Hello Soft Clustering (GMM)

T1. Using 3 mixtures, initialize your Gaussian with means (3,3), (2,2), and (-3,-3), and standard Covariance, I, the identity matrix. Use equal mix- ture weights as the initial weights. Repeat three iterations of EM. Write down $w_{n,j}$, m_j , $\vec{\mu}_j$, Σ_j for each EM iteration. (You may do the calculations by hand or write code to do so)

$$w_{n,j} = \frac{p(x_n; \vec{\mu_j}, \mathbf{\Sigma}_j)m_j}{\Sigma_j p(x_n; \vec{\mu_j}, \mathbf{\Sigma}_j)m_j} \tag{1}$$

$w_{n,j}$ means the probability that data point n comes from Gaussian number j.

Maximization: Update the model parameters, ϕ, $\vec{\mu_j}$, $\mathbf{\Sigma}_j$.

$$m_j = \frac{1}{N}\Sigma_n w_{n,j} \tag{2}$$

$$\vec{\mu_j} = \frac{\Sigma_n w_{n,j}\vec{x_n}}{\Sigma_n w_{n,j}} \tag{3}$$

$$\mathbf{\Sigma}_j = \frac{\Sigma_n w_{n,j}(\vec{x_n} - \vec{\mu_j})(\vec{x_n} - \vec{\mu_j})^T}{\Sigma_n w_{n,j}} \tag{4}$$

The above equation is used for full covariance matrices. For our small toy example, we will use diagonal covariance matrices, which can be acquired by setting the off-diagonal values to zero. In other words, $\mathbf{\Sigma}_{(i,j)} = 0$, for $i \neq j$.

TODO: Complete functions below including

- Fill relevant parameters in each function.
- Implement computation and return values.

These functions will be used in T1-4.

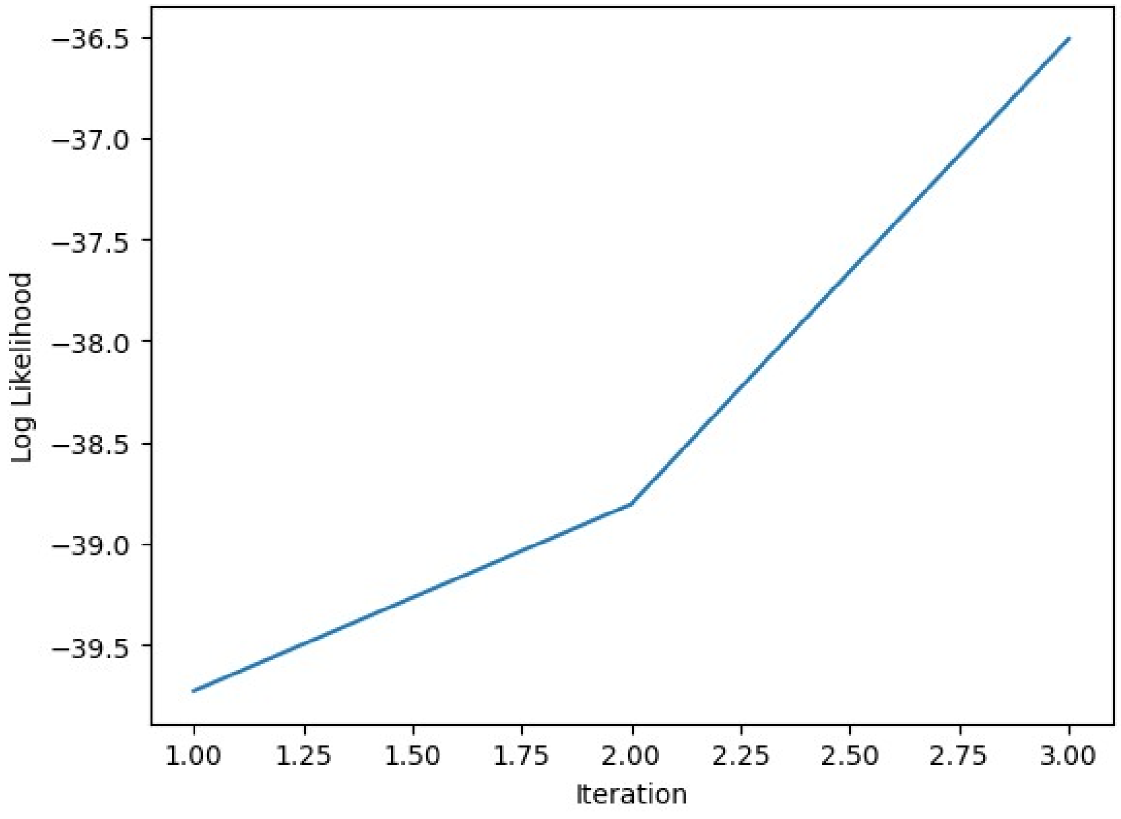
```
import numpy as np
import matplotlib.pyplot as plt

# Hint: You can use this function to get gaussian distribution.
#
https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.multi
```

```
variate_normal.html
from scipy.stats import multivariate_normal

class GMM:
    def __init__(self, mixture_weight, mean_params, cov_params):
        """
        Initialize GMM.
        """
        # Copy construction values.
        self.mixture_weight = mixture_weight
        self.mean_params = mean_params + 1e-20
        self.cov_params = cov_params + 1e-20

        # Initiailize iteration.
        self.n_iter = 0

    def estimation_step(self, data):
        """
        TODO: Perform estimation step. Then, return w_{n,j} in eq. 1)
        """



        # INSERT CODE HERE
        size = self.mean_params.shape[0]
        probability = []
        for i in range(size):
            probability += [multivariate_normal(mean =
self.mean_params[i], cov=self.cov_params[i]).pdf(data)]
        probability = np.array(probability)

        probability_multiplied = np.eye(len(self.mixture_weight)) *
self.mixture_weight @ probability

        w = probability_multiplied / np.sum(probability_multiplied,
axis=0)
        return w


    def maximization_step(self, data, w):
        """
        TODO: Perform maximization step.
            (Update parameters in this GMM model.)
        """
        # INSERT CODE HERE

        size = data.shape[0]

        self.mixture_weight = (1/size)*np.sum(w, axis=1)
        self.mean_params = ((w @ data).T / (w.sum(axis=1)+1e-20)).T
        self.cov_params = np.array([np.dot((w[i].reshape(-1,1)*(data-
self.mean_params[i])).T, (data-self.mean_params[i]))/(np.sum(w[i])+1e-
20) for i in range(self.mean_params.shape[0])])
        self.cov_params = (self.cov_params+1e-20) *
np.eye(self.cov_params.shape[1])


    def get_log_likelihood(self, data):
        """
        TODO: Compute log likelihood.
        """


        # INSERT CODE HERE
        size = self.mean_params.shape[0]
        probability = []
        for i in range(size):
            probability += [multivariate_normal(mean =
self.mean_params[i],
cov=self.cov_params[i]).pdf(data)*self.mixture_weight[i]]
        probability = np.array(probability)

        log_prob = np.log(np.array(probability).sum(axis=0)).sum()

        return log_prob

    def print_iteration(self):
        print("m :\n", self.mixture_weight)
        print("mu :\n", self.mean_params)
        print("covariance matrix :\n", self.cov_params)

print("-------------------------------------------------------------")

    def perform_em_iterations(self, data, num_iterations,
display=True):
        """
        Perform estimation & maximization steps with num_iterations.
        Then, return list of log_likelihood from those iterations.
        """
        log_prob_list = []

        # Display initialization.
        if display:
            print("Initialization")
            self.print_iteration()

        for n_iter in range(num_iterations):

            # TODO: Perform EM step.

            # INSERT CODE HERE
            w = self.estimation_step(data)
            self.maximization_step(data, w)


            # Calculate log prob.
            log_prob = self.get_log_likelihood(data)
            log_prob_list.append(log_prob)

            # Display each iteration.
            if display:
                print(f"Iteration: {n_iter}")
                self.print_iteration()

        return log_prob_list

num_iterations = 3
num_mixture = 3
mixture_weight = [1] * num_mixture # m
mean_params = np.array([[3,3], [2,2], [-3,-3]], dtype = float)
cov_params = np.array([np.eye(2)] * num_mixture)

X, Y = np.array([1, 3, 2, 8, 6, 7, -3, -2, -7]), np.array([2, 3, 2, 8,
6, 7, -3, -4, -7])
data = np.vstack([X,Y]).T

gmm = GMM(mixture_weight, mean_params, cov_params)
log_prob_list = gmm.perform_em_iterations(data, num_iterations)
```

```
Initialization
m :
 [1, 1, 1]
mu :
 [[ 3.  3.]
 [ 2.  2.]
 [-3. -3.]]
covariance matrix :
 [[[1.e+00 1.e-20]
  [1.e-20 1.e+00]]

 [[1.e+00 1.e-20]
  [1.e-20 1.e+00]]

 [[1.e+00 1.e-20]
  [1.e-20 1.e+00]]]
-------------------------------------------------------------
Iteration: 0
m :
 [0.45757242 0.20909425 0.33333333]
mu :
 [[ 5.78992692  5.81887265]
 [ 1.67718211  2.14523106]
 [-4.         -4.66666666]]
covariance matrix :
 [[[4.53619412 0.        ]
  [0.         4.28700611]]

 [[0.51645579 0.        ]
  [0.         0.13152618]]

 [[4.66666668 0.        ]
  [0.         2.88888891]]]
-------------------------------------------------------------
Iteration: 1
m :
 [0.40711618 0.25954961 0.33333421]
mu :
 [[ 6.27176215  6.27262711]
 [ 1.72091544  2.14764812]
 [-3.99998589 -4.6666488 ]]
covariance matrix :
 [[[2.94672736 0.        ]
  [0.         2.93847196]]

 [[0.49649261 0.        ]
  [0.         0.12584815]]

 [[4.66673088 0.        ]
  [0.         2.88900236]]]
-------------------------------------------------------------
Iteration: 2
m :
 [0.36070909 0.30595677 0.33333414]
mu :
 [[ 6.6962644   6.69629468]
 [ 1.91071238  2.27383436]
 [-3.99998673 -4.6666501 ]]
covariance matrix :
 [[[1.73961067 0.        ]
  [0.         1.73929602]]

 [[0.62898406 0.        ]
  [0.         0.1988491 ]]

 [[4.66672942 0.        ]
  [0.         2.88899545]]]
-------------------------------------------------------------
```

T2. Plot the log likelihood of the model given the data after each EM step. In other words, plot $\log \prod_n p(\vec{x}_n \vee \varphi, \vec{\mu}, \Sigma)$. Does it goes up every iteration just as we learned in class?

```
# TODO
def plot_log_likelihood(log_prob_list, num_iterations):
        plt.figure()
        plt.plot(range(1, num_iterations + 1), log_prob_list)
        plt.xlabel('Iteration')
        plt.ylabel('Log Likelihood')
        plt.show()

plot_log_likelihood(log_prob_list, num_iterations)
```



```
ANS : Yes, they went up after each iterations
```

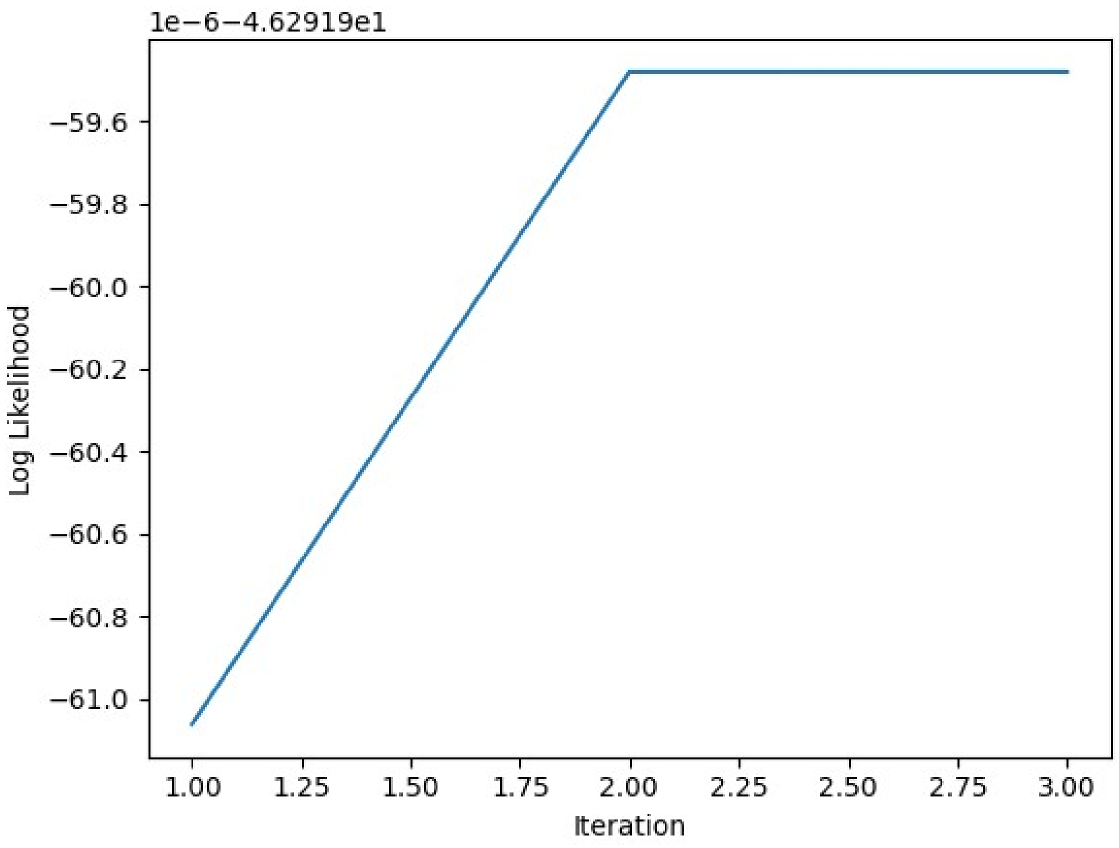
T3. Using 2 mixtures, initialize your Gaussian with means (3,3) and (-3,-3), and standard Covariance, I, the identity matrix. Use equal mixture weights as the initial weights. Repeat three iterations of EM. Write down $w_{n,j}$, m_j , $\vec{\mu}_j$, Σ_j for each EM iteration.

```
num_mixture = 2
mixture_weight = [1] * num_mixture

mean_params = np.array([[3,3], [-3,-3]], dtype = float)
cov_params = np.array([np.eye(2)] * num_mixture)

# INSERT CODE HERE
gmm2 = GMM(mixture_weight=mixture_weight, mean_params=mean_params,
cov_params=cov_params)
log_prob_list2 = gmm2.perform_em_iterations(data, num_iterations)
```

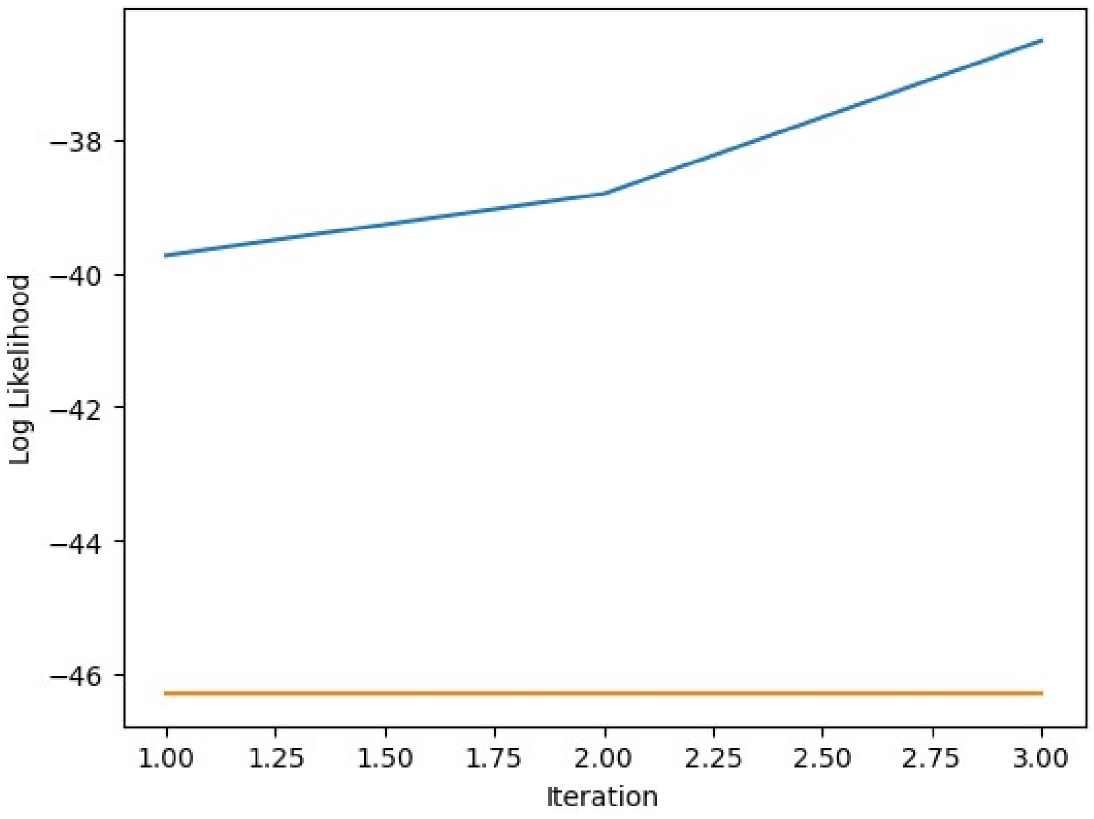
```
Initialization
m :
 [1, 1]
mu :
 [[ 3.  3.]
 [-3. -3.]]
covariance matrix :
 [[[1.e+00 1.e-20]
  [1.e-20 1.e+00]]

 [[1.e+00 1.e-20]
  [1.e-20 1.e+00]]]
-------------------------------------------------------------
Iteration: 0
m :
 [0.66666666 0.33333334]
mu :
 [[ 4.50000001  4.66666667]
 [-3.99999997 -4.66666663]]
covariance matrix :
 [[[6.91666665 0.        ]
  [0.         5.88888889]]

 [[4.66666677 0.        ]
  [0.         2.8888891 ]]]
-------------------------------------------------------------
Iteration: 1
m :
 [0.66669436 0.33330564]
mu :
 [[ 4.49961311  4.66620178]
 [-3.99993241 -4.66651231]]
covariance matrix :
 [[[6.91944755 0.        ]
  [0.         5.89275124]]

 [[4.66806942 0.        ]
  [0.         2.89103318]]]
-------------------------------------------------------------
Iteration: 2
m :
 [0.66669453 0.33330547]
mu :
 [[ 4.49961084  4.66619903]
 [-3.99993206 -4.66651141]]
covariance matrix :
 [[[6.91946372 0.        ]
  [0.         5.8927741 ]]

 [[4.66807754 0.        ]
  [0.         2.89104566]]]
-------------------------------------------------------------
```

T4. Plot the log likelihood of the model given the data after each EM step. Compare the log likelihood between using two mixtures and three mixtures. Which one has the better likelihood?

```
# TODO: Plot log_likelihood from T3
plot_log_likelihood(log_prob_list2, num_iterations)
```



```
# TODO: Plot Comparision of log_likelihood from T1 and T3

plt.figure()
plt.plot(range(1, num_iterations + 1), log_prob_list)
plt.plot(range(1, num_iterations + 1), log_prob_list2)
plt.xlabel('Iteration')
plt.ylabel('Log Likelihood')
plt.show()
```



```
ANS : The 3 mixtures one is better because it has better log
likelihood
```

The face database

```python
# Download facedata for google colab
# !wget -nc
https://github.com/ekapolc/Pattern_2024/raw/main/HW/HW03/facedata_mat.
zip
# !unzip facedata_mat.zip

import scipy.io
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.image as mpimg
from skimage import img_as_float

# Change path to your facedata.mat file.
facedata_path = 'facedata.mat'

data = scipy.io.loadmat(facedata_path)
data_size = data['facedata'].shape
```

```
%matplotlib inline
data_size

(40, 10)
```

Preprocess xf

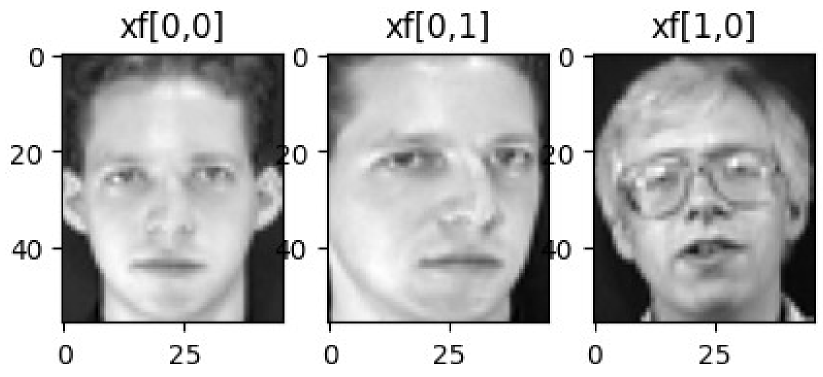
```
xf = np.zeros((data_size[0], data_size[1], data['facedata']
[0,0].shape[0], data['facedata'][0,0].shape[1]))
for i in range(data['facedata'].shape[0]):
    for j in range(data['facedata'].shape[1]):
        xf[i,j] = img_as_float(data['facedata'][i,j])

# Example: Ploting face image.
plt.imshow(xf[0,0], cmap = 'gray')
plt.show()
```

T5. What is the Euclidean distance between xf[0,0] and xf[0,1]? What is the Euclidean distance between xf[0,0] and xf[1,0]? Does the numbers make sense? Do you think these numbers will be useful for face verification?

```
def L2_dist(x1, x2):
    """
    TODO: Calculate L2 distance.
    """
    return np.sqrt( ((x1-x2)**2).sum() )

# Test L2_dist
def test_L2_dist():
    assert L2_dist(np.array([1, 2, 3]), np.array([1, 2, 3])) == 0.0
    assert  L2_dist(np.array([0, 0, 0]), np.array([1, 2, 3])) ==
np.sqrt(14)

test_L2_dist()

print('Euclidean distance between xf[0,0] and xf[0,1] is',
L2_dist(xf[0,0], xf[0,1]))
print('Euclidean distance between xf[0,0] and xf[1,0] is',
L2_dist(xf[0,0], xf[1,0]))
```

```
Euclidean distance between xf[0,0] and xf[0,1] is 10.037616294165492
Euclidean distance between xf[0,0] and xf[1,0] is 8.173295099737281
```

```
# TODO: Show why does the numbers make sense
fig, axes = plt.subplots(1, 3,figsize=(5, 3))
axes[0].imshow(xf[0,0], cmap = 'gray')
axes[0].set_title('xf[0,0]')
axes[1].imshow(xf[0,1], cmap = 'gray')
axes[1].set_title('xf[0,1]')
axes[2].imshow(xf[1,0], cmap = 'gray')
axes[2].set_title('xf[1,0]')
plt.show()
```



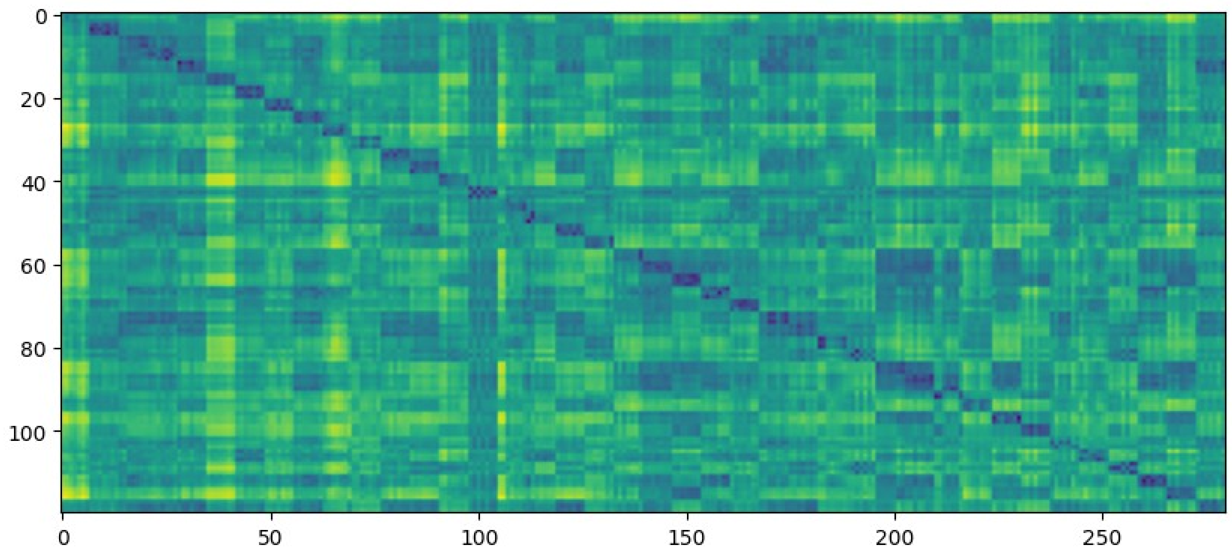
```
ANS : The [1, 0] face's posture is more similiar to the original than
the [0, 1] one's
```

T6. Write a function that takes in a set of feature vectors T and a set of feature vectors D, and then output the similarity matrix A. Show the matrix as an image. Use the feature vectors from the first 3 images from all 40 people for list T (in order x[0, 0], x[0, 1], x[0, 2], x[1, 0], x[1, 1], ...x[39, 2]). Use the feature vectors from the remaining 7 images from all 40 people for list D (in order x[0, 3], x[0, 4], x[0, 5], x[1, 6], x[0, 7], x[0, 8], x[0, 9], x[1, 3], x[1, 4]...x[39, 9]). We will treat T as our training images and D as our testing images

```python
def organize_shape(matrix):
    """
    TODO (Optional): Reduce matrix dimension of 2D image to 1D and
merge people and image dimension.
    This function can be useful at organizing matrix shapes.

    Example:
        Input shape: (people_index, image_index, image_shape[0],
image_shape[1])
        Output shape: (people_index*image_index,
image_shape[0]*image_shape[1])
    """
    new_matrix = matrix.copy()
    new_row, new_col = new_matrix.shape[0] * new_matrix.shape[1],
new_matrix.shape[2]*new_matrix.shape[3]
    new_matrix = new_matrix.reshape((new_row, new_col))
    return new_matrix


def generate_similarity_matrix(A, B):
    """
    TODO: Calculate similarity matrix M,
    which M[i, j] is a distance between A[i] and B[j].
    """

    # INSERT CODE HERE
    similarity_matrix = np.zeros((A.shape[0], B.shape[0]))
    for i in range(A.shape[0]):
        for j in range(B.shape[0]):
            similarity_matrix[i][j] = L2_dist(A[i], B[j])

    return similarity_matrix

def test_generate_similarity_matrix():
    test_A = np.array([[1, 2],[3,4]])
    test_B = np.array([[1, 2], [5, 6], [7, 8]])
    expected_matrix = np.sqrt(np.array([[0, 32, 72], [8, 8, 32]]))
    assert (generate_similarity_matrix(test_A, test_B) ==
expected_matrix).all()
```

```
test_generate_similarity_matrix()

#TODO: Show similariry matrix between T and D.

# INSERT CODE HERE
T = organize_shape(xf[:, :3])
D = organize_shape(xf[:, 3:])

similarity_matrix = generate_similarity_matrix(T, D)

plt.figure(figsize=(10, 10))
plt.imshow(similarity_matrix)
```

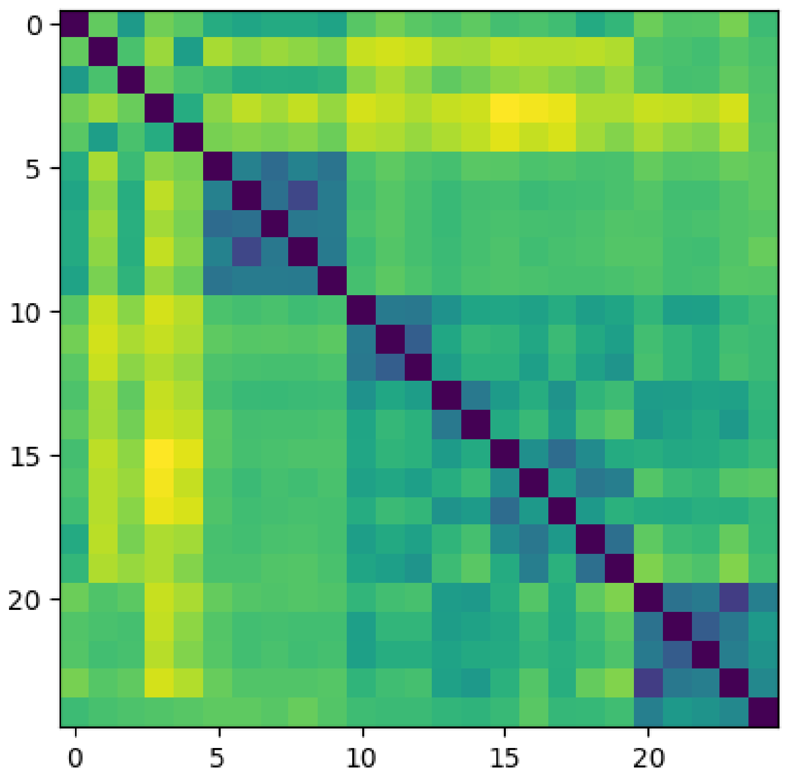
```
<matplotlib.image.AxesImage at 0x1c4d251c050>
```



T7. From the example similarity matrix above, what does the black square between [5:10,5:10] suggest about the pictures from person number 2? What do the patterns from person number 1 say about the images from person 1?

```
# INSERT CODE HERE

new_T = organize_shape(xf[:5, :5])

plt.imshow(generate_similarity_matrix(new_T, new_T))
```

```
<matplotlib.image.AxesImage at 0x1c4d243a4d0>
```



```
    ANS : For person 1, the images are simliar, while the others are
not.
```

T8. Write a function that takes in the similarity matrix created from the previous part, and a threshold t as inputs. The outputs of the function are the true positive rate and the false alarm rate of the face verification task (280 Test images, tested on 40 people, a total of 11200 testing per threshold). What is the true positive rate and the false alarm rate for t = 10?

```python
def evaluate_performance(similarity_matrix, threshold):
    """
    TODO: Calculate true positive rate and false alarm rate from given
similarity_matrix and threshold.
    """

    # INSERT CODE HERE
    y_pred = np.zeros((40, 280))
    y_actual = np.zeros((40, 280))
    for i in range(40):
        for j in range(280):
            y_pred[i, j] = similarity_matrix[3*i:3*i+3, j].min() <
threshold
```

```
            y_actual[i, j] = i == j//7

    tp = np.where(y_pred==y_actual, y_pred, 0).sum()
    tn = np.where(y_pred==y_actual, 1-y_pred, 0).sum()
    fp = np.where(y_pred!=y_actual, 1-y_actual, 0).sum()
    fn = np.where(y_pred!=y_actual, y_actual, 0).sum()

    true_pos_rate = tp/(tp+fn)
    false_positive_rate = fp/(tn+fp)
    false_neg_rate=fn/(tp+fn)
    true_neg_rate=tn/(tn+fp)
    return true_pos_rate, false_positive_rate, false_neg_rate,
true_neg_rate

# Quick check
# (true_pos_rate, false_neg_rate) should be (0.9928571428571429,
0.33507326007326005)
evaluate_performance(similarity_matrix, 9.5)
```

```
(0.9928571428571429,
 0.33507326007326005,
 0.007142857142857143,
 0.6649267399267399)
```

```
# INSERT CODE HERE

evaluate_performance(similarity_matrix, 10)
```

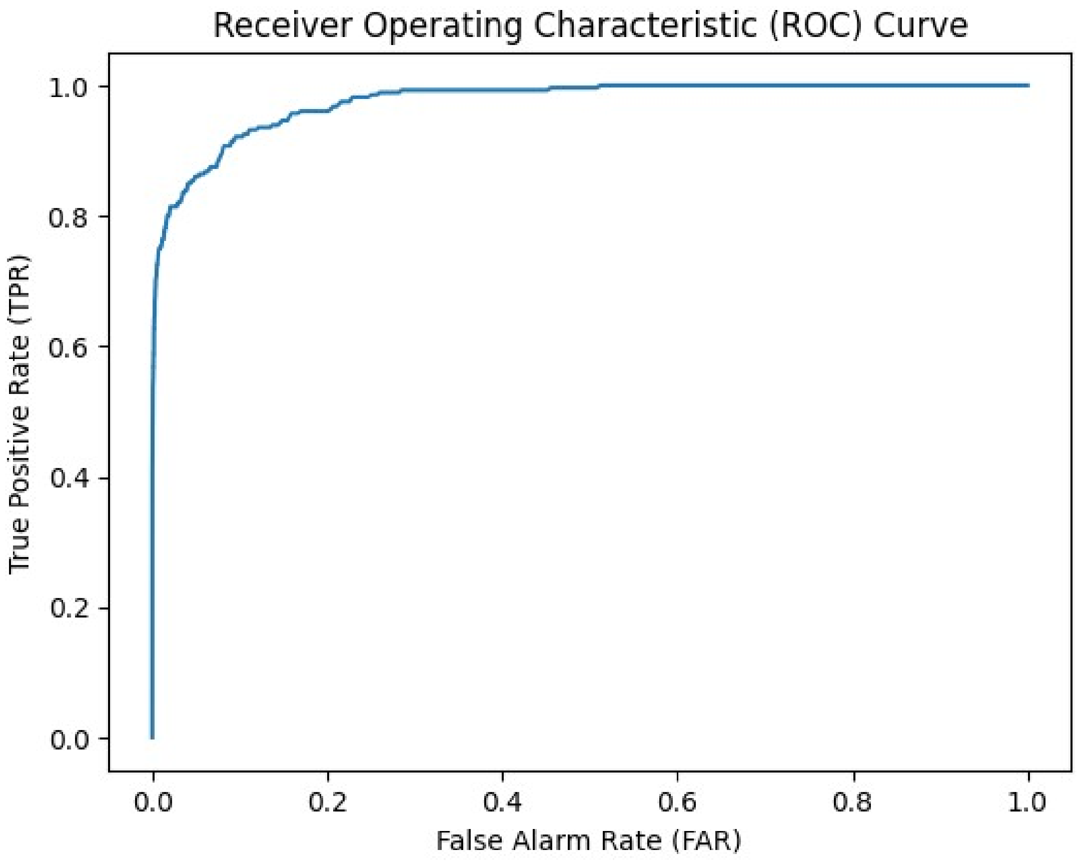
```
(0.9964285714285714,
 0.4564102564102564,
 0.0035714285714285713,
 0.5435897435897435)
```

```
    ANS: For threshold=10, the rates are 0.9964285714285714 and
0.4564102564102564 respectively.
```

T9. Plot the RoC curve for this simple verification system. What should be the minimum threshold to generate the RoC curve? What should be the maximum threshold? Your RoC should be generated from at least 1000 threshold levels equally spaced between the minimum and the maximum. (You should write a function for this).

```
def calculate_roc(input_mat):
    """
    TODO: Calculate a list of true_pos_rate and a list of
false_neg_rate from the given matrix.
    """

    # INSERT CODE HERE
    tpr_list, fpr_list, fnr_list, tnr_list=[],[],[],[]

    for threshold in np.linspace(np.min(input_mat), np.max(input_mat),
num=1000):
        tpr, fpr, fnr, tnr = evaluate_performance(similarity_matrix,
threshold)

        tpr_list.append(tpr)
        fpr_list.append(fpr)
        fnr_list.append(fnr)
        tnr_list.append(tnr)

    tpr_list = np.array(tpr_list)
    fpr_list = np.array(fpr_list)
    fnr_list = np.array(fnr_list)
    tnr_list = np.array(tnr_list)

    return tpr_list, fpr_list, fnr_list, tnr_list

tpr_list, fpr_list, fnr_list,
tnr_list=np.array([]),np.array([]),np.array([]),np.array([])


def plot_roc(input_mat, label=None, show=None):
    """
    TODO: Plot RoC Curve from a given matrix.
    """
    # INSERT CODE HERE
    global tpr_list,fpr_list,fnr_list,tnr_list

    tpr_list, fpr_list, fnr_list, tnr_list = calculate_roc(input_mat)

    if label:
        plt.plot(fpr_list, tpr_list, label=label)
    else:
        plt.plot(fpr_list, tpr_list)
    plt.plot(fpr_list, tpr_list)
    plt.xlabel('False Alarm Rate (FAR)')
    plt.ylabel('True Positive Rate (TPR)')
    plt.title('Receiver Operating Characteristic (ROC) Curve')
    if show == None:
        plt.show()

    return fnr_list

# INSERT CODE HERE

tpr_list,fpr_list,fnr_list,tnr_list=np.array([]),np.array([]),np.array
([]),np.array([])
```

```
plot_roc(similarity_matrix)
```



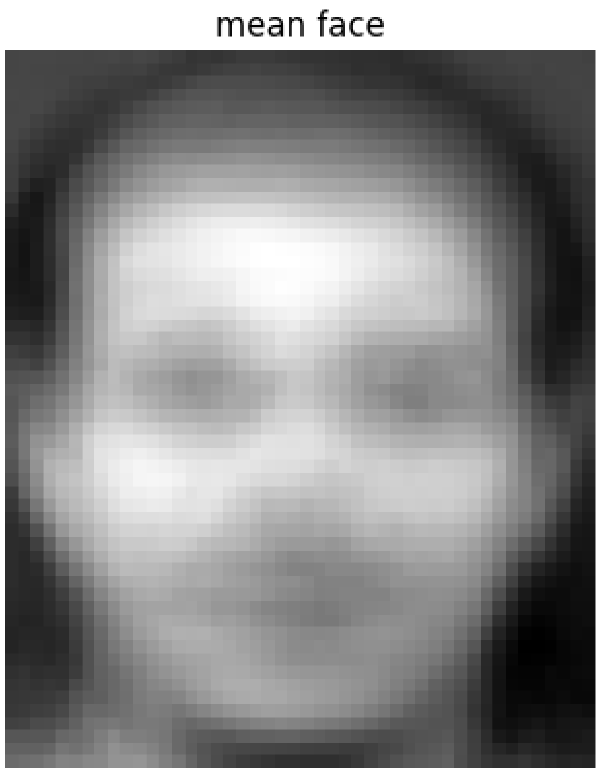
```
array([1.        , 0.99642857, 0.99642857, 0.99642857, 0.99642857,
       0.99642857, 0.99642857, 0.99642857, 0.99285714, 0.99285714,
       0.99285714, 0.99285714, 0.99285714, 0.99285714, 0.99285714,
       0.99285714, 0.99285714, 0.99285714, 0.99285714, 0.99285714,
       0.99285714, 0.99285714, 0.99285714, 0.99285714, 0.99285714,
       0.99285714, 0.99285714, 0.99285714, 0.99285714, 0.99285714,
       0.99285714, 0.99285714, 0.99285714, 0.99285714, 0.99285714,
       0.99285714, 0.99285714, 0.99285714, 0.99285714, 0.99285714,
       0.99285714, 0.99285714, 0.98928571, 0.98928571, 0.98928571,
       0.98928571, 0.98928571, 0.98571429, 0.98571429, 0.98571429,
       0.98571429, 0.98214286, 0.98214286, 0.98214286, 0.98214286,
       0.98214286, 0.98214286, 0.97857143, 0.97857143, 0.97857143,
       0.97857143, 0.97857143, 0.97857143, 0.97857143, 0.97857143,
       0.97857143, 0.97857143, 0.975     , 0.97142857, 0.97142857,
       0.97142857, 0.96785714, 0.96785714, 0.96785714, 0.96785714,
       0.96785714, 0.96785714, 0.96785714, 0.96785714, 0.96785714,
       0.96785714, 0.96785714, 0.96785714, 0.96785714, 0.96785714,
       0.96428571, 0.96428571, 0.96428571, 0.96428571, 0.96428571,
       0.96428571, 0.96071429, 0.95714286, 0.95357143, 0.95357143,
       0.95357143, 0.95      , 0.95      , 0.95      , 0.95      ,
       0.95      , 0.95      , 0.94642857, 0.94642857, 0.94642857,
       0.94642857, 0.94642857, 0.94642857, 0.94285714, 0.94285714,
       0.94285714, 0.94285714, 0.93928571, 0.93928571, 0.93571429,
       0.93214286, 0.925     , 0.925     , 0.92142857, 0.92142857,
       0.92142857, 0.91785714, 0.91785714, 0.91785714, 0.91428571,
       0.91071429, 0.91071429, 0.90714286, 0.90714286, 0.90357143,
       0.90357143, 0.9       , 0.9       , 0.89642857, 0.89642857,
       0.88928571, 0.88928571, 0.88928571, 0.88214286, 0.88214286,
       0.87142857, 0.86785714, 0.86428571, 0.86071429, 0.85714286,
       0.85357143, 0.84642857, 0.84642857, 0.84642857, 0.84285714,
       0.83928571, 0.83928571, 0.82857143, 0.825     , 0.82142857,
       0.81785714, 0.81428571, 0.81428571, 0.81071429, 0.80714286,
       0.8       , 0.8       , 0.8       , 0.8       , 0.79285714,
       0.79285714, 0.79285714, 0.79285714, 0.78928571, 0.78928571,
       0.78571429, 0.78571429, 0.78214286, 0.77857143, 0.77857143,
       0.77857143, 0.77857143, 0.775     , 0.76785714, 0.76785714,
       0.76785714, 0.76428571, 0.76428571, 0.76428571, 0.75714286,
       0.75357143, 0.75      , 0.75      , 0.75      , 0.74642857,
       0.74285714, 0.73928571, 0.73571429, 0.72857143, 0.72857143,
       0.72142857, 0.72142857, 0.72142857, 0.72142857, 0.72142857,
       0.72142857, 0.72142857, 0.71785714, 0.71071429, 0.71071429,
       0.70714286, 0.70714286, 0.70714286, 0.70714286, 0.70357143,
       0.7       , 0.68571429, 0.67857143, 0.66428571, 0.66071429,
       0.66071429, 0.65714286, 0.65714286, 0.65714286, 0.65357143,
       0.64285714, 0.63928571, 0.63571429, 0.63214286, 0.62857143,
       0.625     , 0.61785714, 0.61785714, 0.61785714, 0.61785714,
       0.6       , 0.6       , 0.59642857, 0.59642857, 0.59642857,
       0.59285714, 0.58928571, 0.58571429, 0.58214286, 0.575     ,
       0.56785714, 0.56785714, 0.56071429, 0.55714286, 0.55714286,
       0.55357143, 0.54642857, 0.53928571, 0.53928571, 0.53214286,
       0.53214286, 0.53214286, 0.52857143, 0.52142857, 0.51785714,
       0.51785714, 0.50357143, 0.5       , 0.48928571, 0.48571429,
       0.48214286, 0.48214286, 0.48214286, 0.475     , 0.47142857,
       0.46071429, 0.45714286, 0.45      , 0.43928571, 0.43214286,
       0.43214286, 0.43214286, 0.42857143, 0.425     , 0.425     ,
       0.425     , 0.41785714, 0.41785714, 0.41428571, 0.41071429,
       0.41071429, 0.41071429, 0.40714286, 0.40357143, 0.40357143,
       0.39642857, 0.38928571, 0.38571429, 0.38571429, 0.38571429,
       0.37142857, 0.37142857, 0.36428571, 0.35714286, 0.35714286,
       0.35      , 0.34285714, 0.33928571, 0.32857143, 0.325     ,
       0.325     , 0.31428571, 0.31071429, 0.30714286, 0.3       ,
       0.29642857, 0.29285714, 0.28214286, 0.275     , 0.275     ,
       0.275     , 0.275     , 0.26428571, 0.26428571, 0.26071429,
       0.26071429, 0.25714286, 0.25357143, 0.25      , 0.25      ,
       0.25      , 0.25      , 0.25      , 0.24642857, 0.24642857,
       0.24642857, 0.24642857, 0.24285714, 0.23571429, 0.23571429,
       0.23571429, 0.23571429, 0.22857143, 0.225     , 0.225     ,
       0.21785714, 0.21785714, 0.21071429, 0.21071429, 0.20357143,
       0.2       , 0.2       , 0.2       , 0.2       , 0.19285714,
       0.18571429, 0.18571429, 0.18571429, 0.18571429, 0.18571429,
       0.18571429, 0.18571429, 0.18571429, 0.18571429, 0.18214286,
       0.18214286, 0.17857143, 0.17857143, 0.17857143, 0.175     ,
       0.17142857, 0.16428571, 0.16428571, 0.16071429, 0.16071429,
       0.16071429, 0.15357143, 0.15      , 0.15      , 0.15      ,
       0.14642857, 0.14642857, 0.14642857, 0.14285714, 0.13928571,
       0.13928571, 0.13928571, 0.13928571, 0.13571429, 0.13571429,
       0.13571429, 0.13571429, 0.13214286, 0.13214286, 0.13214286,
       0.12857143, 0.12857143, 0.125     , 0.125     , 0.125     ,
       0.125     , 0.125     , 0.11785714, 0.11428571, 0.11071429,
       0.10714286, 0.10357143, 0.09285714, 0.09285714, 0.09285714,
       0.09285714, 0.09285714, 0.08571429, 0.08571429, 0.08214286,
       0.07857143, 0.07857143, 0.07857143, 0.07857143, 0.07857143,
       0.075     , 0.075     , 0.075     , 0.06785714, 0.06785714,
       0.06785714, 0.06785714, 0.06785714, 0.06428571, 0.06428571,
       0.06428571, 0.06428571, 0.06428571, 0.06428571, 0.06428571,
       0.06071429, 0.06071429, 0.06071429, 0.06071429, 0.05714286,
       0.05357143, 0.05357143, 0.05357143, 0.05357143, 0.05      ,
       0.04285714, 0.04285714, 0.04285714, 0.04285714, 0.03928571,
       0.03928571, 0.03928571, 0.03928571, 0.03928571, 0.03928571,
       0.03928571, 0.03928571, 0.03928571, 0.03928571, 0.03928571,
       0.03928571, 0.03571429, 0.03214286, 0.03214286, 0.02857143,
       0.025     , 0.025     , 0.025     , 0.025     , 0.01785714,
       0.01785714, 0.01785714, 0.01785714, 0.01785714, 0.01785714,
       0.01785714, 0.01428571, 0.01428571, 0.01428571, 0.01071429,
       0.01071429, 0.01071429, 0.01071429, 0.01071429, 0.01071429,
       0.01071429, 0.00714286, 0.00714286, 0.00714286, 0.00714286,
       0.00714286, 0.00714286, 0.00714286, 0.00714286, 0.00714286,
       0.00714286, 0.00714286, 0.00714286, 0.00714286, 0.00714286,
       0.00714286, 0.00714286, 0.00714286, 0.00714286, 0.00714286,
       0.00714286, 0.00714286, 0.00714286, 0.00714286, 0.00714286,
       0.00714286, 0.00714286, 0.00714286, 0.00714286, 0.00714286,
       0.00714286, 0.00714286, 0.00714286, 0.00714286, 0.00714286,
       0.00714286, 0.00714286, 0.00714286, 0.00714286, 0.00714286,
       0.00714286, 0.00714286, 0.00714286, 0.00714286, 0.00714286,
       0.00714286, 0.00714286, 0.00357143, 0.00357143, 0.00357143,
       0.00357143, 0.00357143, 0.00357143, 0.00357143, 0.00357143,
       0.00357143, 0.00357143, 0.00357143, 0.00357143, 0.00357143,
       0.00357143, 0.00357143, 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ,
       0.        , 0.        , 0.        , 0.        , 0.        ])

ANS: maximum and minimum value in the similiarity matrix
```

T10. What is the EER (Equal Error Rate)? What is the recall rate at 0.1% false alarm rate? (Write this in the same function as the previous question)

```
# You can add more parameter(s) to the function in the previous
question.

# EER should be either 0.9071428571428571 or 0.9103759398496248
depending on method.
# Recall rate at 0.1% false alarm rate should be 0.5428571428571428.

print("EER :",tpr_list[np.argmin(np.abs(fpr_list - fnr_list))])
print("Recall at 0.001 false neg :",tpr_list[np.argmin(np.abs(fpr_list
- 0.001))])

EER : 0.9071428571428571
Recall at 0.001 false neg : 0.5428571428571428

ANS:
```

T11. Compute the mean vector from the training images. Show the vector as an image (use numpy.reshape()). This is typically called the meanface (or meanvoice for speech signals). You answer should look exactly like the image shown below.

```
# INSERT CODE HERE
meanface = np.mean(T.reshape(120,56*46),axis=0)

plt.title('mean face')
plt.axis('off')
plt.imshow(meanface.reshape(56, 46), cmap='gray')
plt.show()
```



T12. What is the size of the covariance matrix? What is the rank of the covariance matrix?

```
# TODO: Find the size and the rank of the covariance matrix.
print("size: ",T.shape[1], T.shape[1])
print("rank: ", min(T.shape[1], T.shape[0] - 1))
```

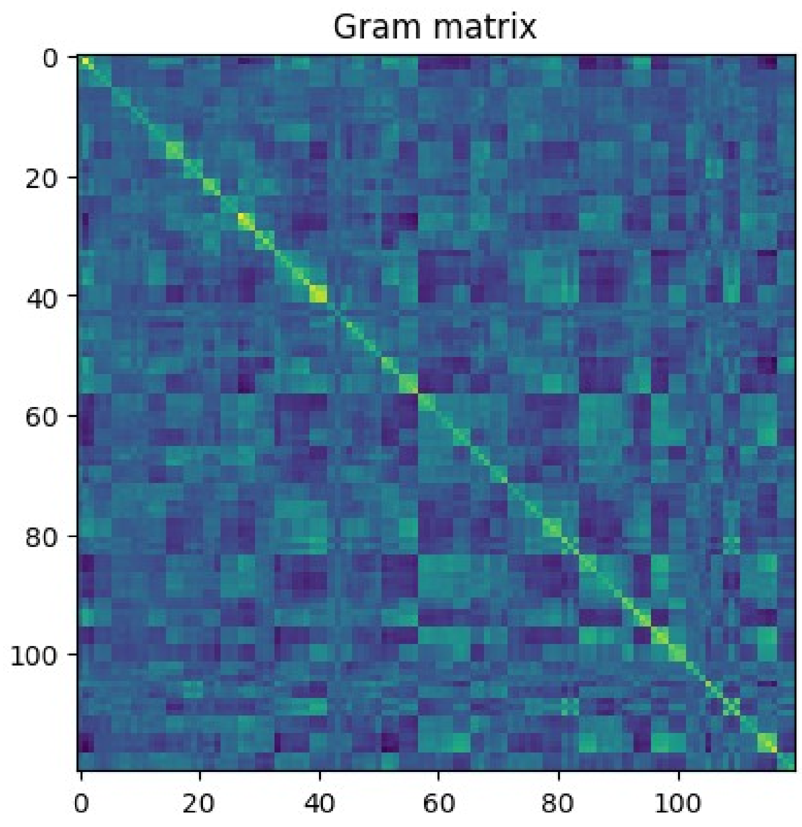
```
size:  2576 2576
rank:  119
```

```
ANS: size: (2576, 2576), rank: 119
```

T13. What is the size of the Gram matrix? What is the rank of Gram matrix? If we compute the eigenvalues from the Gram matrix, how many non- zero eigenvalues do we expect to get?

```
# TODO: Compute gram matrix.
gram_matrix = np.matmul(T-meanface, (T-meanface).T)

plt.title(f'Gram matrix')
plt.imshow(gram_matrix)
```

```
<matplotlib.image.AxesImage at 0x1c4d251e750>
```



```
# TODO: Show size and rank of Gram matrix.

print("ANS:")
print("size", gram_matrix.shape)
print("rank", gram_matrix.shape[0])
print("non-zero eigenvalues", gram_matrix.shape[0]-1)
```

```
ANS:
size (120, 120)
rank 120
non-zero eigenvalues 119
```

T14. Is the Gram matrix also symmetric? Why?

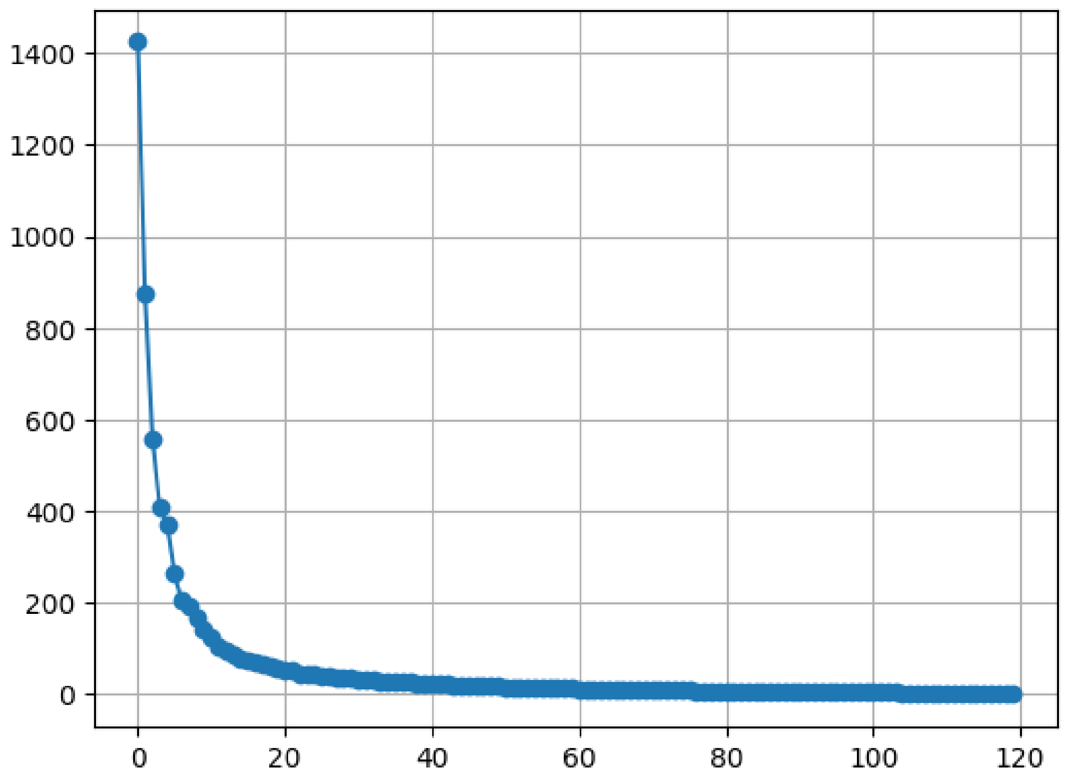
```
ANS: Yes because it is from X multiply transpose of X
```

T15. Compute the eigenvectors and eigenvalues of the Gram matrix, v 0 and λ. Sort the eigenvalues and eigenvectors in descending order so that the first eigenvalue is the highest, and the first eigenvector corresponds to the best direction. How many non-zero eigenvalues are there? If you see a very small value, it is just numerical error and should be treated as zero.

```
# Hint:
https://numpy.org/doc/stable/reference/generated/numpy.linalg.eigh.htm
l

def calculate_eigenvectors_and_eigenvalues(matrix):
    """
    TODO: Calculate eigenvectors and eigenvalues,
    then sort the eigenvalues and eigenvectors in descending order.

    Hint:
https://numpy.org/doc/stable/reference/generated/numpy.linalg.eigh.htm
l
    """

    # INSERT CODE HERE
    eigenvalues, eigenvectors = np.linalg.eigh(matrix)
    eigenvalues = eigenvalues[::-1]
    eigenvectors = eigenvectors[:, ::-1]

    return eigenvalues, eigenvectors

eigenvalues, eigenvectors =
calculate_eigenvectors_and_eigenvalues(gram_matrix)

def test_eigenvalues_eigenvectors():
    # Dot product of an eigenvector pair should equal to zero.
    assert np.round(eigenvectors[10].dot(eigenvectors[20]), 10) == 0.0

    # Check if eigenvalues are sorted.
    assert list(eigenvalues) == sorted(eigenvalues, reverse = True)

test_eigenvalues_eigenvectors()

gram_eigenvalues, gram_eigenvectors =
calculate_eigenvectors_and_eigenvalues(gram_matrix)
print("ANS:")
print("non-zero eigenvalues of gram matrix",
np.sum(gram_eigenvalues>1e-2))
print("shape gram_eigenvalues", gram_eigenvalues.shape)
print("Shape gram_eigenvectors", gram_eigenvectors.shape)

ANS:
non-zero eigenvalues of gram matrix 119
shape gram_eigenvalues (120,)
Shape gram_eigenvectors (120, 120)
```

T16. Plot the eigenvalues. Observe how fast the eigenvalues decrease. In class, we learned that the eigenvalues is the size of the variance for each eigenvector direction. If I want to keep 95% of the variance in the data, how many eigenvectors should I use?

```
# INSERT CODE HERE
plt.plot(np.arange(len(eigenvalues)),eigenvalues,"-",marker="o")
plt.grid(True)
plt.show()

total_variance = gram_eigenvalues.sum()

idx = 0
curr_var = 0
while curr_var < total_variance*0.95 or idx >= len(gram_eigenvalues):
    curr_var += gram_eigenvalues[idx]
    idx += 1
print(idx)
```

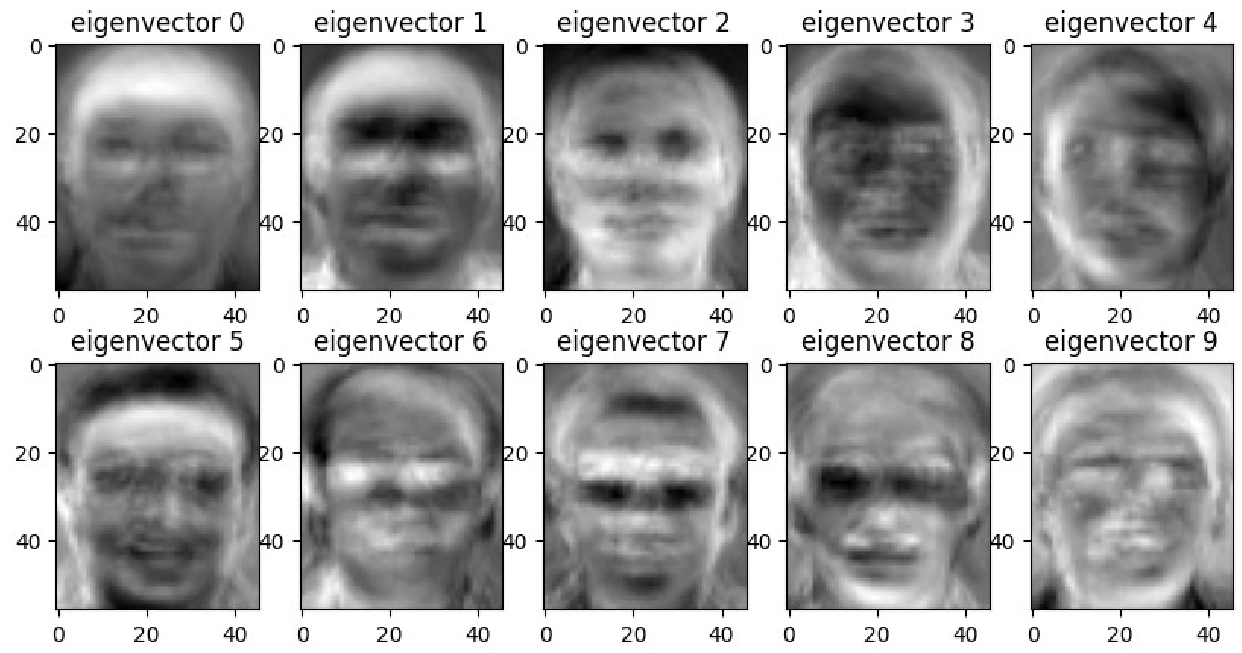


```
64

ANS: 64 eigenvalues
```

T17. Compute $\vec{v}$. Don't forget to renormalize so that the norm of each vector is 1 (you can use numpy.linalg.norm). Show the first 10 eigenvectors as images. Two example eigenvectors are shown below. We call these images eigenfaces (or eigenvoice for speech signals).

```
# TODO: Compute v, then renormalize it.

# INSERT CODE HERE
X_train = T.reshape(120,-1).T
X_hat = X_train - meanface.reshape(-1,1)

v = X_hat @ eigenvectors
v /= np.linalg.norm(v, axis=0)

def test_eignevector_cov_norm(v):
    assert (np.round(np.linalg.norm(v, axis=0), 1) == 1.0).all()

test_eignevector_cov_norm(v)

# TODO: Show the first 10 eigenvectors as images.
plt.figure(figsize=(10, 5))
```

```
for i in range(10):
    plt.subplot(2, 5, i+1)
    plt.title("eigenvector "+str(i))
    plt.imshow(v[:, i].reshape(56,46), cmap='gray')
```



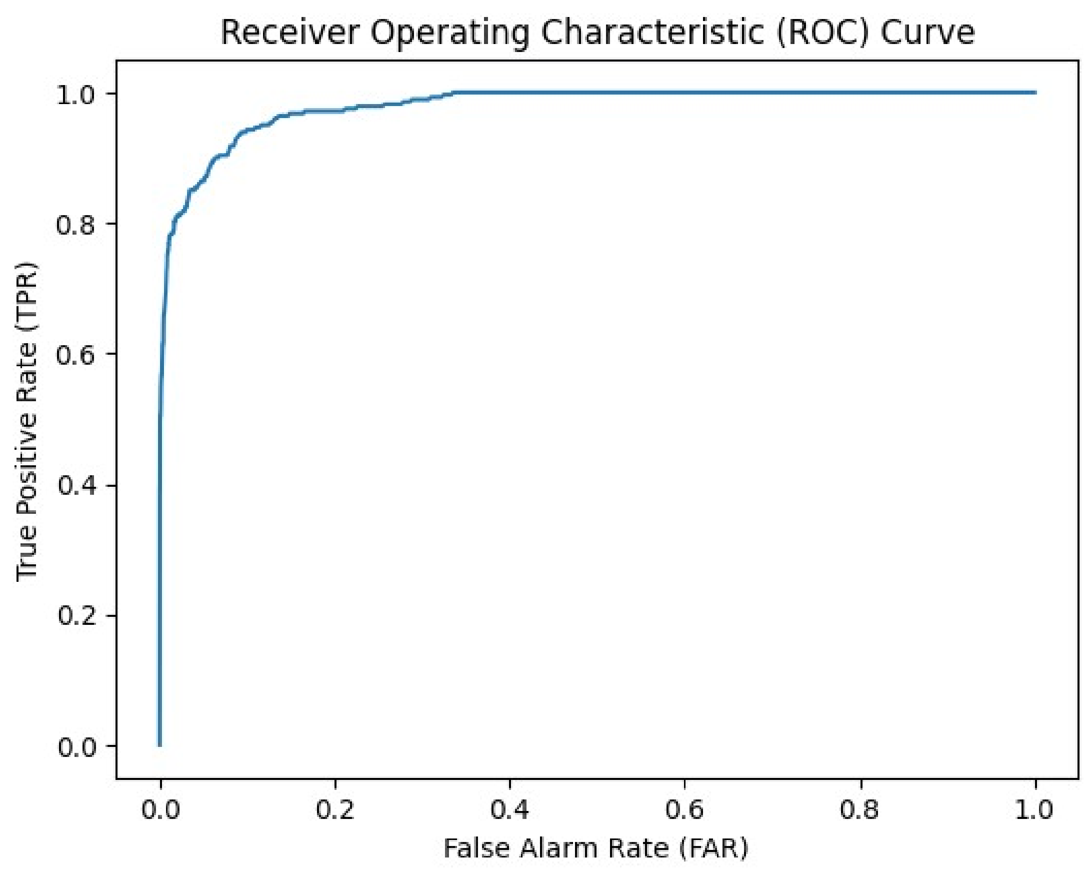
T18. From the image, what do you think the first eigenvector captures? What about the second eigenvector? Look at the original images, do you think biggest variance are capture in these two eigenvectors?

```
ANS: The first eigenvector is white around the hair area, meaning that
the hair is the area with most variance. The second one represents the
hair, eyes, mouths. No because these areas don't cover all the face
areas.
```

T19. Find the projection values of all images. Keep the first k = 10 projection values. Repeat the simple face verification system we did earlier using these projected values. What is the EER and the recall rate at 0.1% FAR?

```
def calculate_projection_vectors(matrix, meanface, v, k=None):
    """
    TODO: Find the projection vectors on v from given matrix and
meanface.
    """

    # INSERT CODE HERE
    if k==None:
```

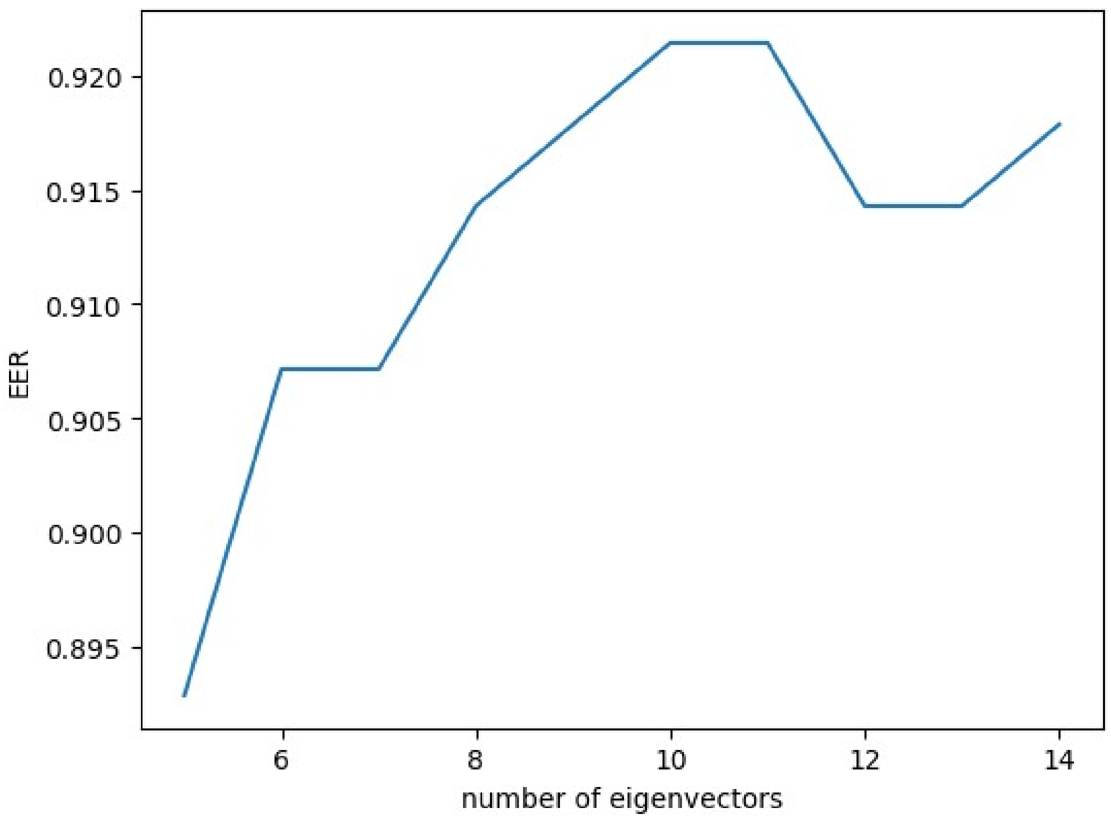
```python
        projection_vectors = np.matmul(matrix-meanface, v[:, :])
    else:
        projection_vectors = np.matmul(matrix-meanface, v[:, :k])

    return projection_vectors

# TODO: Get projection vectors of T and D, then Keep first k
projection values.
k = 10
T_reduced = calculate_projection_vectors(T, meanface, v)[:, :k]
D_reduced = calculate_projection_vectors(D, meanface, v)[:, :k]


def test_reduce_dimension():
    assert T_reduced.shape[-1] == k
    assert D_reduced.shape[-1] == k

test_reduce_dimension()

# TODO: Get similarity matrix of T_reduced and D_reduced
reduced_similarity_matrix = generate_similarity_matrix(T_reduced,
D_reduced)

# TODO: Find EER and the recall rate at 0.1% FAR.
plot_roc(reduced_similarity_matrix)

print("ANS:")
print("EER",tpr_list[np.argmin(np.abs(fpr_list - fnr_list))])
print("Recall",tpr_list[np.argmin(np.abs(fpr_list - 0.001))])
```



```
ANS:
EER 0.9178571428571428
Recall 0.5071428571428571
```

T20. What is the k that gives the best EER? Try k = 5, 6, 7, 8, 9, 10, 11, 12, 13, 14.

```
# INSERT CODE HERE
ks = list(range(5, 15))
T_reduced = calculate_projection_vectors(T,meanface,v)
D_reduced = calculate_projection_vectors(D,meanface,v)
y=[]
for k in ks:
    print("k",k)
    similarity_matrix =
generate_similarity_matrix(T_reduced[:,:k],D_reduced[:,:k])

tpr_list,fpr_list,fnr_list,tnr_list=calculate_roc(similarity_matrix)
    eer=tpr_list[np.argmin(np.abs(fpr_list - fnr_list))]
    print("EER",eer)
    y.append(eer)
    print("Recall",tpr_list[np.argmin(np.abs(fpr_list - 0.001))])
    print()
plt.plot(ks,y)
plt.xlabel("number of eigenvectors")
plt.ylabel("EER")

k 5
EER 0.8928571428571429
Recall 0.25357142857142856

k 6
EER 0.9071428571428571
Recall 0.35

k 7
EER 0.9071428571428571
Recall 0.4107142857142857

k 8
EER 0.9142857142857143
Recall 0.40714285714285714

k 9
EER 0.9178571428571428
Recall 0.45

k 10
EER 0.9214285714285714
Recall 0.5178571428571429

k 11
EER 0.9214285714285714
Recall 0.5035714285714286

k 12
EER 0.9142857142857143
Recall 0.5107142857142857

k 13
EER 0.9142857142857143
Recall 0.5142857142857142

k 14
EER 0.9178571428571428
Recall 0.5035714285714286


Text(0, 0.5, 'EER')
```

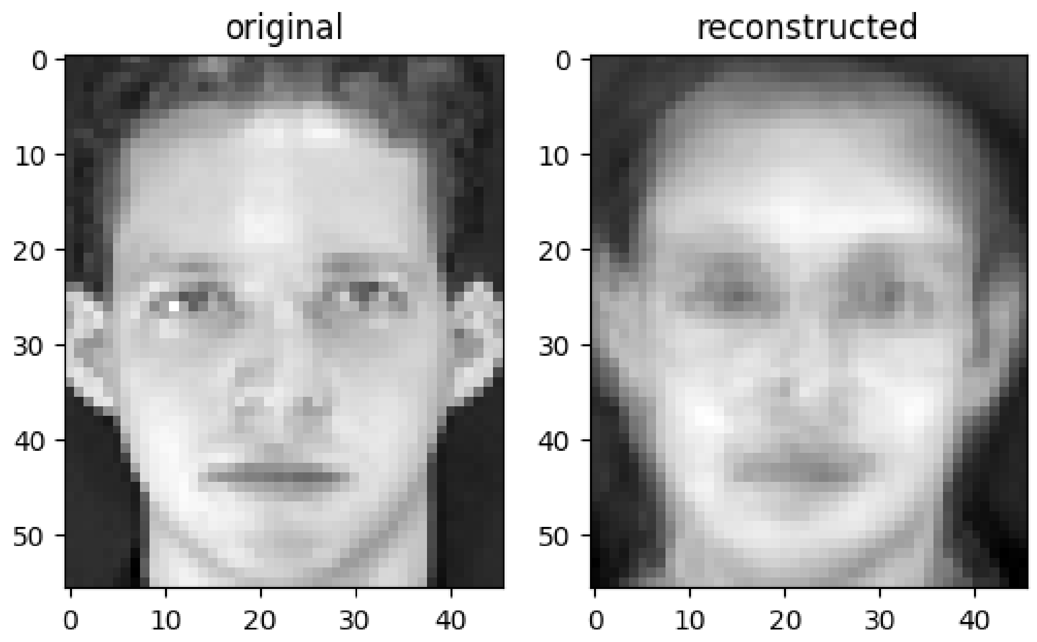


```
ANS: k=10, 11 have the best EER
```

OT2

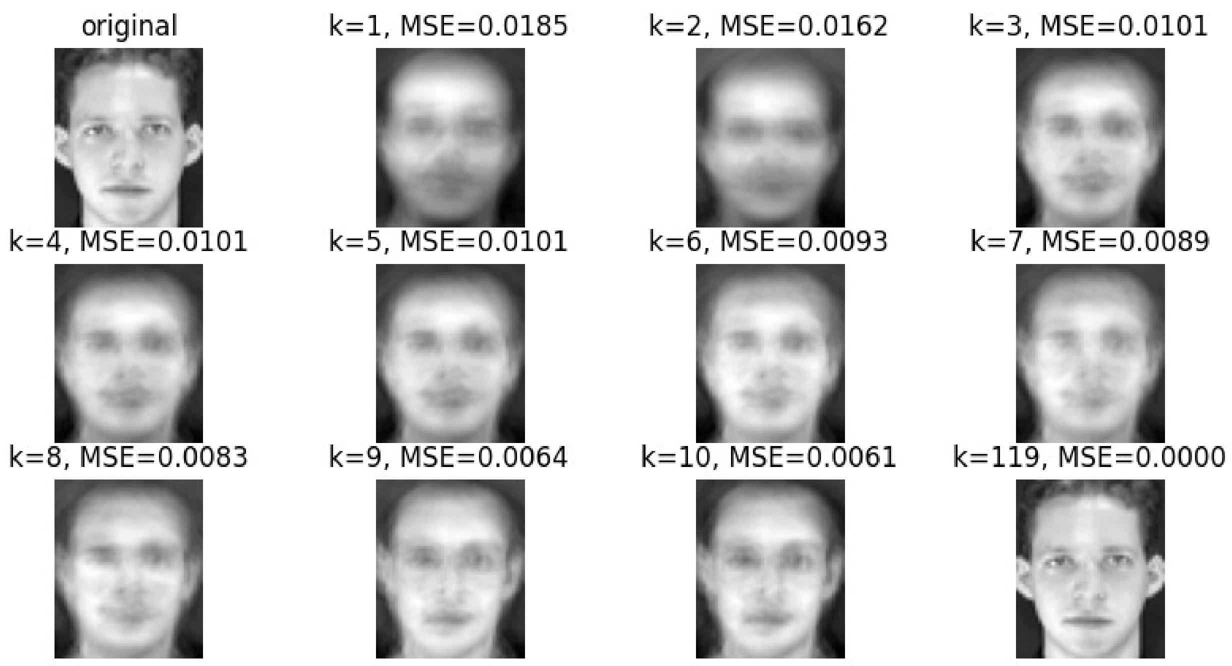
```
def MSE(x, y):
    return ((x-y)**2).mean()

first_image = T[0]
first_image_reduced = calculate_projection_vectors(T[0], meanface, v,
10)
first_image_reconstructed = meanface + np.matmul(first_image_reduced,
v[:, :10].T)

plt.subplot(1, 2, 1)
plt.imshow(first_image.reshape(56, 46), cmap='gray')
plt.title('original')

plt.subplot(1, 2, 2)
plt.imshow(first_image_reconstructed.reshape(56, 46), cmap='gray')
plt.title('reconstructed')

print("MSE", MSE(first_image, first_image_reconstructed))
```

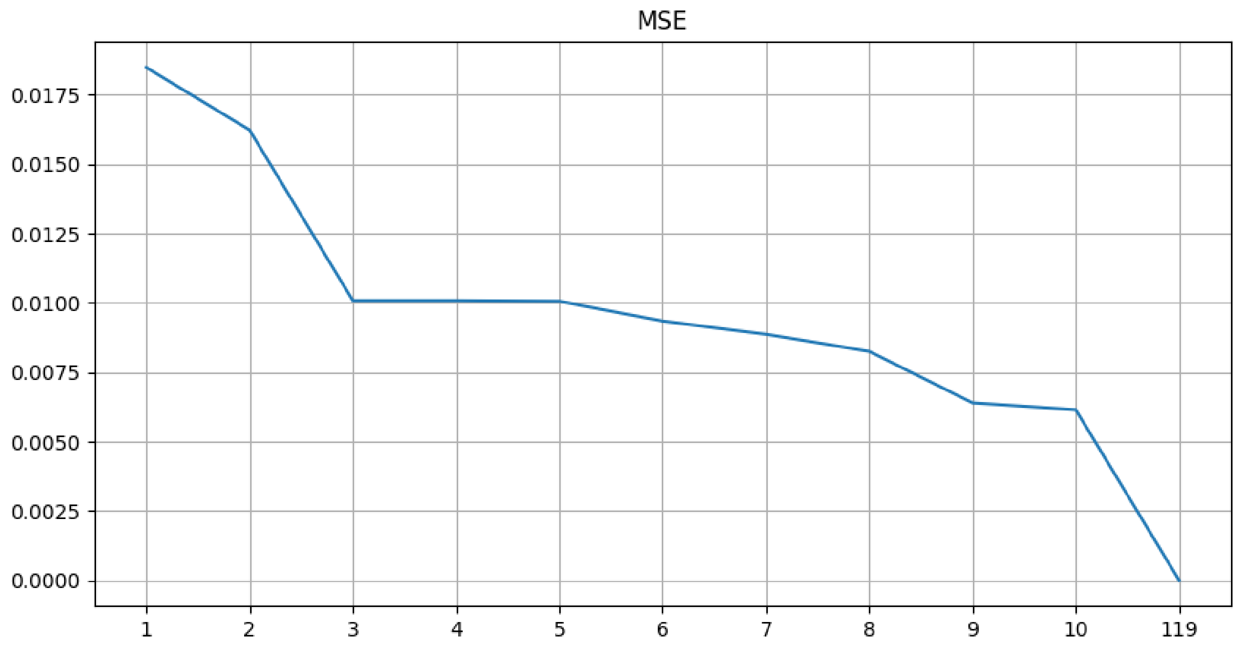
```
MSE 0.006148335016488305
```



OT3

```
x = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 119])
y = np.zeros_like(x, dtype=float)
plt.figure(figsize=(10, 5))
plt.subplot(3, 4, 1)
plt.imshow(T[0].reshape(56, 46), cmap='gray')
plt.title('original')
plt.axis('off')

for idx, k in enumerate(x):
    plt.subplot(3, 4, idx+2)
    image_reduced = calculate_projection_vectors(T[0], meanface, v, k)
    image_reconstructed = meanface + np.matmul(image_reduced, 
v[:, :k].T)
    mse = MSE(T[0], image_reconstructed)

    y[idx] = mse
    plt.imshow(image_reconstructed.reshape(56, 46), cmap='gray')
    plt.title(f'k={k}, MSE={mse:.4f}')
    plt.axis('off')

plt.show()

plt.figure(figsize=(10, 5))
plt.plot(np.arange(len(y)), y)
plt.xticks(np.arange(len(x)), x)
plt.grid()
plt.title('MSE')
plt.show()
```





OT4

```
size_per_original_image = 56*46
print("total", size_per_original_image*1e6)
print("compressed",  1e6*10*4 + 10*56*46*4 + 56*46*4)

total 2576000000.0
compressed 40113344.0
```

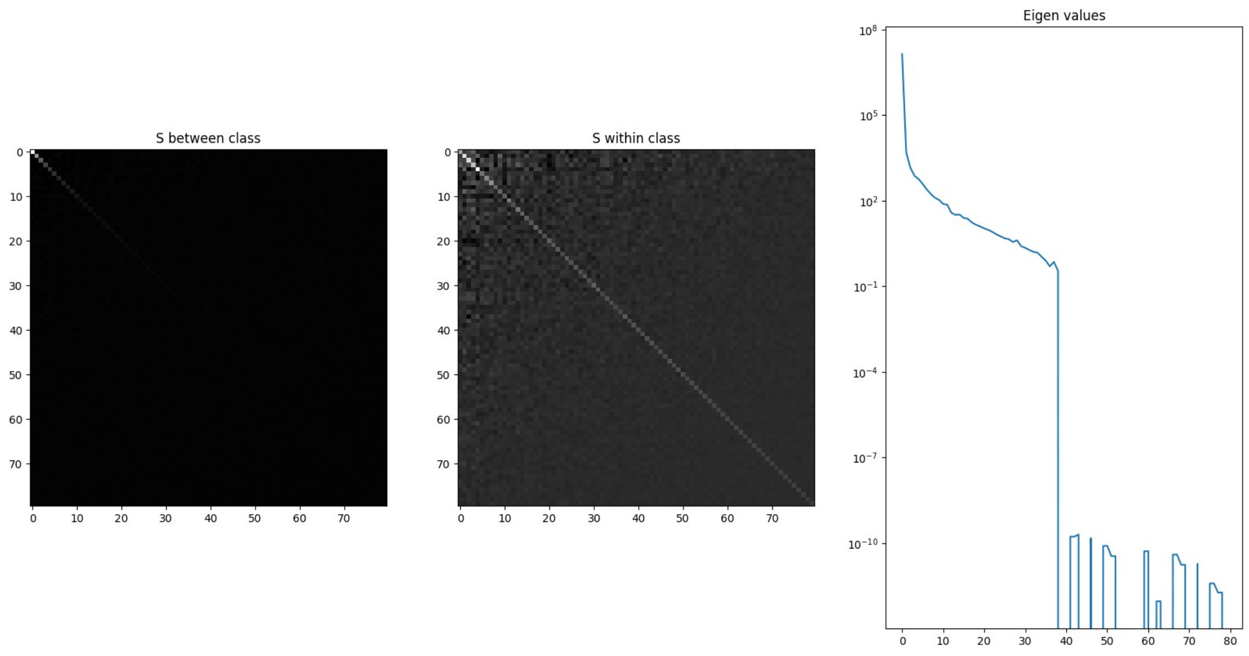
T21. In order to assure that S_W is invertible we need to make sure that S_W is full rank. How many PCA dimensions do we need to keep in order for S_W to be full rank? (Hint: How many dimensions does S_W have? In order to be of full rank, you need to have the same number of linearly independent factors)

```
ANS: 80, (120, 80), (280, 80)

# TODO: Define dimension of PCA.
n_dim = T.shape[0] - 40
print("Dimension", n_dim)

# TODO: Find PCA of T and D with n_dim dimension.
T_reduced = calculate_projection_vectors(T, meanface, v, n_dim)
D_reduced = calculate_projection_vectors(D, meanface, v, n_dim)
print(T_reduced.shape, D_reduced.shape)

Dimension 80
(120, 80) (280, 80)
```

T22. Using the answer to the previous question, project the original input to the PCA subspace. Find the LDA projections. To find the inverse, use −1 numpy.linalg.inv. Is S_W S_B symmetric? Can we still use numpy.linalg.eigh? How many non-zero eigenvalues are there?

```
# TODO: Find the LDA projection.
T_reduced_by_class = T_reduced.reshape((40, 3, -1))
class_mean = T_reduced_by_class.mean(axis=1)
all_mean = class_mean.mean(axis=0).reshape(1, -1)

s_b = np.array([(class_mean[i]-all_mean).T @(class_mean[i]-all_mean)
for i in range(class_mean.shape[0])]).sum(axis=0)

s_wi = np.array([(T_reduced_by_class[i]-class_mean[i]).T @
(T_reduced_by_class[i]-class_mean[i]) for i in
range(class_mean.shape[0])])
s_w = s_wi.sum(axis=0)

LDA = np.linalg.inv(s_w) @ s_b
print("LDA is symmetric", np.allclose(LDA, LDA.T))
LDA_eigenvalues, LDA_eigenvectors = np.linalg.eig(LDA)

LDA_eigenvectors = LDA_eigenvectors.real
LDA_eigenvalues = LDA_eigenvalues.real

plt.figure(figsize=(20, 10))
plt.subplot(1, 3, 1)
plt.imshow(s_b, cmap='gray')
plt.title("S between class")
plt.subplot(1, 3, 2)
plt.imshow(s_w, cmap='gray')
plt.title("S within class")
plt.subplot(1, 3, 3)
plt.title("Eigen values")
plt.plot(LDA_eigenvalues)
plt.yscale('log')

plt.show()
```

```
LDA is symmetric False
```



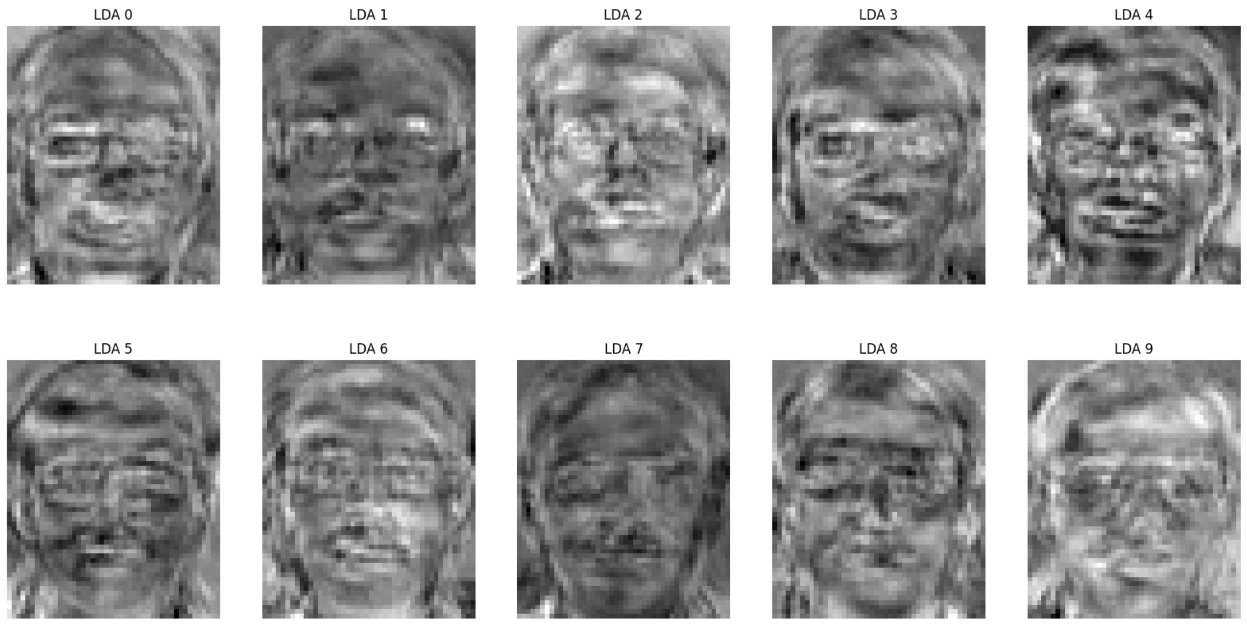
```
# TODO: Find how many non-zero eigenvalues there are.

np.where(LDA_eigenvalues>1e-20, 1, 0).sum()
```

```
60
```

```
ANS: 39
```

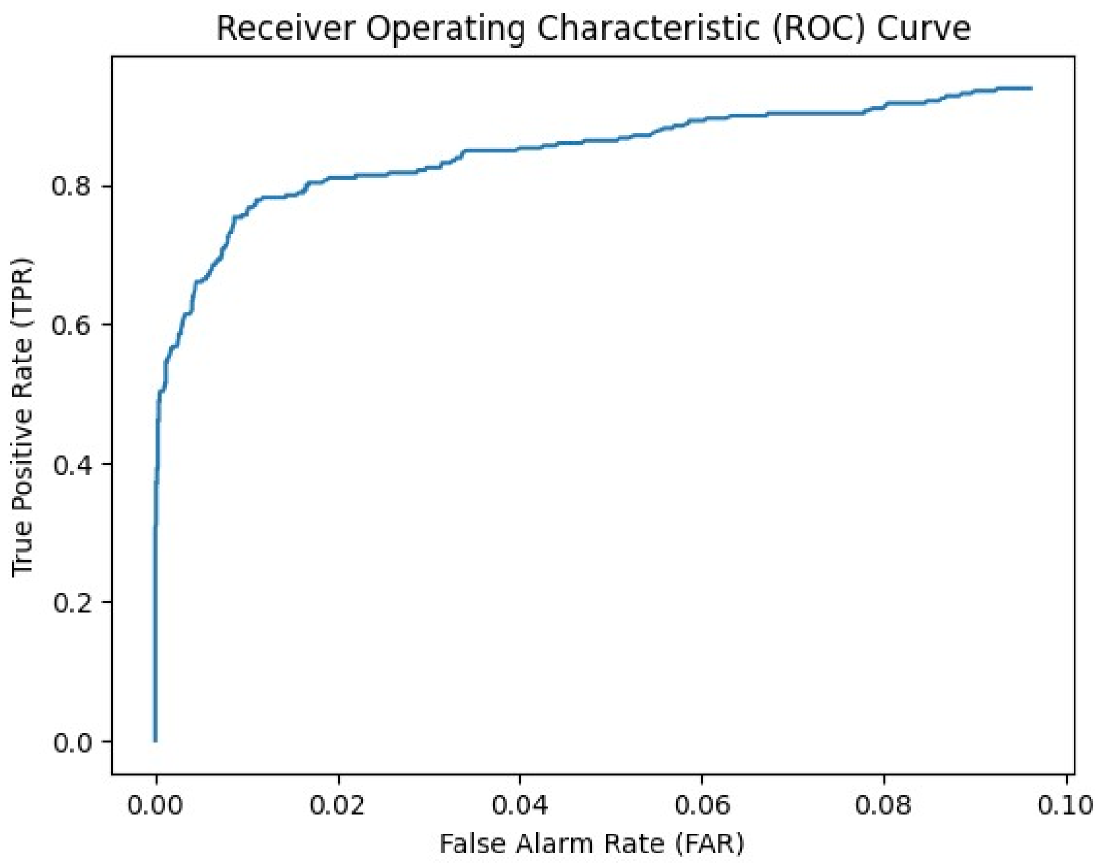
T23. Plot the first 10 LDA eigenvectors as images (the 10 best projections). Note that in this setup, you need to convert back to the original image space by using the PCA projection. The LDA eigenvectors can be considered as a linear combination of eigenfaces. Compare the LDA projections with the PCA projections.

```
# INSERT CODE HERE

best_10_LDA = LDA_eigenvectors[:, :10]
T_LDA = np.matmul(T_reduced, best_10_LDA)
T_eigenface = np.matmul(v[:, :n_dim], best_10_LDA)

D_LDA = np.matmul(D_reduced, best_10_LDA)
D_eigenface = np.matmul(v[:, :n_dim], best_10_LDA)

plt.figure(figsize=(20, 10))
for idx in range(10):
    plt.subplot(2, 5, idx+1)
    plt.imshow(T_eigenface[:, idx].reshape(56, 46), cmap='gray')
    plt.title(f"LDA {idx}")
    plt.axis('off')

plt.show()
```



T24. The combined PCA+LDA projection procedure is called fisherface. Calculate the fisherfaces projection of all images. Do the simple face verification experiment using fisherfaces. What is the EER and recall rate at 0.1% FAR?

```
# INSERT CODE HERE
reduced_similarity_matrix = generate_similarity_matrix(T_LDA, D_LDA)
eer = plot_roc(reduced_similarity_matrix)
```

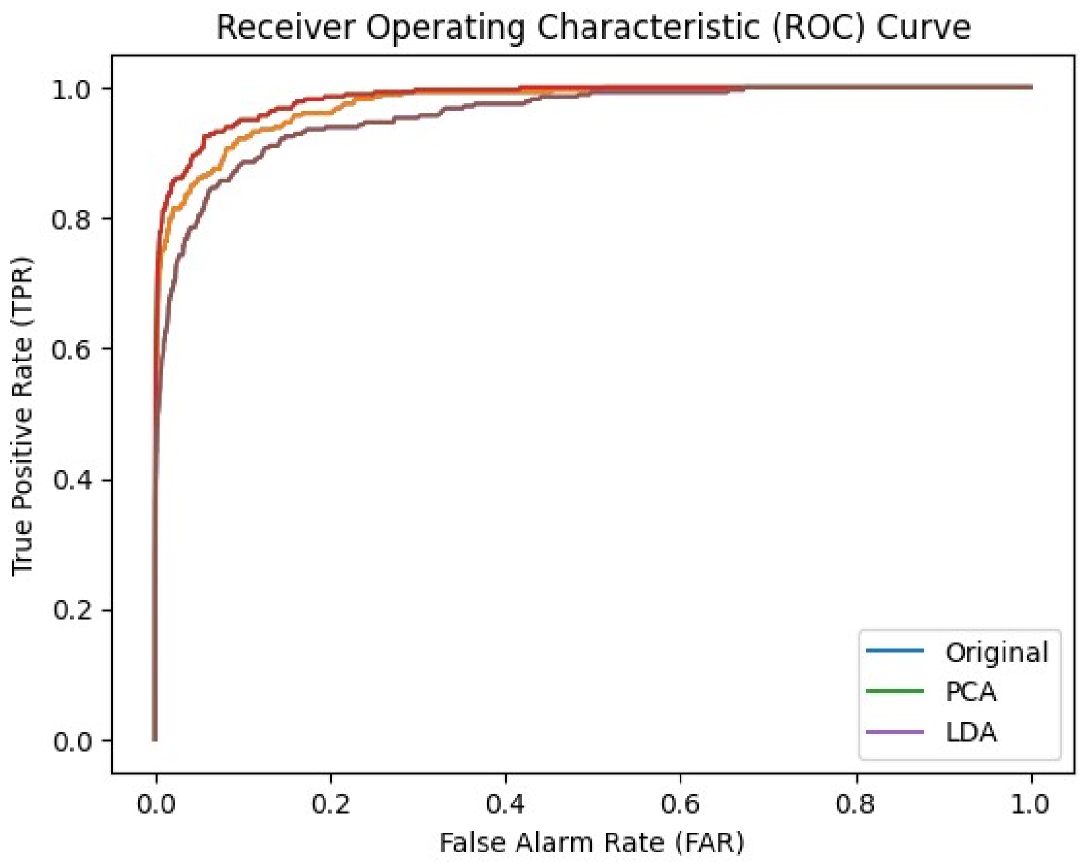


```
print("ANS:")
print("EER",tpr_list[np.argmin(np.abs(fpr_list - fnr_list))])
print("Recall",tpr_list[np.argmin(np.abs(fpr_list - 0.001))])

ANS:
EER 0.9178571428571428
Recall 0.5071428571428571

ANS:
```

T25.Plot the RoC of all three experiments (No projection, PCA, andFisher) on the same axes. Compare and contrast the three results. Submit yourwriteup and code on MyCourseVille.

```
# INSERT CODE HERE

setup = {
    'Original': (T, D),
    'PCA': (T_reduced, D_reduced),
    'LDA': (T_LDA, D_LDA)
}
eer = []

for label, (curr_T, curr_D) in setup.items():
    similarity_matrix = generate_similarity_matrix(curr_T, curr_D)
    plot_roc(similarity_matrix, label=label, show=False)

    print(label)
    print("EER",tpr_list[np.argmin(np.abs(fpr_list - fnr_list))])
    print("Recall",tpr_list[np.argmin(np.abs(fpr_list - 0.001))])
    print()

plt.legend()
plt.show()
```

```
Original
EER 0.9071428571428571
Recall 0.5428571428571428

PCA
EER 0.9321428571428572
Recall 0.6071428571428571

LDA
EER 0.8857142857142857
Recall 0.40714285714285714
```


Receiver Operating Characteristic (ROC) Curve
True Positive Rate (TPR)
False Alarm Rate (FAR)
0.0
0.2
0.4
0.6
0.8
1.0
Original
PCA
LDA

```
ANS:
```