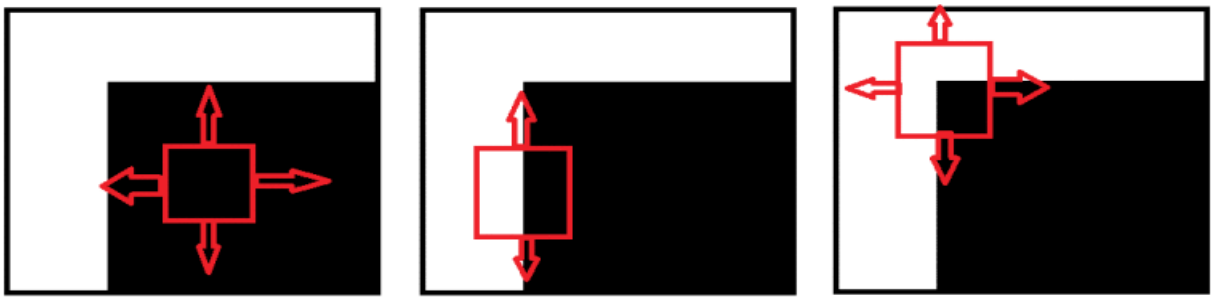
Engn4528 Clab – 2
Report

Yadu Krishna Choyi
u6728671

Contents

Task 1: Harris corner detector

The Harris corner detector is a corner detecting algorithm which uses differential of the corner score, and is usually the best and reliable corner detecting algorithm. The algorithm works by using a window which is moved across an over the image. A large change in intensity in this window represents an interest point. This intuition is explained below.



Flat region: no change in all directions

Edge: no change along the edge direction

Corner: significant changes in all directions

The change in intensity is calculated using the following formula. Here u,v is the shift along the coordinates, I(x,y) is the intensity of the image at point (x,y), and w(x,y) is a window function that gives the weight of the change in intensity. For this task, we are given a gaussian function window.

$$E(u,v) = \sum_{x,y} w(x,y)\left[I(x+u,y+v) - I(x,y)\right]^2$$

Window function — Shifted intensity — Intensity

Since the shift (u,v) is usually small, we can use Taylor series expansion on the above equation to obtain the following.

$$E(u,v) \cong \begin{bmatrix} u,v \end{bmatrix} \; M \; \begin{bmatrix} u \\ v \end{bmatrix}$$

$$M = \sum_{x,y} w(x,y) \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix}$$

For the given task, the Harris response is computed using the following code.

```
1:    for i in range(h_r.shape[0]):
2:        for j in range(h_r.shape[1]):
3:            x_sum = np.sum(Ix2[i:i+window_s, j:j+window_s])
4:            y_sum = np.sum(Iy2[i:i+window_s, j:j+window_s])
5:            xy_sum = np.sum(Ixy[i:i+window_s, j:j+window_s])
6:            det = x_sum*y_sum -xy_sum**2
7:            trace =x_sum + y_sum
```

```
8:              r = det - thresh*trace**2
9:              h_r[i,j]=r
```

Here, x_sum, y_sum and xy_sum in line 3, 4, and 5 corresponds to I_x^2, I_y^2, and I_xI_y respectively. Using these values, the variables(determinant and trace) for computing the Harris response is computed[Line 6 and 7]. Using the determinant and trace, the Harris response is computed for all the pixels in the image[Line 8] and stored in a matrix.

The following code performs non maximal suppression on the Harris response matrix.

```
1:    def nms (r):
2:        corner = []
3:        mean = r[r>0].mean()
4:        for i in range(r.shape[0]-2):
5:            for j in range(r.shape[1]-2):
6:                window = r[i:i+3,j:j+3]
7:                max_value = np.amax(window)
8:                index =unravel_index(window.argmax(),window.shape)
9:                if max_value > mean:
10:                   x = i+index[0]
11:                   y = j+index[1]
12:                   corner.append([x,y])
13:     return np.array(corner)
```

Non maximal suppression is performed by using a moving window of size 3×3[Line 6] with step size 1 on the Harris response matrix. The max of the values in the window is compare to the mean of the Harris response matrix[Line 9]. All max values corresponding to all window position is suppressed if the it is less than the mean. The new filtered corner point coordinates are stored in a list[Line 12]. This process removes clutter and refines the output by enabling visualization of important interest points only.

The above function returns a list of coordinates which corresponds to all the interest points/corners in the input image. We can visualize these corner points by changing the color of the pixels to red by setting the pixel value to [255,0,0].

1.1: Factors that affect Harris corner are:

1: Sigma: Varying this parameter changes the gaussian window distribution. A small value of sigma will result in detection of small interest features, and a large sigma value will result in detection of larger interest features, ignoring all small interest features.

2: Thresh: Empirical constant that controls the effect of trace on Harris response.

3: Window size in calculating the intensity change: This changes the area of effect of gaussian filter.

4: Window size of non maximal suppression: Increasing this window size will result in the suppression of more values, resulting in a more refined output.

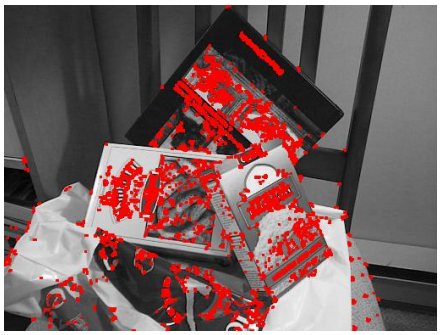
1.2: Test results for Harris corner detection algorithm.

The output of the Harris corner detector along with the outputs using the inbuilt function is given below.[Note: The shift in color when using inbuilt function is because the images are read in BGR format(cv2.imread()), but saved as RGB format. This does not effect the feature output]

Harris_1 (Custom)

Harris_1 (Inbuilt)

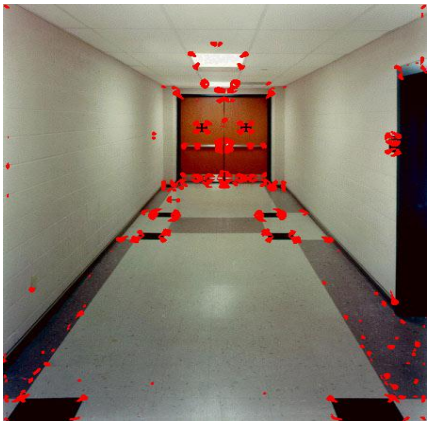
Harris_2 (Custom)



Harris_2 (Inbuilt)

Harris_3 (Custom)

Harris_3 (Inbuilt)



Harris_4 (Custom)

Harris_4 (Inbuilt)

Task 2 : K-means Clustering

The k-means algorithm involves 3 simple steps:

1: Select k random points from the image/dataset as initial centroids.
2: Calculate distance of a point to all centroid. This point is assigned to a centroid to which its distance is the least.
3: Recompute the centroid based on all points in that clustering.
4: Repeat steps **2** and **3**.

2.1: Simple k-means implementation using L, a and b as dimensions.

```
1:    for iter in range(10):
2:            cluster = {k: [] for k in keys}
3:            # find pixels that belong to closest centroid
4:            for i in range(ds.shape[0]):
5:                for j in range(ds.shape[1]):
6:                    cp = ds[i,j] #current pixel
7:                    temp_dist = float('inf')
8:                    counter = 0
9:                    index = 0
10:                   for k in cent:
11:                     # calculate distance from current point to all centroid
12:                     dist = (cp[0]-k[0])**2 + (cp[1]-k[1])**2 + (cp[2]-
                                k[2])**2
13:                       if dist < temp_dist:
14:                           temp_dist = dist
15:                           index = counter
16:                       counter = counter + 1
17:                   # Assign point to the cluster
18:                   if iter != 9:
19:                       cluster[index].append(cp)
20:                   else:
21:                       cluster[index].append([i,j])
22:           kk = cluster.keys()
23:           c = 0
24:           # recompute cluster centroids
25:           if iter != 9:
26:               for i in kk:
27:                   m = np.mean(cluster[i],axis = 0)
28:                   cent[c] = m
29:                   c += 1
```

Ideally we loop and recompute centroids till convergence is reached, but good results can be achieved using 10 iteration[Line 1]. [Line 2] creates an empty dictionary with the centroids as keys. This dictionary will store all the cluster points, and each cluster can be accessed using the centroids as keys. We then loop through all the pixels in the image [Line 4 and 5]. The distance between the current pixel and all the k centroids are calculated[Line 12]. The current pixel belongs to that cluster/centroid whose distance is the least. This is done in Line 13 – Line 21. Once all the pixels are clustered, the centroids are recomputed[Line 25 – Line 29]. The new centroid is the mean of all the points in that cluster. This entire process is repeated 10 times.

2.2: Simple k-means implementation using L, a, b, x and y as dimensions.

The main code implementation is the same as above. The crucial difference in this case is that instead of using just the l, a, b representation, we add the position values (x,y) of each pixel to the l, a, b matrix. This means that the matrix is now of the shape **Image height * Image width * 5** instead of **Image height * Image width * 3**. This dataset matrix is then used for further calculation as before. The code snippet given below shows how the dataset matrix was generated.

```
1:        #fill ax with l, a, b, x, and y features
2:        for i in range(ax.shape[0]):
3:            for j in range(ax.shape[1]):
4:                ax[i,j,0] = lab[i,j,0]
5:                ax[i,j,1] = lab[i,j,1]
6:                ax[i,j,2] = lab[i,j,2]
7:                ax[i,j,3] = i;
8:                ax[i,j,4] = j;
```

Here, **ax** is the dataset matrix. We copy over the l, a, and b values from the lab format image[Line 4-Line 6], and then also add the coordinates of each pixel to **ax**[Line 7 and Line 8].

2.3: Comparing output of k-means using l, a, b as input vs using l, a, b, x, y as input(Varying parameter k).


lab format, k = 2


lab format, k = 5


lab format, k = 8


lab format, k = 10


labxy format, k = 2


labxy format, k = 5


labxy format, k = 8

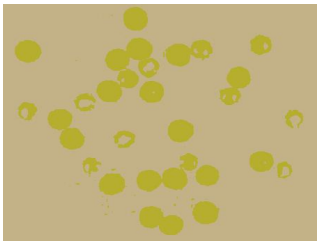

labxy format, k = 10

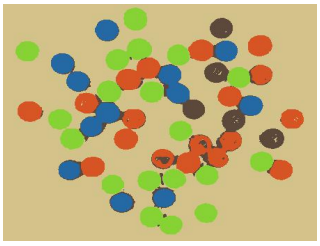
Note: Each element in the cluster is assigned the color of its centroid for visual representation.

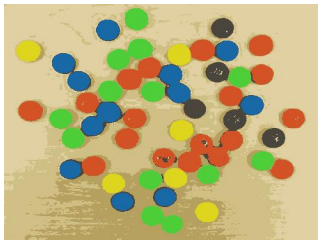
Looking at the output both the formats, we can see that using image coordinates (x and y) results in clusters that are localized not only based on the color of the objects, but also on the pixel coordinates.

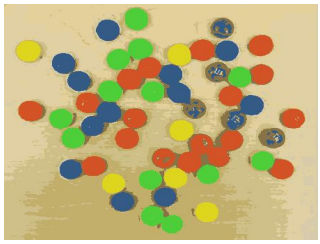
This means that two objects in the image that has similar color, but are far away from each other gets allocated to different clusters. This is not the case when using just the lab format as input.

Using just the lab format results in clustering just based on colors. Objects in the image that have similar colors gets clustered together regardless of its position in the image. Given below is the output of the two k-means algorithm using mandm.png as input.


lab format, k = 2


lab format, k = 5


lab format, k = 8


lab format, k = 10


labxy format, k = 2


labxy format, k = 5


labxy format, k = 8


labxy format, k = 10

2.4: K-means ++.

K-means++ is an improvement to the vanilla k-means algorithm. It is a more robust clustering algorithm as it eliminates the chances of selecting centroids that are close to each other. Improper selection of initial centroids can affect the output, as well as the convergence speed. Instead of randomly selecting points from the dataset to be the initial centroids, we select data points based on the probability of the points derived from its distance to the centroids. The steps involved in k-means++ are summarized below.

1: Randomly select one data point to be the initial centroid. The rest k-1 centroids are selected based on this initial centroid.

2: Calculate the distance of all the data points from the initial centroid.

3: Convert all the distances calculated above to probabilities by dividing each distance by the sum of all distances.

4: Compute the cumulative probabilities.

5: Select a random number **r** which has a range of 0-1.

6: Select a point to be the new centroid such that the cumulative probability is just greater than **r**.

7: Calculate the distance of all data points to the selected centroids.

8: Select the minimum distance to the centroids.

9: Repeat steps 4-8 till the desired number of centroids are selected.
Given below is the code implementation of k-means++ initialization algorithm.

```
0:    cent[0] = random.choice(random.choice(ax))#randomly select first initial
                centroid
1:    # Applying kmeans ++ centroid initialization algorithm
2:    for i in range(1,k):
3:        d = []
4:        for j in range(ax.shape[0]):
5:            for l in range(ax.shape[1]):
6:                dist = []
7:                for m in range(i):
8:                    dist.append(np.sum((cent[m]-ax[j,l])**2)) # Find distance of
                        pixel to already calculated centroids
9:                d.append(np.min(dist))# Find the minimum distance
10:       prob = d/np.sum(d) # Convert all distance to probabilities
11:       cp = np.cumsum(prob) # Find cumilative probabilities
12:       cp = cp.reshape(ax.shape[0],ax.shape[1])
13:       r = random.random() # Select a random number between 0 and 1
14:       # Select a point whose probability is just greater than r and assign it
            as new centroid
15:       index_x = np.where(cp>r)[0][0]
16:       index_y = np.where(cp>r)[1][0]
17:       cent[i] = ax[index_x,index_y]
```

The first centroid is selected randomly[Line 0] and then we loop through k-1 times[Line 2]. We then loop through all the pixels in the input image[Line 4, Line 5]. For each pixel, we calculate the distance to all the available centroids[Line 8] and select the minimum distance[Line 9]. We then convert the distance to probabilities[Line 10] and then to cumulative probabilities[Line 11]. Line 12 converts the 1d array containing cumulative probabilities to a 2d array with dimension of the input image to facilitate indexing. The value of r is selected[Line 13] and is compared to the cumulative probabilities. The index of the element whose value is just greater than r is selected[Line 15 and Line 16]. The element corresponding to this index in the input image is then selected to be the new centroid[Line 17].

The k-means ++ algorithm is slower in performance when compared to the simple implementation as this algorithm involves more computation for the initial centroid selection. This also means that since the initial centroids are selected more optimally, the convergence speed will be less than the simple implementation.

2.5: Comparing output of simple k-means vs k-means ++.

1: Input: peppers.png

K-means, k = 2

K-means, k = 5

K-means, k = 8

K-means, k = 10

K-means++, k = 2

K-means++, k = 5

K-means++, k = 8

K-means++, k = 10

2: Input: mandm.png

K-means, k = 2

K-means, k = 5

K-means, k = 8

K-means, k = 10

K-means++, k = 2

K-means++, k = 5

K-means++, k = 8

K-means++, k = 10

Task 3: Face recognition using Eigenface

The term eigenface represents a set of eigenvectors which is derived from the covariance matrix of a probability distribution. These eigenfaces are generated from a set of training images by applying principle component analysis. A face can then be considered to be some combination of eigenfaces, which can allow one to check which faces in the training set best matches the testing image.

3.1: Importance of alignment: Each eigenface encodes information about the human face. If faces are not aligned, the information encoded is most likely to be the difference in illumination in training images due to the varying position of the face in each image. The eigenface generated from such training set has very little encoded information about the actual features of a face, instead it contains other irrelevant information encoded, which leads to poor recognition performance.

3.2.1: Read all input images as a 2d vector map.

```
1:    image_list = [] # Array to store all training images
2:        # Read all train images
3:     for filename in glob.glob('Yale-FaceA/trainingset/*.png'):
4:         im=Image.open(filename)
5:         image_list.append(im)

6:          # Convert images to single 2-dimensional array with 195*231(resolution
           of images) rows and     144 columns(num of training images)
7:          im_matrix = []
8:          for image in image_list:
9:              im = (np.array(image)).astype(float)
10:       im_matrix.append(im.flatten('F'))
11:    im_matrix = np.stack(im_matrix).T
```

Images from the training set are first read and saved into a list[Line 1 – Line 5]. These images are then converted into numpy array representation[Line 9] and then flattened into a 1d array[Line 10]. Each 1d array is the image vector in high dimensional space. The result is a 2d matrix consisting of all the images in the training set represented as vectors. This matrix is then used for further calculation.

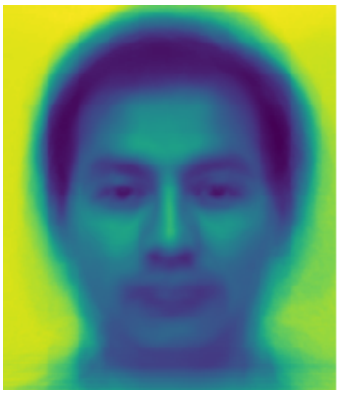
3.2.2: Performing PCA.

```
1:    mean = np.mean(im_matrix,1) # Compute mean image
2:    r_mean = mean.reshape(195*231,1)
3:    normalised_face = im_matrix - r_mean  # Normalize the training images
4:    cov = (normalised_face.T @ normalised_face) / 144 # Compute covariance to
      minimize computation
5:    e_val, e_vec = np.linalg.eig(cov)# Compute eigen vectors and eigen values
```

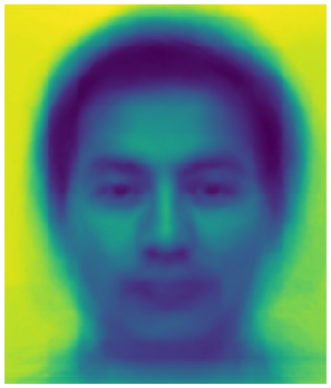
The above code snippet calculates the eigenvalues and eigenvectors from the face vector matrix generated in the previous step. The first step is to calculate the mean face from the face vector matrix[Line 1]. This mean face is then used to normalize all the vectors in the image vector matrix[Line 3]. Using the normalized face vectors, we calculate the covariance[Line 4]. Here the covariance matrix is calculated using **Transpose(Normalized_vector) dot (Normalized_vector)** so that the resulting covariance matrix is of size 135 * 135. This results in better performance. On the other hand if the

covariance matrix is calculated using **(Normalized_vector) dot Transpose(Normalized_vector)**, then the resulting covariance matrix will be of size 45045 * 45045. The eigenvalues and eigenvectors are the calculated using the covariance matrix[Line 5].

The mean face generated from the original training dataset(135 images) and the new training dataset(144 images) are given below.



Mean face with 135 faces



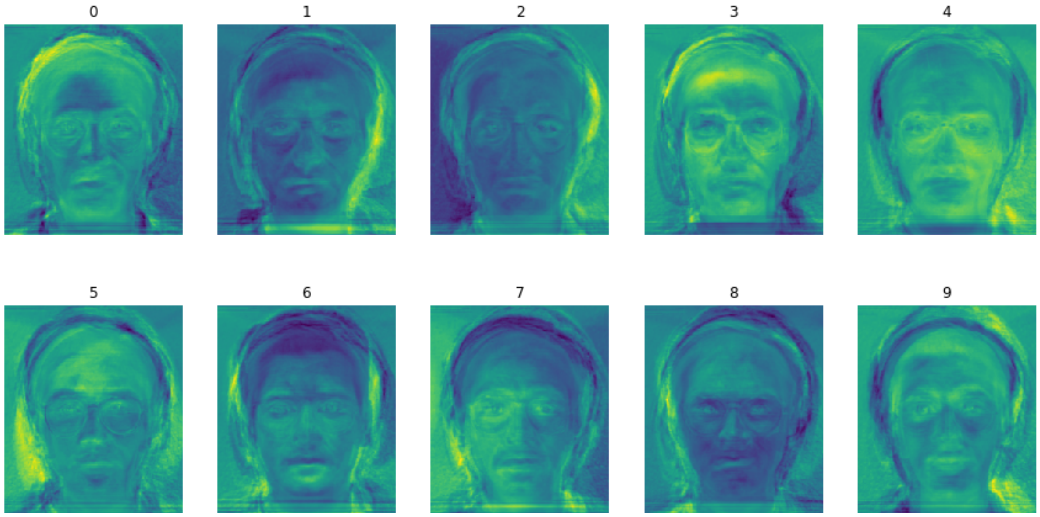
Mean face with 144 faces

3.2.3 Top k eigenfaces.

The code given below extracts the top k (k = 10) eigenvectors[Line 1] from the eigenvector matrix and then converts them into eigenfaces[Line 2]. These eigenfaces are then used to compute the weight matrix.

```
1:    top_k = e_vec[0:10,:]
2:    eigen_face = top_k.dot(normalised_face.T)
3:    weights = (normalised_face.T).dot(eigen_face.T)
```

Given below is the top 10 eigenfaces visualized.

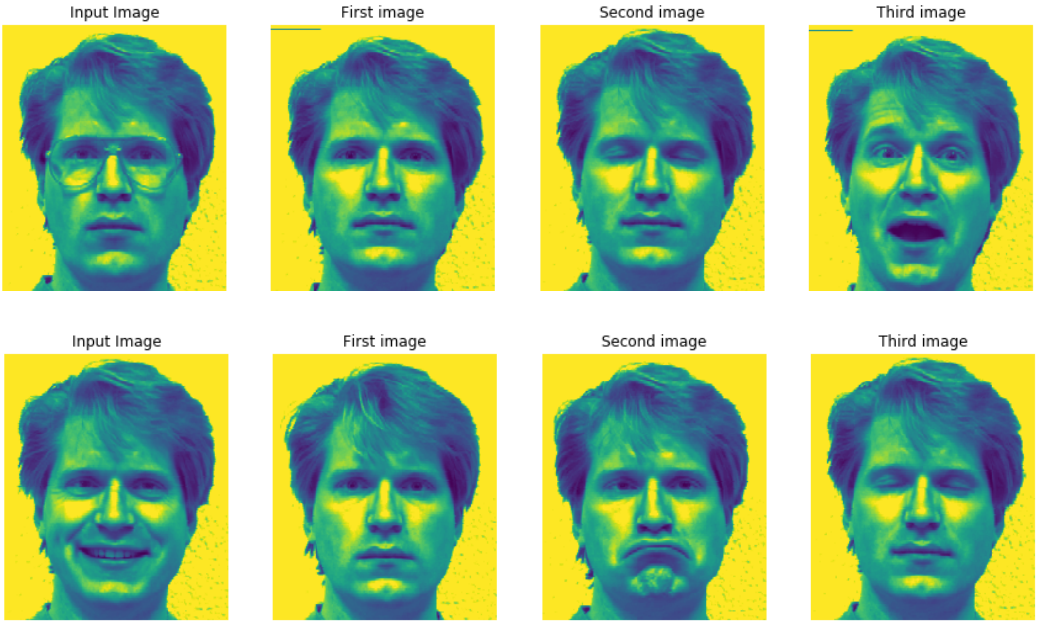


3.2.4: Test set output.

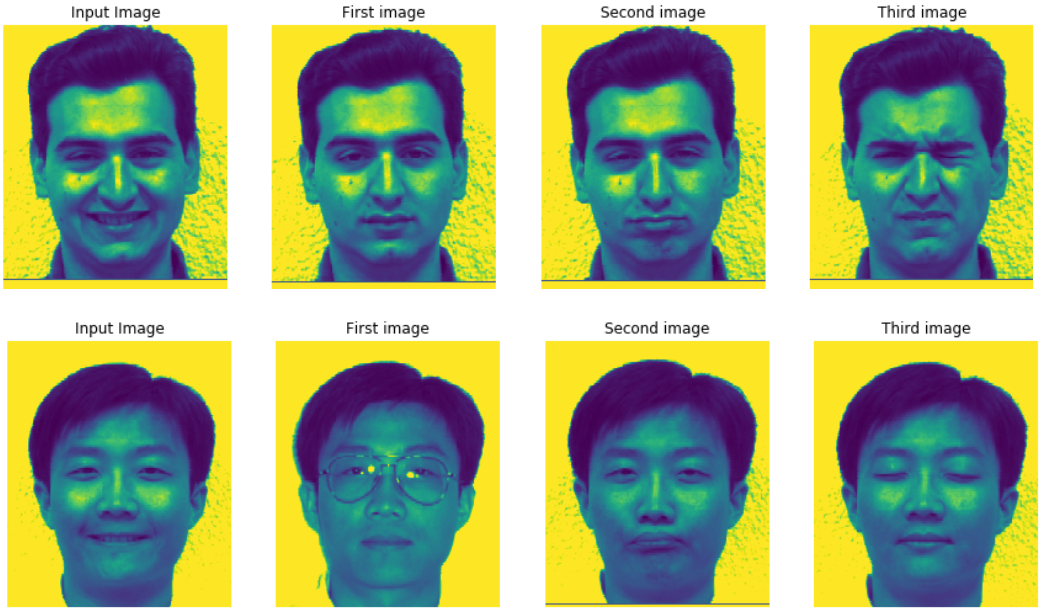
```
1:      # Normalise the test image
2:      test_img = (test_img_input.flatten('F').T).reshape(195*231,1)
3:      test_norm = test_img - r_mean
4:      # Compute weight of test image
5:      test_w = (test_norm.T).dot(eigen_face.T)
6:      # Compute distance between test weight and training weights
7:      dist = np.linalg.norm(test_w-weights,axis=1)
8:      # Find index of top 3 training images based on distance
9:      top_k = nsmallest(3,dist)
10:     index = []
11:     counter = 0
12:     for i in dist:
13:         for j in top_k:
14:             if i == j:
15:                 index.append(counter)
16:         counter += 1
```

The above code snippet uses the input test image and applies the nearest neighbor algorithm to compute the top 3 faces that are the most similar to the test image. We start by normalizing the input test image. The test image is first flattened to a 1d vector representation[Line 2] and is normalized by subtracting the mean from the vector[Line 3]. Similar to computing the weight matrix during the training step, the weight matrix of the test image is calculated[Line 5]. We then compute the distance between the single weight vector of the test image and the weight vectors of the eigenface[Line 7]. The index value of faces corresponding to the nearest three weights vectors computed during training is extracted[Line 9 – Line 16]. These index correspond to the face vector in the original 2d vector representation of the face images. Each index can then be used to extract a single face vector of size 45045. This vector is then reshaped to 195*231 (Image resolution) so as to display the matching faces.

Given below are the outputs generated for the test set images.




Input Image
First image
Second image
Third image
Input Image
First image
Second image
Third image
Input Image
First image
Second image
Third image
Input Image
First image
Second image
Third image
Input Image
First image
Second image
Third image
Input Image
First image
Second image
Third image



3.2.5: Top 3 images similar to my face image.

3.2.6: Output when trained on 9 extra face image of myself.