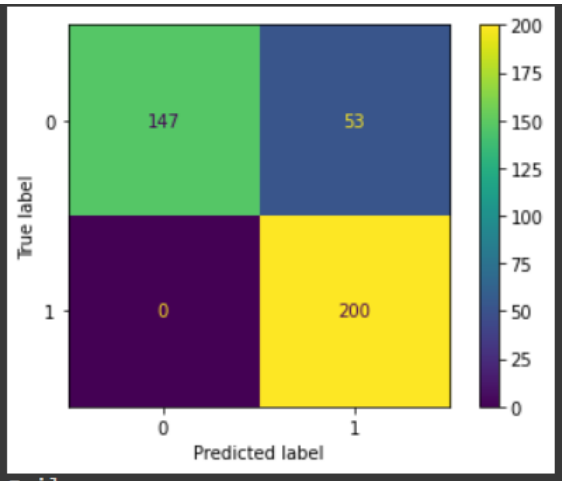
```
[21] acc_300 = faces_vs_non_faces_PCA(faces_non_faces_300_training_features,
                                      faces_non_faces_300_testing_features,
                                      faces_non_faces_300_training_labels,
                                      faces_non_faces_300_testing_labels,
                                      0.9)
```



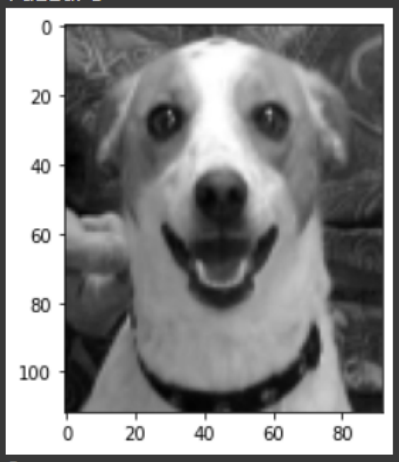
acc_300 = 88.57%

Trying PCA with 400 non-faces images and 400 faces images.

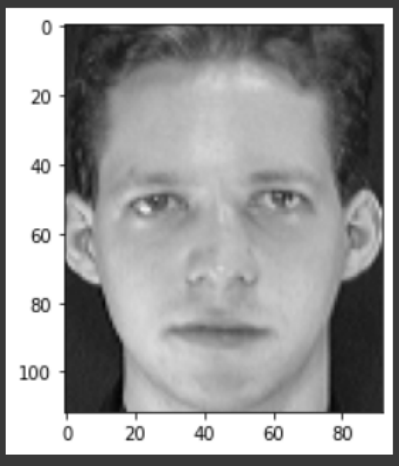
```
[23] acc_400 = faces_vs_non_faces_PCA(faces_non_faces_400_training_features,
                                      faces_non_faces_400_testing_features,
                                      faces_non_faces_400_training_labels,
                                      faces_non_faces_400_testing_labels,
                                      0.9)
```



Failure



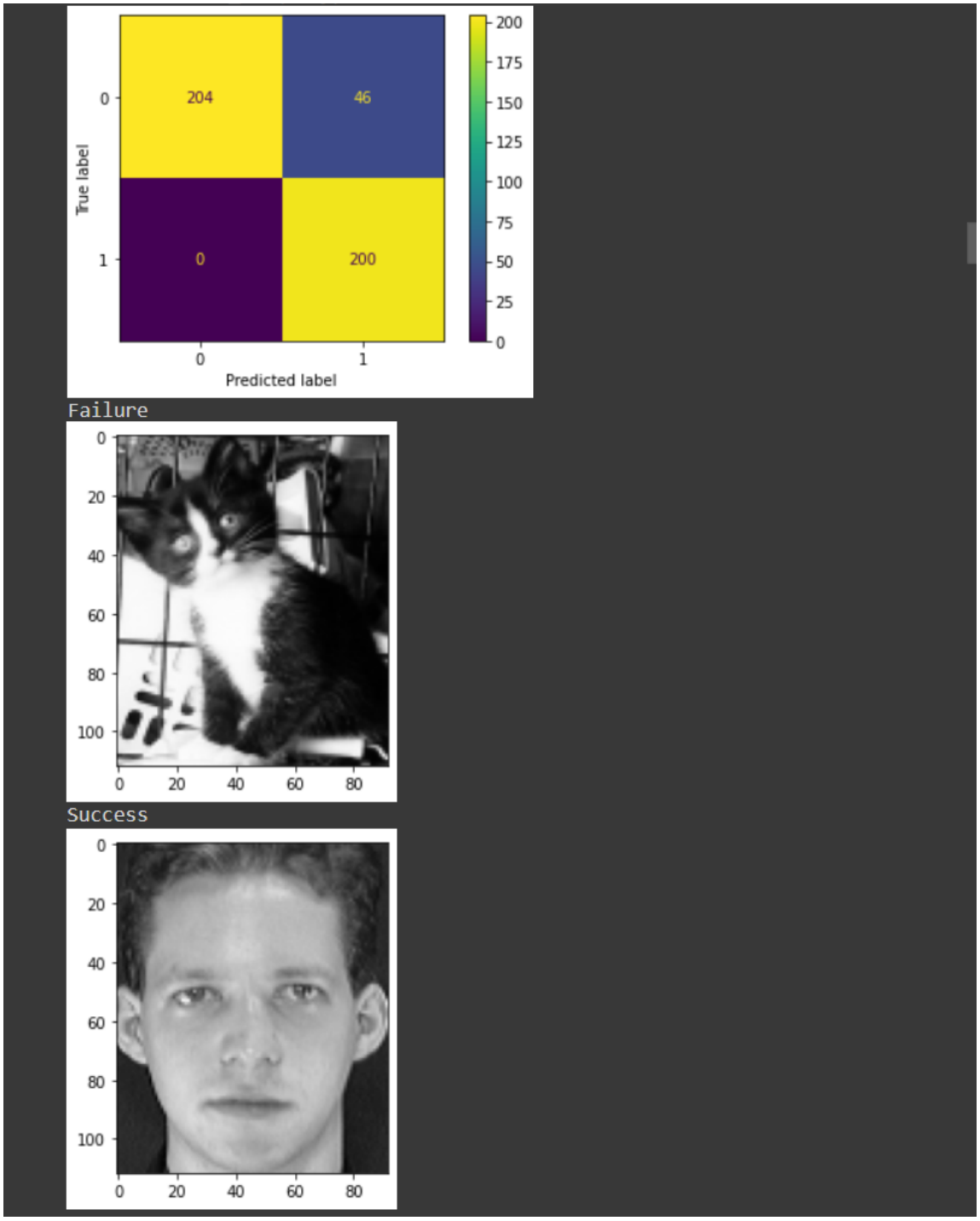
Success



acc_400 = 86.75%

Trying PCA with 500 non-faces images and 400 faces images.

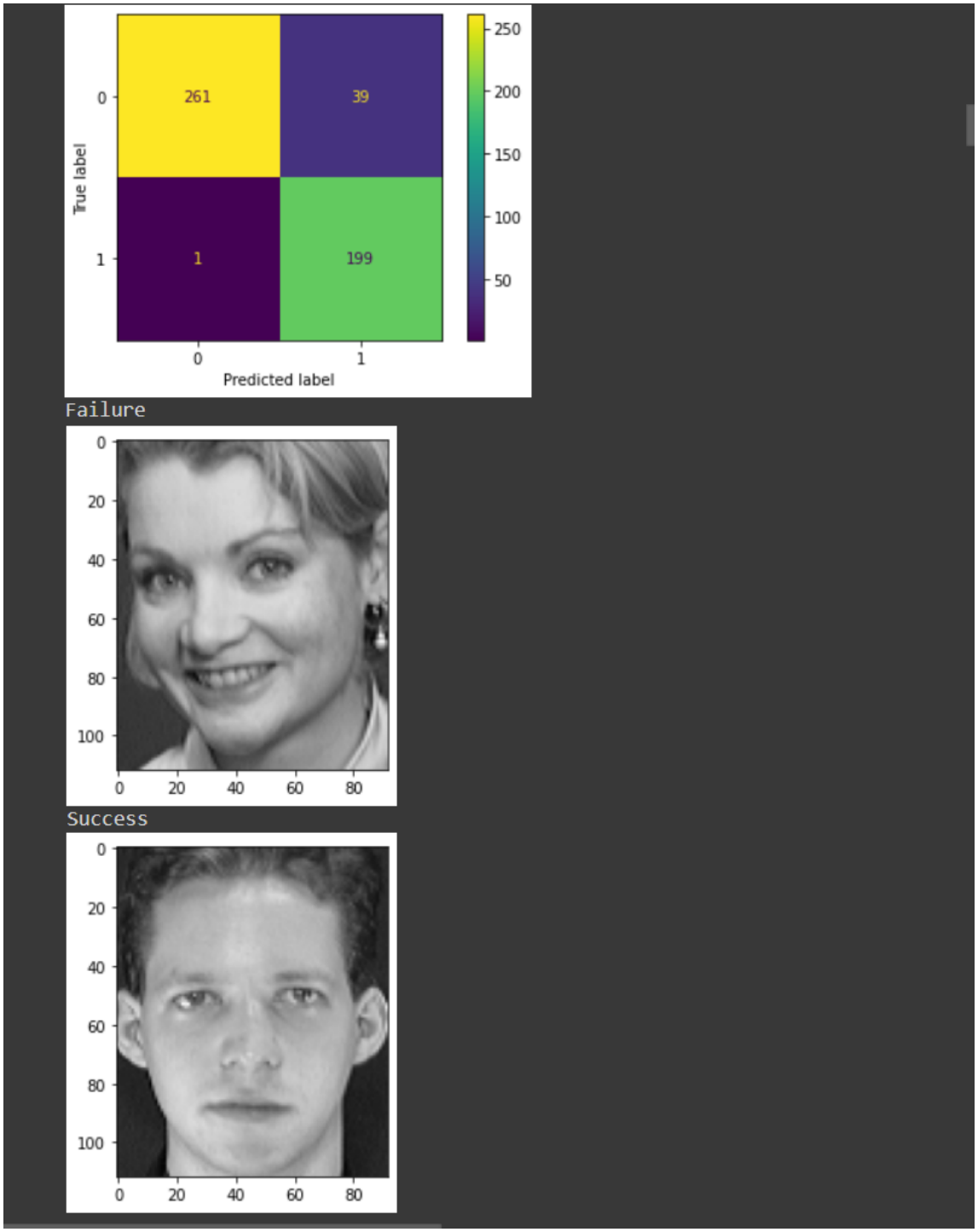
```
[24] acc_500 = faces_vs_non_faces_PCA(faces_non_faces_500_training_features,
                                      faces_non_faces_500_testing_features,
                                      faces_non_faces_500_training_labels,
                                      faces_non_faces_500_testing_labels,
                                      0.9)
```



acc_500 = 89.78%

Trying PCA with 600 non-faces images and 400 faces images.

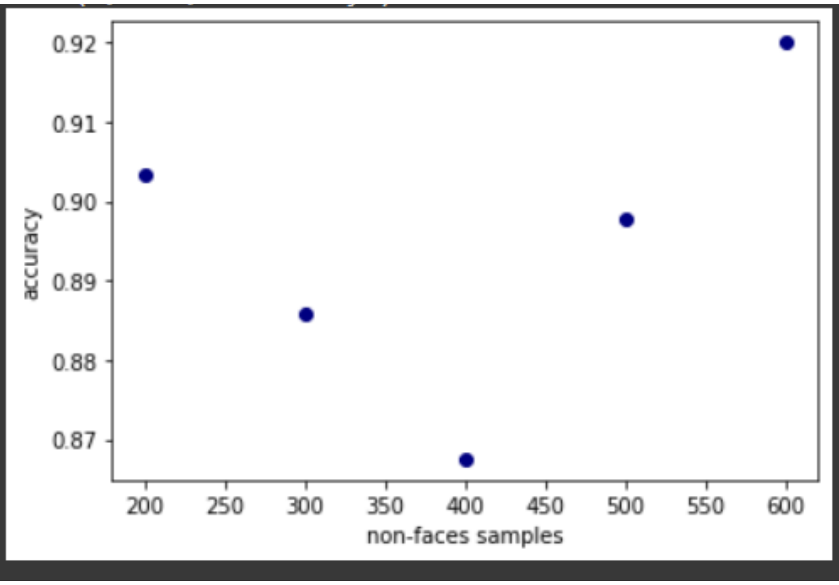
```
[25] acc_600 = faces_vs_non_faces_PCA(faces_non_faces_600_training_features,
                                      faces_non_faces_600_testing_features,
                                      faces_non_faces_600_training_labels,
                                      faces_non_faces_600_testing_labels,
                                      0.9)
```



acc_600 = 92%

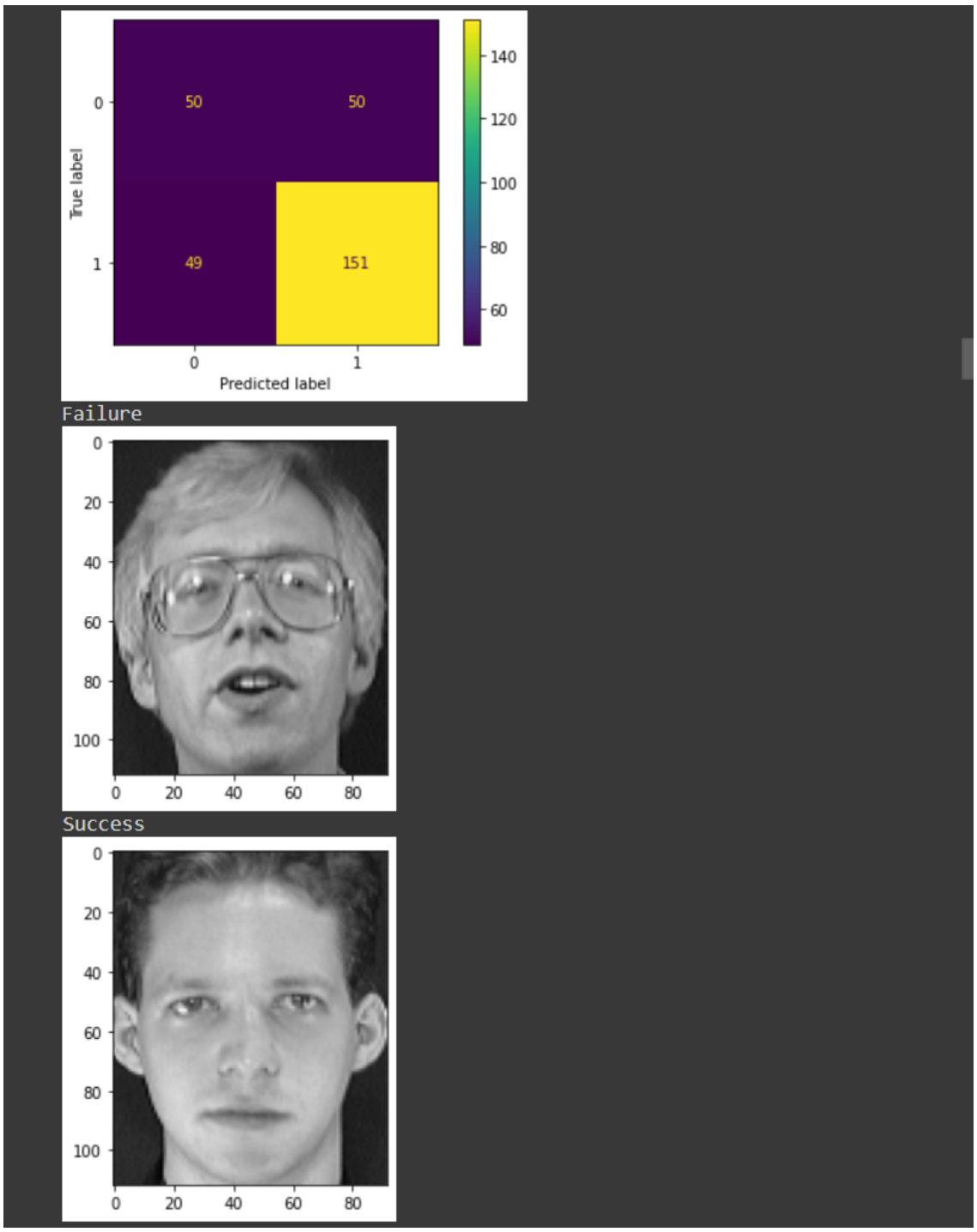
Plotting of accuracy vs number of non-faces samples

```
[32] acc_PCA = [acc_200, acc_300, acc_400, acc_500, acc_600]
     samples = [200, 300, 400, 500, 600]
     # plotting
     plt.scatter(samples, acc_PCA, color='navy')
     plt.xlabel('non-faces samples')
     plt.ylabel('accuracy')
```



Trying LDA with 200 non-faces images and 400 faces images.

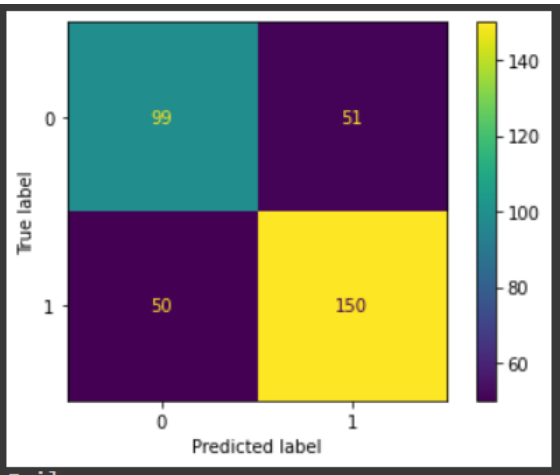
```
[ ]  acc_LDA_200 = faces_vs_non_faces_LDA(faces_non_faces_200_training_features,
                                          faces_non_faces_200_testing_features,
                                          faces_non_faces_200_training_labels,
                                          faces_non_faces_200_testing_labels)
```



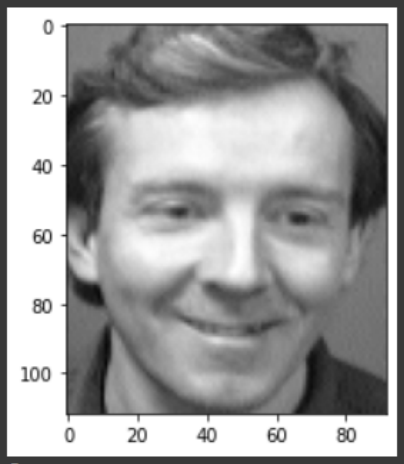
acc_LDA_200 = 67%

Trying LDA with 300 non-faces images and 400 faces images.

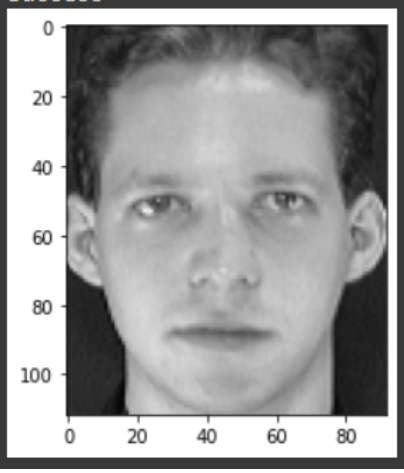
```
[ ]  acc_LDA_300 = faces_vs_non_faces_LDA(faces_non_faces_300_training_features,
                                          faces_non_faces_300_testing_features,
                                          faces_non_faces_300_training_labels,
                                          faces_non_faces_300_testing_labels)
```



Failure



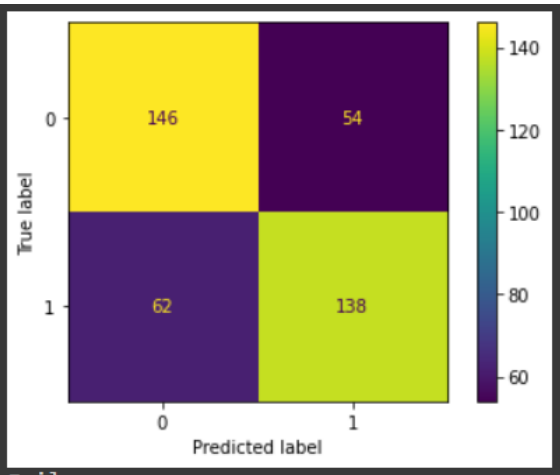
Success



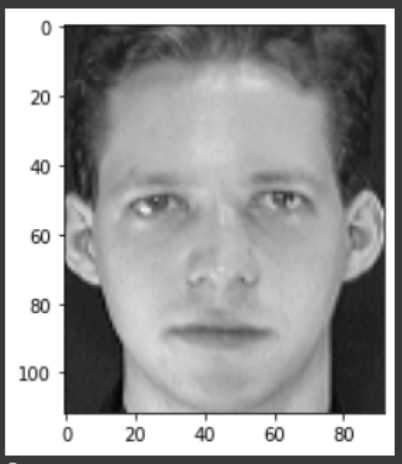
acc_LDA_300 = 71.14%

Trying LDA with 400 non-faces images and 400 faces images.

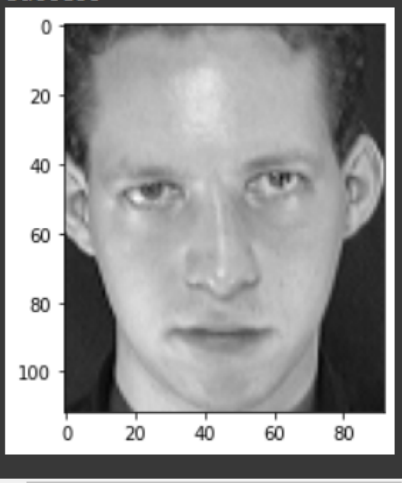
```
[ ]  acc_LDA_400 = faces_vs_non_faces_LDA(faces_non_faces_400_training_features,
                                          faces_non_faces_400_testing_features,
                                          faces_non_faces_400_training_labels,
                                          faces_non_faces_400_testing_labels)
```



Failure



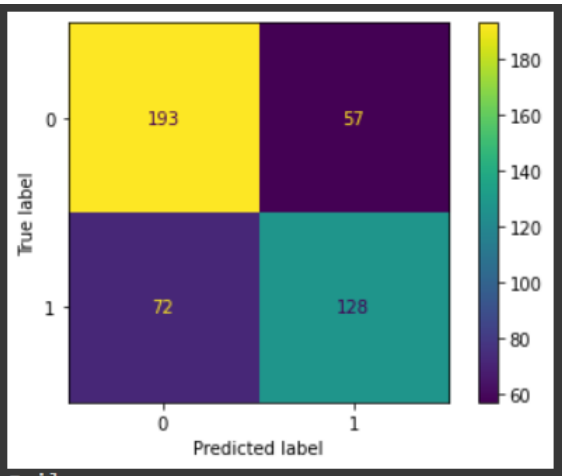
Success



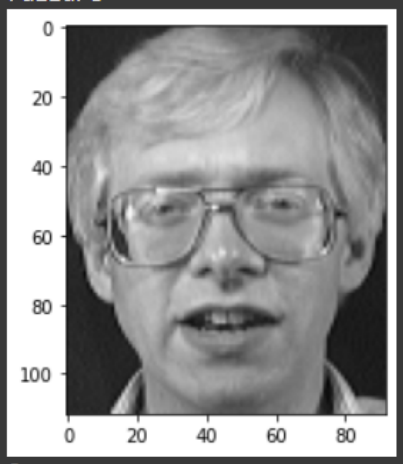
acc_LDA_400 = 71%

Trying LDA with 500 non-faces images and 400 faces images.

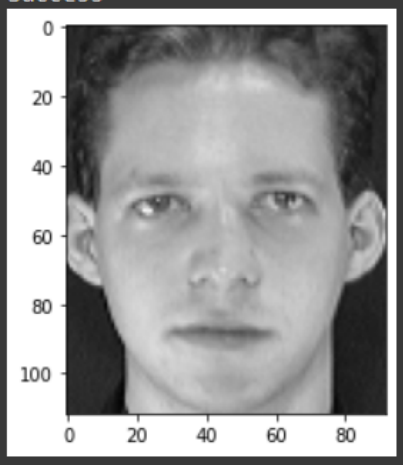
```
[ ]  acc_LDA_500 = faces_vs_non_faces_LDA(faces_non_faces_500_training_features,
                                          faces_non_faces_500_testing_features,
                                          faces_non_faces_500_training_labels,
                                          faces_non_faces_500_testing_labels)
```



Failure



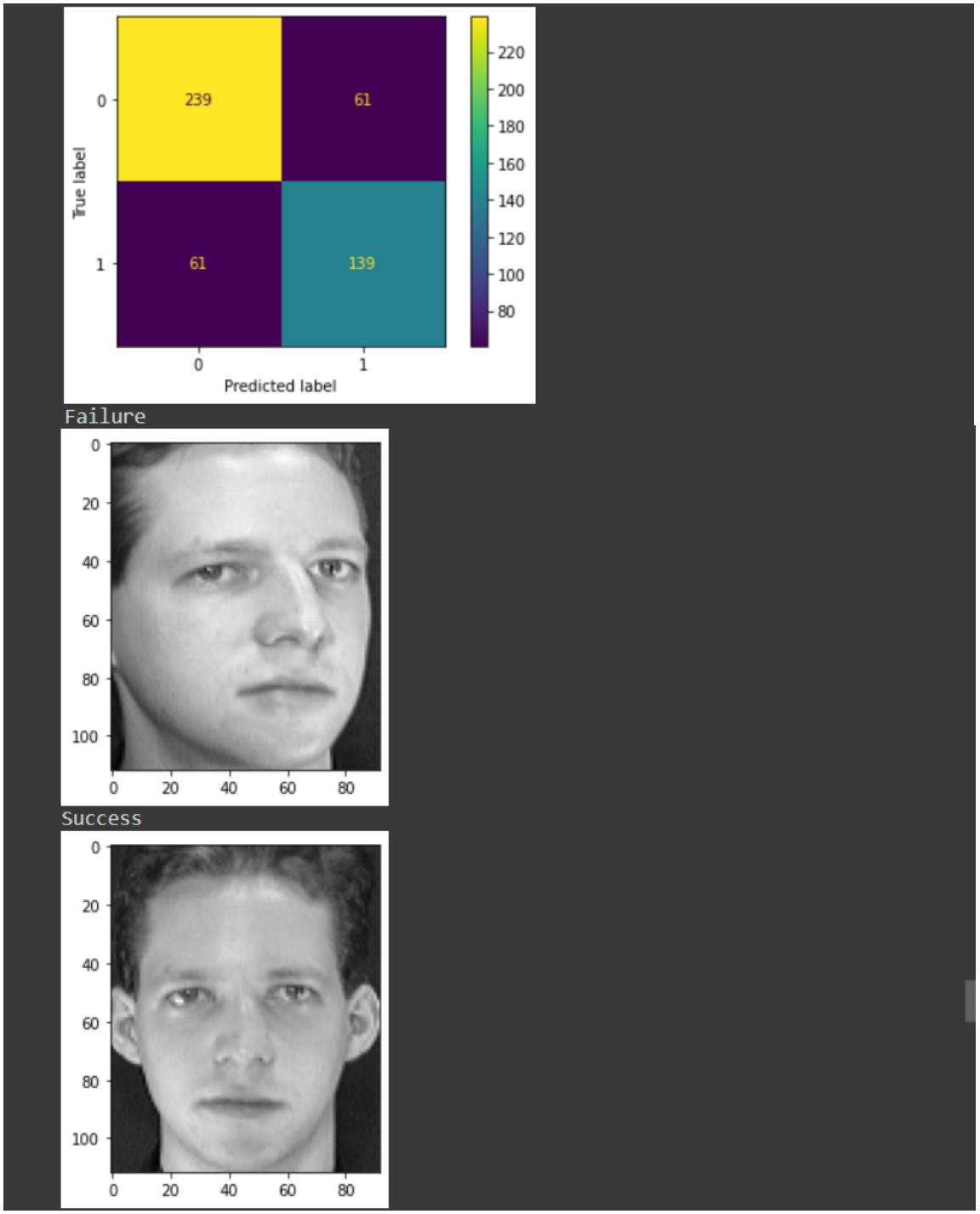
Success



acc_LDA_500 = 71.33%

Trying LDA with 600 non-faces images and 400 faces images.

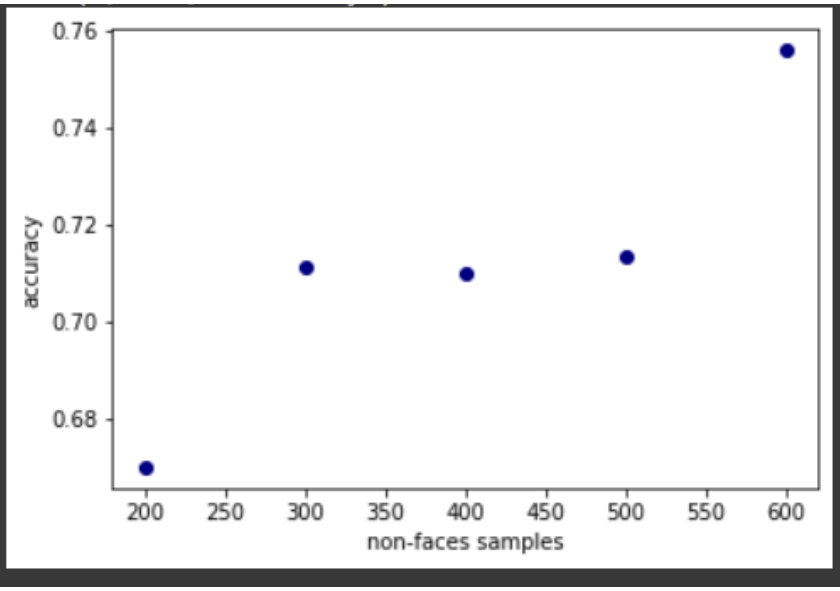
```
[ ]  acc_LDA_600 = faces_vs_non_faces_LDA(faces_non_faces_600_training_features,
                                          faces_non_faces_600_testing_features,
                                          faces_non_faces_600_training_labels,
                                          faces_non_faces_600_testing_labels)
```



acc_LDA_600 = 75.6%

Plotting of accuracy vs number of non-faces samples

```
[35] acc_LDA = [0.67, 0.7114, 0.71, 0.7133, 0.756]
     samples = [200, 300, 400, 500, 600]
     # plotting
     plt.scatter(samples, acc_LDA, color='navy')
     plt.xlabel('non-faces samples')
     plt.ylabel('accuracy')
```



Bonus

Generating the data matrix and label vector & Splitting the dataset into training and testing sets

```
[20] # Training
     # Generating Data Matrix D
     D_training = []
     for i in range(1,41):
       for j in range(1,8):
         a_training = cv2.imread('/content/s'+ str(i) + '/' + str(j) + '.pgm', cv2.IMREAD_GRAYSCALE).flatten()
         D_training.append(np.asarray(a_training, dtype=np.uint16))
     dataset_training = pd.DataFrame(D_training)
     display(dataset_training)

     # Generating label vector y
     y_training = np.empty((0))
     for i in range(1,41):
       x_training = np.arange(7)
       z_training = np.full_like(x_training, i)
       y_training = np.concatenate((y_training, z_training))
     labels_training = pd.DataFrame(y_training, columns=['id'])
     display(labels_training)


     # Testing
     # Generating Data Matrix D
     D_testing = []
     for i in range(1,41):
       for j in range(8,11):
         a_testing = cv2.imread('/content/s'+ str(i) + '/' + str(j) + '.pgm', cv2.IMREAD_GRAYSCALE).flatten()
         D_testing.append(np.asarray(a_testing, dtype=np.uint16))
     dataset_testing = pd.DataFrame(D_testing)
     display(dataset_testing)
```

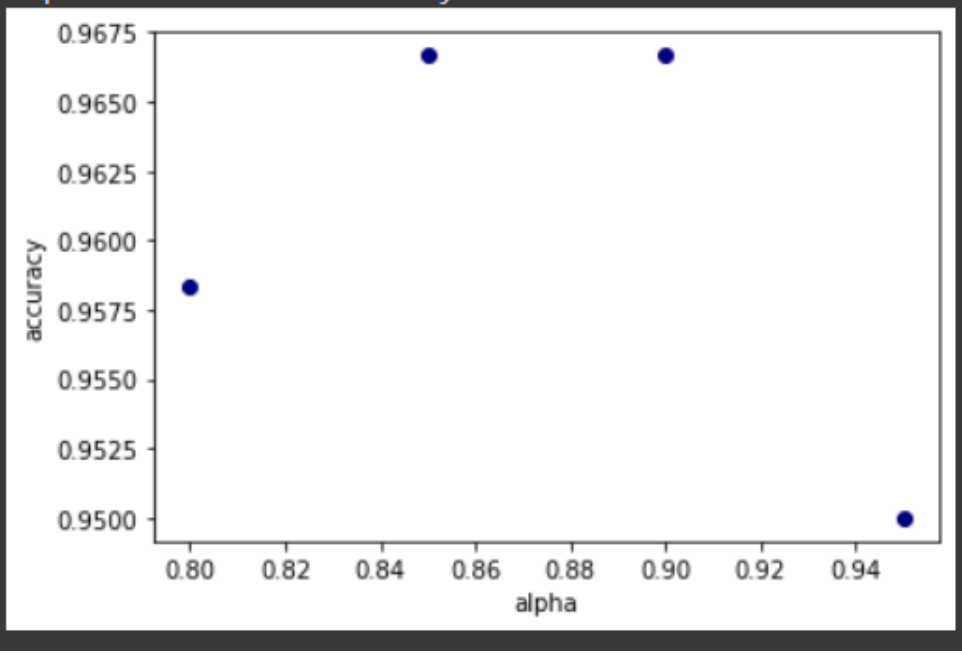
```
     # Generating label vector y
     y_testing = np.empty((0))
     for i in range(1,41):
       x_testing = np.arange(3)
       z_testing = np.full_like(x_testing, i)
       y_testing = np.concatenate((y_testing, z_testing))
     labels_testing = pd.DataFrame(y_testing, columns=['id'])
     display(labels_testing)
```

Using OpenCV library, each image is read as a matrix (2d array) of dimensions 112x92. Then using flatten() it's converted to a 1d array of 10304 elements and added to the dataset.

Splitting the dataset and labels into training and testing sets by changing the number of instances per subject to be 7 in the training set and keeping 3 instances per subject for testing.

Running PCA Algorithm with Different Alpha

```
alpha =  0.8     accuracy =  0.9583333333333334
alpha =  0.85    accuracy =  0.9666666666666667
alpha =  0.9     accuracy =  0.9666666666666667
alpha =  0.95    accuracy =  0.95
```



Since alpha value and accuracy are not correlated, we can say that there is no relation between them.

Running LDA algorithm

```
[ ] proj_train_LDA_bonus, proj_test_LDA_bonus, predicted_LDA_bonus, acc_LDA_bonus = apply_LDA_bonus(dataset_training, dataset_testing)

    0.9416666666666667
```

Classifier Tuning

```
print('PCA')
tuning(proj_train_PCA_bonus, proj_test_PCA_bonus, labels_training, labels_testing)
```

```
PCA
[0.9583333333333334, 0.95, 0.9, 0.8583333333333333]
```