Clab – 2 Report

Engn 4528

Australian National University

Prateek Arora

Student Id – u6742441

15th May, 2020

1. Harris Corner Detection

1) **Read and understand the above corner detection code [2]**
   Code given above explains how derivates is used to calculate the intensities and f special function is used for the calculation of Ixx and Iyy which we help in making the intensity matrix.

2) **Complete the missing parts, rewrite them to 'harris.py' as a python script, and design appropriate function signature [2].**

```
##########################################################################################################
    # Task: Compute Harris Corners
    height, width = bw.shape  ##Gets the shape of image
    result_image = np.zeros((height, width)) ## Assigning the zeros to an array of same size as image
    R = np.zeros((height, width)) ## Assigning the zeros to an array of same size as image
    max_value = 0


    ## Looping over the dimensions of image
    for i in range(height):
        for j in range(width):
            determinant = Ix2[i][j]*Iy2[i][j] - (Ixy[i][j]**2) ## Find determinant of using the intensity
            trace_matrix = Ix2[i][j] + Iy2[i][j] ## Finding the trace
            R[i][j] = determinant - 0.01 * (trace_matrix**2) ##Getting rhe corners for the matrix
            if R[i][j] > max_value:
                max_value = R[i][j] ## Saving the maximum value got

##########################################################################################################
```

```
#####################################################################
    # Task: Perform non-maximum suppression and
    #       thresholding, return the N corner points
    #       as an Nx2 matrix of x and y coordinates

    # Looping over the dimensions of the image
    for k in range(height - 1):
        for l in range(width - 1):
            if (R[k][l] > 0.01 * max_value and R[k][l] > R[k-1][l-1] and R[k][l] > R[k-1][l+1] and R[k][l] > R[k+1][l-1]
                and R[k][l] > R[k+1][l+1]):
                result_image[k][l] = 1 ##Assigning the values 1 if the above condition is satisfied
    return result_image
#####################################################################
```

[1]

In the above code, we have been given the intensities as well as the derivative respectively. Using that, we have to form an intensity matrix and by setting the threshold to 0.01, we can calculate the R matrix. The R matrix is calculated by subtracting the trace of the intensities from the by the determinant of the intensities. The R matrix will help using non suppression by selecting the corners appropriately. In non-maximum suppression, we will find the corners by comparing the values that we got from R and will compare the value from the surroundings and if the value is still the greatest, this means intensity change is highest in that point which means that we have found the corner. After finding the corner, we can set the value of corner equal to 1 and then we can find the x and y coordinates where value is 1. This will help us to detect the corners in the image.

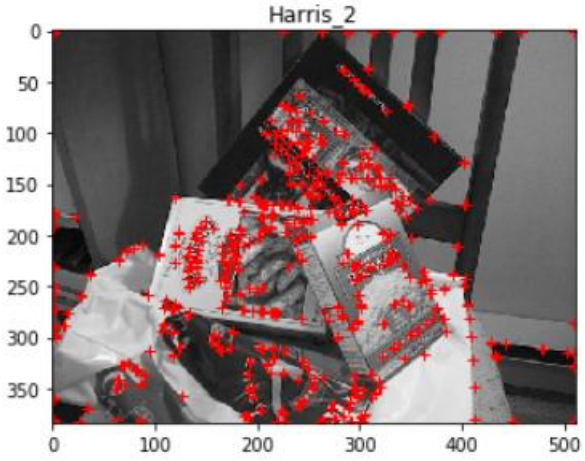
3) **Comment on block #5 (corresponding to line #13 in "harris.py") and every line of your solution in block #7[2]**

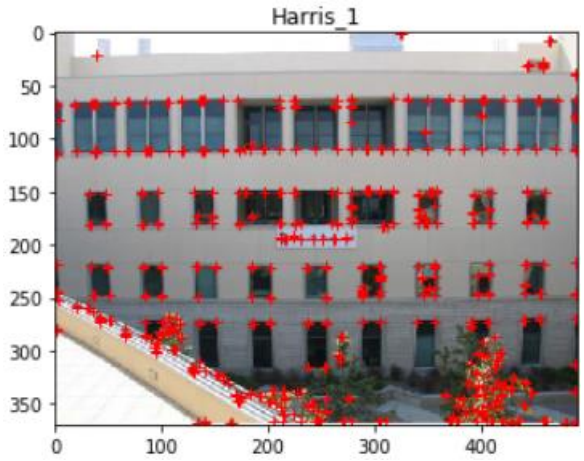
Comments have been added in every line of the code so as to make a readable code. Other comments pictures are given below.

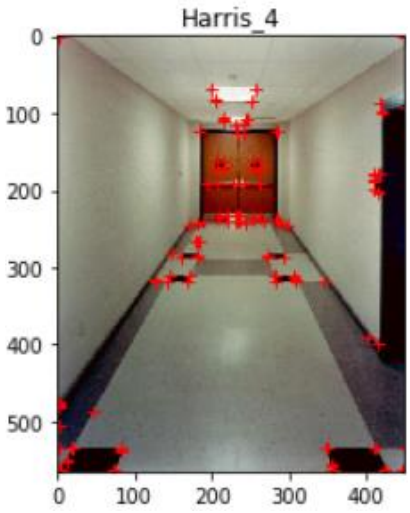
```
    g = fspecial((max(1, np.floor(3 * sigma) * 2 + 1), max(1, np.floor(3 * sigma) * 2 + 1)), sigma) ## fspecial function returns the
gaussian lower pass filter  of size h which is returned the fspecial function with the standard deviation of sigma which is equal to 2

    Iy2 = conv2(np.power(Iy, 2), g) # Over here, we convolute intensity matrix(Iy**2) with the gaussian filter formed
    Ix2 = conv2(np.power(Ix, 2), g) # Over here, we convolute intensity matrix(Ix**2) with the gaussian filter formed
    Ixy = conv2(Ix * Iy, g) # We convolute intensity matrix(Ix*Iy) with the gaussian filter formed
```

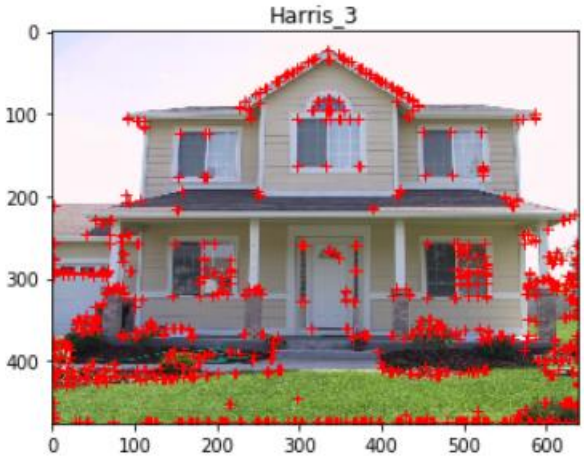
4) Test this function on the provided four test images (can be downloaded from Wattle). Display your results by marking the detected corners on the input images (using circles or crosses, etc) [2]

The above function which has been made from scratch throws some results in which the corners of the image are represented as red cross mark.

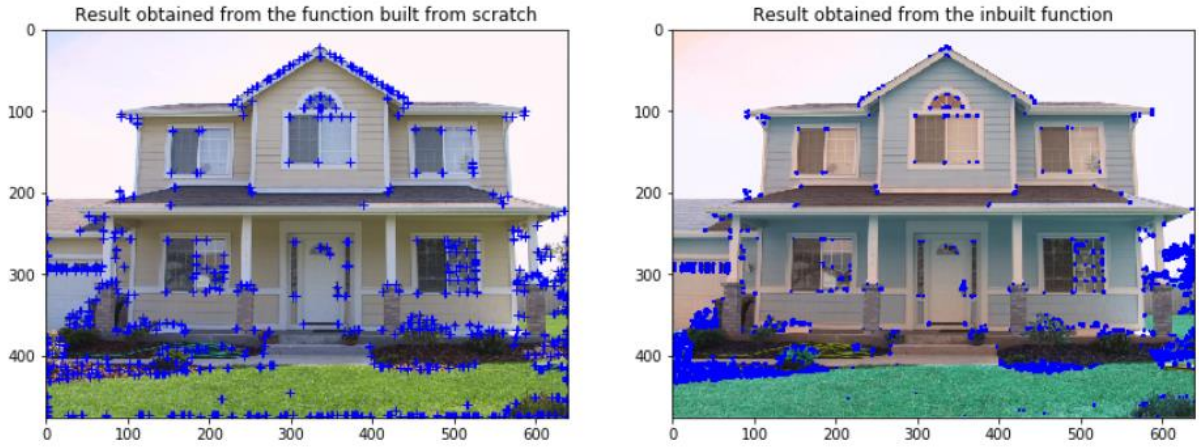








5) **Compare your results with that from python's built-in function *cv2.cornerHarris()* (0.5 mark), and discuss the factors that affect the performance of Harris corner detection (1 mark) [2]**



The first image is the result obtained from the function which is made from the scratch and the second image is the result obtained from the inbuilt function.

Factors:

So, if the threshold is decreased to a certain, then it starts to count some points as the counters which are not the corners and if we increase the threshold, then we start to miss some of the corners in the picture. When we increase the window size in the harris corner detection, number of responses per corners increases rapidly.

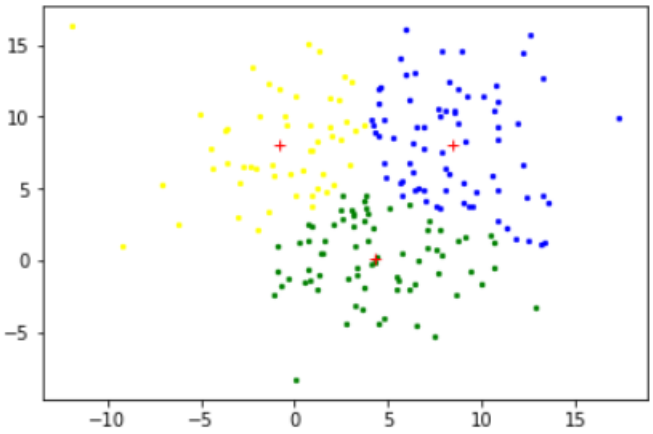
2. K means clustering and Colour Image Segmentation

1) **Implement your own K-means function *my_kmeans()*.The input is the data points to be processed and the number of clusters, and the output is several clusters of points (you can use a cell array for clusters). Make sure each step in K-means is correct and clear, and comment on key code fragments [2]**

```
def my_kmeans(matrix,k,given_iterations):

    clusters = np.zeros(matrix.shape[0]) #Assign the clusters of each data point
    id1 = np.random.randint(matrix.shape[0], size=k) ## Selecting random index from the data points given

    distances_from_centroids = np.zeros((matrix.shape[0],k)) ## Array Initialization to store the distances from centroids
    new_centroid = matrix[id1,:] # Initialization of centroid
    max_iterations = 0 # Number of iterarions

    while max_iterations <= given_iterations:
        max_iterations += 1

        clusters = find_clusters(distances_from_centroids,matrix,new_centroid,k) # This functions helps in finding the clusters

        for j in range(k):
            new_centroid[j] = np.mean(matrix[clusters == j], axis=0) #Finding new centroids by taking mean of points in cluster

    return new_centroid , clusters
```

```
def find_clusters(dist,matrix,centroid,k):

    for i in range(k):

        dist[:,i] = np.linalg.norm(matrix - centroid[i],axis = 1) # Geting the distance from the centroid to the points given
        clusters = np.argmin(dist,axis = 1) # Extract the minimum distance and assigning the point to that particular cluster

    return clusters
```

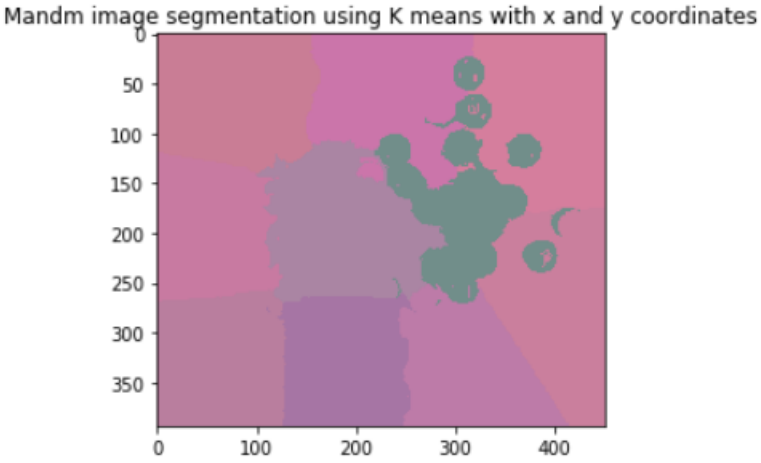
```
new_array = 5 * np.random.randn(200,2) + 5 # Making an array elements which represents gaussian distribution
matrix = np.asmatrix(new_array)
k = 3 # Number of clusters

new_centroid,clusters = my_kmeans(matrix,k,given_iterations=1000) ## Running the function and getting the centroids and clusters
colors=['green', 'blue', 'yellow'] ## Choosing colors for the clusters formed

for i in range(200):
    plt.scatter(matrix[i, 0], matrix[i,1], s=5, color = colors[int(clusters[i])]) ## Plotting cluster points with colors
plt.plot(new_centroid[:,0], new_centroid[:,1], 'r+') # Plotting final centroids using red cross sign.
plt.show()
```



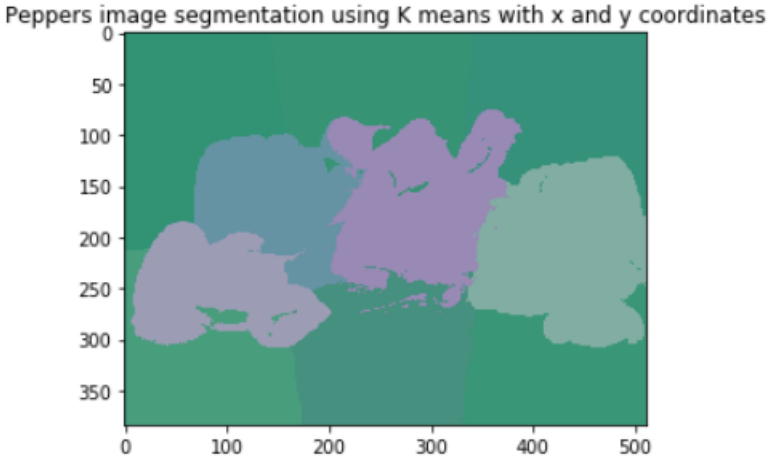
In K means algorithm, we basically choose random points from the dataset for the centroids. After that, distance of points is calculated from the centroids and then is assigned to the cluster which has the least distance from the point. After that, the mean of the points of the same cluster are taken and then new centroids are formed from that. This continues till the algorithm converges and number of iterations are over.

2) **Apply your K-means function to colour image segmentation. Each pixel should be represented as a 5-D vector that encodes: (1) L∗ - lightness of the colour; (2) a∗ - the colour position between red and green; (3) b∗ - the position between yellow and blue; (4) x, y - pixel coordinates. Please compare segmentation results (1) using different numbers of clusters (1 mark), and (2) with and without pixel coordinates (1 mark) [2]**

```
image = cv2.imread('mandm.png',1)  # Reading the image
lab = cv2.cvtColor(image, cv2.COLOR_BGR2LAB) # Converting the image's color space from RGB to LAB
image_5d_array = []

m_mandm = lab.shape[0]
n_mandm = lab.shape[1]

for i in range(m_mandm):
    for j in range(n_mandm):
        image_5d_array.append([lab[i,j,0],lab[i,j,1],lab[i,j,2],i,j]) ## Making of an 5d list

image_5d_array = np.asarray(image_5d_array) # Converting that list to the array
centers, labels = my_kmeans(image_5d_array,10,100)

image_1 = centers[labels]
image_1 = image_1.reshape(394,451,5)

plt.imshow(image_1[:,:,:3])
plt.title("Mandm image segmentation using K means with x and y coordinates")
plt.show()
```

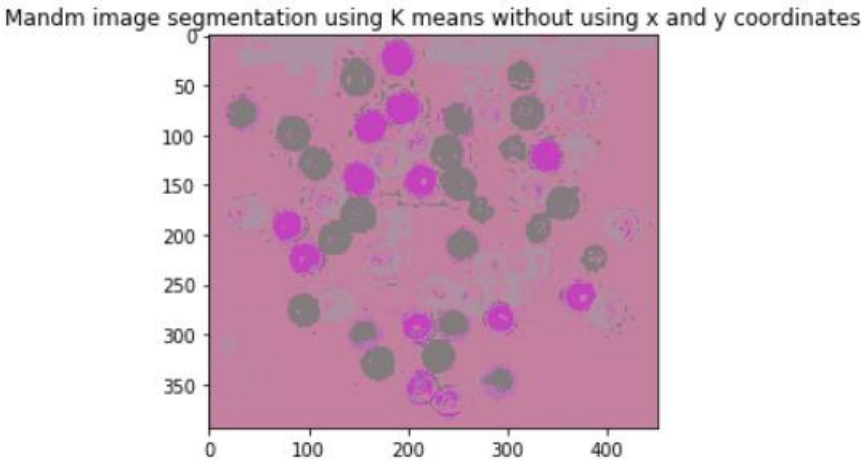



```
image = cv2.imread('peppers.png',1) # Reading the image
lab = cv2.cvtColor(image, cv2.COLOR_BGR2LAB) # Converting the image's color space from RGB to LAB
image_5d_array = []

m_peppers = lab.shape[0]
n_peppers = lab.shape[1]

for i in range(m_peppers):
    for j in range(n_peppers):
        image_5d_array.append([lab[i,j,0],lab[i,j,1],lab[i,j,2],i,j]) ## Making of an 5d list

image_5d_array = np.asarray(image_5d_array) # Converting that list to the array
centers, labels = my_kmeans(image_5d_array,10,100)

image_1 = centers[labels]
image_1 = image_1.reshape(384,512,5)

plt.imshow(image_1[:,:,:3])
plt.title("Peppers image segmentation using K means with x and y coordinates")
plt.show()
```

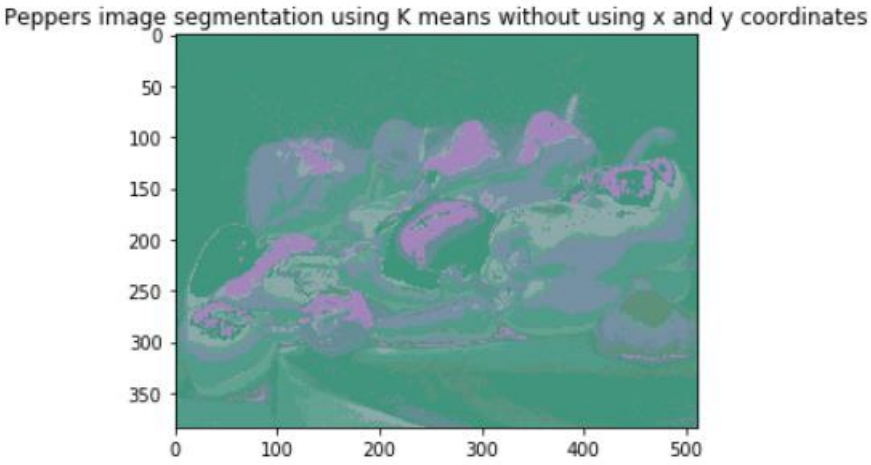


The above images are the plots when we consider the x and y coordinates.

```
image = cv2.imread('mandm.png',1) # Reading the image
lab = cv2.cvtColor(image, cv2.COLOR_BGR2LAB) # Converting the image's color space from RGB to LAB
image_3d_array = []

m_mandm = lab.shape[0]
n_mandm = lab.shape[1]

for i in range(m_mandm):
    for j in range(n_mandm):
        image_3d_array.append([lab[i,j,0],lab[i,j,1],lab[i,j,2]]) ## Making of an 5d list

image_3d_array = np.asarray(image_3d_array) # Converting that list to the array
centers, labels = my_kmeans(image_3d_array,10,100)

image_1 = centers[labels]
image_1 = image_1.reshape(394,451,3)

plt.imshow(image_1)
plt.title("Mandm image segmentation using K means without using x and y coordinates")
plt.show()
```



```
image = cv2.imread('peppers.png',1) # Reading the image
lab = cv2.cvtColor(image, cv2.COLOR_BGR2LAB) # Converting the image's color space from RGB to LAB
image_3d_array = []

m_peppers = lab.shape[0]
n_peppers = lab.shape[1]

for i in range(m_peppers):
    for j in range(n_peppers):
        image_3d_array.append([lab[i,j,0],lab[i,j,1],lab[i,j,2]]) ## Making of an 5d list

image_3d_array = np.asarray(image_3d_array) # Converting that list to the array
centers, labels = my_kmeans(image_3d_array,10,100)

image_1 = centers[labels]
image_1 = image_1.reshape(384,512,3)

plt.imshow(image_1)
plt.title("Peppers image segmentation using K means without using x and y coordinates")
plt.show()
```



The above images are the plots when we don't consider the x and y coordinates.

So, from this we can deduce that when we consider the x and y axis coordinates, the results of the of image segmentation are bad with respect to the result that we achieved without using the x and y coordinates. In this algorithm of image segmentation, k, i.e., number of clusters are set 10 and the iterations are set to 100. The only reason for setting these many iterations is because by using a lot of iterations, there is higher probability that the algorithm has converged or the final centroids are optimum which means there cannot be changes in the centroids further.

3) **The standard K-means algorithm is sensitive to initialization (e.g. initial cluster centres/seeds). One possible solution is to use K-means++, in which the initial seeds are forced to be far away from each other (to avoid local minimum). Please read the material http://ilpubs.stanford.edu:8090/778/1/2006-13.pdf, summarize the key steps in the report (0.5 mark), and then implement K-means++ in your standard algorithm as a new initialization strategy (0.5 mark). Compare the image segmentation performance (*e.g.*, convergence speed and segmentation results) of this new strategy, with that of standard K-means, using different numbers of clusters and the same 5-D point representation from previous question (0.5 mark). [2]**

The only difference between the K means algorithm and K means algorithm is that, in k means centroids are initialized randomly whereas in K means ++ algorithm, the centroids are chosen from the dataset in such a way that all points are farthest from each other. It overcomes the limitation of poor initialization of centroids which leads to bad clusters. So, after the initial points have been selected, then we try to find the distance of other points from the centroids and assign the point to the cluster whose distances is the shortest from the centroid. After this, we find the new centroid by taking the mean of the datapoints belonging in that cluster. This algorithm continues till there is convergence as well as the number of iterations are completed.

3. Face Recognition using Eigenface

1) **Unzip the face images and get an idea what they look like. Then take 10 different frontal face images of yourself, convert them to grayscale, and align and resize them to match the images in Yale-Face. Explain why alignment is necessary for Eigen-face [2]**

```
my_image_array = []

for a in glob.glob('Yale-FaceA/trainingset/*.png'):
    my_image = cv2.imread(a,0)  ## Reading the images from the training dataset
    my_image = np.asarray(my_image,dtype = 'uint8') ## Store the values in an array
    my_image_array.append(my_image) ## Append the values in a list

image_extracted = my_image_array[:][135:144] # These images are "my images" which are basically are not of the same size as oth

for i in range(9):
    image_extracted[i] = cv2.resize(image_extracted[i],(195,231)) ## Resizing the images

cv2.imwrite("Yale-FaceA/trainingset/subject16.confused.png",image_extracted[0])  ## Saving the images after resizing for the fu
cv2.imwrite("Yale-FaceA/trainingset/subject16.doubt.png",image_extracted[1])  ## Saving the images after resizing for the furth
cv2.imwrite("Yale-FaceA/trainingset/subject16.fine.png",image_extracted[2])   ## Saving the images after resizing for the furth
cv2.imwrite("Yale-FaceA/trainingset/subject16.happy.png",image_extracted[3])   ## Saving the images after resizing for the furt
cv2.imwrite("Yale-FaceA/trainingset/subject16.ignorance.png",image_extracted[4])   ## Saving the images after resizing for the
cv2.imwrite("Yale-FaceA/trainingset/subject16.laughing.png",image_extracted[5])  ## Saving the images after resizing for the fu
cv2.imwrite("Yale-FaceA/trainingset/subject16.shocked.png",image_extracted[6])  ## Saving the images after resizing for the fur
cv2.imwrite("Yale-FaceA/trainingset/subject16.smile.png",image_extracted[7])  ## Saving the images after resizing for the furth
cv2.imwrite("Yale-FaceA/trainingset/subject16.thinking.png",image_extracted[8])  ## Saving the images after resizing for the fu
```

Over here, glob.glob has been used so that we can extract images from the training set as given above. 0 is placed after the image reading in cv.imread function because it will return 2d image instead of 3d image. After making the changes in the shape using the resize function and setting it to the shape of (231,195). The reason in the resize function dimension are reversed because the resize function takes first width and then height into account. After that image is saved in the same file for the usage.

Alignment is necessary for eigen face because we are subtracting the mean face from the matrix to make the normalized matrix accordingly and if we don't do that, the images pixels will not be aligned due to which could image on the edge also and the range of the values will be different and it could lead to bad estimation.

2) Train an Eigen-face recognition system. Specifically, at a minimum your face recognition system should do the following [2]:

i) Read all the 135 training images from Yale-Face, represent each image as a single data point in a high dimensional space (the entire dataset is converted to a 2D map) and collect all the data points into a big data matrix [2].

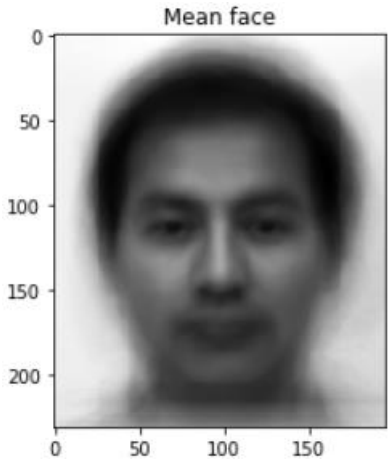
```
image_array_formed = []
flattened = []

for a in glob.glob('Yale-FaceA/trainingset/*.png'):
    my_image = cv2.imread(a,0) # Reading the images from the training dataset
    my_image = np.asarray(my_image,dtype = 'uint8')
    image_array_formed.append(my_image) ## Append the values in the list

image_array_formed = image_array_formed[:][:135]

for j in range(len(image_array_formed)):
    k = image_array_formed[j].flatten() ## Flattening of the image
    flattened.append(k) ## Appending the flattened images in a list

A t matrix = np.matrix(flattened) ## Converting the flattened image list to an matrix
```

Looping around training image dataset using glob.glob provides us the array of the images which could be appended to the list. The reason for selecting image_array_formed[:][:135] these images is because initially, we are not concerned about the images of our faces. Image is flattened use flatten function to get the 2d flattened list. After the flattened list, it is converted to the matrix. So, at the end, images are converted into big 2d matrix which Is our goal that is achieved.

ii) Perform PCA on the data matrix (1 mark), and display the mean face (1 mark). Given the size of input image, direct eigen decomposition of covariance matrix would be slow. Read lecture notes and find a faster way to compute eigen values and vectors, explain the reason (1 mark) and implement it in your code (1 mark) [2].

```
mean_matrix = np.mean(A_t_matrix.T, 1)  # Calculating mean of the matrix
mean_array = np.asarray(mean_matrix, dtype = 'uint8')
mean_array = np.reshape(mean_array,(231,195)) # Reshapeing the image so as to plot the mean face
plt.title("Mean face")
plt.imshow(mean_array, cmap = "gray")
plt.show()

A_t_matrix = A_t_matrix.T - mean_matrix #Normalizing the matrix

Covariance_matrix = (A_t_matrix.T@A_t_matrix/(len(A_t_matrix) - 1))  ## Calculating the covariance of the matrix

u,v = np.linalg.eigh(Covariance_matrix) ## Calculating the eigenvector and eigenvalues

u = u.tolist()
u_index = sorted(range(len(u)), key=lambda k: u[k]) # Getting the index of the sorted array
u = sorted(u, reverse=True)
u = np.matrix(u)

v = v.tolist()
v.sort(key = lambda u_index: u_index) # Sorting the eigenvectors according to the index of the eigenvalues
v = np.asmatrix(v)
v = v[::-1]

eigen_faces = np.dot(A_t_matrix,v) # Dot product of the two matrix to make eigenface

eigen_faces = eigen_faces.T
```

PCA is basically reducing the dimension of the image and extracting the maximum information out of it. So, we need covariance matrix for the calculating the eigenfaces which completes the PCA procedure. So, initially, mean matrix is extracted from the image matrix. The following image displays the mean image of training set:



The mean matrix is subtracted so as to get the normalized matrix so that covariance matrix can be extracted from that using the formula

Covariance = $\frac{1}{M}\ A\ A^T$.

So, if we use the old way, that is finding the covariance matrix, then it has a high order of complexity because as we know that A matrix formed from the image has (45045, 135) dimension and if we multiply the matrix, then we will get a matrix of (45045,45045) which is of a big dimension and it can cause a memory error in your laptop if you have a basic one. So, to avoid that we can use the equation of

$$Av = \boldsymbol{\lambda} v$$

where v is the eigenvector of the A and **λ** is the eigen value.

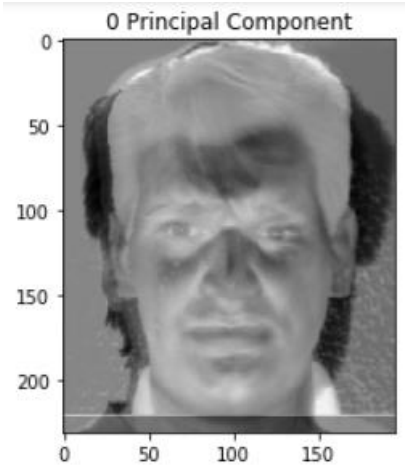
From this, equation we can deduce that the eigenvectors of $A\,A^T$ and $A^T A$ are the same

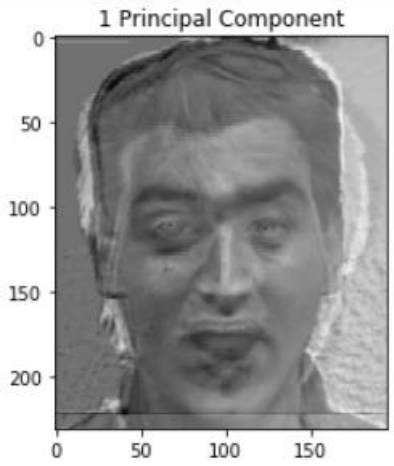
So, if we calculate the covariance matrix using $A^T A$, then we will get the dimension of (135, 135) and it is of very less complexity as that of $A\,A^T$. So, after calculating the eigenvalues and eigenvectors, we need to arrange them in descending order so that we can get the highest principal components, After that, we can multiply the eigenvectors with the image matrix. This provide us the eigenface. Values of this eigenfaces will be same when we calculate the eigenvector for $A\,A^T$ which has high complexity. So, in this way, we can reduce the complexity as well as make the computation of the program much faster. So, for the better of the program, this has been implemented in the program as we can see in the picture above for the programming.

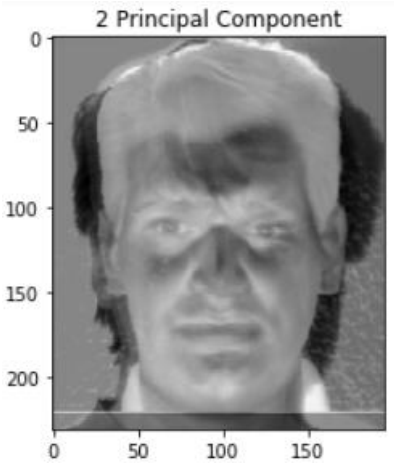
iii) Determine the top k principal components and visualize the top-k eigen- faces in your report (1 mark). You can choose k=10 or k=15 [2].

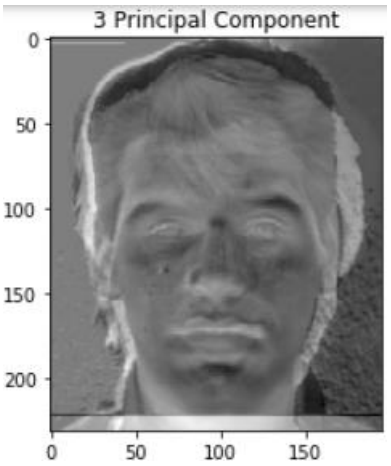
So, as to get the top k principal components, we want to have eigenvectors with the largest eigenvalues. So, to obtain largest values, we want to arrange the eigenvalues in the descending order and according to that index, we should sort the eigenvector in descending order. From, this we can extract the top k principal components. This is done in the previous part when we were calculating eigenfaces which can be seen from the above image. For printing the top 10 eigenfaces, then following code is written:

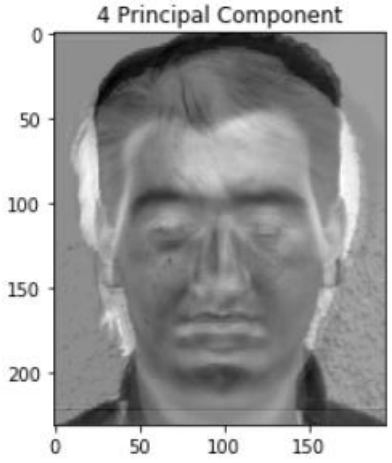
```
k = 10

### Plotting of the top k principal components

for i in range(k):
    new_im = eigen_faces[:][i]
    new_im = new_im.T
    new_im = np.reshape(new_im,(231,195))
    plt.imshow(new_im, cmap = "gray")
    plt.title(str(int(i)) + " " + "Principal Component")
    plt.show()
```

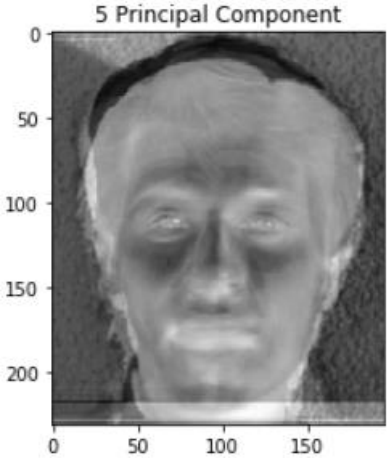


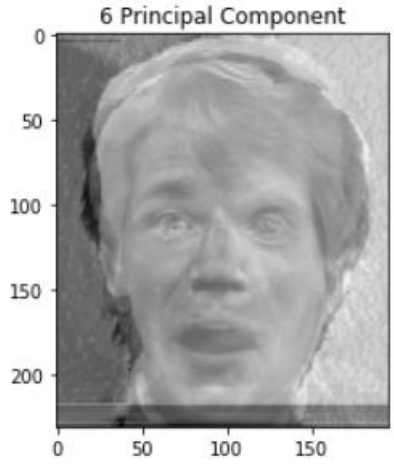

1 Principal Component
0
50
100
150
200
0
50
100
150

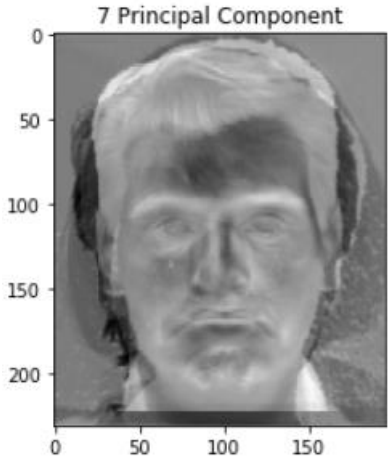

2 Principal Component
0
50
100
150
200
0
50
100
150


3 Principal Component
0
50
100
150
200
0
50
100
150

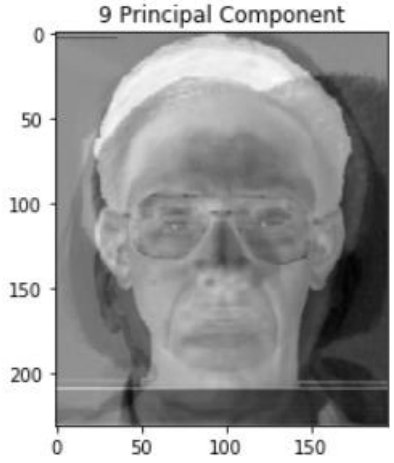

4 Principal Component
0
50
100
150
200
0
50
100
150


5 Principal Component
0
50
100
150
200
0
50
100
150


6 Principal Component
0
50
100
150
200
0
50
100
150


7 Principal Component
0
50
100
150
200
0
50
100
150


8 Principal Component
0
50
100
150
200
0
50
100
150


9 Principal Component
0
50
100
150
200
0
50
100
150

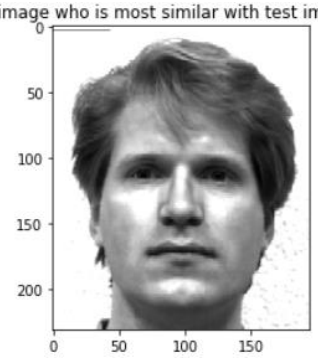
iv) For each of the 10 test images in Yale-Face, read in the image, determine its projection onto the basis spanned by the top k eigenfaces. Use this projection for a nearest-neighbour search over all the 135 faces, and find out which three face images are the most similar. Show these top 3 faces next to the test image in your report (1.5 marks). Report and analyse the recognition accuracy of your method (1 mark) [2].

```
training_weights = A_t_matrix.T@eigen_faces[:10][:].T # Top 10 principal components is used for the training weights

testing_image_array_formed = []
testing_flattened = []

for i in glob.glob('Yale-FaceA/testset/*.png'):
    im = cv2.imread(i,0) # Reading the test image dataset
    im = np.asarray(im, dtype = 'uint8') ## Conversion of the matrix
    testing_image_array_formed.append(im) ## Appending the values in the list

## Conversion of test image to appropriate size#########
my_test_image = testing_image_array_formed[:][10] ## Extracting "my image" from the dataset to change the size of the image

my_test_image = cv2.resize(my_test_image,(195,231)) ## Resizing the image

cv2.imwrite('Yale-FaceA/testset/subject16.normal.png',my_test_image) # Storing the image the test dataset
################################################
testing_image_array_formed = testing_image_array_formed[:10] ## "My image" is not considered for testing because it is not requ

for j in range(len(testing_image_array_formed)):
    k = testing_image_array_formed[j].flatten()  ## Flattening of the image
    testing_flattened.append(k) ## Appending the flattened image into a list
B_testing_matrix = np.matrix(testing_flattened)

Normalized_test = B_testing_matrix.T - mean_matrix # Normalization of the test image matrix

testing_weights = Normalized_test.T@eigen_faces[:10][:].T # Top 10 principal components is used for the testing weights

val = np.zeros((10,135))
diff_array = []

for i in range(10):
    for j in range(135):
        val[i][j] = np.linalg.norm(training_weights[:][j] - testing_weights[:][i]) # Calculating the norm for the image so as t

sort_index_array1 = np.zeros((10,3))
```

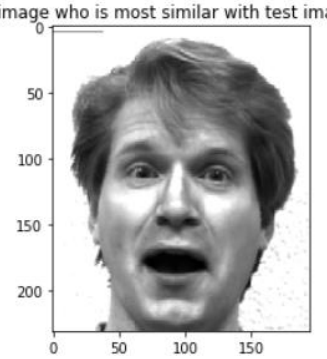
```
for b in range(10):
    sort_index_array = sorted(range(len(val[b][:])), key=lambda k: val[b][:][k]) # Sorting the array and extracting the index a
    sort_index_array1[b][:] = sort_index_array[:3]

## Plotting of the images
for l in range(10):
    for m in range(3):
        plt.title("Training image who is most similar with test image of index" + str(int(l)))
        plt.imshow(image_array_formed[int(sort_index_array1[:][l][m])], cmap = "gray")
        plt.show()
    plt.title("Testing Image of Index" + str(int(l)))
    plt.imshow(testing_image_array_formed[l], cmap = "gray")
    plt.show()
```

So, before reading the testing images, training weights should be calculated so as to compare that with the testing weights to calculate the top 3 similar images. Training weight is calculated by multiplying the normalized matrix with the eigen faces. After that, we can read all the test images using the glob.glob . Similarly, we remove the last image from the dataset because the last image in the test set was my image which was inserted afterwards by me and we don't need that image in initial questions. After that weights are obtained for testing set by multiplying the eigen face value with the normlaized test set, then it is compared with the trainig so as to extract the image closest to the testing image by sorting the index of the matrix accordingly. After that, top 3 images are chosen and then the image is plotted accordingly. The following images shows the top 3 images which are similar to the test set.

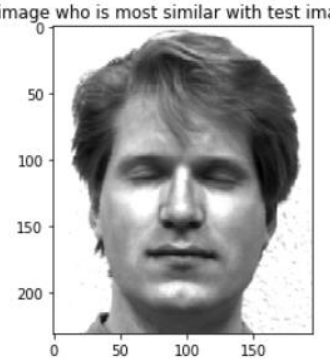
Training image who is most similar with test image of index0



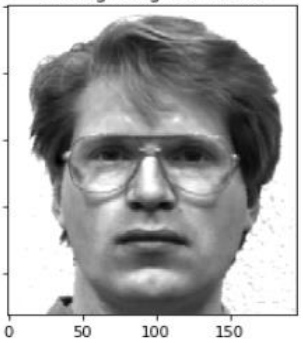
Training image who is most similar with test image of index0



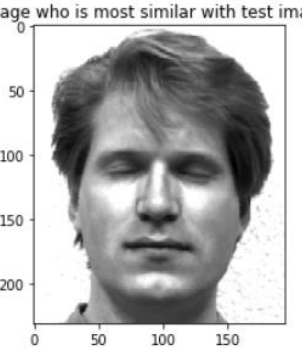
Training image who is most similar with test image of index0



Testing Image of Index0

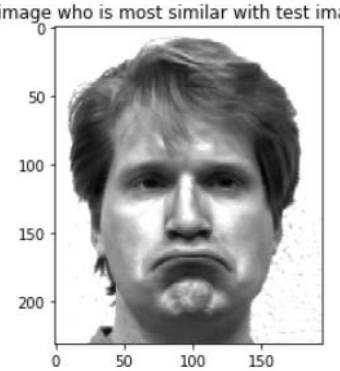


Training image who is most similar with test image of index1



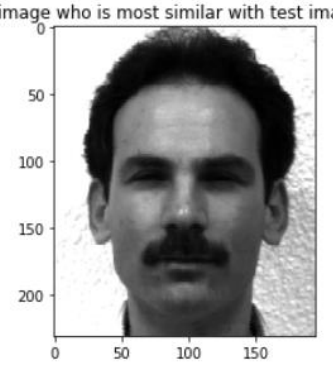
Training image who is most similar with test image of index1

Training image who is most similar with test image of index1

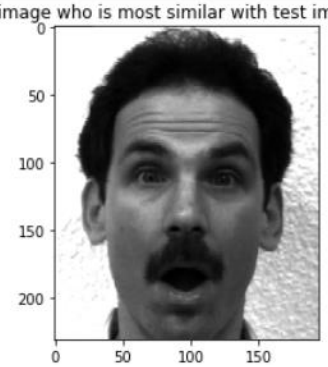


Testing Image of Index1

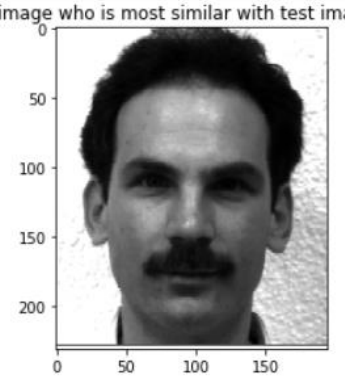
Training image who is most similar with test image of index2



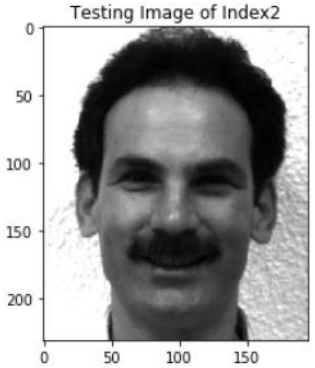
Training image who is most similar with test image of index2



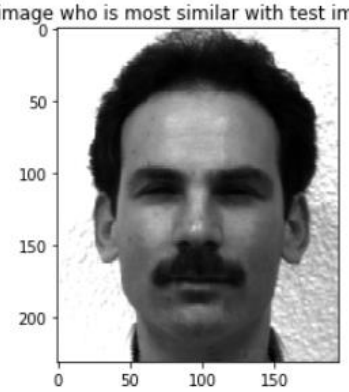
Training image who is most similar with test image of index2



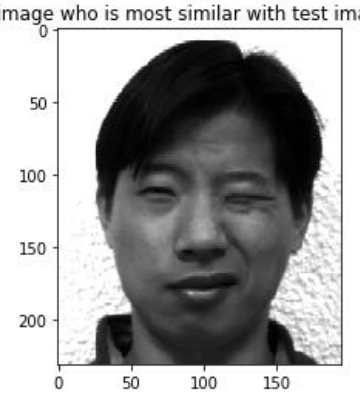
Testing Image of Index2



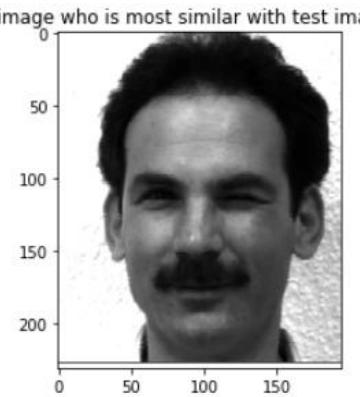
Training image who is most similar with test image of index3



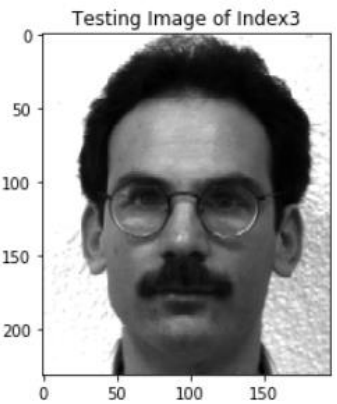
Training image who is most similar with test image of index3



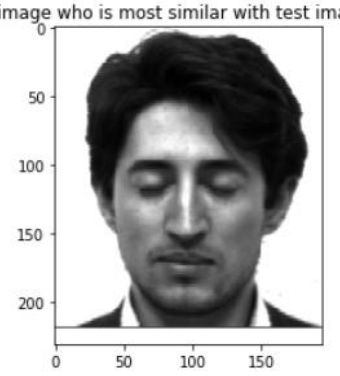
Training image who is most similar with test image of index3



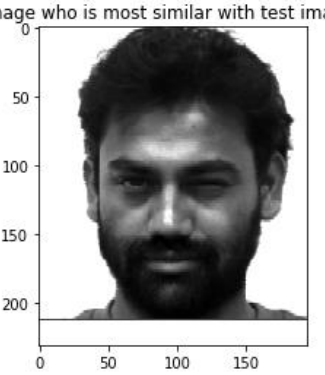
Testing Image of Index3



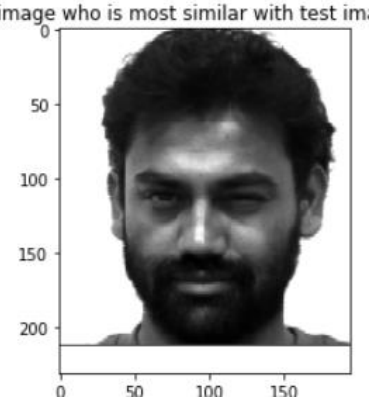
Training image who is most similar with test image of index4



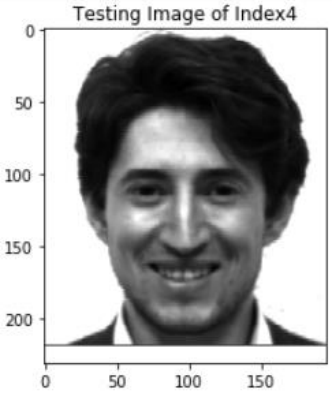
Training image who is most similar with test image of index4



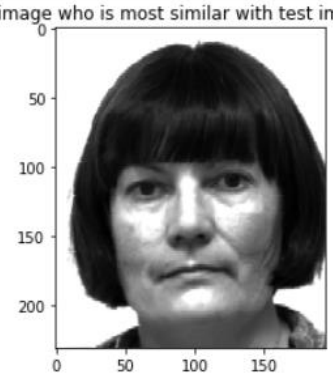
Training image who is most similar with test image of index4



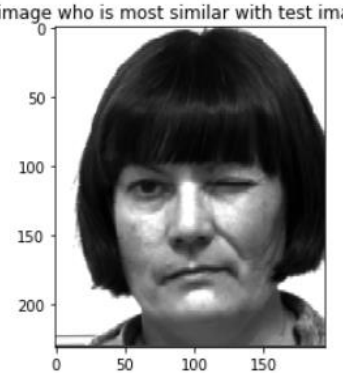
Testing Image of Index4



Training image who is most similar with test image of index5



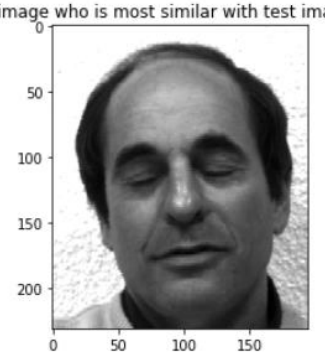
Training image who is most similar with test image of index5



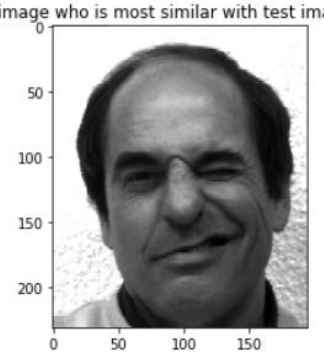
Training image who is most similar with test image of index5

Testing Image of Index5

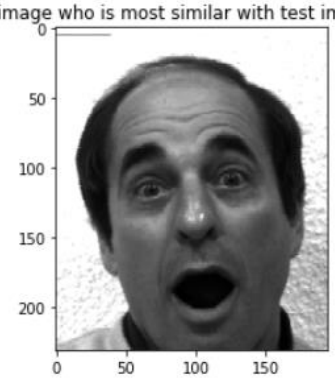
Training image who is most similar with test image of index6



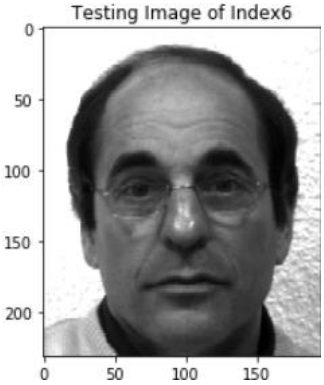
Training image who is most similar with test image of index6



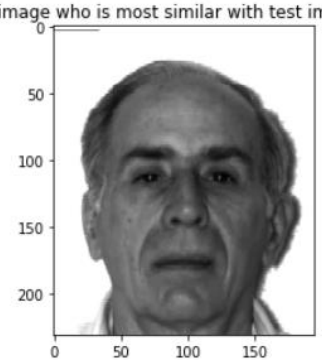
Training image who is most similar with test image of index6



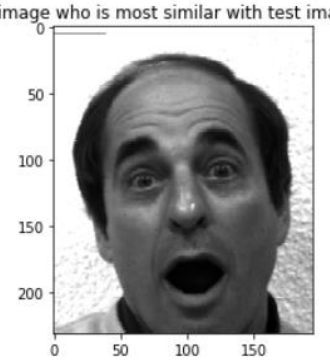
Testing Image of Index6



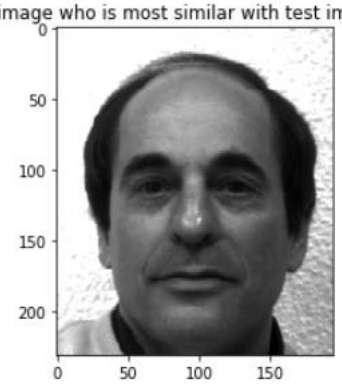
Training image who is most similar with test image of index7



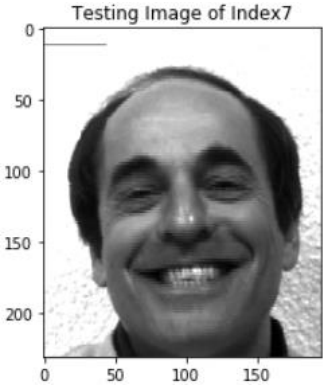
Training image who is most similar with test image of index7



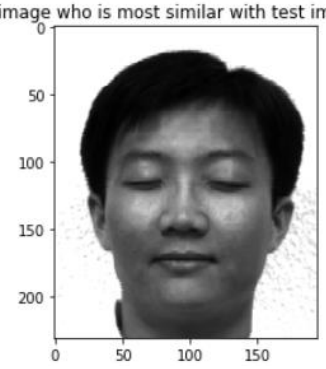
Training image who is most similar with test image of index7



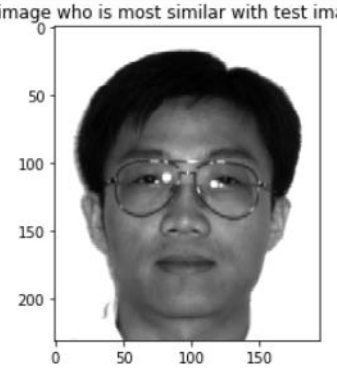
Testing Image of Index7



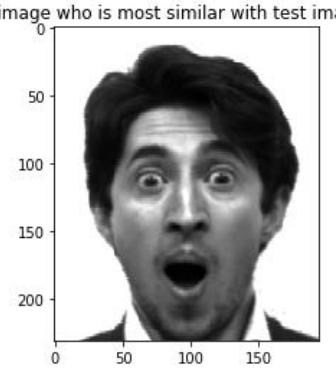
Training image who is most similar with test image of index8



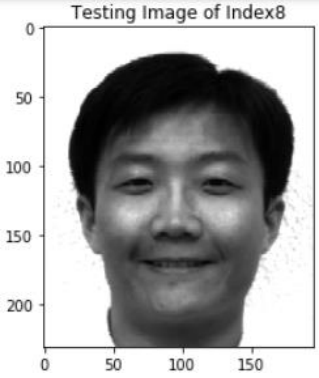
Training image who is most similar with test image of index8



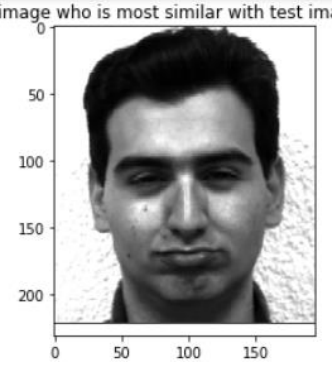
Training image who is most similar with test image of index8



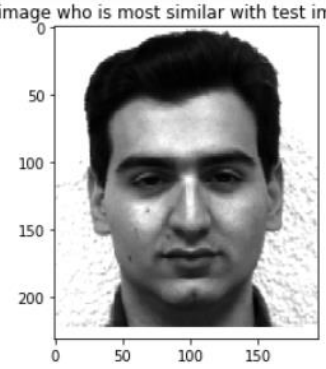
Testing Image of Index8

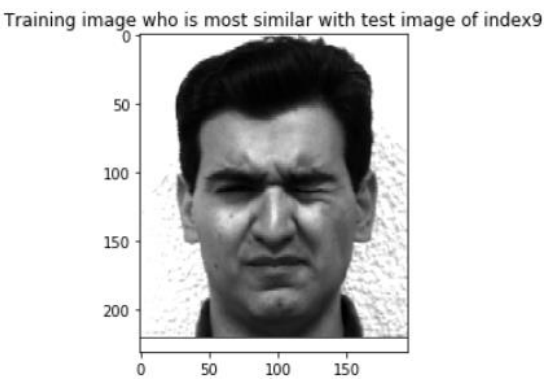


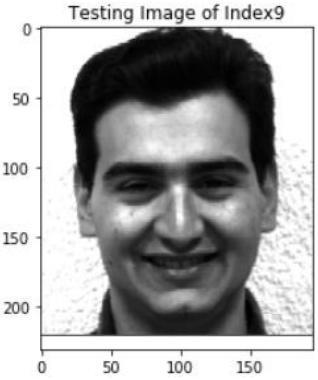
Training image who is most similar with test image of index9



Training image who is most similar with test image of index9







So, as we can deduce from the above, model works really good for image recognition. Although, there are some mistakes that are made by the model in face recognition which will hamper the accuracy of the model. So, for test image at index 3, one image is wrongly classified, for test image indexed at 4, two images are wrong classified, for test image at index 7, one image is wrongly classified and for index 8, one image is wrong classified. So, as we know that there are 30 images in total out of which 25 are correctly classified. So,

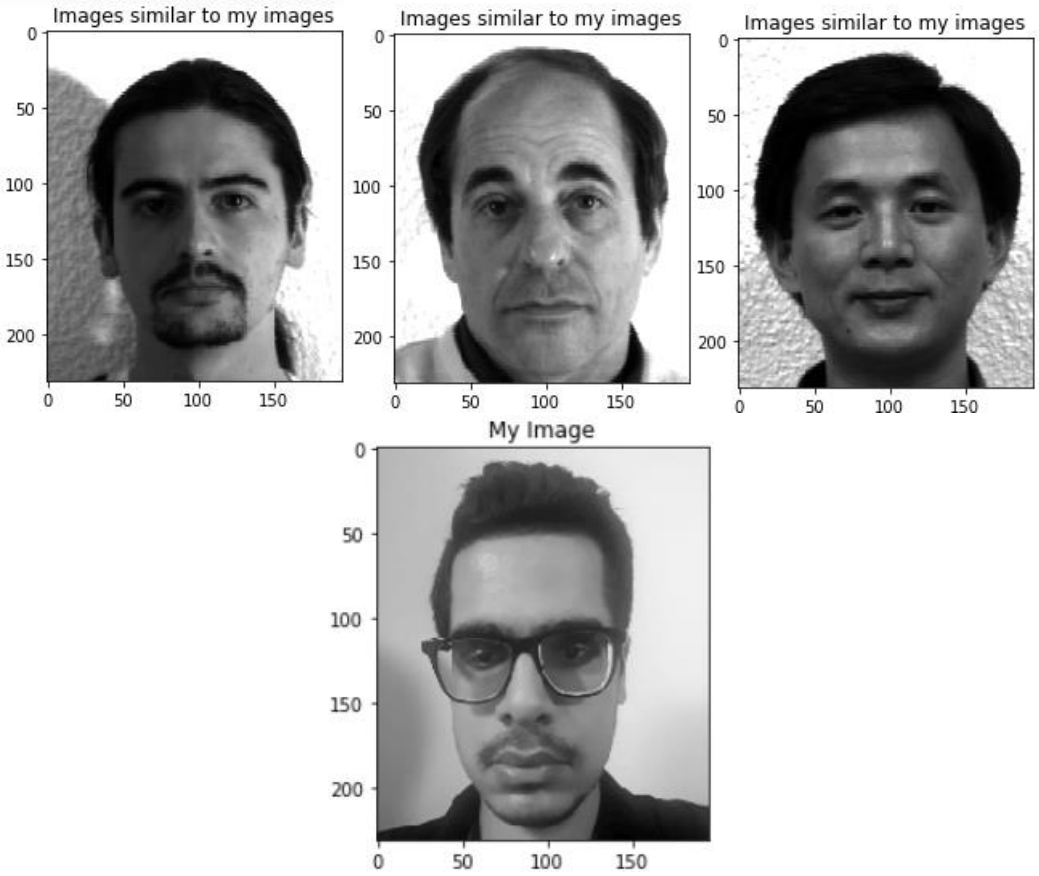
Accuracy for the model will be = (25/30) * 100 = 83.33 %

v) Read in one of your own frontal face images. Then run your face recognizer on this new image. Display the top 3 faces in the training folder that are most similar to your own face (1 mark) [2]

```
my_testing_flattened = []

my_testing_flattened = my_test_image.flatten() # Flattening fo the image
my_B_testing_matrix = np.matrix(my_testing_flattened) ## Storing the image in the the matrix

my_Normalized_test = my_B_testing_matrix.T - mean_matrix ## Normalizing the matrix by subtracting the mean

my_image_testing_weights = my_Normalized_test.T@eigen_faces[:10][:].T # Getting testing weights using the images

val_my = np.zeros((135))

for j in range(135):
    val_my[j] = np.linalg.norm(training_weights[:][j] - my_image_testing_weights) # Calculating the norm for the image so as to

sort_index_array_for_my_image = sorted(range(len(val_my)), key=lambda k: val_my[k]) # Sorting the array and extracting the index

sort_index_array_for_my_image = sort_index_array_for_my_image[:3]

# Plotting the image closest to my image
for h in range(len(sort_index_array_for_my_image)):
    plt.imshow(image_array_formed[sort_index_array_for_my_image[h]],cmap = "gray")
    plt.title("Images similar to my images")
    plt.show()

plt.imshow(my_test_image, cmap = "gray")
plt.title("My Image")
plt.show()
```

My face image is to be resized which was done early. Then, the image is flattened and the matrix derived from that flattening of the image is multiplied with eigen faces so as to get the testing weight and then, norm is used so that, we can extract the distance between the images from the training dataset and the testing image. After that, the list is sorted and the index of that image is extracted which is then plotted by running the loop.

The following images in the training set are closest to my face:

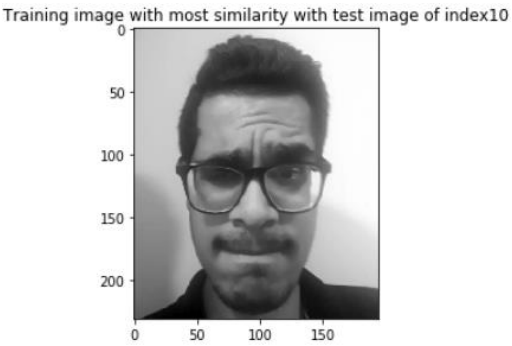


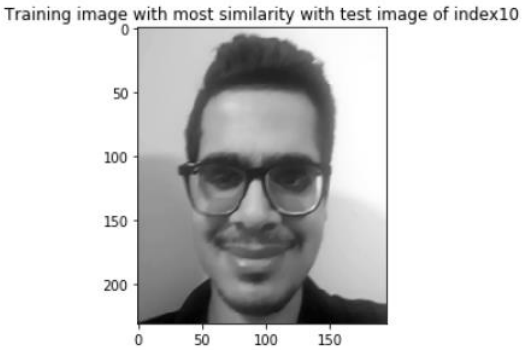
vi) Repeat the previous experiment by pre-adding the other 9 of your face images into the training set (a total of 144 training images). Note that you should make sure that your test face image is different from those included in the training set. Display the top 3 faces that are the closest to your face (1 mark) [2].

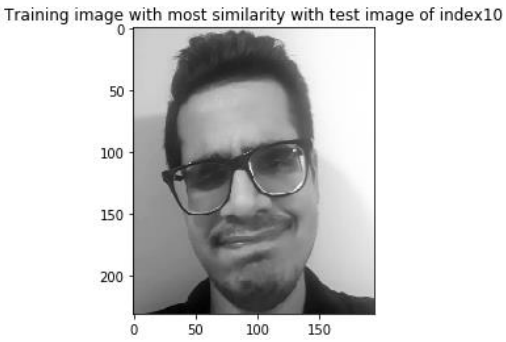
```
image_array_formed = []
flattened = []

for a in glob.glob('Yale-FaceA/trainingset/*.png'):
    my_image = cv2.imread(a,0) # Reading the images from the training dataset
    my_image = np.asarray(my_image,dtype = 'uint8')
    image_array_formed.append(my_image) ## Append the values in the list

for j in range(len(image_array_formed)):
    k = image_array_formed[j].flatten() ## Flattening of the image
    flattened.append(k) ## Appending the flattened images in a list

A_t_matrix = np.matrix(flattened) ## Converting the flattened image list to an matrix

A_t_matrix = A_t_matrix.T - mean_matrix #Normalizing the matrix

Covariance_matrix = (A_t_matrix.T@A_t_matrix/(len(A_t_matrix) - 1))  ## Calculating the covariance of the matrix

u,v = np.linalg.eigh(Covariance_matrix) ## Calculating the eigenvector and eigenvalues

u = u.tolist()
u_index = sorted(range(len(u)), key=lambda k: u[k]) # Getting the index of the sorted array
u = sorted(u, reverse=True)
u = np.matrix(u)

v = v.tolist()
v.sort(key = lambda u_index: u_index) # Sorting the eigenvectors according to the index of the eigenvalues
v = np.asmatrix(v)
v = v[::-1]

eigen_faces = np.dot(A_t_matrix,v) # Dot product of the two matrix to make eigenface

eigen_faces = eigen_faces.T

training_weights = A_t_matrix.T@eigen_faces[:10][:].T # Top 10 principal components is used for the training weights

testing_image_array_formed = []
testing_flattened = []

for i in glob.glob('Yale-FaceA/testset/*.png'):
    im = cv2.imread(i,0) # Reading the test image dataset
    im = np.asarray(im, dtype = 'uint8') ## Conversion of the matrix
    testing_image_array_formed.append(im) ## Appending the values in the list
```

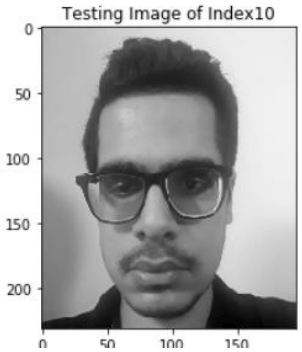
```
for j in range(len(testing_image_array_formed)):
    k = testing_image_array_formed[j].flatten()  ## Flattening of the image
    testing_flattened.append(k) ## Appending the flattened image into a list
B_testing_matrix = np.matrix(testing_flattened)

Normalized_test = B_testing_matrix.T - mean_matrix # Normalization of the test image matrix

testing_weights = Normalized_test.T@eigen_faces[:10][:].T # Top 10 principal components is used for the testing weights

val = np.zeros((11,144))
diff_array = []

for i in range(11):
    for j in range(144):
        val[i][j] = np.linalg.norm(training_weights[:][j] - testing_weights[:][i]) # Calculating the norm for the image

sort_index_array1 = np.zeros((11,3))

for b in range(11):
    sort_index_array = sorted(range(len(val[b][:])), key=lambda k: val[b][:][k])
    sort_index_array1[b][:] = sort_index_array[:3]

for m in range(3):
    plt.title("Training image with most similarity with test image of index" + str(int(10)))
    plt.imshow(image_array_formed[int(sort_index_array1[:][10][m])], cmap = "gray")
    plt.show()
plt.title("Testing Image of Index" + str(int(10)))
plt.imshow(testing_image_array_formed[10], cmap = "gray")
plt.show()
```

The coding for this part is almost the same but the difference is that my images are also used in the training dataset due to which the mean face also changes a bit. The process remains the same, i.e, we have to calculate the norm of the training and testing weights which will provide us with the image close to the test image from the training dataset. Over here, we are more concerned about the testing of the model on my image and getting the results accordingly from the training dataset. So, when my image is used for testing, we get the following results:









So, from this, we can conclude that model which is made is capable to extracted the right image with respect to my test image in the test dataset as well as model demonstrates really good accuracy.

References:

[1] https://stackoverflow.com/questions/3862225/implementing-a-harris-corner-detector

[2] C lab2 questions assignment pdf