Clab-2 Report

ENGN6528

Name: Fengdan Cui

UID Master: U6589143

09/05/2020

CONTENT

Task-1: Harris Corner Detector

1.1 Rewrite codes and design appropriate function signature

The change of the first derivative of the corner point in each direction is the largest and Harris' mathematical formula is:

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

Calculate the partial derivative according to the Taylor series, and finally get a Harris matrix formula:

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

And Harris corner response value is:

$$R = detM - k(traceM)^2$$

$$detM = \lambda_1 \lambda_2$$

$$traceM = \lambda_1 + \lambda_2$$

Therefore, the Harris corner detector algorithm can be summarized:

1. Calculate the gradient I_x, I_y of the image along x, y direction.
2. According to the result of the first step, apply Gaussian blur and calculate ${I_x}^2$, ${I_y}^2$ and $I_x * I_y$.
3. Define the Harris matrix M of each pixel and calculate the two eigenvalues of the matrix.
4. Calculate the R value of each pixel.
5. Use 3x3 window to perform non-maximum suppression.
6. Based on corner detection results, mark the key points on the original image.

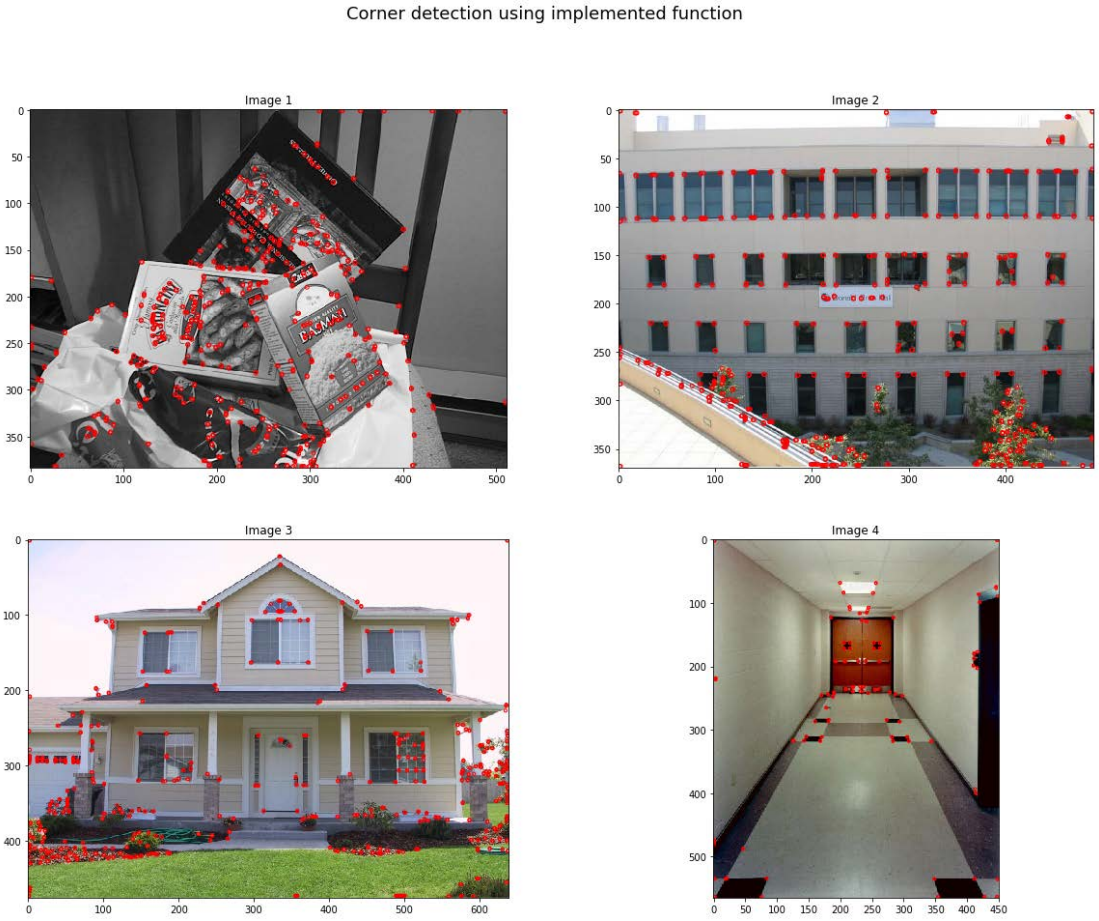
The implemented function can be found in ***HarrisCorner.py*** and the codes have also been added to appendices at the end of this report. There are two main functions which aim to detect corners and perform non-maximum suppression respectively.

1.2 Test function on the provided four test images

The test results are obtained by calling ***corner_detection(img_files)***, where the input is a list contains of image files. Inside the function, I called my implemented functions ***corners = non_max_sup(harris_corneress(img_gray))*** to process the images and return a Nx2 list that contains all coordinates of detected corners. Then I marked the original image with a red hollow circle and display final images in python console. The

visualized results can be seen as following:

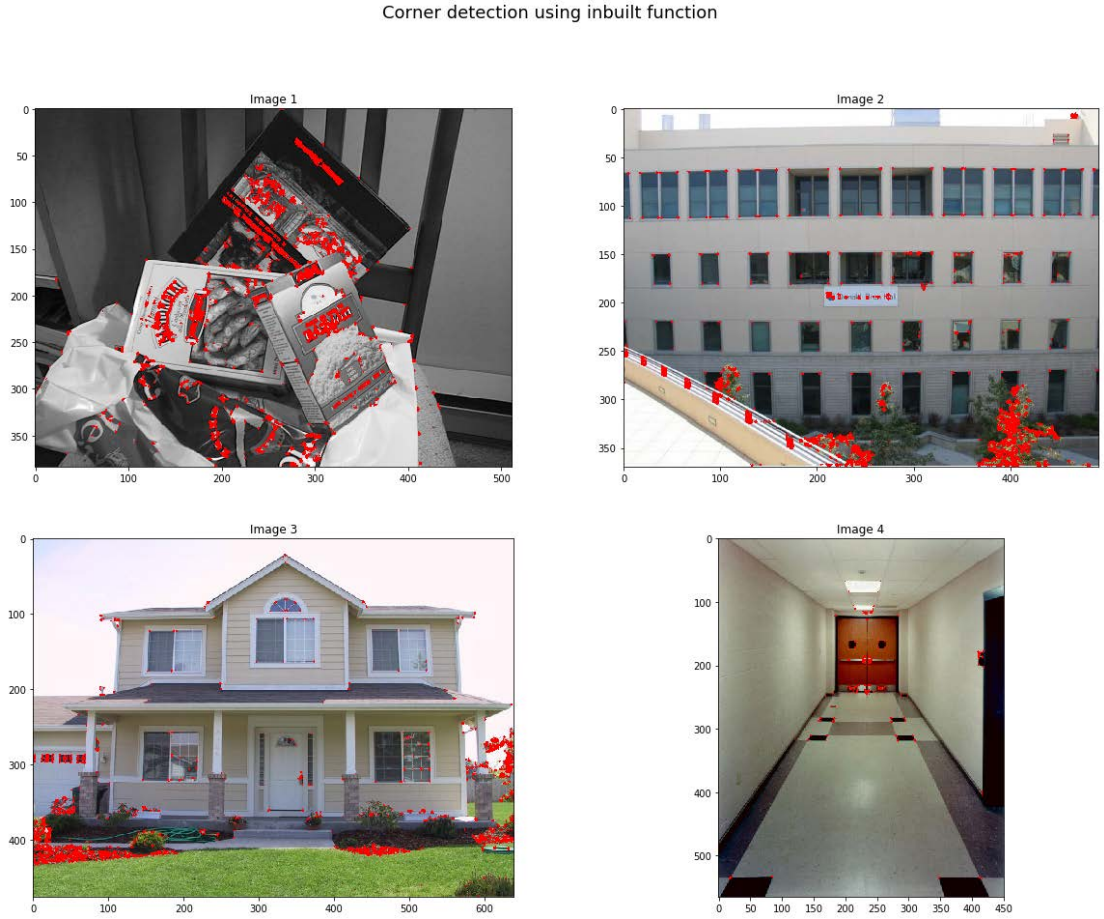
Graph 1 Perform implemented function for corner detection



1.3 Compare results with built-in function *cv2.cornerHarris()*

Applying the inbuilt function ***cv2.cornerHarris(img_gray, 3, 3, 0.04)*** to process same images and get four more marked images, which have been shown below. It can be seen in the two graphs, the detection results are almost the same.

Graph 2 Perform built-in function for corner detection



1.4 Factors that affect the performance of Harris corner detection

Although Harris algorithm performs well in detecting corners of a specific image, it is not invariant to scale change, which means some corners may not be detected when the image is enlarged. I cropped an image and enlarged it 50 times, then called ***cv2.cornerHarris()*** to the new image to detect corners. The comparison result can be seen in the graph below, some corners that can be detected in the original image cannot be detected in the new image. It is obvious that the change of image scale has importance influence on Harris corner detection.

Graph 3 Effect of different image sizes on corner detection

In addition, the choice of threshold can also influence the corner detection result. Regardless of the method I implement or the built-in method, only those pixels which are larger than the optimal threshold will be marked and such threshold is depend on the image. Therefore, it is necessary to determine suitable threshold.

Finally, noise is also a factor that has effect on Harris corner detection. In the detecting process, a critical step is to calculate the corner response R for each pixel based on the gradients along both two directions, so that the noise interference will cause inaccurate calculation results. Using the right method to reduce noise and smooth the image is important to decrease the effect on corner detection result.

Task-2: K-Means Clustering and Color Image Segmentation

2.1 Implement K-means function *my_kmeans()*

The principle of K-means algorithm is to divided a given sample set into K clusters according to the distances between the samples. The points within the cluster are connected as closely as possible, and the distance between the clusters is as large as possible. The process of my implemented function can be summarized as follows:

1. Randomly initialize k center points using ***rands = random.sample(range(0, r), k)*** and ***centers = samples[rands, :]***.
2. According to the distance of each sample to the center points, assign them to the center point closest to it, forming k clusters. The clusters are stored in a dictionary where the keys are 0, 1, 2, …, meaning the index of centers.
3. Recalculate the centers of each cluster based on the previous classification result and update the centers array. Store the label of each sample to a list, which is like 0, 1, 2, …, representing the class that they belong to.

4. Repeat step 2 and 3 until the number of iterations is reached.

The codes of function ***my_kmeans()*** can be found in ***ImageSegmentation.py*** and also in appendices section of this report.

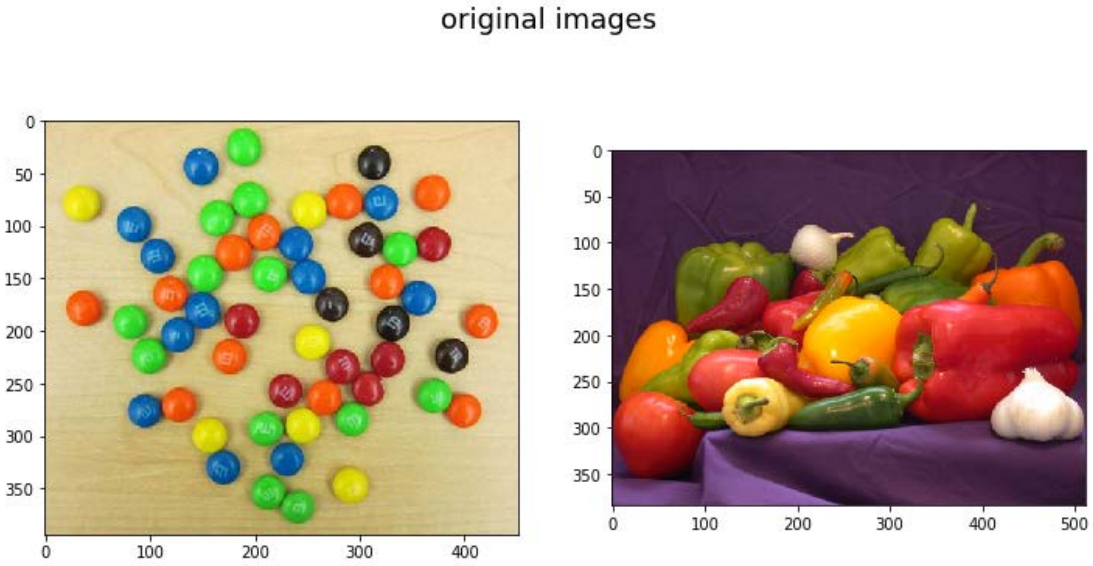
2.2 Apply implemented K-means function to color image segmentation

The process for applying the function to image segmentation is:

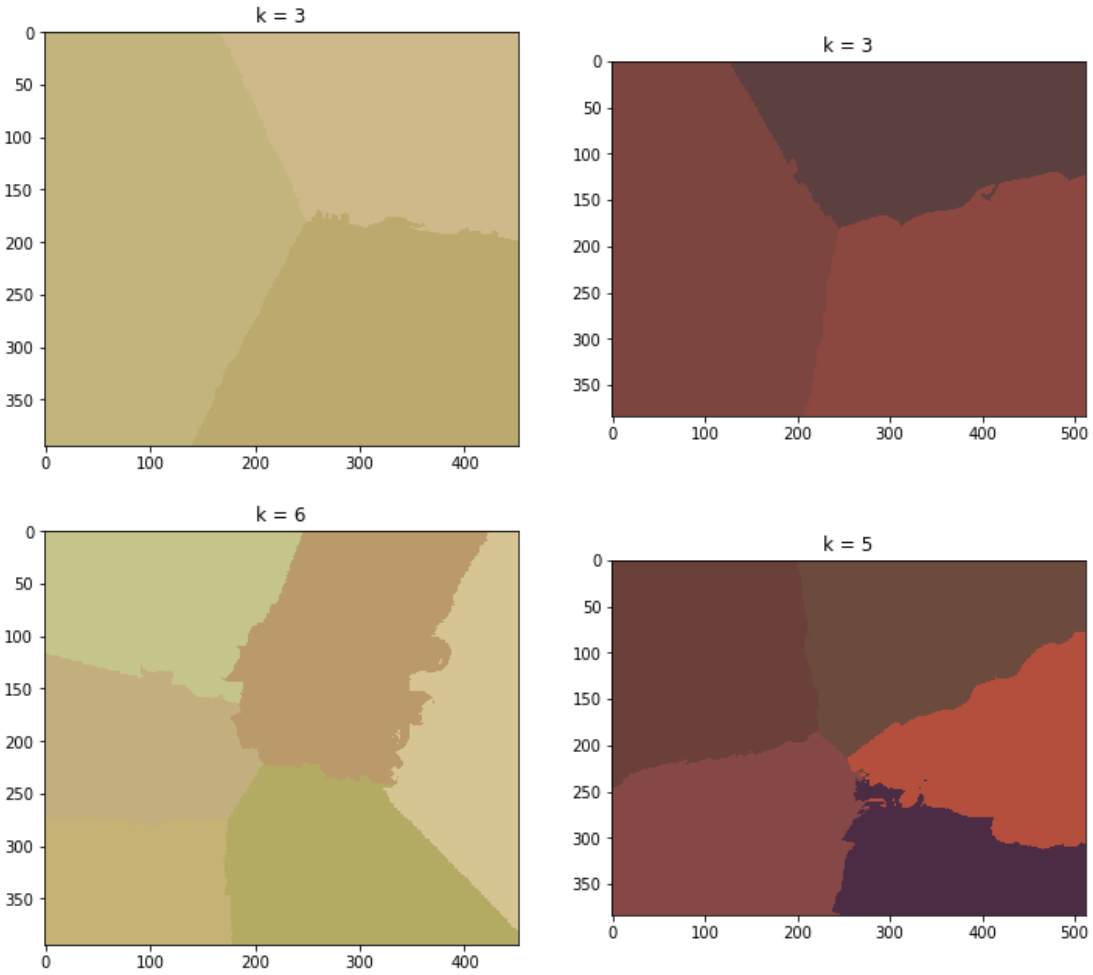
1. Encode the pixels of image: using ***cv2.cvtColor(image, cv2.COLOR_RGB2LAB)*** to transfer color space from RGB to LAB, then iterating pixels and assigning each pixel with a 5-D vector. Returning an array with the shape of (h x w) x 5 that contains all encoded pixels.
2. Call implemented function ***my_kmeans()*** to get the labels of all pixels and centers of clusters.
3. Assign pixel values for each pixel and reshape the array to the shape of image using ***out = center[labels.flatten()].reshape((img.shape))***. Then the assigned ***out*** is the final image matrix and display for segmentation visualization.

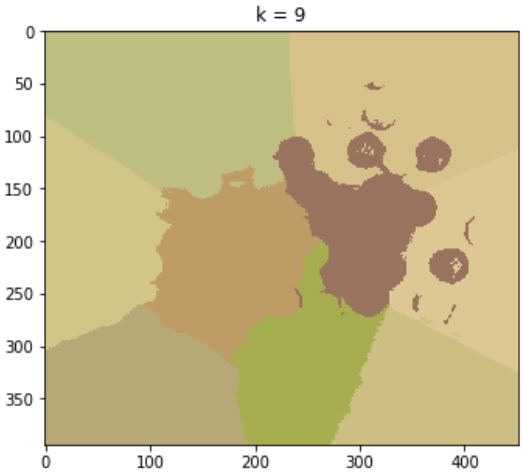
The codes can be found in ***ImageSegmentation.py***, where there are three functions used for this task, ***image_encode(image, coordinates = True)*** to encode the input image, ***my_kmeans(samples, k, init = 'random', max_iter = 10)*** to find clusters and return labels and centers, and ***display_k(images, ks, inits, coordinates = True)*** to display final segmented result. The different segmentation results generated by using different number of cluster and whether to include coordinates are shown in the graphs below. It is clear when the number of clusters is close to the number of color types included in the image, the image segmentation effect is better. If there are coordinate values included in the data after encoding images such like Graph 5, the results show all different color clusters. If there isn’t coordinate values in encoded data such like Graph 6, the results illustrate the image segmentation according to different clusters.

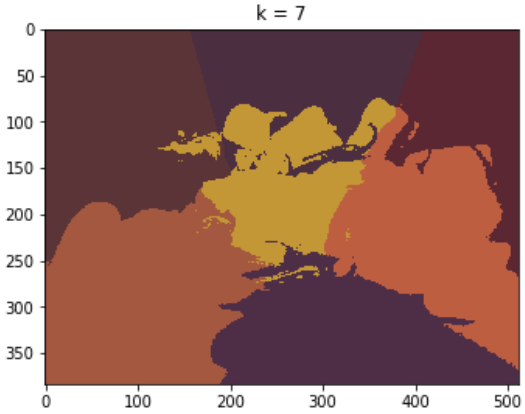
Graph 4 Original images for segmentation



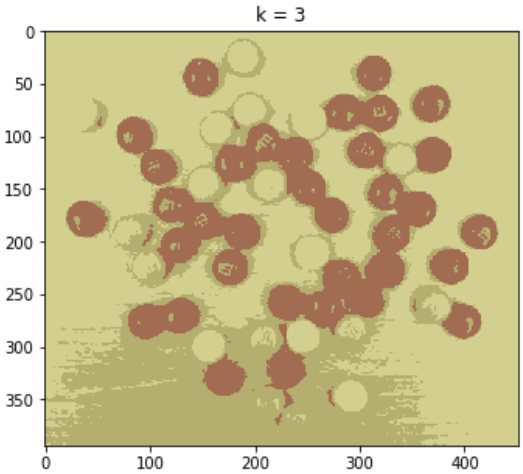
Graph 5 Segmentation results by k-means with coordinates with different k

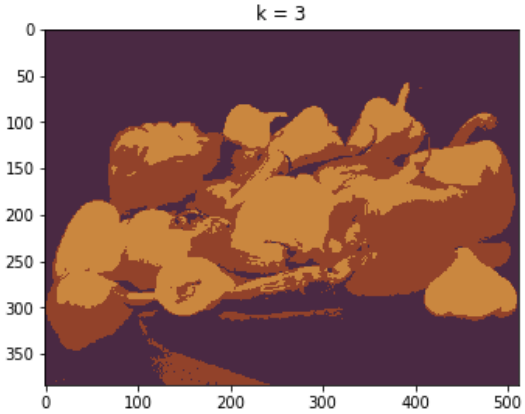


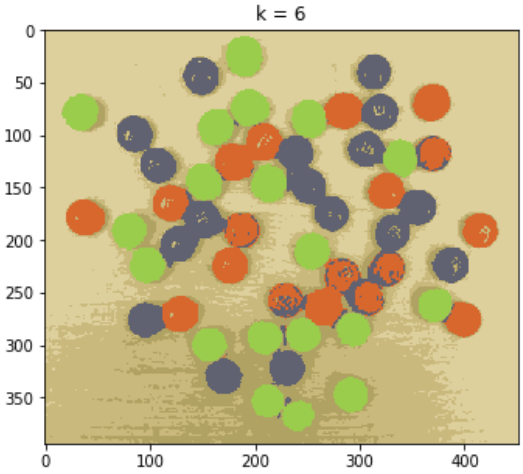


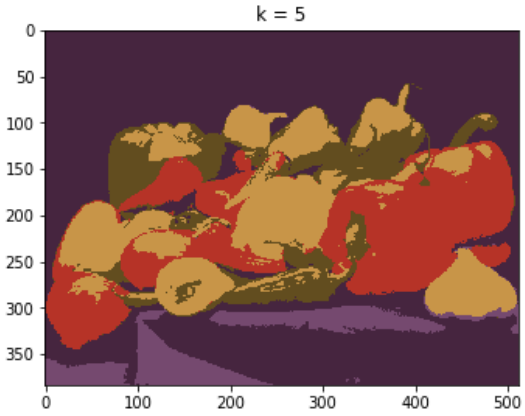


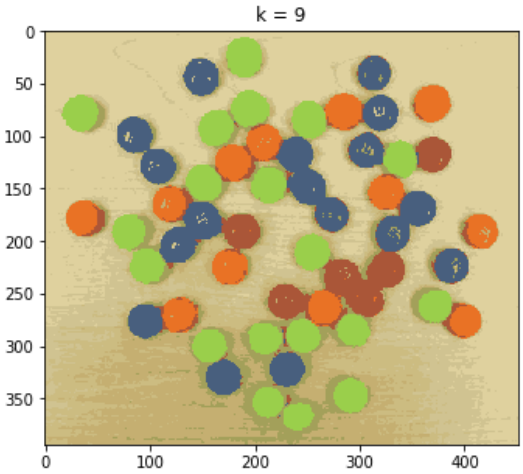
Graph 6 Segmentation results by k-means without coordinates with different k

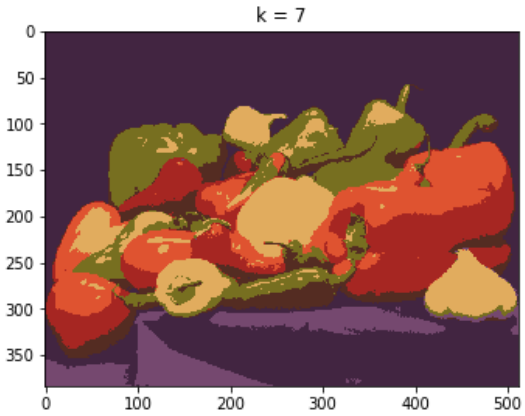












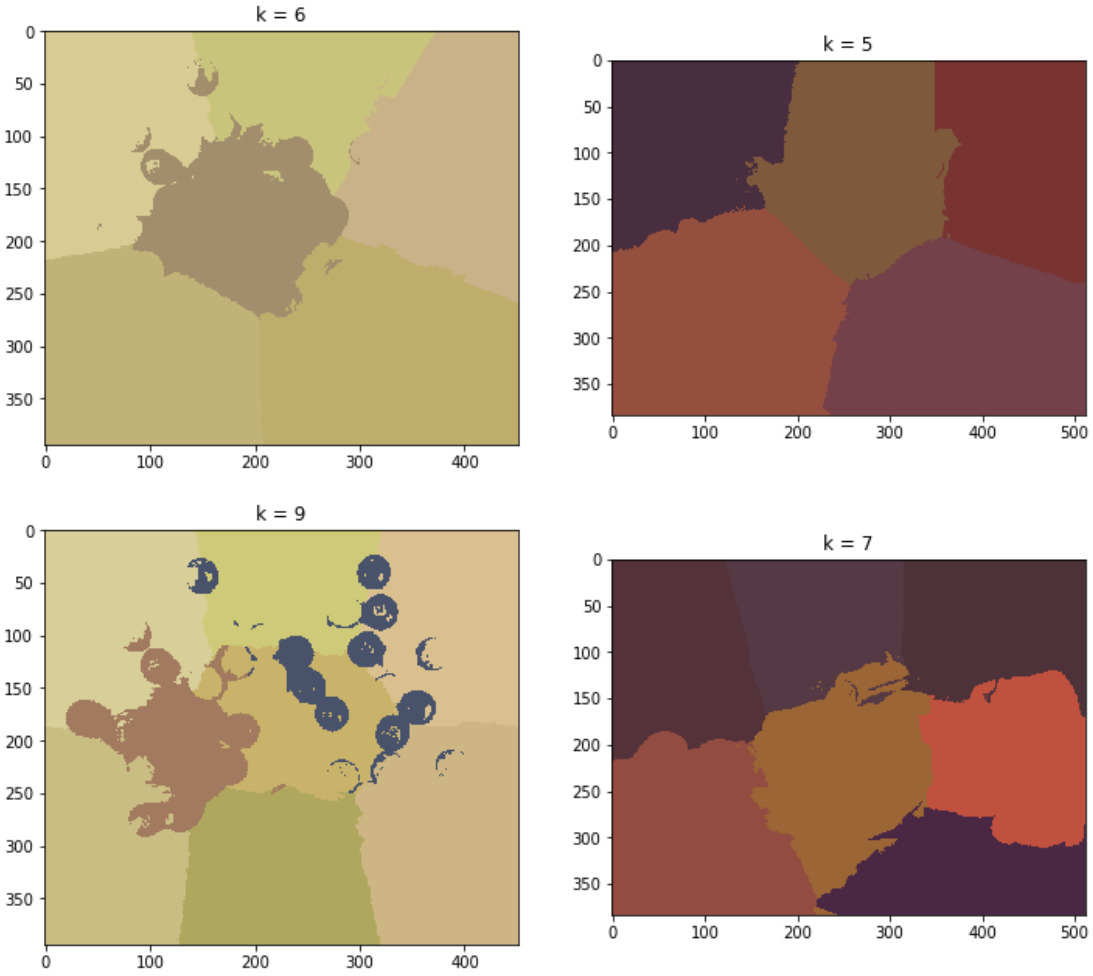
2.3 Summarize, implement K-means++ and compare the image segmentation performance

The basic principle of K-Means ++ algorithm when initializing cluster centers is to make the distance between cluster centers as far as possible, which improves the shortcomings of k-means due to the random selection of initial centers, which leads to unstable clustering accuracy. The process is as follows:

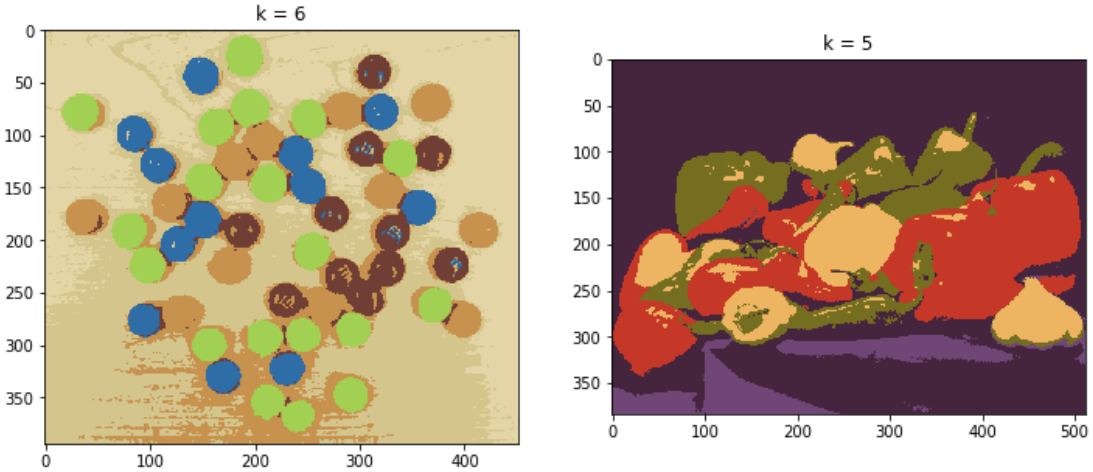
1. Randomly select a sample in the data set as the first initial clustering center.
2. Calculate the distance between each sample in the data set and the cluster center that has been initialized, and select the shortest distance.
3. Assuming that the distance from each point to its nearest center point is ***dist***, with a probability proportional to ***dist***, select a point as a new center and add it to the centers list.
4. Repeat step 2 and 3 until k centers are selected.
5. Use K-Means algorithm to calculate the final clustering results for k centers.

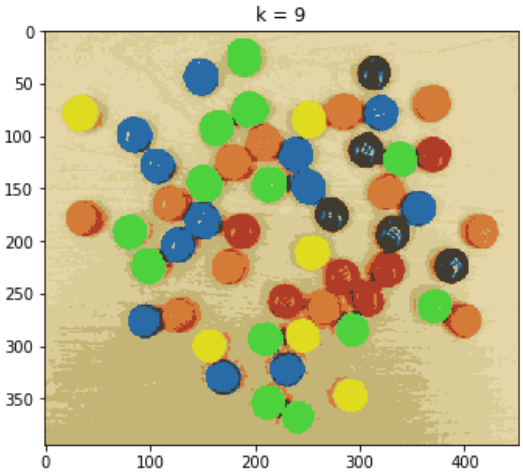
The implemented function can be found in ***ImageSegmentation.py***, where the function ***kmeans_plus_init(samples, k)*** is used to initialized centers by performing k-means++ algorithm. The codes can also be found in the appendices section in this report. Then performing ***my_kmeans(samples, k, init = 'random', max_iter = 10)*** and ***display_k(images, ks, inits, coordinates = True)*** to segment both two images. By using different numbers of clusters, the segmentation results can be seen in the follow graphs where the original images are encoded without coordinates. It can be seen in such graphs, the results using k-means++ is better than simple k-means. Especially when the number of clusters is 9, the result of the image segmented using k-means++ is more obvious.

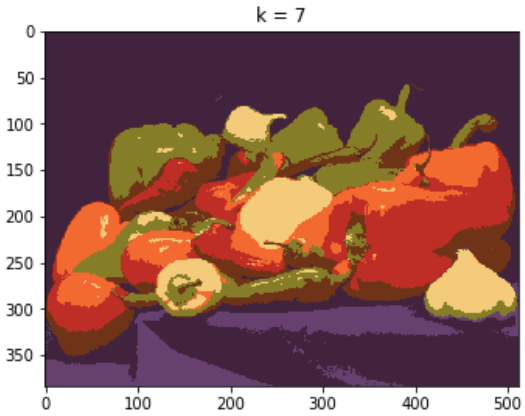
Graph 7 Segmentation results by k-means++ with coordinates with different k



Graph 8 Segmentation results by k-means++ without coordinates with different k







Task-3: Face Recognition using Eigenface

3.1 Explain why alignment is necessary for Eigen-face

Alignment is necessary for Eigen-face recognition algorithm because it is not robust to misalignment and background variation. In details, the first step for getting Eigen-face is to calculate the mean face of a large number of training faces, if the alignment or background are different of some images, for example, if there are two images that are upside down, then the average image obtained from such images is meaningless for performing PCA. Moreover, the purpose of PCA is to extract the main feature components from the original data by linear transformation. Aligned images are helpful for locating feature points more accurately, such as the position of eyes. Therefore, the images used for Eigen-face must be the same size and must be aligned.

3.2 Perform PCA on the data matrix

The process of performing PCA on face images can be summarized as follows:

1. Assume the acquired face vector set S is a matrix of the shape of ***mxn*** where ***m*** is the number of images and ***n*** is the number of all pixels of an image.

   $$S = \{\Gamma_1, \Gamma_2, \Gamma_3, \dots, \Gamma_m\}$$

   Calculate the average image:

   $$\Psi = \frac{1}{m}\sum_{i=1}^{m} \Gamma_i$$

   The result is an N-dimensional vector, if we restore it back to the form of an image, we can get the mean face, which is displayed below.
2. Calculate the difference between each image and the average image:

   $$\Phi_i = \Gamma_i - \Psi$$

3. Calculate the covariance matrix:

$$C = \frac{1}{m}\sum_{i=1}^{m} \Phi_i {\Phi_i}^T$$

Assume $A = \{\Phi_1, \Phi_2, \Phi_3, \dots, \Phi_m\}$ and $A \in R^{nxm}$, then $C = \frac{1}{m} AA^T$.

4. Calculate the eigenvalues and eigenvectors of C.

The covariance matrix is a high-dimensional matrix because $C \in R^{nxn}$, direct decomposition is time-consuming so that the last step can be done in an easy way. As for the eigen decomposition of $A^T A$:

$$A^T A v_i = \lambda_i v_i$$

$$AA^T A v_i = \lambda_i A v_i$$

$$C A v_i = \lambda_i A v_i$$

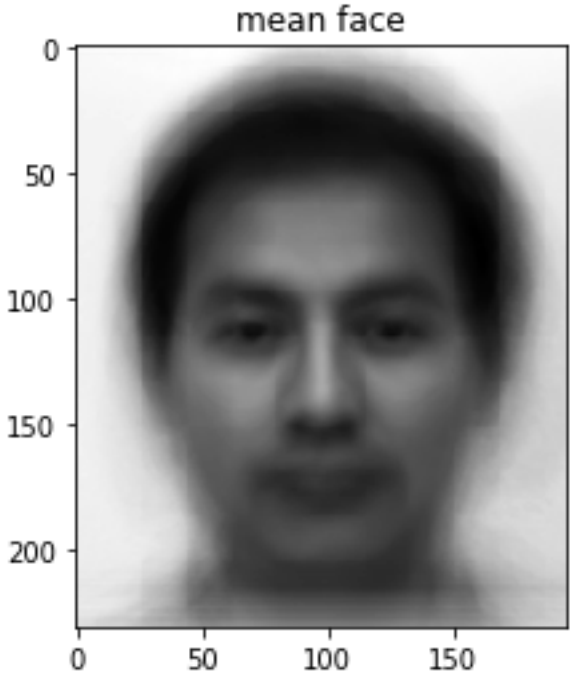
$$C u_i = \lambda_i u_i \quad u_i = A v_i$$

Through a series of derivations, we can see the eigenvalues of AA^T and $A^T A$ are same and the eigenvectors are $u_i = A v_i$. $AA^T \in R^{nxn}$, where there can be at most n eigenvalues and eigenvectors but $A^T A \in R^{mxm}$, where there can be at most m eigenvalues and eigenvectors. The m eigenvalues in the $A^T A$ correspond to the m largest eigenvalues of AA^T. Therefore, when n is much greater than m, we can fast decompose the $A^T A$.

Once the eigenvectors have been obtained, we can compute the eigenvectors of covariance matrix as $u_i = A v_i$.

The codes for implement PCA can be found in ***FaceRegnition.py*** and also in the appendices of this report. The returned values are the average of training images, the eigenvectors of covariance matrix and differences which are mentioned above.

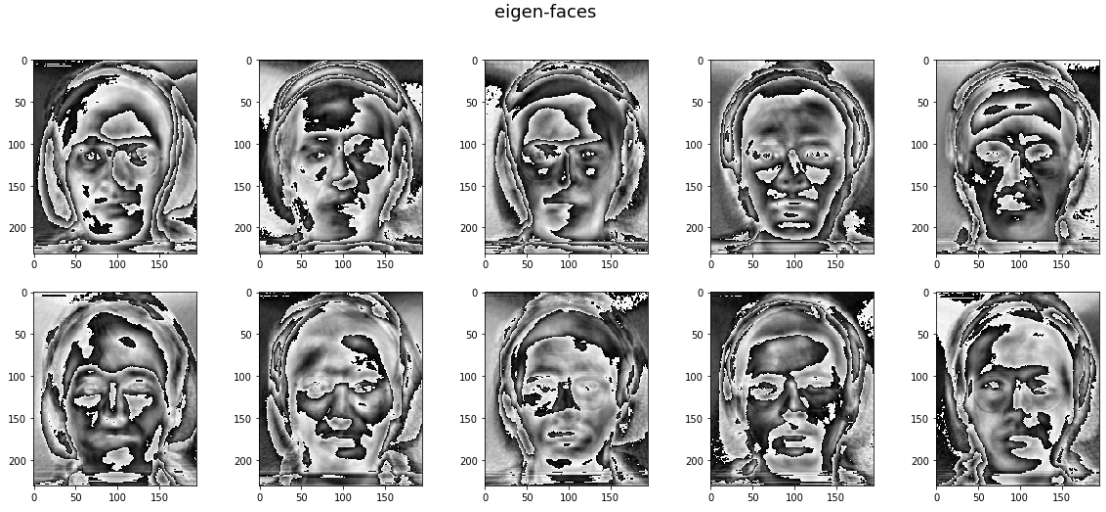
Graph 9 The mean face of training images



3.3 Visualize the top-k eigen-faces

Inside the PCA function, using ***eig_sort_index = np.argsort(-eig_vals)*** to sort the decomposed eigenvalues and ***k_index = eig_sort_index[:k]*** to extract the indexes of the first k values according to the input parameters k. Then ***cov_vects = diff * eig_vects[:, k_index]*** which is the k largest eigenvectors of covariance matrix as well as eigen faces. The top-k eigen-faces with k = 10 can be seen in the graph below and the codes for implementing such functions can be found in ***FaceRegnition.py*** and also in the appendices section.

Graph 10 Top-10 eigen-faces of training images



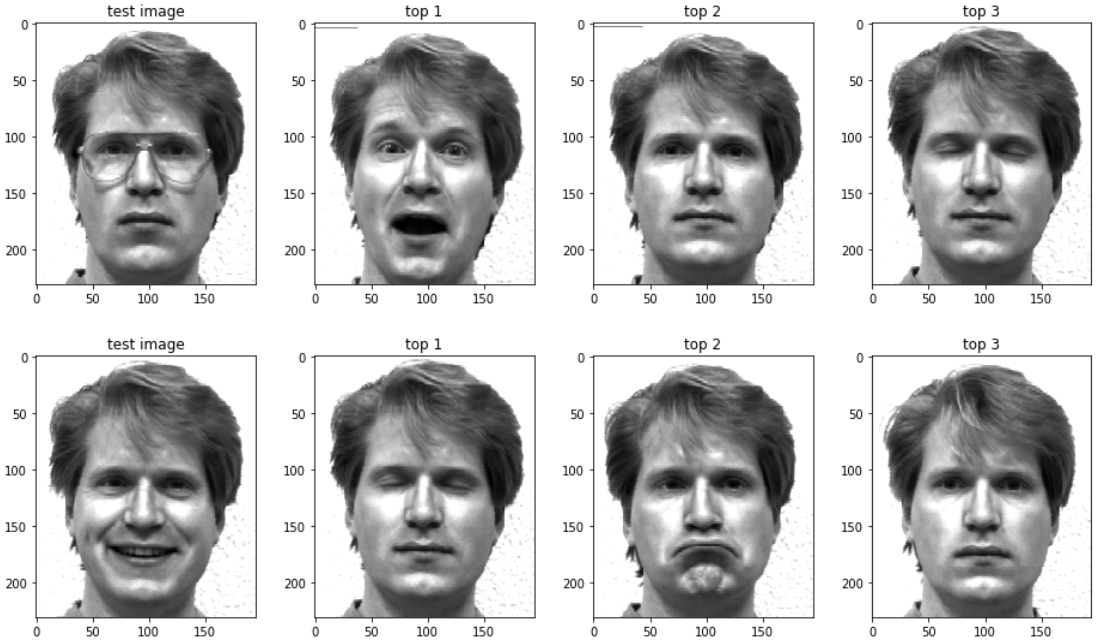
3.4 Top 3 faces in the training folder that are most similar to each test image

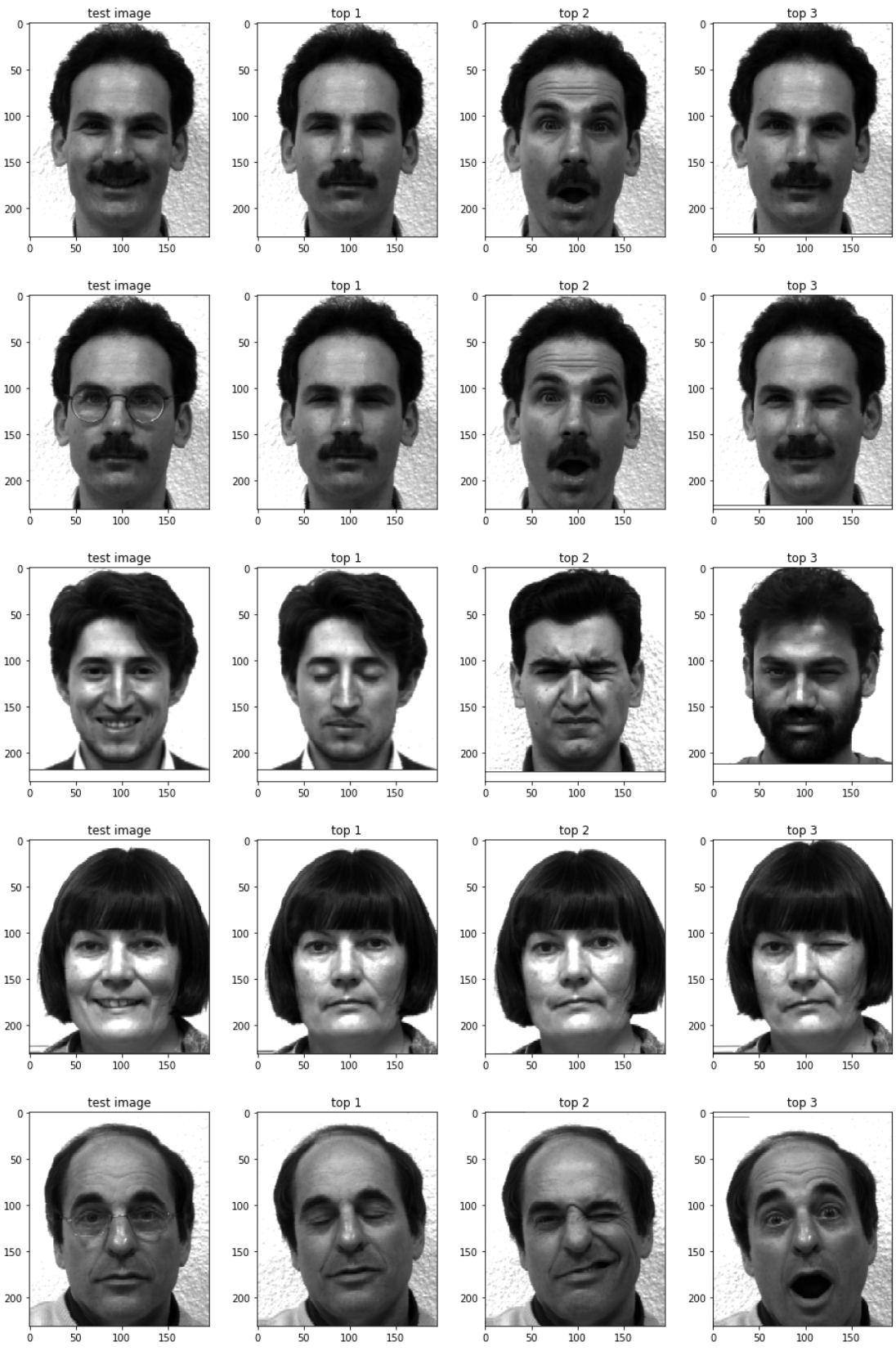
According to the returned values of PCA, including average of data, the top k eigenfaces and differences, for an unknown image, we can apply nearest-neighbor search method for detecting similar faces among training images. Specific steps are as follows:

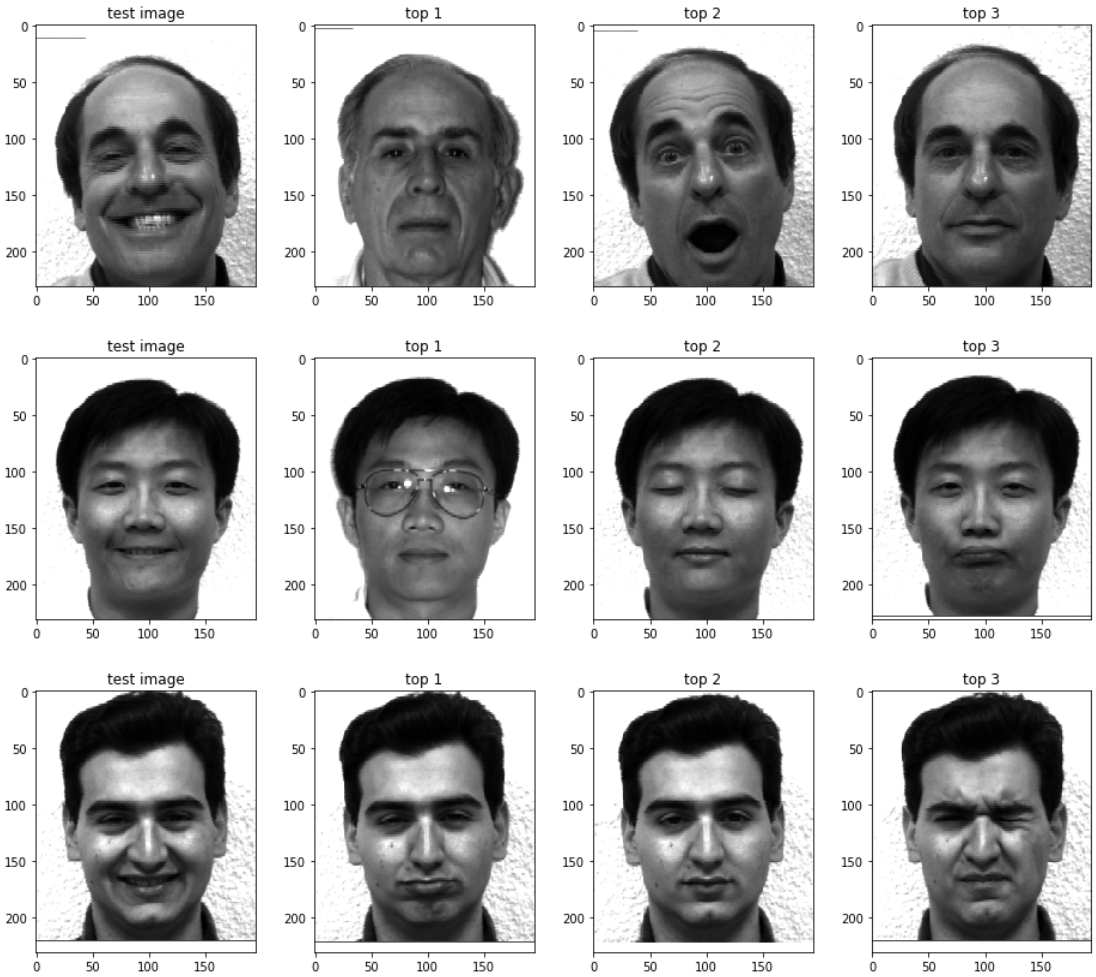
1. Calculate the difference between test image and returned mean face, $\Phi = \Gamma - \Psi$
2. Calculate the mapped vector of test image, $\widehat{\Phi} = \sum_{i=1}^{k} w_i u_i \quad w_i = u_i{}^T \Phi$
3. Calculate the mapped vector of the training images. Then compute the distances between test vector and train vectors and store them into an array.
4. Sort the array and return three face images are the most similar with the test image.

The codes for implementing face recognizer can be found in ***FaceRegnition.py*** and also in the appendices section. The results can be shown in the following graph, where the first image of each line is the original test image and the remaining images are top-3 similar faces. As the results shown, the similar faces of 90% of the test faces can be accurately detected. The reason why the eighth image is erroneously detected may be the interference of the image background and the inevitable errors in the process of feature extraction and recognition.

Graph 11 Top-3 similar faces with test set



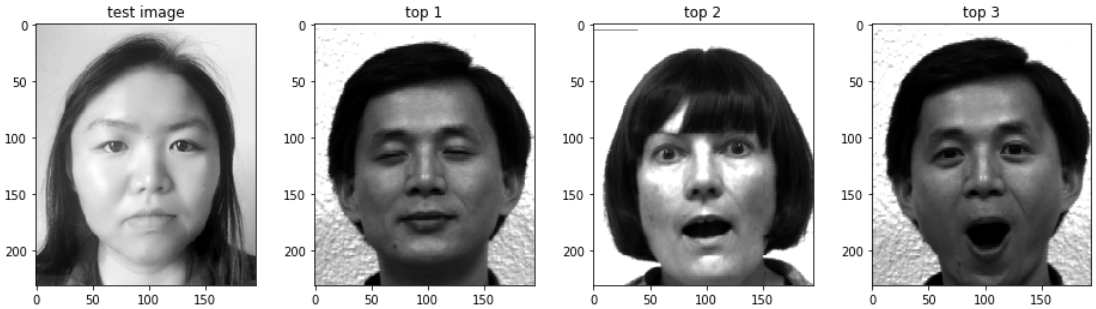

test image
top 1
top 2
top 3
test image
top 1
top 2
top 3
test image
top 1
top 2
top 3
test image
top 1
top 2
top 3
test image
top 1
top 2
top 3



3.5 Top 3 faces in the training folder that are most similar to my face

By applying the implemented face recognizer to my face, the detected three similar faces are shown in the graph below.

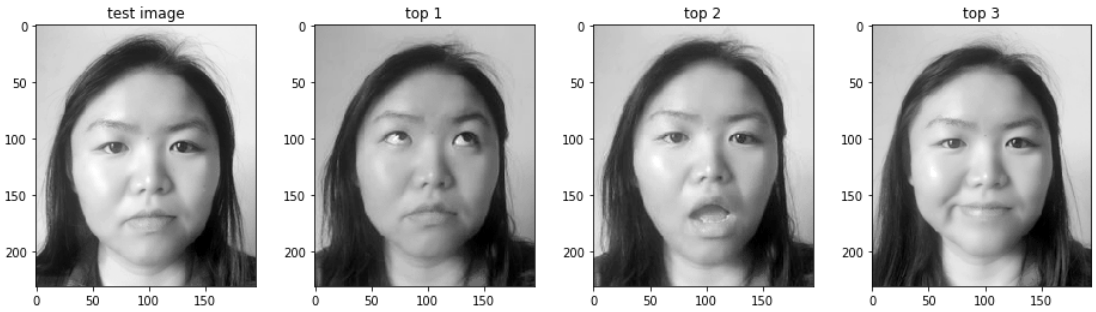
Graph 12 Top-3 similar faces with my face



3.6 Top 3 faces that are the closest to my face after pre-adding the other 9 of my face images

After adding other nine of my faces to the training set and apply the PCA and face recognize, the detected three similar faces are shown in the graph below. It is obvious the result is accurate completely.

Graph 13 Top-3 similar faces with my face after extending training images



References

https://docs.opencv.org/2.4/modules/imgproc/doc/feature_detection.html?highlight=cornerharris
https://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_ml/py_kmeans/py_kmeans_opencv/py_kmeans_opencv.html

Appendices

harris_corneress(image, sigma = 2, k = 0.04)

```
# Compute corner response R

def harris_corneress(image, sigma = 2, k = 0.04):

    h, w = image.shape

    # R to store cornerness score
    R = np.zeros((h, w))

    dx = np.array([[-1, 0, 1], [-1, 0, 1], [-1, 0, 1]])
    dy = dx.transpose()

    # calculating gradients in x and y direction
    Ix = conv2(image, dx)
    Iy = conv2(image, dy)

    # appLy Gaussian filter
    g = fspecial((max(1, np.floor(3 * sigma) * 2 + 1), max(1,
np.floor(3 * sigma) * 2 + 1)), sigma)
    Iy2 = conv2(np.power(Iy, 2), g)
    Ix2 = conv2(np.power(Ix, 2), g)
    Ixy = conv2(Ix * Iy, g)

    for i in range(h):
        for j in range(w):

            # harris matrix
            M = [[Ix2[i, j], Ixy[i, j]], [Ixy[i, j], Iy2[i, j]]]
```

```
            # singular value decomposition
            u, s, v = np.linalg.svd(M)

            # omputing lambda values
            [lmda1, lmda2] = s
            lambda_product = lmda1 * lmda2
            lambda_sum = lmda1 + lmda2

            # computing corneress scores and store in R
            R[i, j] = lambda_product - k * (lambda_sum**2)

    return R
```

non_max_sup(R, thresh = 0.01)

```
    # Find points with large corner response: R > threshold
    # Take only the points of local maxima of R

    def non_max_sup(R, thresh = 0.01):

        # get the height and width of image for interation later,
the shape of R is the same as image
        h, w = R.shape

        #  considering  only  those  points  that  are  larger  than
threshold
        thresh *= R.max()

        # an Nx2 matrix for storing the final x and y coordinates
        out = list()

        # 3 x 3 window, start from 1 to len - 1 for local maxima of
each window
        for r in range(1, h - 1):
            for c in range(1, w - 1):

                # If the point is greater than the largest point among
the surrounding points,
                # then it is greater than all the points around
                Rmax = 0

                # iterating all surrounding points 3x3
```

```
            for i in [r - 1, r + 1]:
                for j in [c - 1, c + 1]:

                    # comparing and replacing max
                    if R[i, j] > Rmax:
                        Rmax = R[i, j]

            # pick out points of local maxima and larger than
threshold
            if (R[r, c] > thresh) & (R[r, c] >= Rmax):

                # the index in matrix is the reverse order of the
coordinates of image,
                # so add [c, r] not [r, c] to list
                out.append([c, r])

    return out
```

my_kmeans(samples, k, init = 'random', max_iter = 10)

```
def my_kmeans(samples, k, init = 'random', max_iter = 10):

    # number of training: r = samples.shape[0]
    # number of features: c = samples.shape[1]
    r, c = samples.shape

    # centers initialization by different methods
    if init == 'random':

        rands = random.sample(range(0, r), k)   # generate different
randoms
        centers = samples[rands, :]            # samples.shape[1] x
k

    if init == 'plus':
        centers = kmeans_plus_init(samples, k)

    # store clusters {0: [points], 1: [points], ..., k: [points]}
    clusters = {i: [] for i in range(k)}
    n = 0

    while n < max_iter:
        labels = []
```

```
        # generate clusters
        for i, s in enumerate(samples):
            distance = []
            for ce in centers:
                distance.append(dist_eclu(s, ce))

            #  find  out  the  minimal  distance  and  return  the
corresponding index - one of the centers
            min_i = np.argmin(distance)

            # store the sample in dictionary under the key (one center)
            clusters[min_i].append(s)

            # give a label for a smaple, 0, 1, 2, ..., k
            labels.append(min_i)

        # recalculate the central value
        centers = np.zeros((k, c))
        for i, value in enumerate(clusters.values()):
            mean = np.mean(value, axis = 0)
            centers[i] = mean

        print("iterating " + str(n + 1) + " time(s) ......")

        n += 1

    print(centers)

    return np.array(labels), centers

# looking for the nearest distances between a point with all centers
def nearest(point, centers):

    min_dist = float("inf")

    for c in centers:
        distance  = dist_eclu(point, c)

        if min_dist > distance:
            min_dist = distance

    return min_dist
```

kmeans_plus_init(samples, k)

```
def kmeans_plus_init(samples, k):

    r, c = samples.shape
    rand = np.random.randint(0, r)
    centers = np.zeros((k, c))
    centers[0] = samples[rand, :]    # randomly choose the first
center

    for n in range(1, k):

        # distances between a point with all centers
        distances = []
        for s in samples:
            distances.append(nearest(s, centers[0:k]))
        total = np.sum(distances)
        weights = [x / total for x in distances]

        # select the point as new center that is far away from exist
center within probability
        prob = np.random.random()
        total = 0
        x = -1
        while total < prob:
            x += 1
            total += weights[x]
        centers[n] = samples[x]

    return centers
```

PCA(self, data, k)

```
# perform PCA on the data matrix , return average, covariance vector,
and difference
    def PCA(self, data, k):

        data = np.transpose(data)
        average = np.mean(data, axis = 1)   # average of all images
        diff = data - average              # differences between all
images and the average

        eig_vals,  eig_vects  =  np.linalg.eig(diff.T  *  diff)    #
```

```
eigenvalues and eigenvectors

       eig_sort_index = np.argsort(-eig_vals)  # sort eigenvalues
from large to small
       k_index  =  eig_sort_index[:k]                  #  select  k
eigenvectors with largest eigenvalues

       cov_vects = diff * eig_vects[:, k_index]

       return average, cov_vects, diff
```

face_recognizer(self, test, average, cov_vects, diff_train)

```
# three face images that are the most similar with the given image
   # perform nearest neighbor
   def face_recognizer(self, test, average, cov_vects, diff_train):

       test = np.transpose(test)
       distances = []

       diff = test - average
       test_vect = cov_vects.T * diff

       for train in diff_train.T:

          train_vect = cov_vects.T * train.T
          distances.append(np.linalg.norm(test_vect - train_vect))
# calculate second norm

       sort_index = np.argsort(distances)

       return sort_index[:3]
```