HW3

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 , $\vec{j}$, Σ_j for each EM iteration. (You may do the calculations by hand or write code to do so)

Figure 1: Screenshot 2023-12-21 140423.png

TODO: Complete functions below including

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

These functions will be used in T1-4.

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

```python
from math import log


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):
        w = np.empty((len(data), len(self.mixture_weight)))
        for i in range(len(data)):
            for j in range(len(self.mixture_weight)):
                w[i][j] = self.mixture_weight[j] * multivariate_normal.pdf(
                    data[i], self.mean_params[j], self.cov_params[j]
                )
            w[i] /= np.sum(w[i])
        return w

    def maximization_step(self, data, w):
        """
        TODO: Perform maximization step.
            (Update parameters in this GMM model.)
        """
        self.mixture_weight = np.mean(w, axis=0)

        for i in range(len(self.mixture_weight)):
            self.mean_params[i] = np.sum(
                (w[:, i].reshape(-1, 1) * data) / np.sum(w[:, i]), axis=0
            )

        for i in range(len(self.mixture_weight)):
            for j in range(len(data)):
                self.cov_params[i] += w[j][i] * np.outer(
                    data[j] - self.mean_params[i], data[j] - self.mean_params[i]
                )
            self.cov_params[i] /= np.sum(w[:, i])

    def get_log_likelihood(self, data):
        """
        TODO: Compute log likelihood.
        """
        # INSERT CODE HERE
        log_prob = 0
        for i in range(len(data)):
            prob = 0
            for j in range(len(self.mixture_weight)):
                prob += self.mixture_weight[j] * multivariate_normal.pdf(
                    data[i], self.mean_params[j], self.cov_params[j]
                )
            log_prob += log(prob)
        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 = []
```

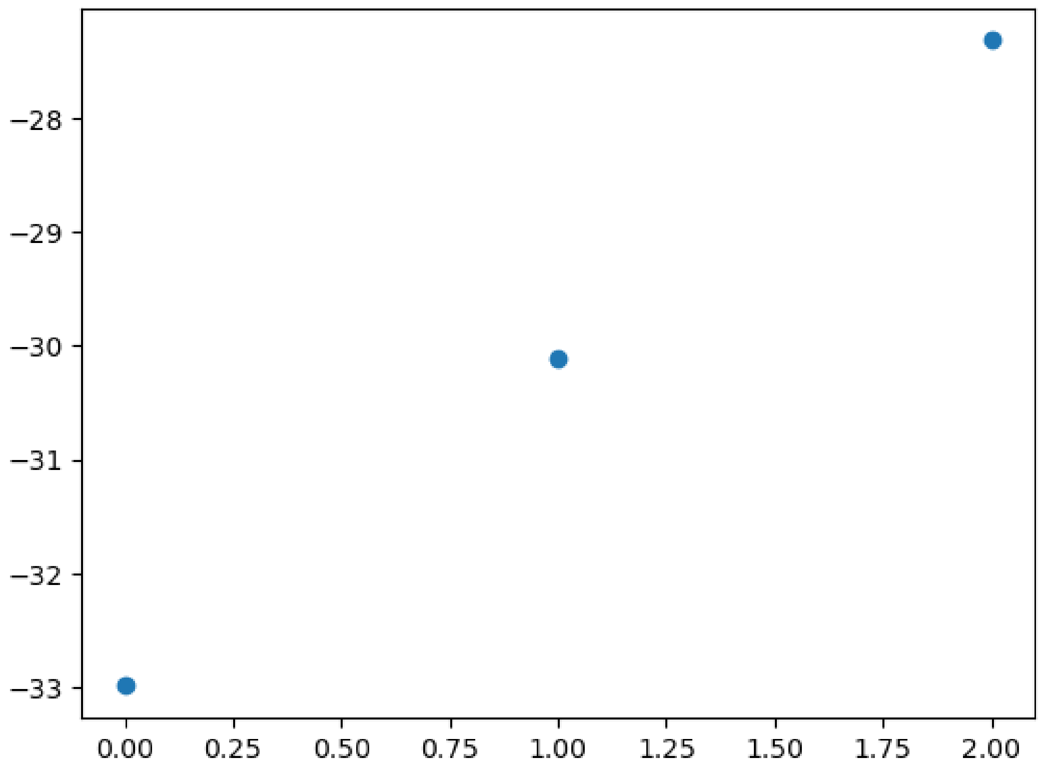
```python
        # Display initialization.
        if display:
            print("Initialization")
            self.print_iteration()

        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
```

```python
num_iterations = 3
num_mixture = 3
mixture_weight = [1 / 3] * 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 :
 [0.3333333333333333, 0.3333333333333333, 0.3333333333333333]
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 :
 [[[4.7790215  4.39754618]
  [4.39754618 4.52983349]]

 [[1.04784828 0.19950142]
  [0.19950142 0.66291866]]

 [[5.00000001 3.33333335]
  [3.33333335 3.22222225]]]
------------------------------------------------------------
Iteration: 1
m :
 [0.43157401 0.23509255 0.33333343]
mu :
 [[ 5.99337631  6.01503835]
 [ 1.75703518  2.18979297]
 [-3.9989529  -4.66554675]]
covariance matrix :
 [[[5.28734911 5.08097657]
  [5.08097657 5.02807493]]

 [[1.05876077 0.33019619]
  [0.33019619 0.46708534]]

 [[6.34050367 4.45266725]
  [4.45266725 3.97239822]]]
------------------------------------------------------------
Iteration: 2
m :
 [0.42527311 0.24150965 0.33321724]
mu :
 [[ 6.06314449  6.06816499]
 [ 1.74195349  2.19309469]
 [-3.99896534 -4.66580103]]
covariance matrix :
 [[[5.15904134 5.07970284]
  [5.07970284 5.04545699]]

 [[1.0647511  0.39483531]
  [0.39483531 0.37070081]]

 [[6.79180837 4.82951865]
  [4.82951865 4.22577279]]]
------------------------------------------------------------
```

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

```
likelihood = np.array(log_prob_list)
plt.scatter(np.arange(3), likelihood)
```



```
ANS : The log likelihood goes up as expected.
```

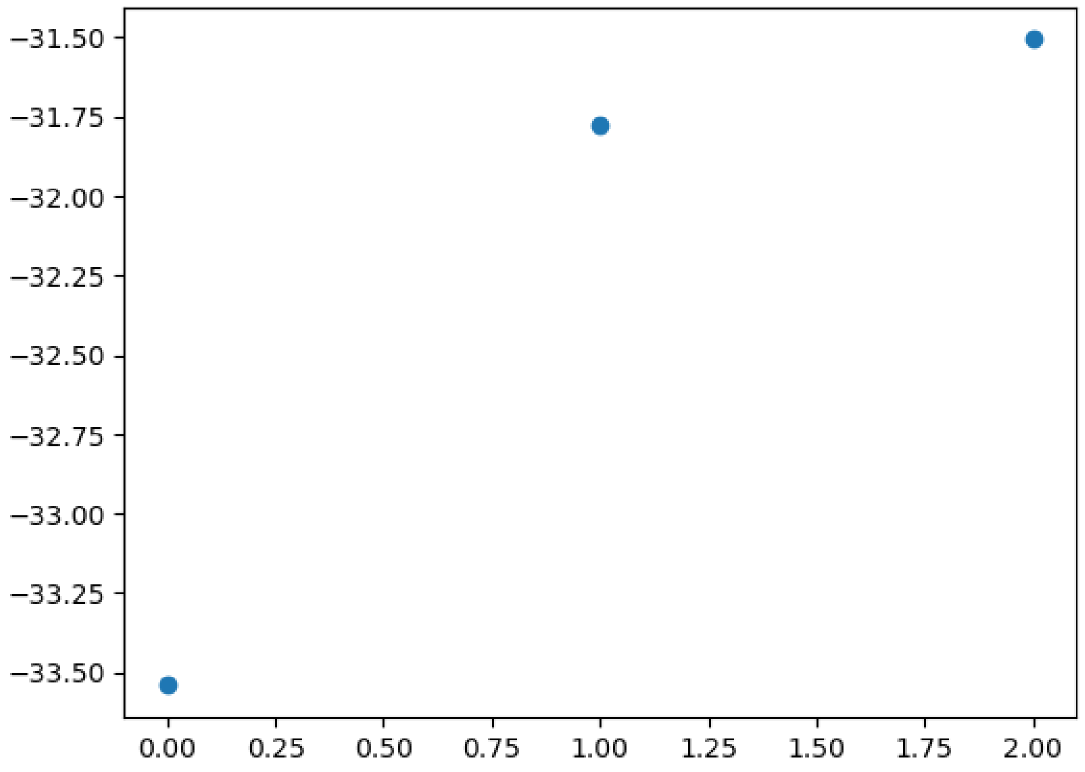
T3. Using 2 mixtures, initialize your Gaussian with means (3,3) and (-3,-3), and standard Covariance, I, the identity matrix. Use equal mixture weights as the initial weights. Repeat three iterations of EM. Write down $w_{n,j}$, m_j , $\vec{j}$, Σ_j for each EM iteration.

```
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, mean_params, cov_params)
log_prob_list2 = gmm2.perform_em_iterations(data, num_iterations)
```

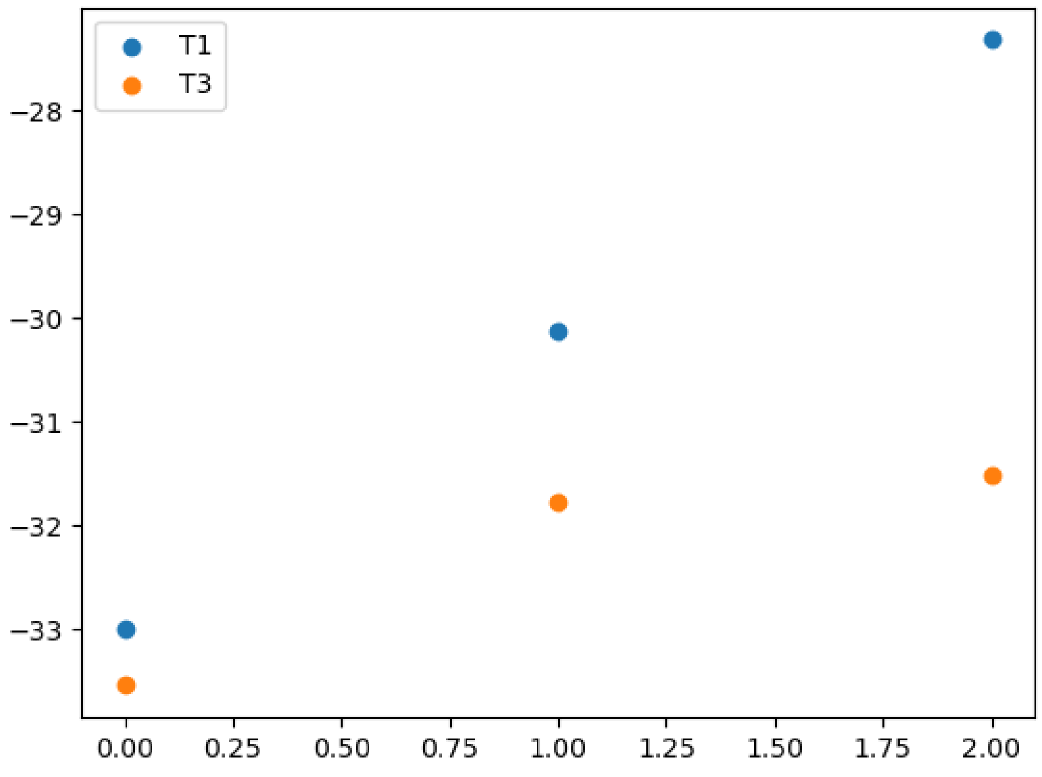
```
Initialization
m :
 [1, 1]
mu :
 [[ 3.  3.]
 [-3. -3.]]
covariance matrix :
 [[[1. 0.]
  [0. 1.]]

 [[1. 0.]
  [0. 1.]]]
--------------------------------------------------------------
Iteration: 0
m :
 [0.66666666 0.33333334]
mu :
 [[ 4.50000001  4.66666667]
 [-3.99999997 -4.66666663]]
covariance matrix :
 [[[7.08333332 6.33333333]
  [6.33333333 6.05555555]]

 [[5.0000001  3.33333349]
  [3.33333349 3.22222243]]]
--------------------------------------------------------------
Iteration: 1
m :
 [0.66963856 0.33036144]
mu :
 [[ 4.46591863  4.63178178]
 [-4.00738256 -4.67991709]]
covariance matrix :
 [[[8.3182437  7.61893814]
  [7.61893814 7.13634743]]

 [[6.39149841 4.48128334]
  [4.48128334 3.98738996]]]
--------------------------------------------------------------
Iteration: 2
m :
 [0.67050251 0.32949749]
mu :
 [[ 4.45593074  4.62156588]
 [-4.00927521 -4.68354405]]
covariance matrix :
 [[[8.58758407 7.89927636]
  [7.89927636 7.3850813 ]]

 [[6.8799934  4.8815273 ]
  [4.8815273  4.25504019]]]
--------------------------------------------------------------
```

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?

```python
likelihood2 = np.array(log_prob_list2)
plt.scatter(np.arange(3), likelihood2)
```



```python
# TODO: Plot Comparision of log_likelihood from T1 and T3
plt.scatter(np.arange(3), likelihood, label="T1")
plt.scatter(np.arange(3), likelihood2, label="T3")
plt.legend()
```



```
ANS : The one with 3 clusters results in a better likelihood.
```

The face database

```
# 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
```

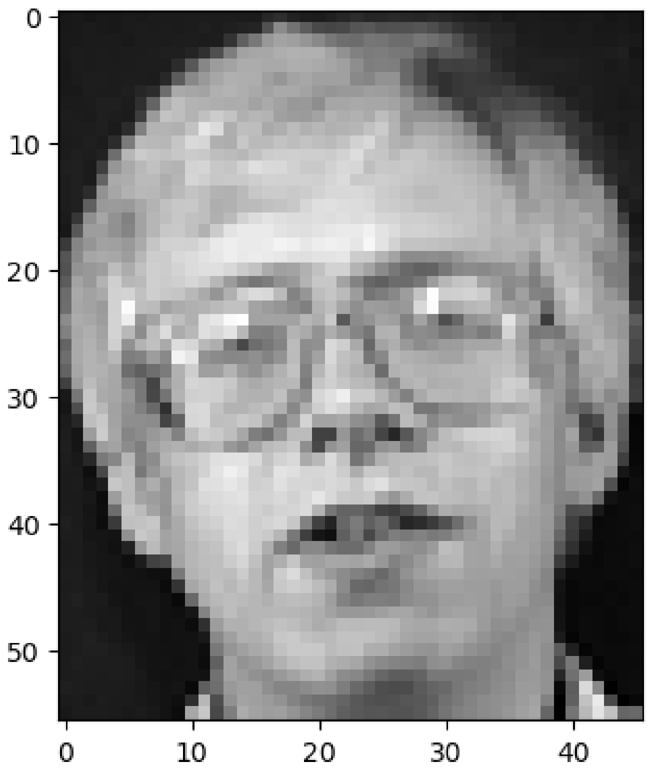
```
(40, 10)
```

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()
```

```
plt.imshow(xf[1, 0], cmap="gray")
```



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.
    """
    # from numpy.linalg import norm
    from numpy.linalg import norm

    return norm(x1 - x2)
    # return np.sqrt(np.sum((x1 - x2) ** 2))


# 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]))
```

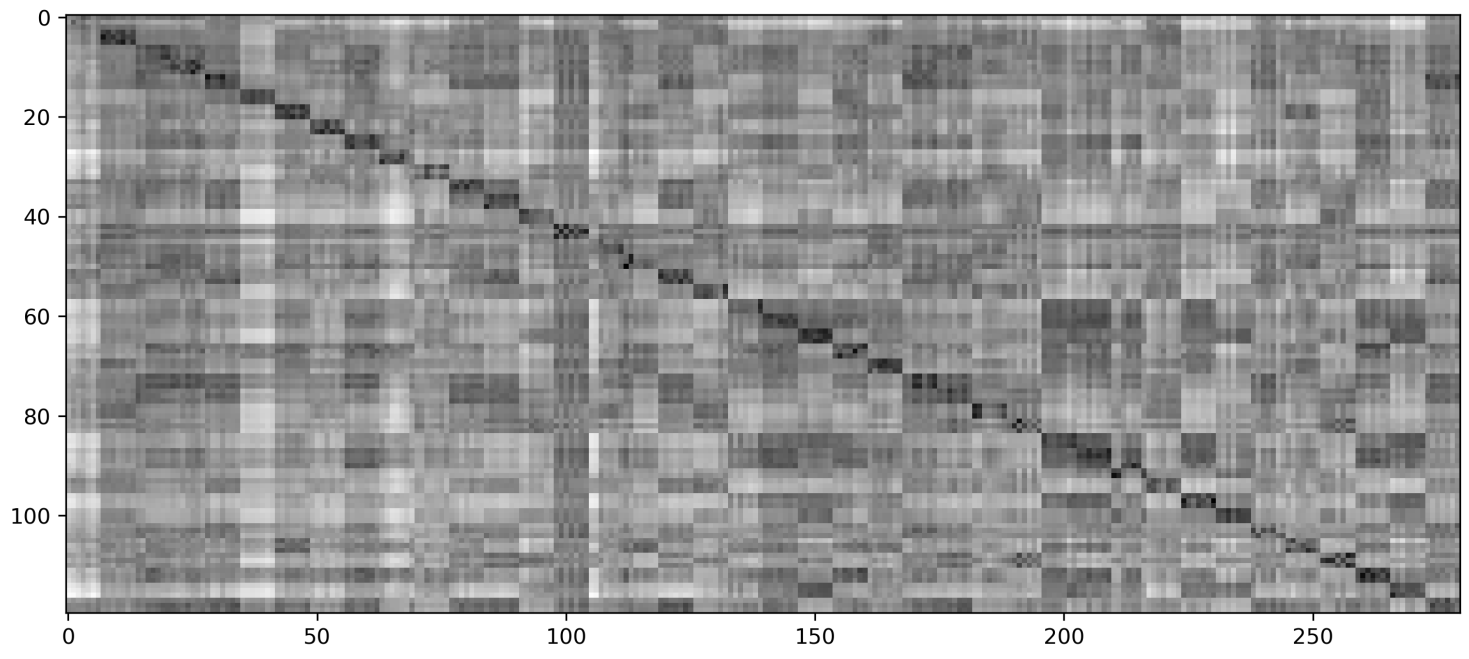
```
Euclidean distance between xf[0,0] and xf[0,1] is 10.037616294165492
Euclidean distance between xf[0,0] and xf[1,0] is 8.173295099737281
```

ANS : Somehow the similarity between different people seems to be less than the same person but different pose, still the numbers will be useful if we compare against the minimum similarity value.

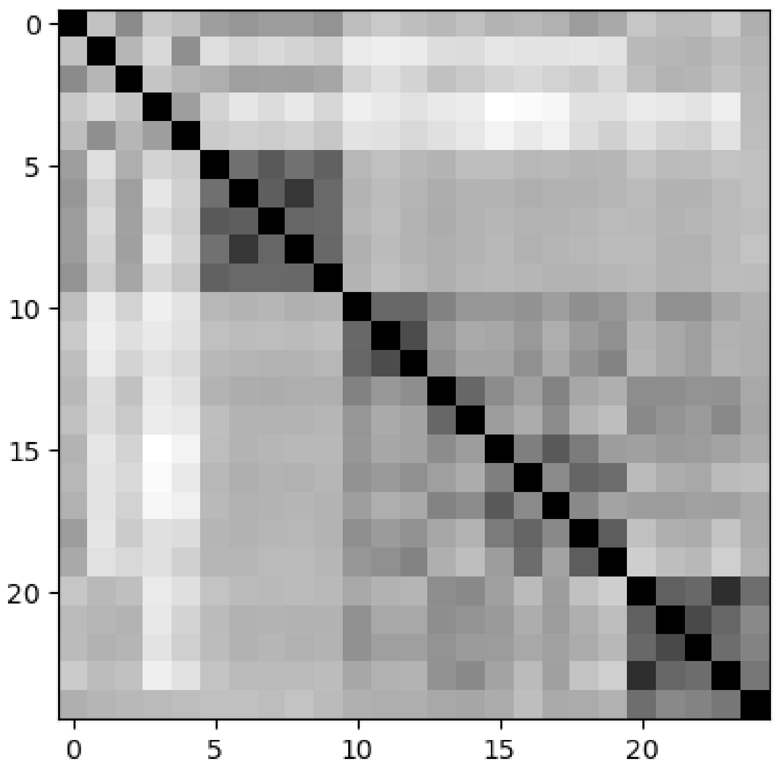
T6. Write a function that takes in a set of feature vectors T and a set of feature vectors D, and then output the similarity matrix A. Show the matrix as an image. Use the feature vectors from the first 3 images from all 40 people for list T (in order x[0, 0], x[0, 1], x[0, 2], x[1, 0], x[1, 1], . . . 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

```
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])
    """
    return


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

    similarity_matrix = np.zeros((len(A), len(B)))
    for i in range(len(A)):
        for j in range(len(B)):
            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 = xf[:, :3].reshape(-1, 46 * 56)
D = xf[:, 3:].reshape(-1, 46 * 56)
similarity_matrix = generate_similarity_matrix(T, D)
# plt.figure(dpi=300, figsize=(20, 20))
# plt.imshow(similarity_matrix)
plt.figure(dpi=300, figsize=(15, 5))
plt.imshow(similarity_matrix, cmap="gray")
```



```
sm = generate_similarity_matrix(
    xf[:5, :5].reshape(-1, 46 * 56), xf[:5, :5].reshape(-1, 46 * 56)
)
plt.imshow(sm, cmap="gray")
```



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?

ANS : The more black a square is the more similar those two faces are. This means person #2 have more similar poses than person #1.

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?

```
def predict(
    similarity_matrix: np.ndarray,
    threshold,
    count=40,
    train_faces=3,
):
    sm = similarity_matrix.reshape(count, train_faces, -1)
    return (sm.min(axis=1) < threshold).reshape(count, count, -1)
```

```
def evaluate_performance(
    similarity_matrix, threshold, count=40, train_faces=3, test_faces=7
):
    """
    TODO: Calculate true positive rate and false alarm rate from given similarity_matrix and threshold
    """
    prediction = predict(similarity_matrix, threshold, count, train_faces)
    true_label = np.zeros_like(prediction)
    for i in range(count):
        true_label[i, i, :] = 1
    tpr = np.sum(prediction * true_label) / np.sum(true_label)
    fpr = np.sum(prediction * (1 - true_label)) / np.sum(1 - true_label)
    return tpr, fpr


# Quick check


# (true_pos_rate, false_neg_rate) should be (0.9928571428571429, 0.33507326007326005)

print("")
evaluate_performance(similarity_matrix, 9.5)
```

```
(0.9928571428571429, 0.33507326007326005)
```

```
    ANS: True positive rate = 0.9928571428571429, False positive rate = 0.33507326007326005
```

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).

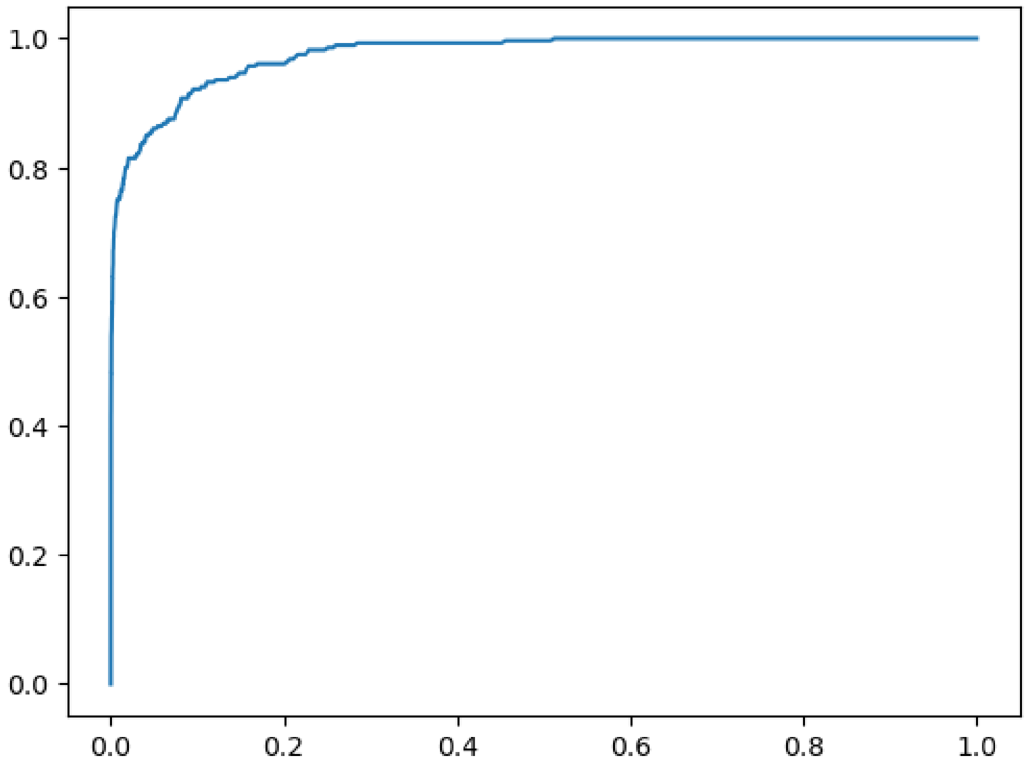
The minimum and maximum threshold is set to be the minimum and maximum similarity value respectively.

```
similarity_matrix.min(), similarity_matrix.max()
```

```
(1.7420153428787781, 17.541726165424688)
```

```
def calculate_roc(input_mat):
    """
    TODO: Calculate a list of true_pos_rate and a list of false_neg_rate from the given matrix.
    """
    tpr_list, far_list = [], []
    # INSERT CODE HERE
    for threshold in np.linspace(input_mat.min(), input_mat.max(), 1000):
        tpr, far = evaluate_performance(input_mat, threshold)
        tpr_list.append(tpr)
        far_list.append(far)
    return tpr_list, far_list


def plot_roc(tpr, far, label=""):
    """
    TODO: Plot RoC Curve from a given matrix.
    """
    plt.plot(far, tpr, label=label)
    # plt.plot([0, 1], [1, 0], "r--")
```

```
# INSERT CODE HERE
tpr1, far1 = calculate_roc(similarity_matrix)


plot_roc(tpr1, far1)
```



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.
```

Recall rate at 0.1% false alarm rate:

```
tpr1[np.argmin(np.abs(np.array(far1) - 0.001))]
```

```
0.5428571428571428
```

ERR:

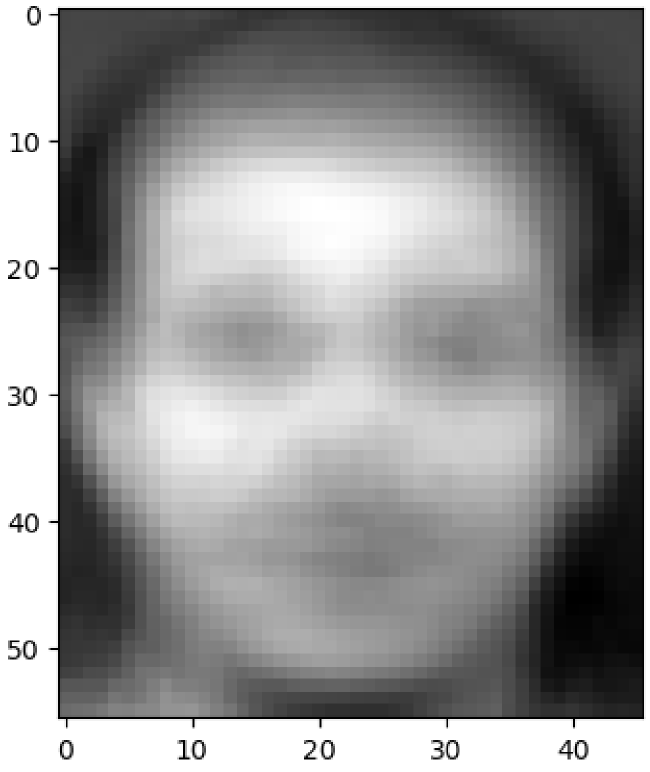
```
far1[np.argmin(np.abs(1 - np.array(tpr1) - np.array(far1)))]
```

```
0.08891941391941392
```

```
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 = T.mean(axis=0).reshape(56, 46)
plt.imshow(meanface, cmap="gray")
```



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.
X = T.T - meanface.reshape(-1, 1)
sigma = np.cov(X)
print(f"size: {sigma.shape}")
print(f"rank: {np.linalg.matrix_rank(sigma)}")
```

```
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 = X.T @ X
print(gram_matrix.shape)
print(f"rank: {np.linalg.matrix_rank(gram_matrix)}")
```

```
(120, 120)
rank: 119
```

We expect the number of non-zero eigenvalues to be 119, the same number as the rank.

T14. Is the Gram matrix also symmetric? Why?

Yes, $(X^TX)^T = X^TX$

T15. Compute the eigenvectors and eigenvalues of the Gram matrix, v 0 and . 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.html


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.html
    """

    eigenvalues, eigenvectors = np.linalg.eigh(matrix)
    idx = eigenvalues.argsort()[::-1]
    eigenvalues = eigenvalues[idx]
    eigenvectors = eigenvectors[:, idx]

    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()
```

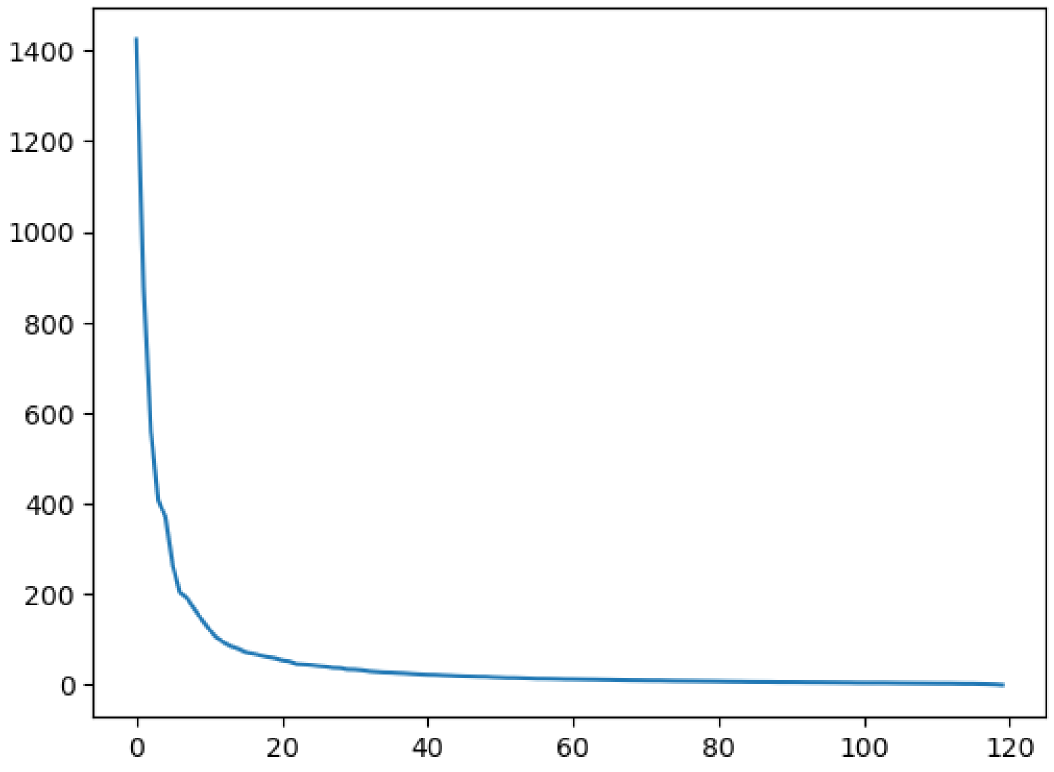
```
np.sum(~np.isclose(eigenvalues, 0))
```

```
119
```

The number of non-zero eigenvalues is 119 as expected.

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
eigenvalues, eigenvectors = calculate_eigenvectors_and_eigenvalues(gram_matrix)
plt.plot(eigenvalues)
```



```
var_explained = np.cumsum(eigenvalues) / np.sum(eigenvalues)
np.argmax(var_explained > 0.95) + 1
```

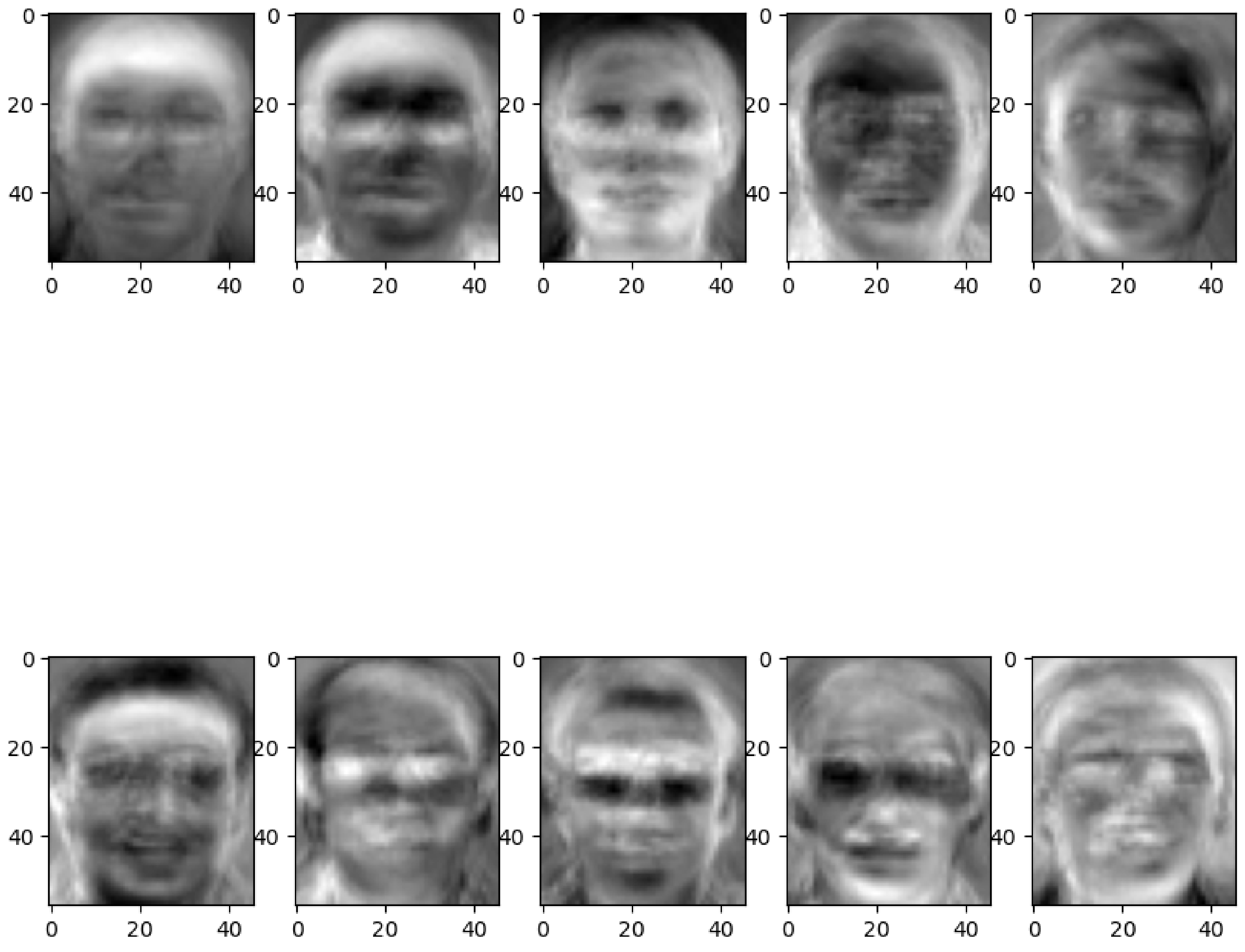
```
64
```

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).

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

# INSERT CODE HERE

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

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


test_eignevector_cov_norm(v)
```

```python
# TODO: Show the first 10 eigenvectors as images.
img = v[:, :10].reshape(56, 46, -1)
plt.figure(figsize=(10, 10))
plt.tight_layout()
for i in range(10):
    plt.subplot(2, 5, i + 1)
    plt.imshow(img[:, :, i], 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: Looking at the white parts of the image, the fist eigenvector seems to capture the hair, the second eigenvector seems to capture the eyes and hair.

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?

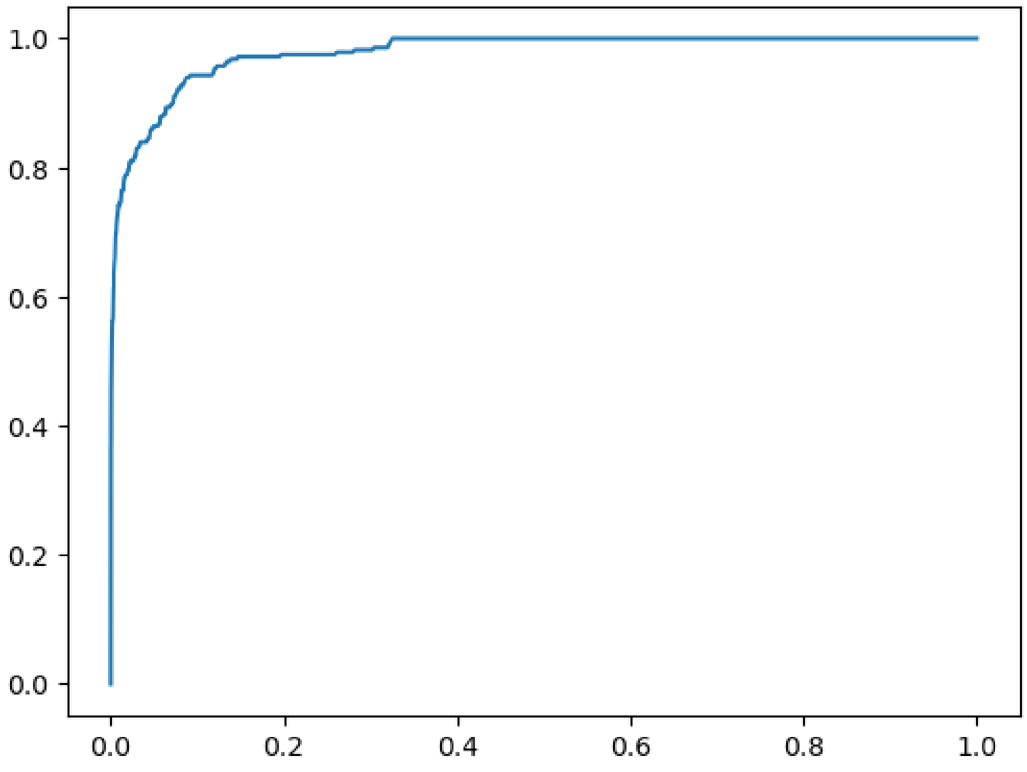
```python
def calculate_projection_vectors(matrix, meanface, v, ndim):
    """
    TODO: Find the projection vectors on v from given matrix and meanface.
    """

    # INSERT CODE HERE
    projection_vectors = v.T @ (matrix.T - meanface.reshape(-1, 1))
    return projection_vectors.T[:, :ndim]
```

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


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


test_reduce_dimension()
```

```python
# TODO: Get similarity matrix of T_reduced and D_reduced
similarity_matrix_reduced = generate_similarity_matrix(T_reduced, D_reduced)
```

```python
# TODO: Find EER and the recall rate at 0.1% FAR.

tpr2, far2 = calculate_roc(similarity_matrix_reduced)
```

```python
plot_roc(tpr2, far2)
```



```
print(tpr2[np.argmin(np.abs(np.array(far2) - 0.001))])
print(far2[np.argmin(np.abs(1 - np.array(tpr2) - np.array(far2)))])
```

```
0.5178571428571429
0.07884615384615384

ANS:
EER: 0.07884615384615384
Recall rate at 0.1% FAR: 0.5178571428571429
```

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

```
for k in range(5, 15):
    T_reduced = calculate_projection_vectors(T, meanface, v, k)
    D_reduced = calculate_projection_vectors(D, meanface, v, k)
    similarity_matrix_reduced = generate_similarity_matrix(T_reduced, D_reduced)
    tpr, far = calculate_roc(similarity_matrix_reduced)
    print(f"k={k} EER={far[np.argmin(np.abs(1 - np.array(tpr) - np.array(far)))]}")
```

```
k=5 EER=0.10705128205128205
k=6 EER=0.09413919413919414
k=7 EER=0.09276556776556777
k=8 EER=0.0858058608058608
k=9 EER=0.08021978021978023
k=10 EER=0.07884615384615384
k=11 EER=0.0782967032967033
k=12 EER=0.0848901098901099
k=13 EER=0.0815934065934066
k=14 EER=0.0826007326007326
```

ANS: k = 11 results in the best EER.

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: The number of dimension to make S_W full rank is $120 - 40 = 80$

```
# TODO: Define dimension of PCA.
n_dim = 80
T_reduced = calculate_projection_vectors(T, meanface, v, n_dim).reshape(-1, 3, 80)
D_reduced = calculate_projection_vectors(D, meanface, v, n_dim).reshape(-1, 7, 80)
# TODO: Find PCA of T and D with n_dim dimension.
```

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?

```
def calculate_lda_projection(matrix, faces, classes=40, n_dim=80):
    reduced = matrix.reshape(-1, faces, n_dim)
    mean = reduced.mean(axis=1)
    global_mean = mean.mean(axis=0)
    Sb = np.zeros((n_dim, n_dim))
    Sw = np.zeros((n_dim, n_dim))
    for i in range(classes):
        for j in range(faces):
            Sw += np.outer(reduced[i, j] - mean[i], reduced[i, j] - mean[i])
    for i in range(classes):
        Sb += np.outer(mean[i] - global_mean, mean[i] - global_mean)
    eigenvalues, eigenvectors = np.linalg.eig(np.linalg.inv(Sw) @ Sb)
    idx = eigenvalues.argsort()[::-1]
    eigenvalues = eigenvalues[idx]
    eigenvectors = eigenvectors[:, idx]

    return eigenvalues, eigenvectors
```

```
# TODO: Find the LDA projection.
eigenvalues, w = calculate_lda_projection(T_reduced, 3)
```

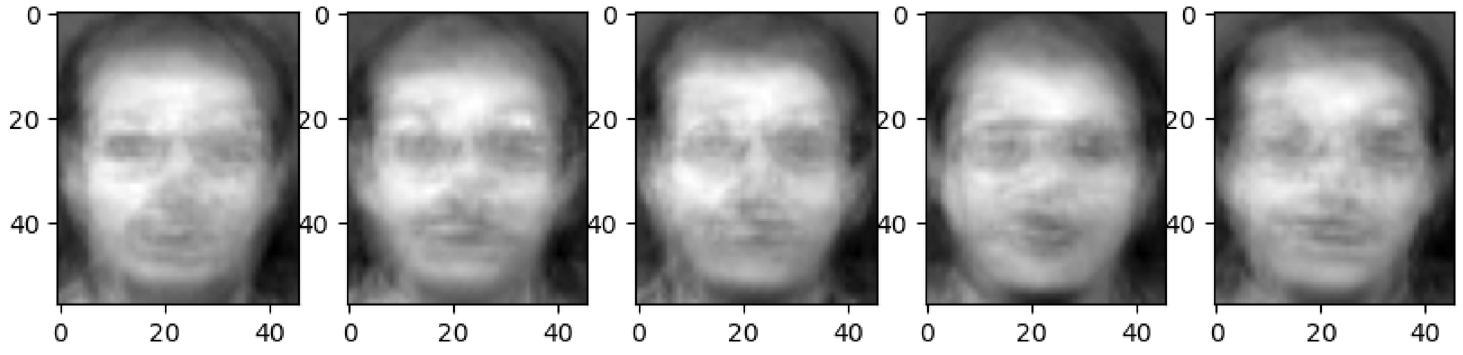
```
# TODO: Find how many non-zero eigenvalues there are.
np.sum(~np.isclose(eigenvalues, 0))
```

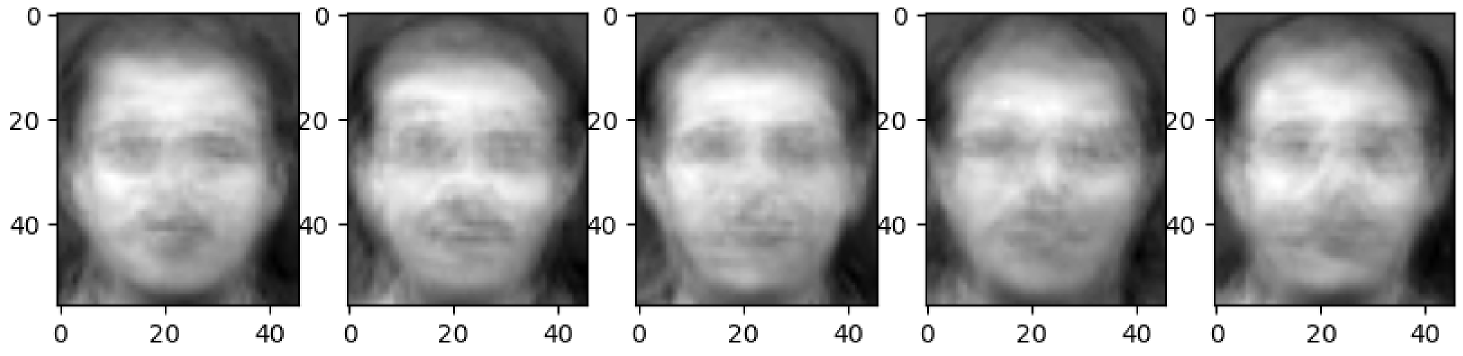
```
39
```

ANS: We can't use `numpy.linalg.eigh` anymore since the product $S_W^{-1} S_B$ is not symmetric. The resulting number of non-zero eigenvalues is 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.

```python
# INSERT CODE HERE
img = (meanface.reshape(-1, 1) + v[:, :80] @ w[:, :10]).reshape(56, 46, -1)
plt.figure(figsize=(10, 10))
plt.tight_layout()
for i in range(10):
    plt.subplot(2, 5, i + 1)
    plt.imshow(np.real(img[:, :, i]), cmap="gray")
```





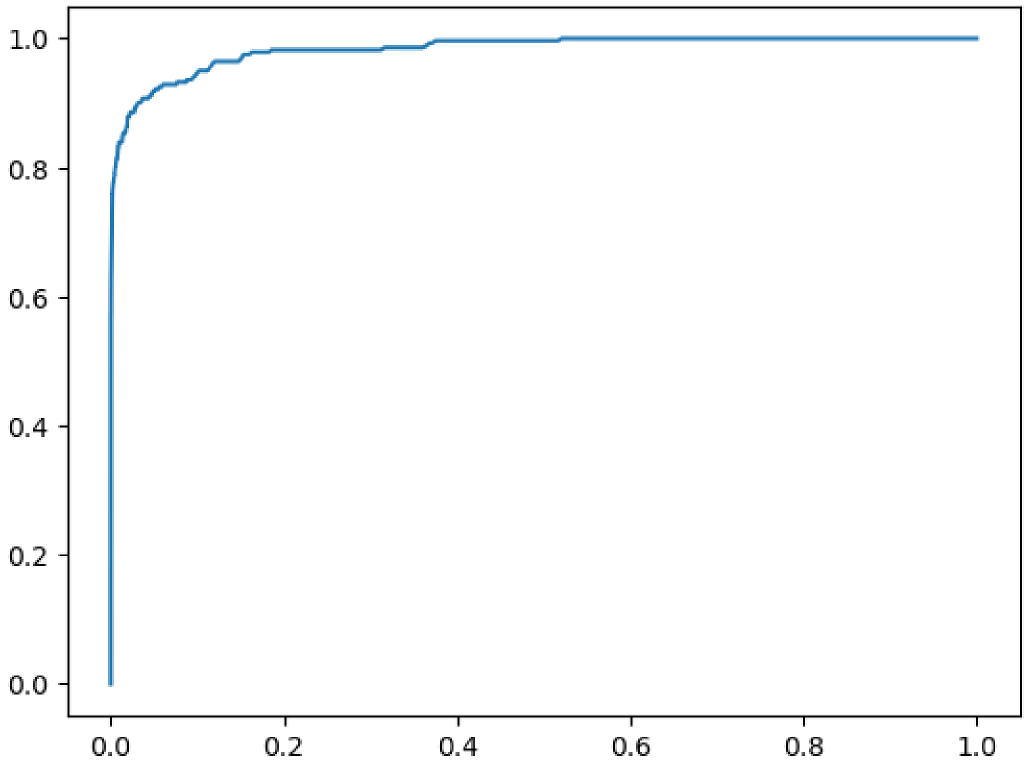
T24. The combined PCA+LDA projection procedure is called fisherface. Calculate the fisherfaces projection of all images. Do the simple face verification experiment using fisherfaces. What is the EER and recall rate at 0.1% FAR?

```python
T_reduced = calculate_projection_vectors(T, meanface, v, 80)
D_reduced = calculate_projection_vectors(D, meanface, v, 80)
```

```
similarity_matrix = generate_similarity_matrix(
    T_reduced @ w[:, :39], D_reduced @ w[:, :39]
)
```

```
# calculate_roc(similarity_matrix)

tpr3, far3 = calculate_roc(similarity_matrix)
plot_roc(tpr3, far3)
print(tpr3[np.argmin(np.abs(np.array(far3) - 0.001))])
print(far3[np.argmin(np.abs(1 - np.array(tpr3) - np.array(far3)))])
```

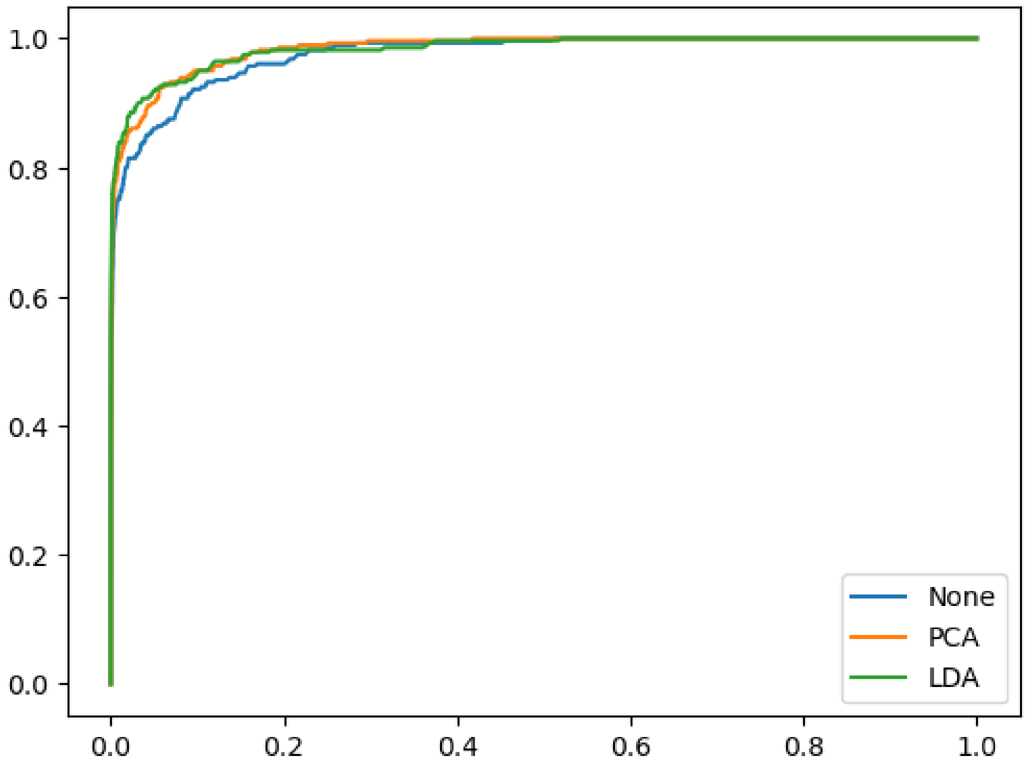
```
0.6821428571428572
0.0706959706959707
```



```
ANS:
EER: 0.0706959706959707
Recall rate at 0.1% FAR: 0.6821428571428572
```

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.

```
plot_roc(tpr1, far1, "None")
plot_roc(tpr2, far2, "PCA")
plot_roc(tpr3, far3, "LDA")
plt.legend()
```



```
ANS: LDA gives the best result, followed by PCA, and naive method.
```