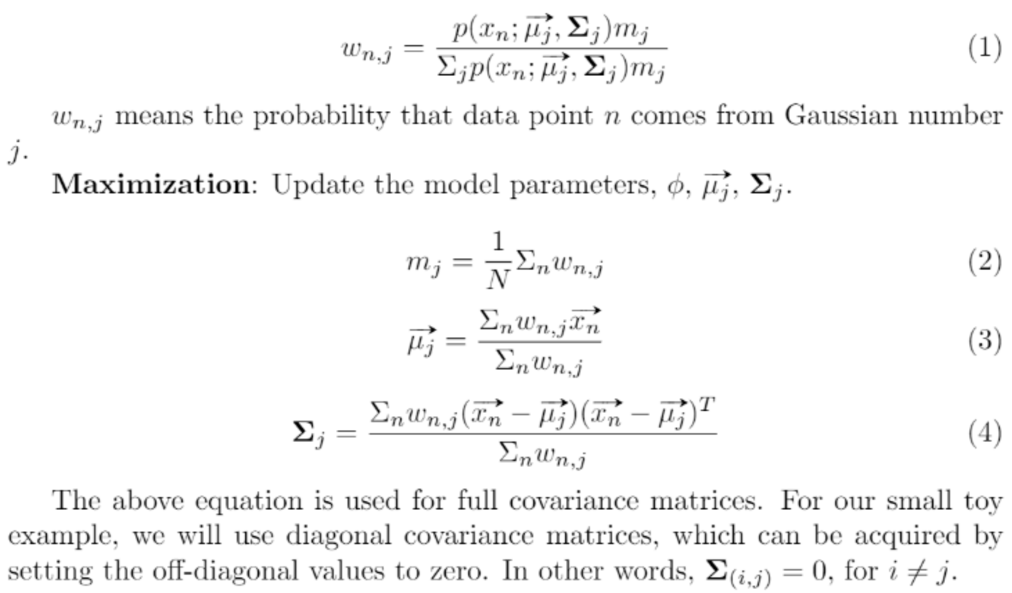
Hello Soft Clustering (GMM)

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.

In [ ]:

```
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.multiv
from scipy.stats import multivariate_normal
```

In [ ]:

```
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
        self.cov_params = cov_params

        # Initiailize iteration.
        self.n_iter = 0

    def estimation_step(self,data, mixture_weight, mean_params, cov_param
        prob  = np.array([multivariate_normal(mean=mean_params[i], cov=cov
        prob_m = np.dot(np.eye(mixture_weight.shape[0]) * mixture_weight,
        w = prob_m / (np.sum(prob_m, axis=0))
        return w


    def maximization_step(self, data, w):
        """
        TODO: Perform maximization step.
            (Update parameters in this GMM model.)
        """
        self.mixture_weight = (1/data.shape[0])*np.sum(w, axis=1)

        self.mean_params = (np.dot(w, data).T/ (w.sum(axis=1)+1e-10)).T

        self.cov_params = np.array([np.dot((w[i].reshape(-1,1)*(data-self
        self.cov_params = (self.cov_params+1e-10) * np.eye(self.cov_param


    def get_log_likelihood(self, data):
        """
        TODO: Compute log likelihood.
        """
        log_likelihood = np.log(np.array([multivariate_normal(mean=self.m

        return log_likelihood

    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):

            w = self.estimation_step(data, self.mixture_weight, self.mean_

            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
```

In [ ]:

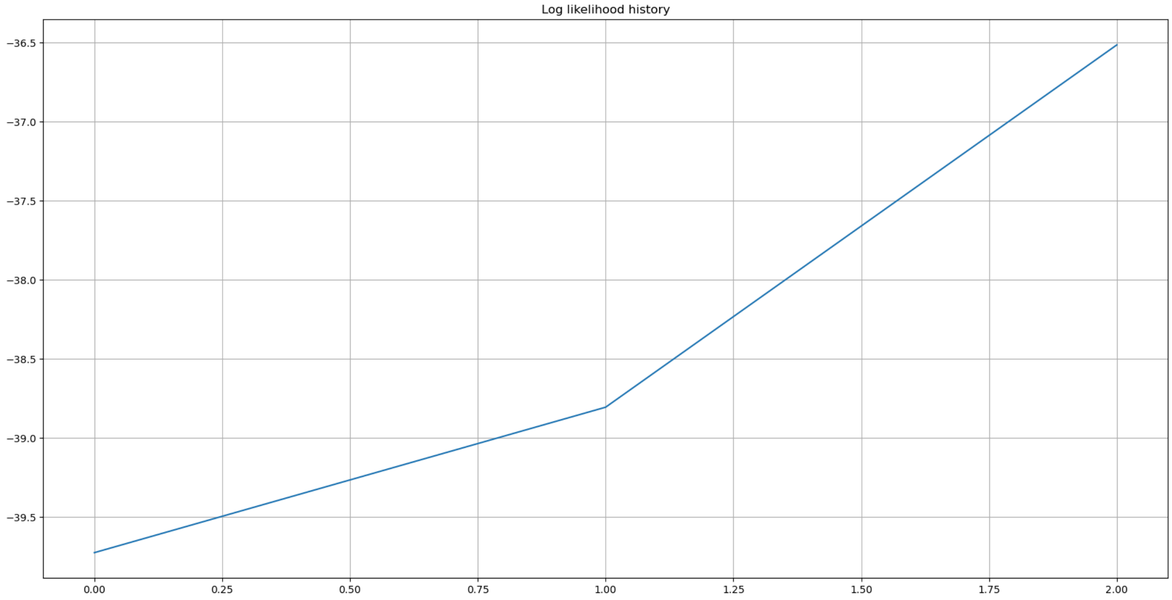
```
# TODO
def plot_log_prob(log_prob_list):
    plt.figure(figsize=(20, 10))
    plt.grid()
    plt.title('Log likelihood history')
    plt.plot(np.arange(len(log_prob_list)), log_prob_list)
    plt.show()
```

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

In [ ]:

```
num_iterations = 3
num_mixture = 3
mixture_weight = np.array([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,
data = np.vstack([X,Y]).T

gmm = GMM(mixture_weight, mean_params, cov_params)
log_prob_list_3 = gmm.perform_em_iterations(data, num_iterations)
plot_log_prob(log_prob_list_3)
```

```
Initialization
m :
 [1 1 1]
mu :
 [[ 3.  3.]
 [ 2.  2.]
 [-3. -3.]]
covariance matrix :
 [[[1. 0.]
  [0. 1.]]

 [[1. 0.]
  [0. 1.]]

 [[1. 0.]
  [0. 1.]]]
-------------------------------------------------------------
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]]]
-------------------------------------------------------------
```



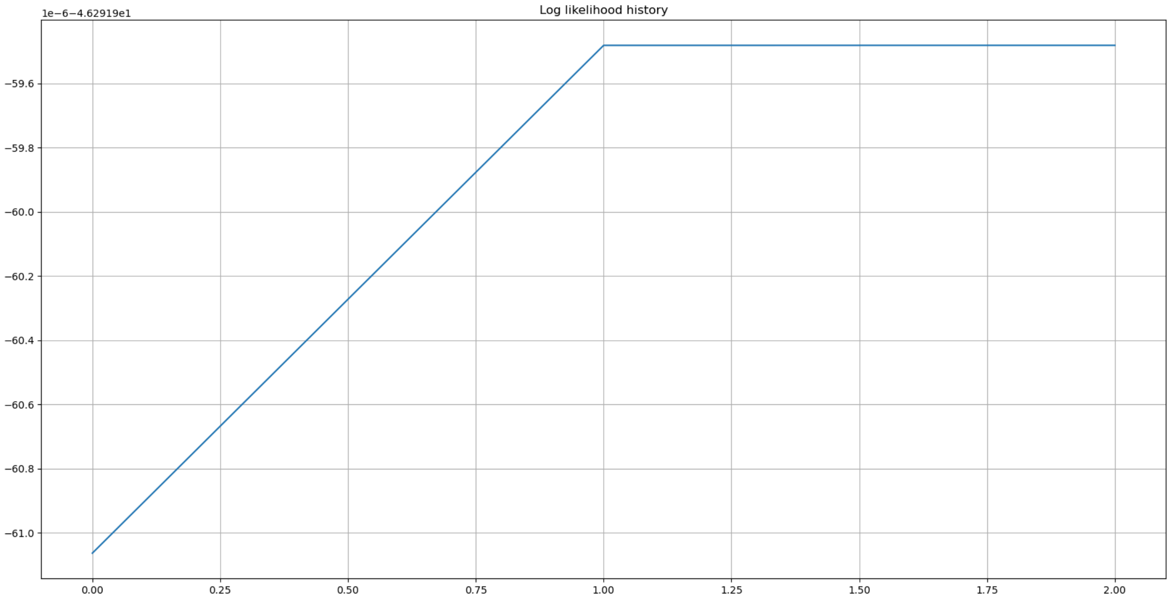
```
ANS : Yes it looklike going to coverge every iteration
```

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.

In [ ]:

```
num_mixture = 2
mixture_weight = [1] * num_mixture
mixture_weight = np.array(mixture_weight)

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

# INSERT CODE HERE
gmm = GMM(mixture_weight, mean_params, cov_params)
log_prob_list_2 = gmm.perform_em_iterations(data, num_iterations)

plot_log_prob(log_prob_list_2)
```

```
Initialization
m :
 [1 1]
mu :
 [[ 3.  3.]
 [-3. -3.]]
covariance matrix :
 [[[1. 0.]
  [0. 1.]]

 [[1. 0.]
  [0. 1.]]]
-------------------------------------------------------------
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 -4.6665114 ]]
covariance matrix :
 [[[6.91946372 0.        ]
  [0.         5.8927741 ]]

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

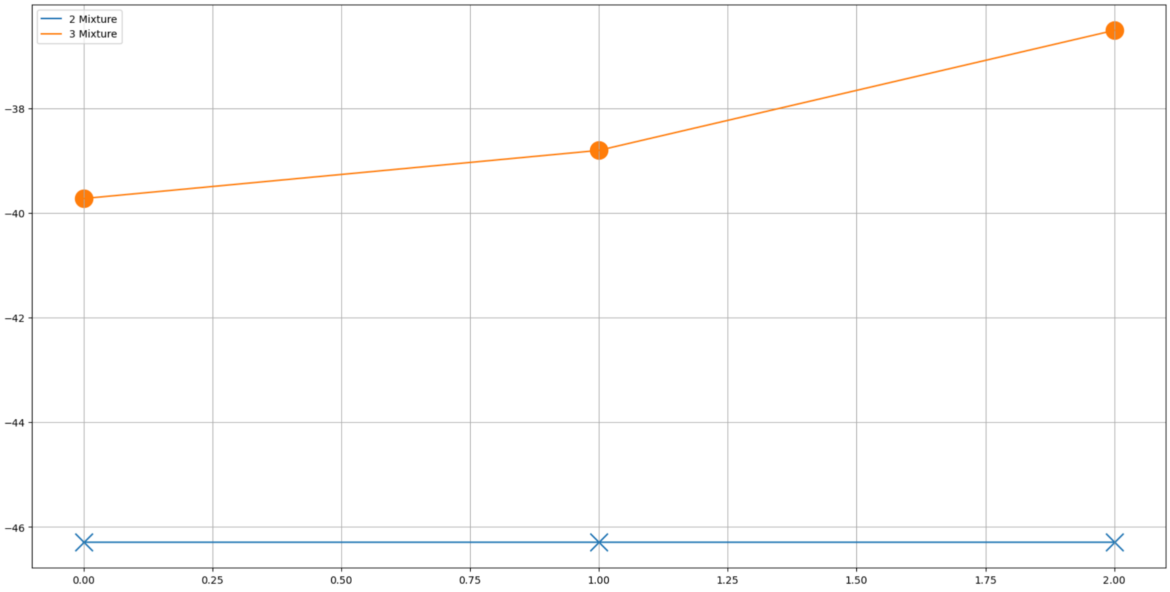


T4. Plot the log likelihood of the model given the data after each EM step. Compare the log likelihood between using two mixtures and three mixtures. Which one has the better likelihood?

In [ ]:

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

plt.figure(figsize=(20, 10))
plt.plot(np.arange(len(log_prob_list_2)), log_prob_list_2, label='2 Mixtu
plt.scatter(np.arange(len(log_prob_list_2)), log_prob_list_2, marker='x',
plt.plot(np.arange(len(log_prob_list_3)), log_prob_list_3, label='3 Mixtu
plt.scatter(np.arange(len(log_prob_list_3)), log_prob_list_3, marker='o',

plt.legend()
plt.grid()
```

```
[-39.725993156389336, -38.80573197942934, -36.512976308824065]
```



```
ANS : Likelihood with 3 mixture is better Since it
converge.
```

The face database

In [ ]:

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

In [ ]:

```
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
```

Out[ ]:

```
(40, 10)
```

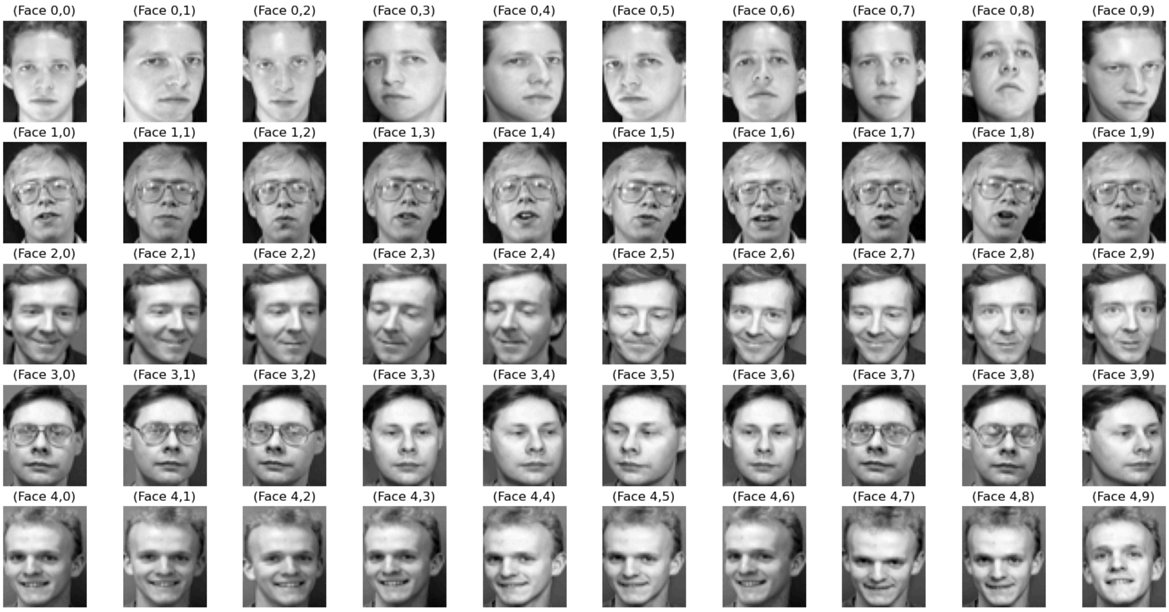
Preprocess xf

In [ ]:

```
xf = np.zeros((data_size[0], data_size[1], data['facedata'][0,0].shape[0]
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])
```

In [ ]:

```
# Example: Ploting face image.
plt.figure(figsize=(20, 10))
for i in range(5):
  for j in range(10):
    plt.subplot(5, 10, i*10+j+1)
    plt.imshow(xf[i,j], cmap='gray')
    plt.title(f'(Face {i},{j})')
    plt.axis('off')
```



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?

In [ ]:

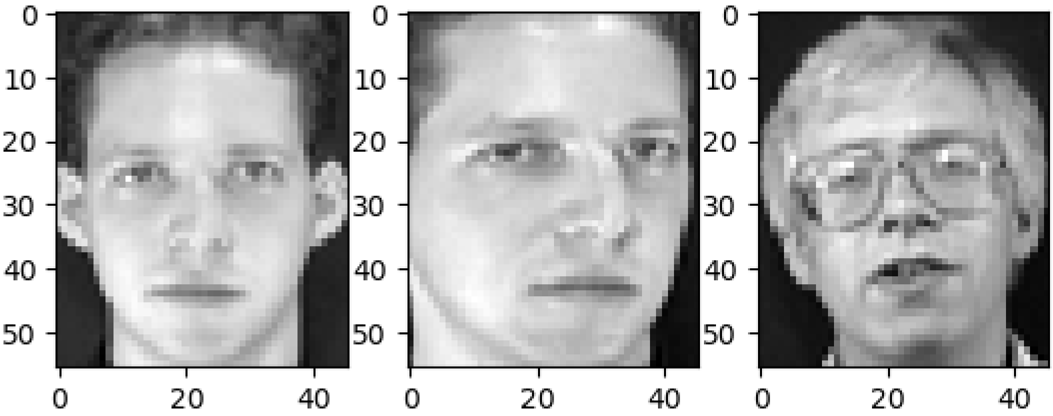
```
def L2_dist(x1, x2):
    distance = np.sqrt(np.sum((x1 - x2)**2))
    return distance

# 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(

test_L2_dist()

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

```
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
```

In [ ]:

```
# TODO: Show why does the numbers make sense
print('Euclidean distance between xf[0,0] and xf[0,0] is', L2_dist(xf[0,0
print('Euclidean distance between xf[0,0] and xf[0,1] is', L2_dist(xf[0,0
print('Euclidean distance between xf[1,0] and xf[0,1] is', L2_dist(xf[1,0
print('Euclidean distance between xf[1,0] and xf[0,0] is', L2_dist(xf[1,0
plt.subplot(1,3,1)
plt.imshow(xf[0,0], cmap = 'gray')
plt.subplot(1,3,2)
plt.imshow(xf[0,1], cmap = 'gray')
plt.subplot(1,3,3)
plt.imshow(xf[1,0], cmap = 'gray')
plt.show()
```

```
Euclidean distance between xf[0,0] and xf[0,0] is 0.0
Euclidean distance between xf[0,0] and xf[0,1] is 10.037616294165492
Euclidean distance between xf[1,0] and xf[0,1] is 11.517134319336666
Euclidean distance between xf[1,0] and xf[0,0] is 8.173295099737281
```



```
ANS : Euclidian values seems make sense when comparing
difference of images in overall. This can apply only when
image has same dimension. It's sensitive to rotation and
distance of objects in images so it might not good for
facial recognition.
```

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

In [ ]:

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

    Example:
        Input shape: (people_index, image_index, image_shape[0], image_sh
        Output shape: (people_index*image_index, image_shape[0]*image_sha
    """

    people_index, image_index, image_shape_0, image_shape_1 = matrix.shap

    # Reshape the matrix to merge people and image dimensions
    reshaped_matrix = matrix.copy().reshape(people_index * image_index, i

    return reshaped_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].
    """
    distance_matrix = np.sqrt(np.sum((A[:, np.newaxis] - B)**2, axis=2))

    return distance_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

test_generate_similarity_matrix()
```

In [ ]:

```
#TODO: Show similariry matrix between T and D.
T = organize_shape(xf[:, :3])
D = organize_shape(xf[:, 3:])
similarity_matrix = generate_similarity_matrix(organize_shape(xf[:,:3]),
similarity_matrix
```

Out[ ]:

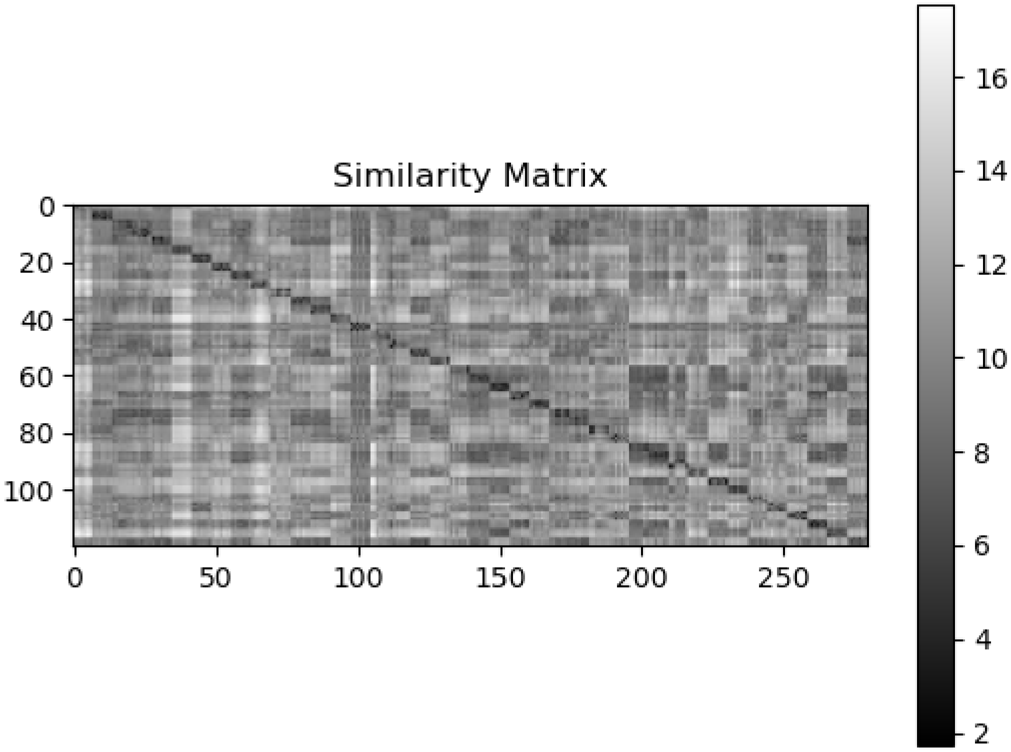
```
array([[10.36960631,  9.84869463,  8.99622801, ...,  9.89638826,
         9.36151948, 10.66062617],
       [11.24987522,  7.4172114 ,  9.88066979, ..., 10.9694266 ,
        10.90238961, 10.93630575],
       [10.22209276,  9.41321639,  9.29987742, ...,  9.99689456,
         9.94499521, 10.41008147],
       ...,
       [11.50315003, 10.77719551, 11.45361976, ...,  5.5842077 ,
         9.25718815,  7.93519019],
       [ 9.82401314, 10.09262219,  9.9006693 , ...,  6.75074775,
         7.45046954,  8.90196597],
       [ 9.75449186,  9.98745234, 10.16210857, ...,  6.51894822,
         7.88934623,  9.55831251]])
```

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?

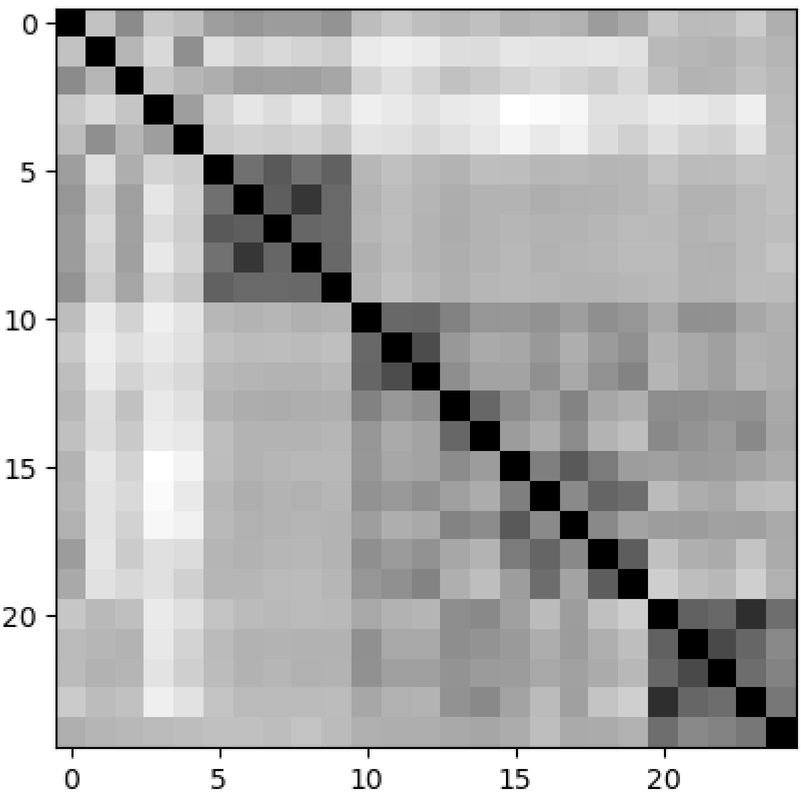
In [ ]:

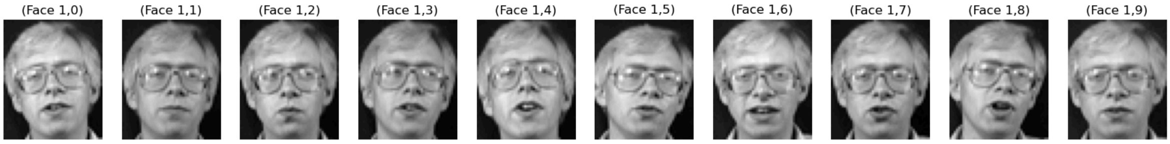
```
# INSERT CODE HERE
# Show the similarity matrix
plt.imshow(similarity_matrix, cmap='gray')
plt.colorbar()
plt.title("Similarity Matrix")
plt.show()

# Extract the relevant squares
person_2_square = similarity_matrix[5:10, 5:10]
person_1_square = similarity_matrix[:3, :3]

# Calculate the average similarity for each square
person_2_avg_similarity = np.mean(person_2_square)
person_1_avg_similarity = np.mean(person_1_square)

print("Average similarity for person number 2:", person_2_avg_similarity)
print("Average similarity for person number 1:", person_1_avg_similarity)

eT = organize_shape(xf[:5, :5])
```

```
plt.imshow(generate_similarity_matrix(eT, eT), cmap='gray')


plt.figure(figsize=(20, 10))
for i in range(2):
    for j in range(10):
        plt.subplot(2, 10, i*10+j+1)
        plt.imshow(xf[i,j], cmap='gray')
        plt.title(f'(Face {i},{j})')
        plt.axis('off')
```



```
Average similarity for person number 2: 9.637038405598823
Average similarity for person number 1: 9.633052437208354
```





ANS : The black square between [5:10,5:10] in the similarity matrix suggests that the images from person number 2 have a low similarity with each other. This means that the images of person number 2 have significant variations or differences among them. On the other hand, the patterns from person number 1 in the similarity matrix suggest that the images from person number 1 have a high similarity with each other. This indicates that the images of person number 1 have a consistent pattern or similarity among them.

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?

In [ ]:

```
def evaluate_performance(similarity_matrix, threshold):
    """
    TODO: Calculate true positive rate and false alarm rate from given si
    """
    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() < thresh
            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_alarm_rate = fp/(tn+fp)


    return true_pos_rate, false_alarm_rate

# Quick check
# (true_pos_rate, false_neg_rate) should be (0.9928571428571429, 0.335073
evaluate_performance(similarity_matrix, 9.5)
```

Out[ ]:

```
(0.9928571428571429, 0.33507326007326005)
```

In [ ]:

```
# INSERT CODE HERE
evaluate_performance(similarity_matrix, 10)
```

Out[ ]:

```
(0.9964285714285714, 0.4564102564102564)
```

```
    ANS: True Positve rate is 0.9964285714285714 and False
Alarm is 0.4564102564102564
```

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).

In [ ]:

```
def calculate_roc(input_mat):
    """
    TODO: Calculate a list of true_pos_rate and a list of false_neg_rate
    """
    tpr_list = []
    far_list = []

    n_iter = 2000
    recall = None
    diff_recal = float('inf')
    eer = 0
    diff = float('inf')

    rag = input_mat.max() - input_mat.min()
    minv = input_mat.min() + 0.20*rag
    maxv = input_mat.max() - 0.20*rag
    for threshold in np.linspace(minv, maxv, n_iter):
        tpr, far = evaluate_performance(input_mat, threshold)
        if abs(far-0.001) < diff_recal:
            recall = tpr
            diff_recal = abs(far-0.001)
        if abs(tpr+far-1) < diff:
            diff = abs(tpr+far-1)
            eer = tpr
        tpr_list.append(tpr)
        far_list.append(far)

    return tpr_list, far_list, eer, recall

def plot_roc(input_mat, label=None):
    """
    TODO: Plot RoC Curve from a given matrix.
    """
    tpr_list, far_list, eer, recall = calculate_roc(input_mat)
    plt.title("Plot of ROC")
    if label:
        plt.plot(far_list, tpr_list, label=label)
    else:
        plt.plot(far_list, tpr_list)
    plt.xlabel('False Alarm Rate')
    plt.ylabel('True Positive Rate')
    print(f"EER : {eer}")
    print(f"Recall at 0.1%% FAR : {recall}")

    return eer
```

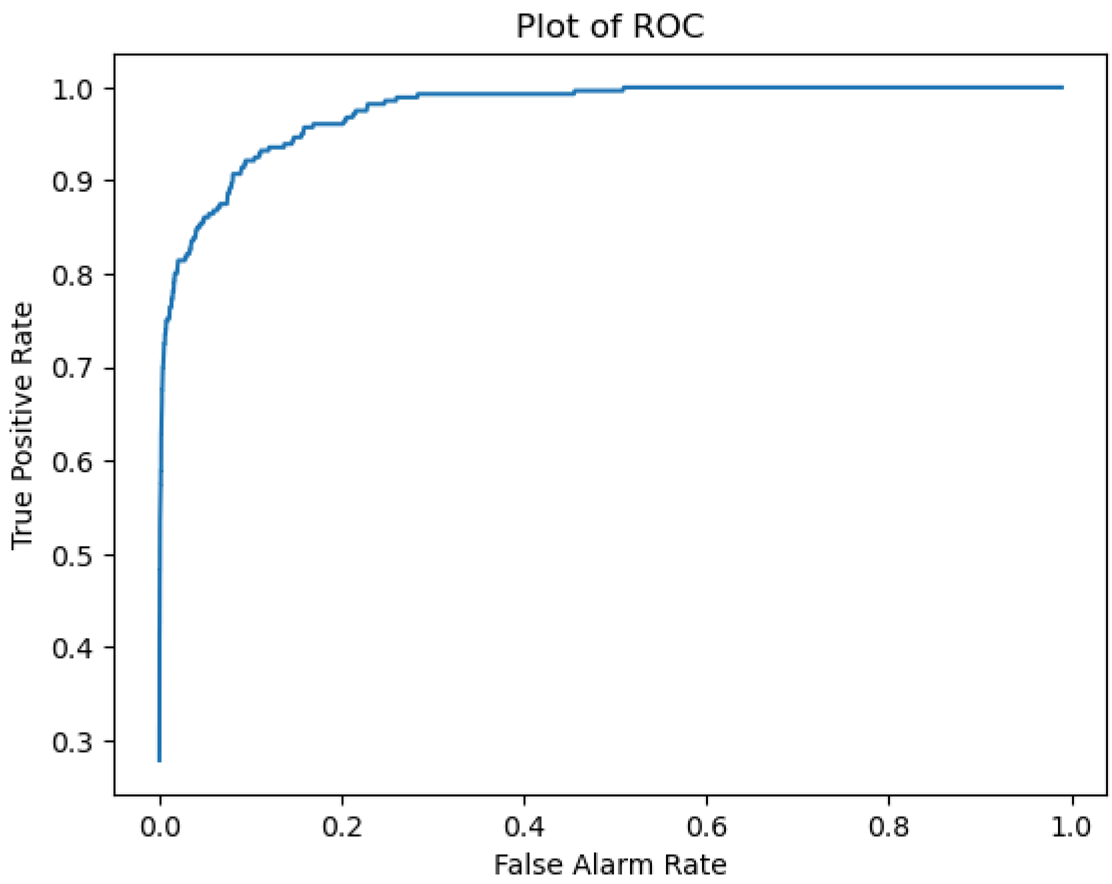
In [ ]:

```
# INSERT CODE HERE
plot_roc(similarity_matrix)
```

```
EER : 0.9107142857142857
Recall at 0.1%% FAR : 0.5464285714285714
```

Out[ ]:

```
0.9107142857142857
```

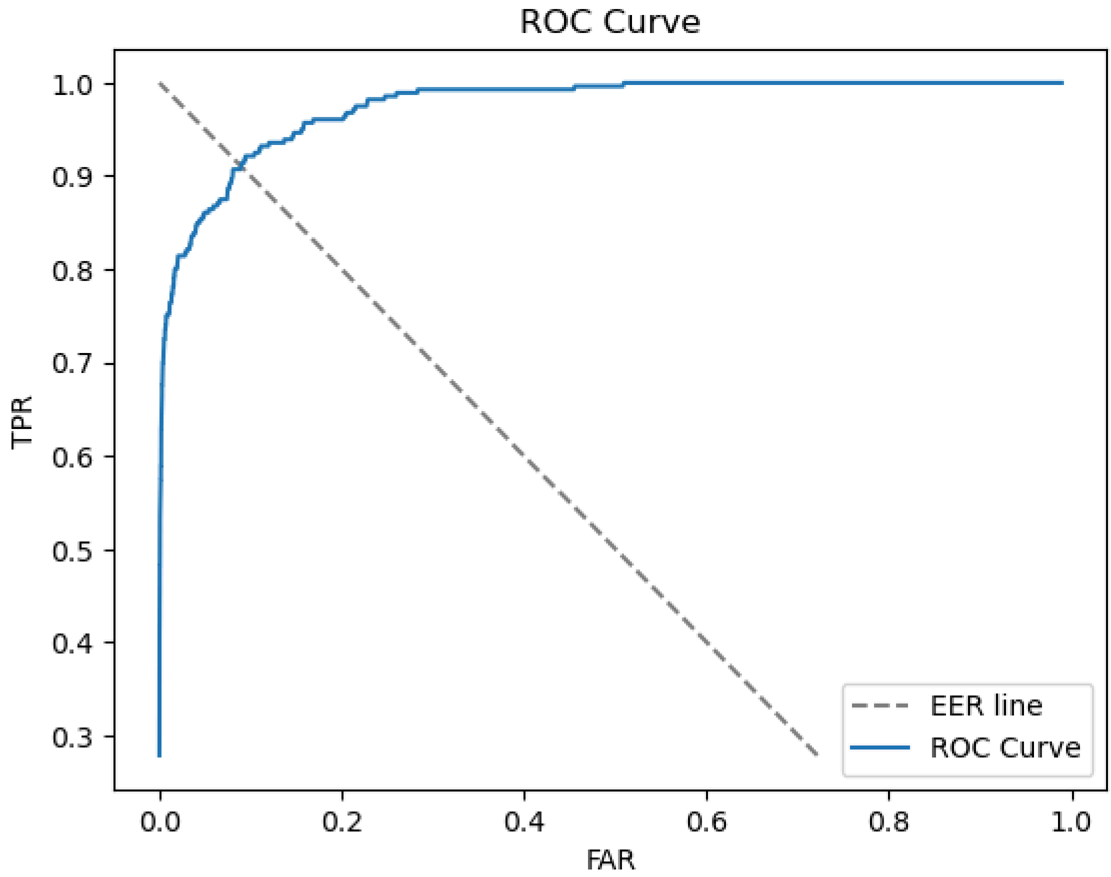


```
ANS: I decide to choose minimum and maximum threshold by
min/max +/- 0.20 * range
```

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)

In [ ]:

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

# EER should be either 0.9071428571428571 or 0.9103759398496248 depending
# Recall rate at 0.1% false alarm rate should be 0.5428571428571428.
def plot_roc(input_mat, display=True,with_err=True,label=None):
    """
    Plot RoC Curve from a given matrix.
    """
    tpr_list, far_list, err, recall = calculate_roc(input_mat)

    # Calculate recall rate at FAR

    if display and with_err:
        plt.plot(1-np.array(tpr_list) , tpr_list, color='gray', linestyle

    # Plot ROC curve
    if display:
        plt.plot(far_list, tpr_list, label=label if label else 'ROC Curve
        plt.xlabel('FAR')
        plt.ylabel('TPR')
        plt.title('ROC Curve')
        plt.legend()
        plt.show()
    print("ERR:", err)
    print("Recall at FAR:", recall)
    return err, recall

# Plot ROC curve with EER and recall rate at 0.1% FAR
plot_roc(similarity_matrix)
```



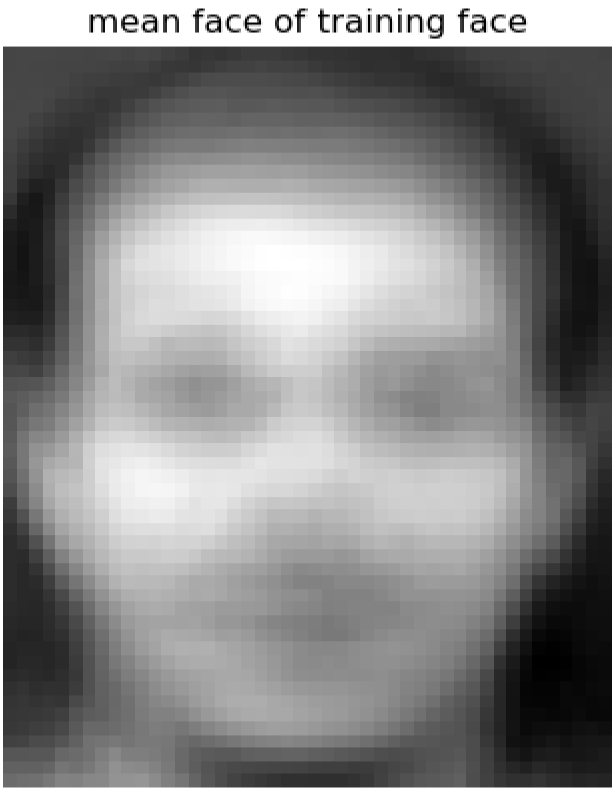
```
ERR: 0.9107142857142857
Recall at FAR: 0.5464285714285714
```

```
Out[ ]: (0.9107142857142857, 0.5464285714285714)
```

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.

```
In [ ]: # INSERT CODE HERE
meanface = T.mean(axis=0)

plt.title('mean face of training 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?

In [ ]:

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

```
Covariance matrix size : (2576,2576)
Rank of Covariance matrix : 119
```

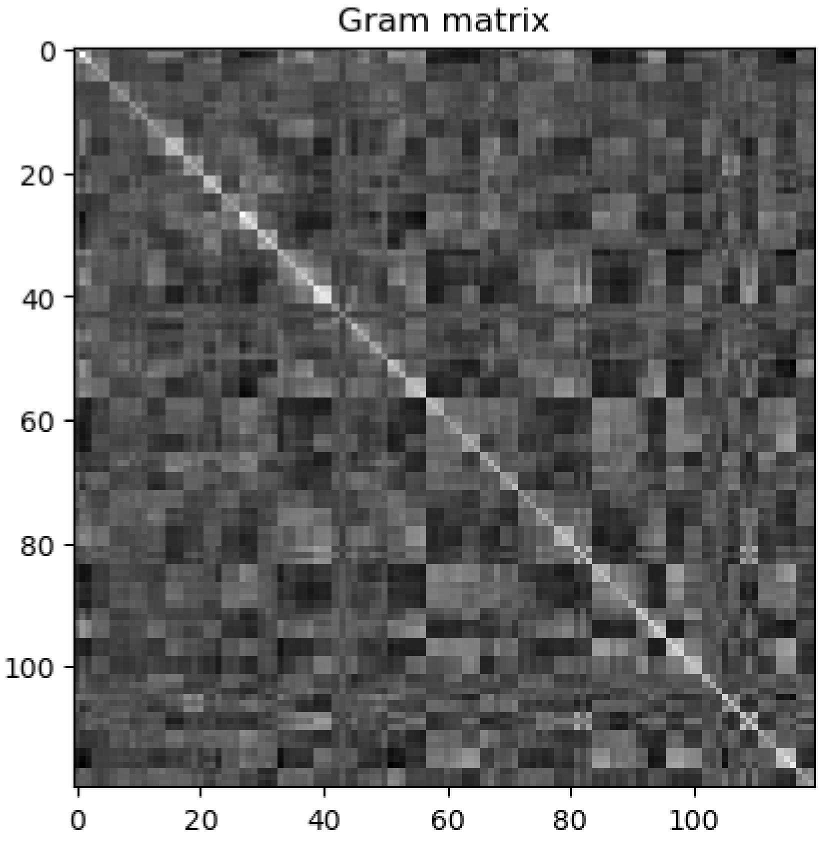
    ANS:

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?

In [ ]:

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

plt.title(f'Gram matrix')
plt.imshow(gram_matrix, cmap='gray')

print(f"""
Size of gram matrix : {gram_matrix.shape}
Rank of gram matrix : {gram_matrix.shape[0]}
Expected non-zero eigenvalues : {gram_matrix.shape[0]-1}
""".strip())
```

```
Size of gram matrix : (120, 120)
Rank of gram matrix : 120
Expected non-zero eigenvalues : 119
```



ANS:

T14. Is the Gram matrix also symmetric? Why?

ANS:

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.

In [ ]:

```python
# Hint: https://numpy.org/doc/stable/reference/generated/numpy.linalg.eig

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.e
    """
    # INSERT CODE HERE
    eigenvalues, eigenvectors = np.linalg.eigh(matrix)

    # Sort the eigenvalues and eigenvectors in descending order
    idx = np.argsort(eigenvalues)[::-1]
    eigenvalues = eigenvalues[idx]
    eigenvectors = eigenvectors[:, idx]

    # Treat very small values as zero
    eigenvalues[np.abs(eigenvalues) < 1e-20] = 0.0
    eigenvectors[np.abs(eigenvectors) < 1e-20] = 0.0

    return eigenvalues, eigenvectors

eigenvalues, eigenvectors = calculate_eigenvectors_and_eigenvalues(gram_m

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()
```

In [ ]:

```
gram_evalues, gram_evectors = calculate_eigenvectors_and_eigenvalues(gram
print(f"""
Amount of non-zero eigenvalues of gram matrix : {np.sum(gram_evalues>1e-2
""".strip())
```

```
Amount of non-zero eigenvalues of gram matrix : 119
```

ANS:

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?

In [ ]:

```
# INSERT CODE HERE
total_variance = gram_evalues.sum()

top95_idx = 0
curr_var = 0
while curr_var < total_variance*0.95 or top95_idx >= len(gram_evalues):
    curr_var += gram_evalues[top95_idx]
    top95_idx += 1
top95_idx -= 1

print(f"""
Total variance : {total_variance:3,.2f}
95% of variace use {top95_idx+1} eigenvalues
""".strip())
```

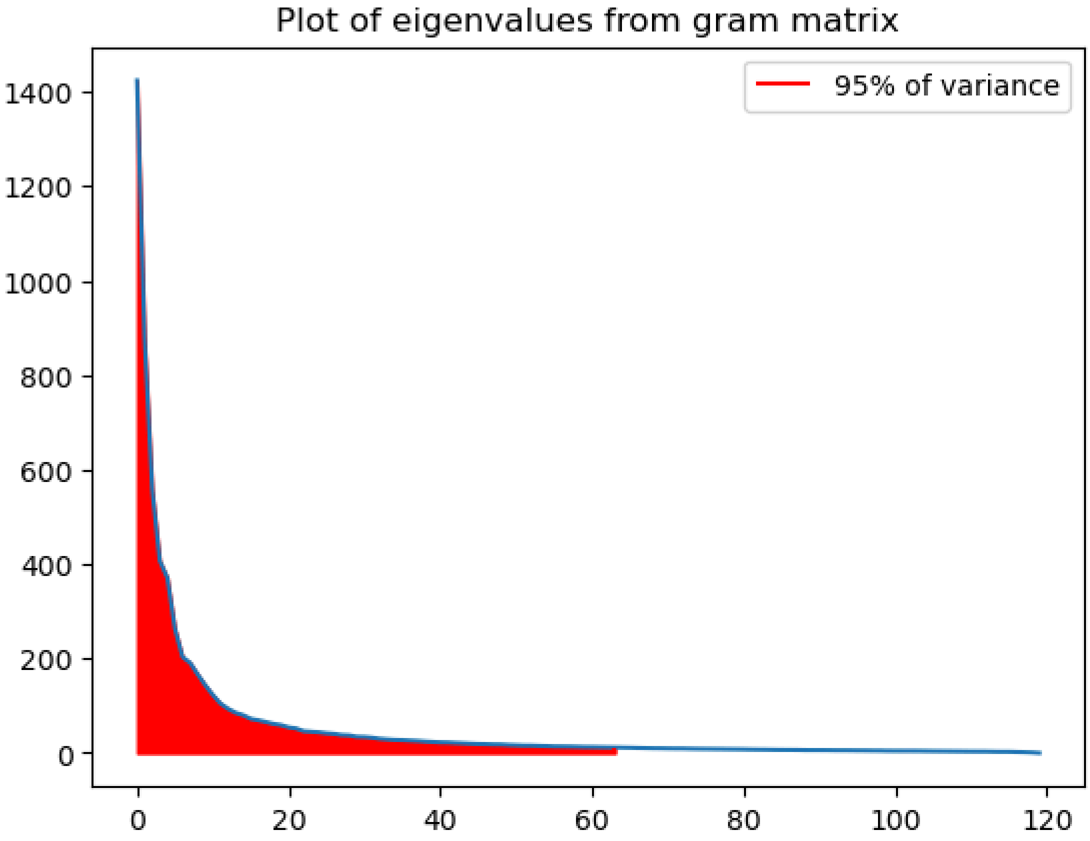
```
Total variance : 6,853.80
95% of variace use 64 eigenvalues
```

In [ ]:

```python
plt.title("Plot of eigenvalues from gram matrix")
plt.plot(gram_evalues)
plt.vlines(top95_idx,
           ymin=0,
           ymax=gram_evalues[top95_idx],
           color='r',
           linestyle='-',
           label='95% of variance')
plt.fill_between(np.arange(top95_idx+1),
                 gram_evalues[:top95_idx+1],
                 color='r')
plt.legend()
```

Out[ ]:

```
<matplotlib.legend.Legend at 0x7950e7285f70>
```



ANS:

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).

In [ ]:

```python
# TODO: Compute v, then renormalize it.
v = np.matmul((T-meanface).T, gram_evectors)
v = v/np.linalg.norm(v, axis=0)
```

```
v.shape
```

Out[ ]:

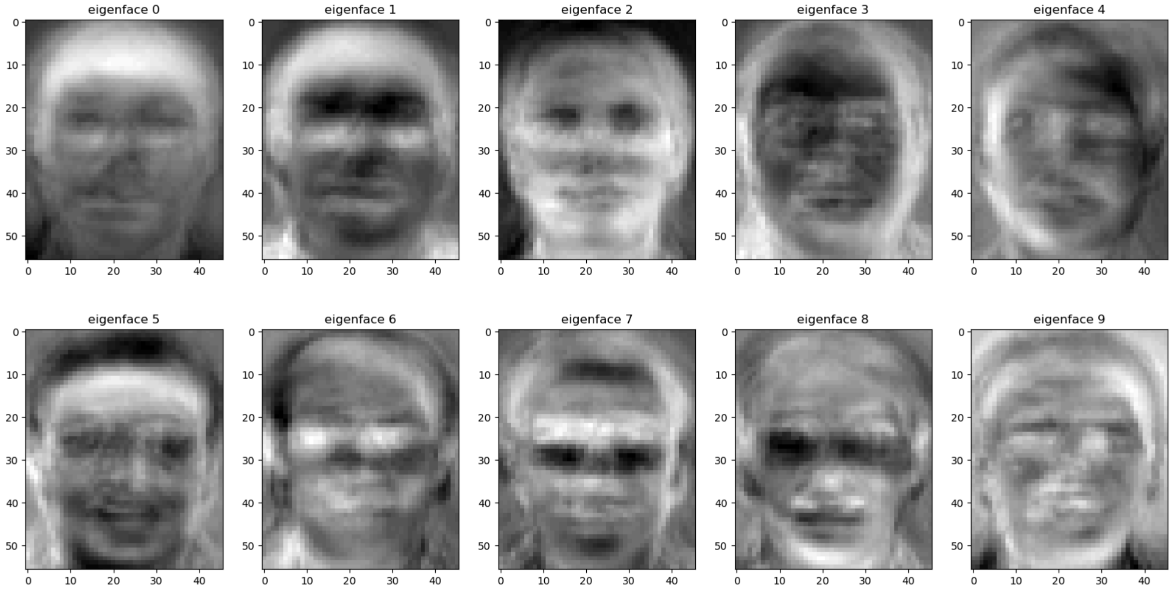
```
(2576, 120)
```

In [ ]:

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

test_eignevector_cov_norm(v)
```

In [ ]:

```
# TODO: Show the first 10 eigenvectors as images.
plt.figure(figsize=(20, 10))
for i in range(10):
    plt.subplot(2, 5, i+1)
    plt.title(f"eigenface {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:

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?

In [ ]:

```
def calculate_projection_vectors(matrix, meanface, v, k):
    """
    TODO: Find the projection vectors on v from given matrix and meanface
    """
    projection_vectors = np.matmul(matrix-meanface, v[:, :k])

    return projection_vectors
```

In [ ]:

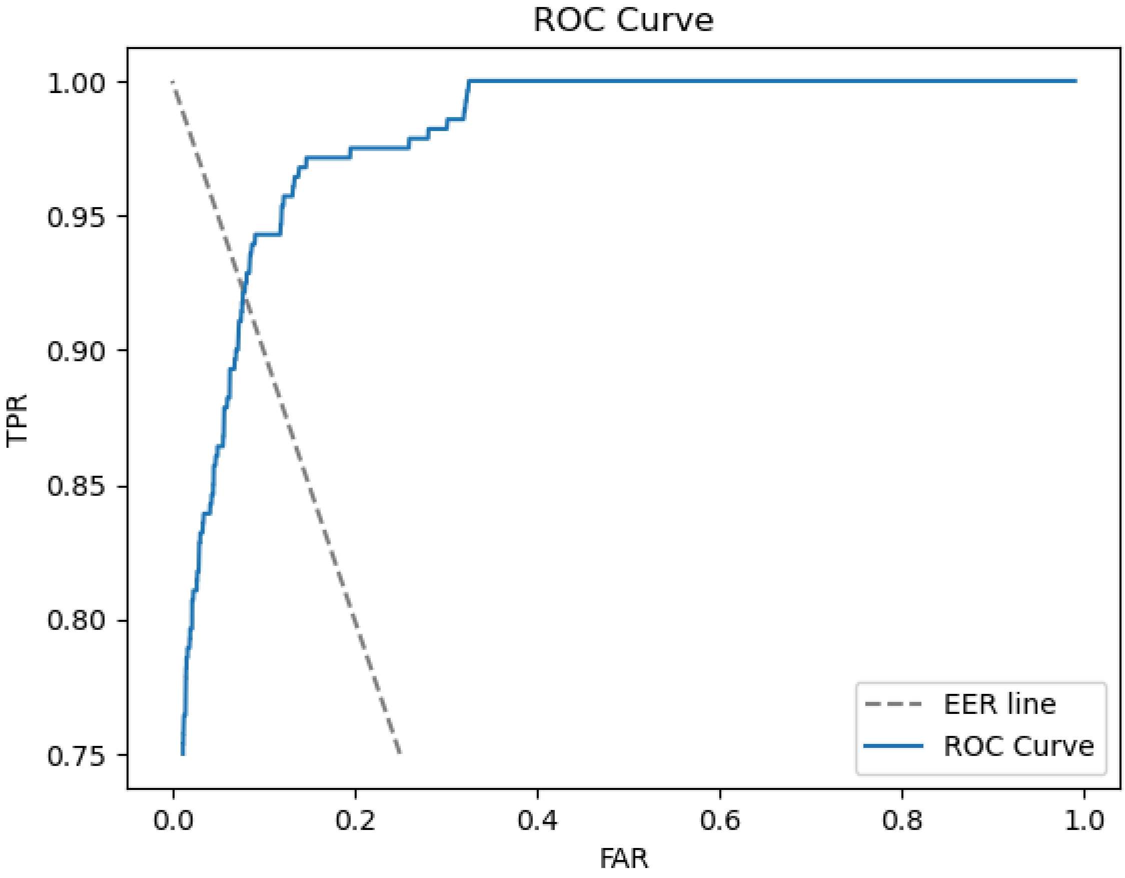
```
# TODO: Get projection vectors of T and D, then Keep first k projection v
k = 10
T_reduced = calculate_projection_vectors(T, meanface, v, k)
D_reduced = calculate_projection_vectors(D, meanface, v, k)


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

test_reduce_dimension()
```

In [ ]:

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

In [ ]:

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



```
ERR: 0.9214285714285714
Recall at FAR: 0.75
```

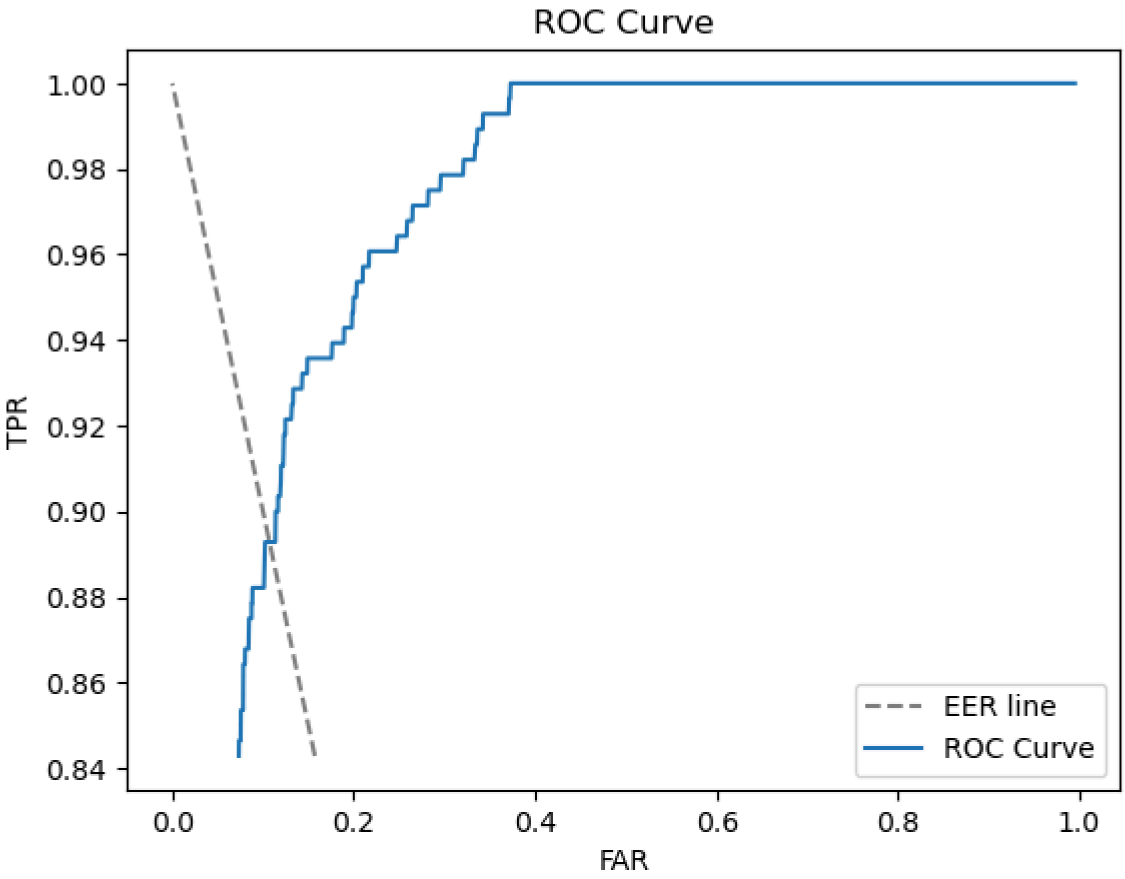
ANS:

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

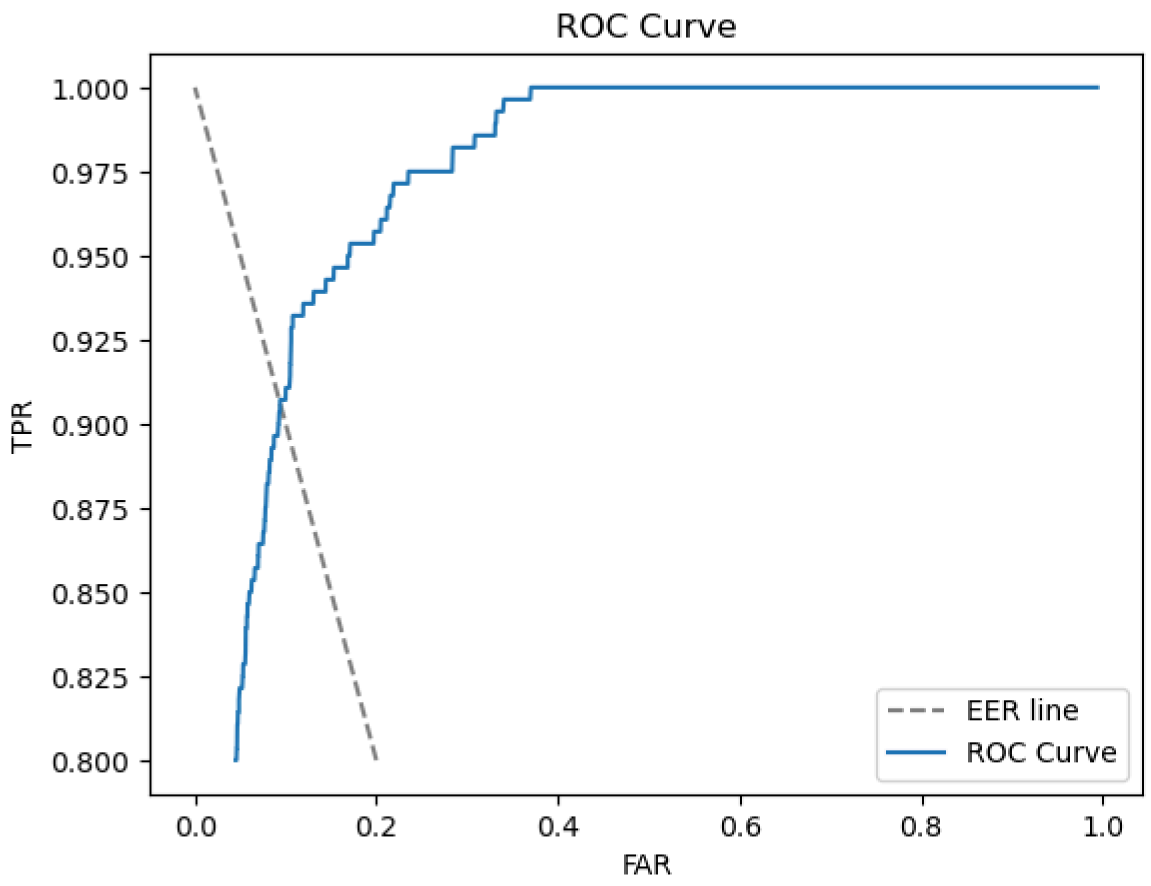
In [ ]:

```
# INSERT CODE HERE
best_eer = -1
best_k = None
for k in range(5, 15):
    print(f"for k = {k}")
    T_reduced = calculate_projection_vectors(T, meanface, v, k)
    D_reduced = calculate_projection_vectors(D, meanface, v, k)


    reduced_similarity_matrix = generate_similarity_matrix(T_reduced, D_r
    err, recall =plot_roc(reduced_similarity_matrix)
    if best_eer == -1 or best_eer > err:
        best_eer = err
        best_k = k

plt.legend()
print(f"Best k : {best_k}")
```

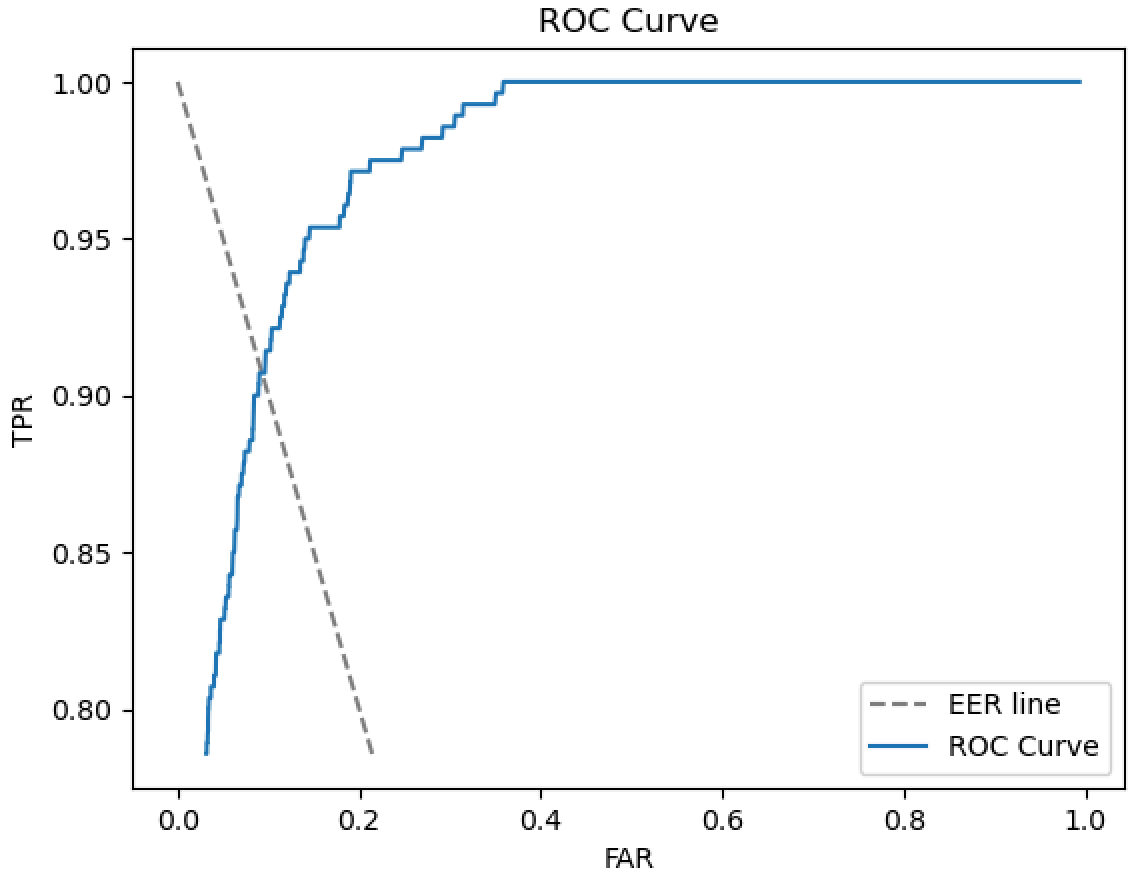
```
for k = 5
```



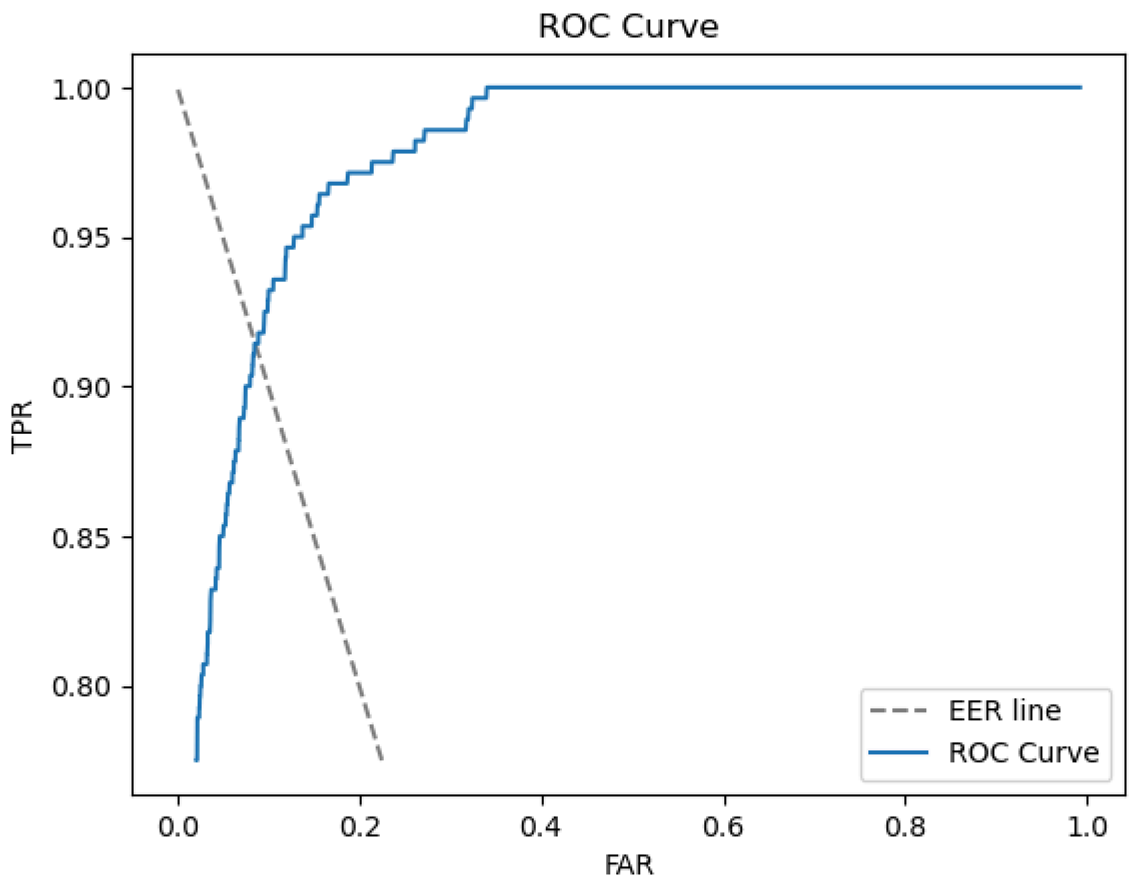
```
ERR: 0.8928571428571429
Recall at FAR: 0.8428571428571429
for k = 6
```



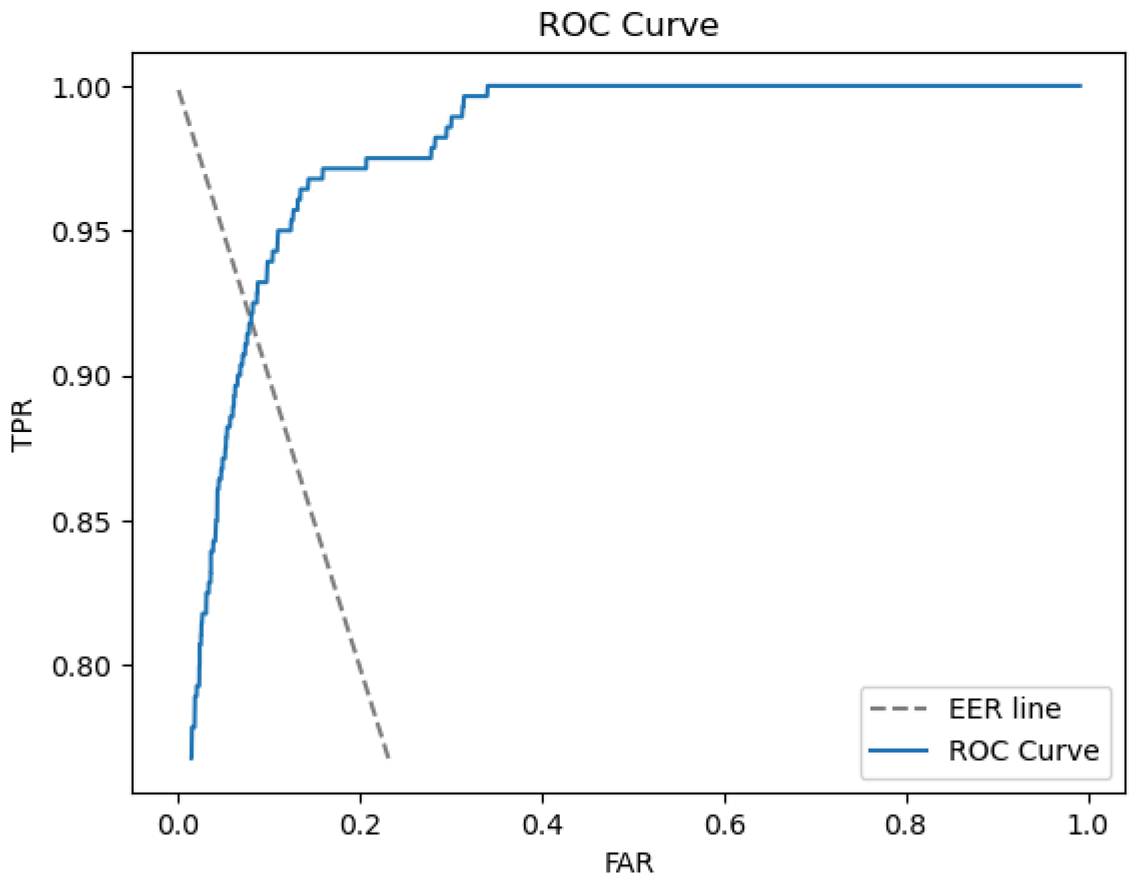
```
ERR: 0.9071428571428571
Recall at FAR: 0.8
for k = 7
```



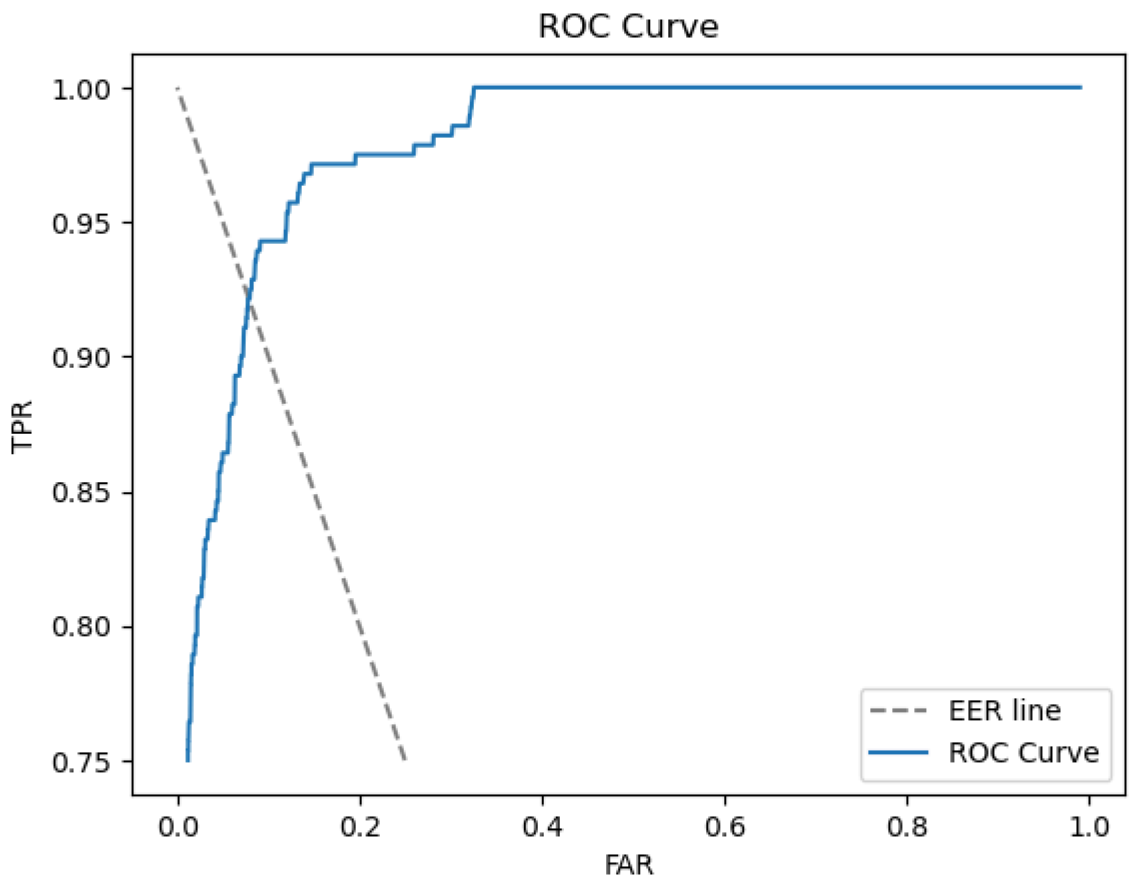
```
ERR: 0.9071428571428571
Recall at FAR: 0.7857142857142857
for k = 8
```



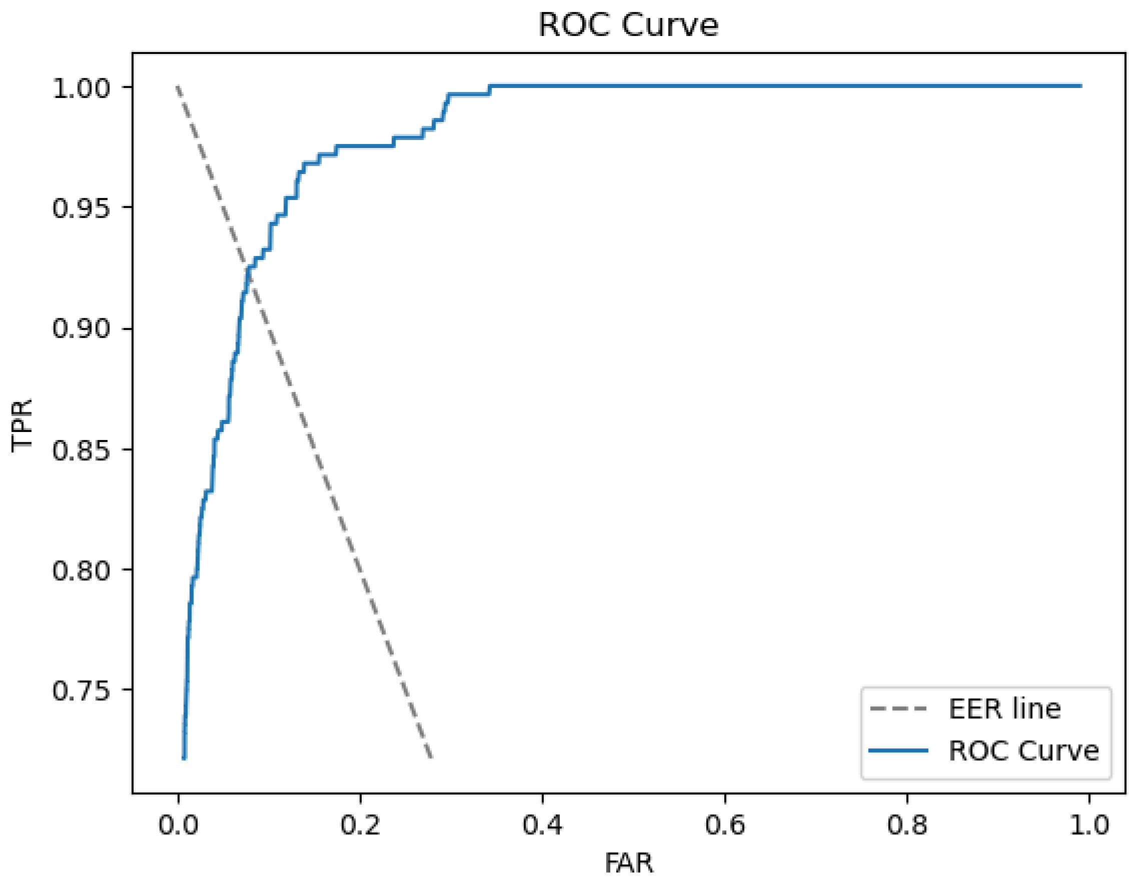
```
ERR: 0.9142857142857143
Recall at FAR: 0.775
for k = 9
```



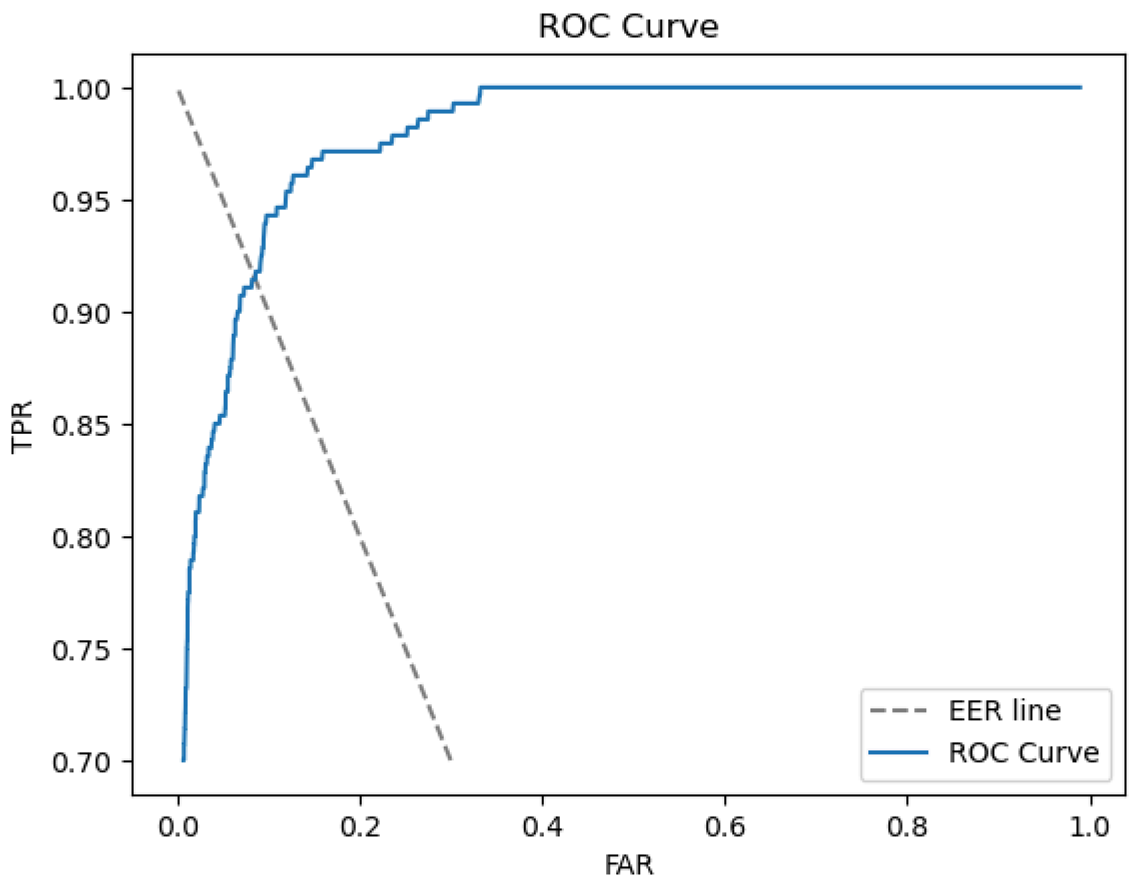
```
ERR: 0.9178571428571428
Recall at FAR: 0.7678571428571429
for k = 10
```



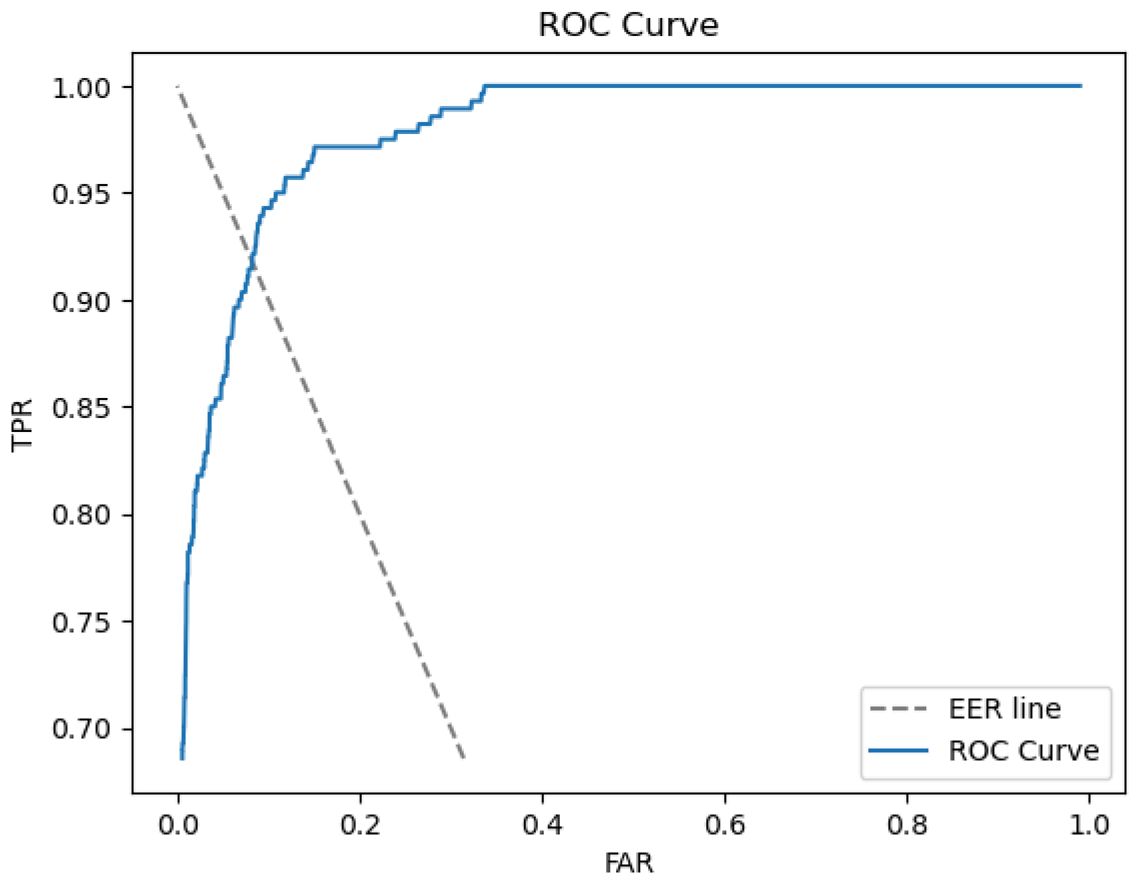
```
ERR: 0.9214285714285714
Recall at FAR: 0.75
for k = 11
```



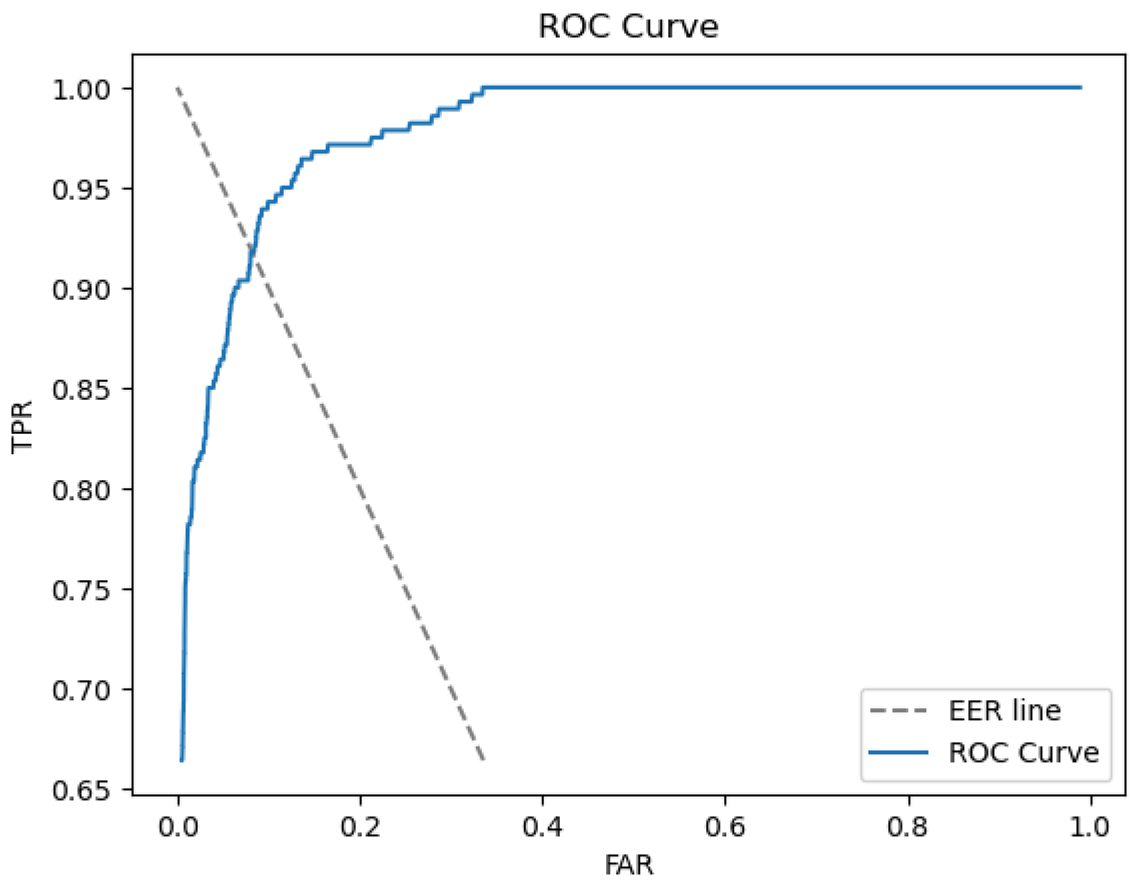
```
ERR: 0.9214285714285714
Recall at FAR: 0.7214285714285714
for k = 12
```



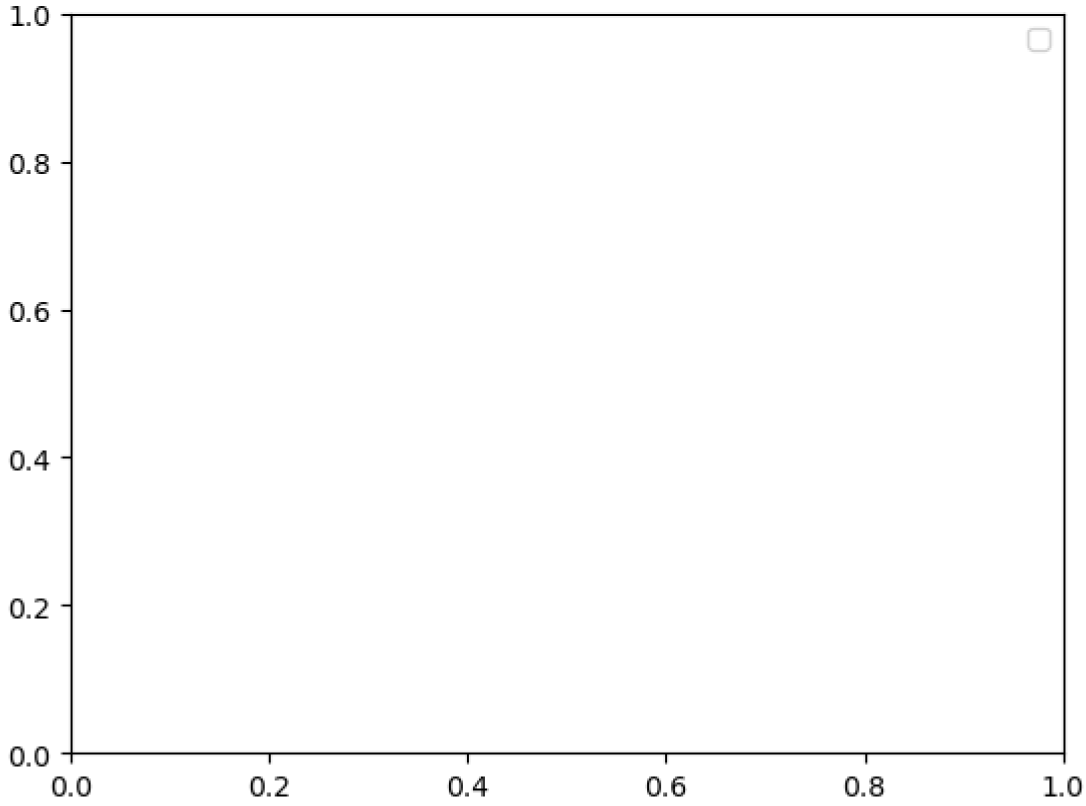
```
ERR: 0.9142857142857143
Recall at FAR: 0.7
for k = 13
```



```
ERR: 0.9178571428571428
Recall at FAR: 0.6857142857142857
for k = 14
```



```
No artists with labels found to put in legend.  Note that artists whose la
bel start with an underscore are ignored when legend() is called with no a
rgument.
ERR: 0.9178571428571428
Recall at FAR: 0.6642857142857143
Best k : 5
```



ANS:

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:

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

        # TODO: Find PCA of T and D with n_dim dimension.
```

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

T22. Using the answer to the previous question, project the original in- put 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?

```
In [ ]: # 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([np.matmul((class_mean[i]-all_mean).T, (class_mean[i]-all_

        s_wi = np.array([np.matmul((T_reduced_by_class[i]-class_mean[i]).T, (T_re
        s_w = s_wi.sum(axis=0)


        LDA = np.matmul(np.linalg.inv(s_w), s_b)
        print(f"LDA is symmetric? : {np.allclose(LDA, LDA.T)}")
        LDA_evalues, LDA_evectors = np.linalg.eig(LDA)

        LDA_evectors = LDA_evectors.real
        LDA_evalues = LDA_evalues.real
```

```
LDA is symmetric? : False
```

ANS:

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.

In [ ]:

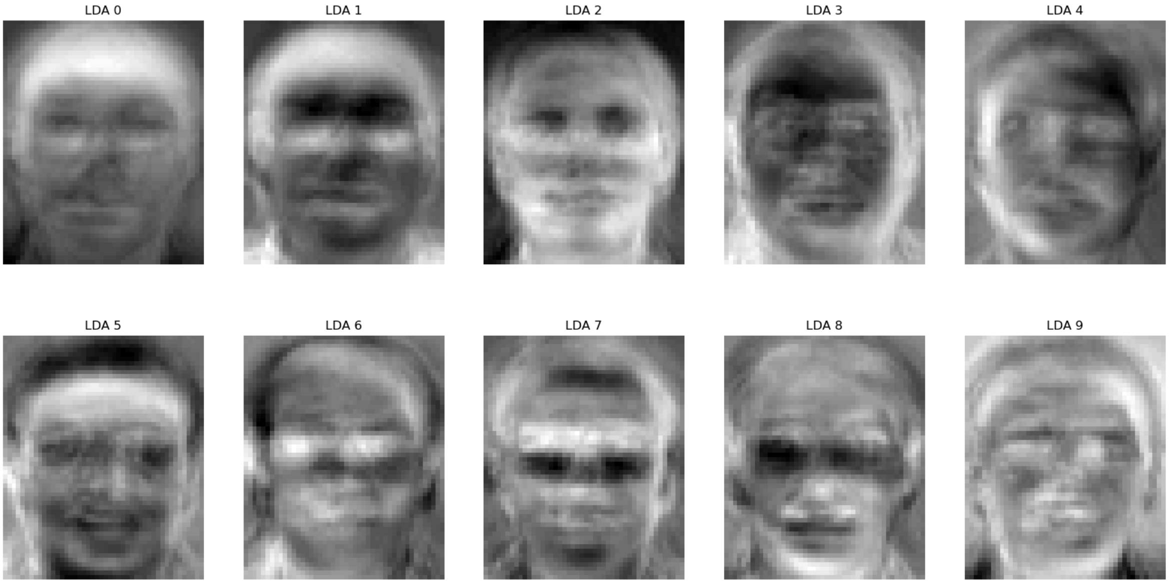
```
# INSERT CODE HERE
print(f"Amount of non-zero eigenvalues : {np.where(LDA_evalues>1e-5, 1, 0
```

```
Amount of non-zero eigenvalues : 39
```

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?

In [ ]:

```
# INSERT CODE HERE
best_10_LDA = LDA_evectors[:, :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(v[:, idx].reshape(56, 46), cmap='gray')
    plt.title(f"LDA {idx}")
    plt.axis('off')

plt.show()

T_LDA.shape, D_LDA.shape
```



Out[ ]:

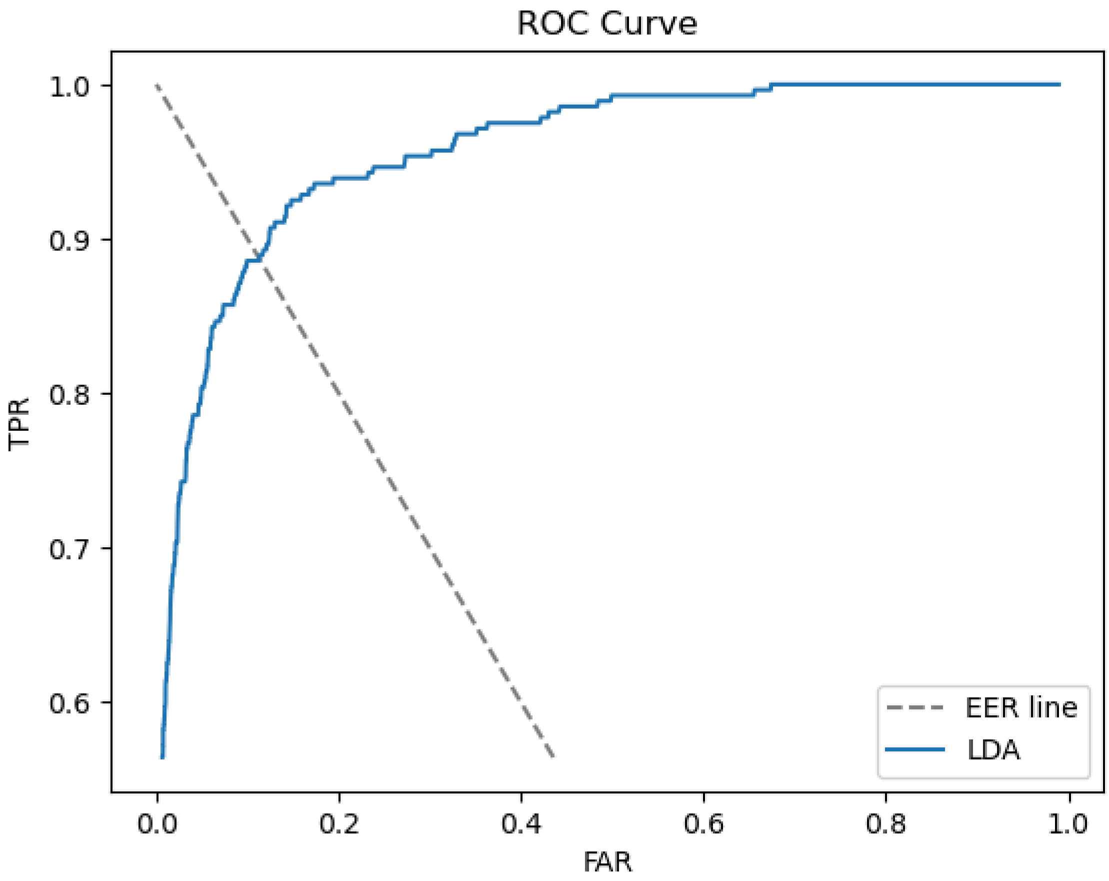
```
((120, 10), (280, 10))
```

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.

In [ ]:

```python
# INSERT CODE HERE
reduced_similarity_matrix = generate_similarity_matrix(T_LDA, D_LDA)
eer = plot_roc(reduced_similarity_matrix, label='LDA')
```



```
ERR: 0.8857142857142857
Recall at FAR: 0.5642857142857143
```

ANS: