Homework 4 – Report

Programming Exercises

Question 1

Part a

Question 1

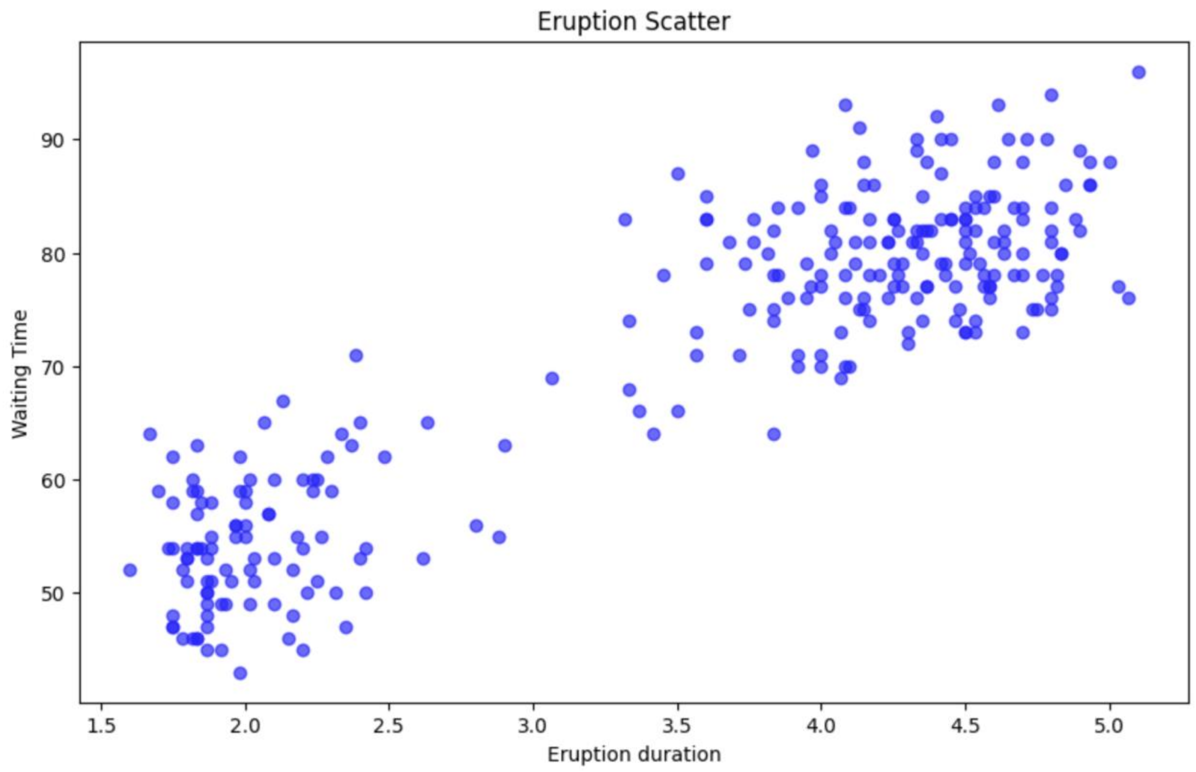
```
import numpy as np
import pandas as pd
from matplotlib import pyplot as plt
```

```
filepath = (r"C:\Users\Sofia Beyerlein\Desktop\Cornell Graduate\Applied Machine Learning\hw4\AML_HW4_Data.csv")
df = pd.read_csv(filepath)
df
```

Part a

```
feature_vectors = df.to_numpy()
```

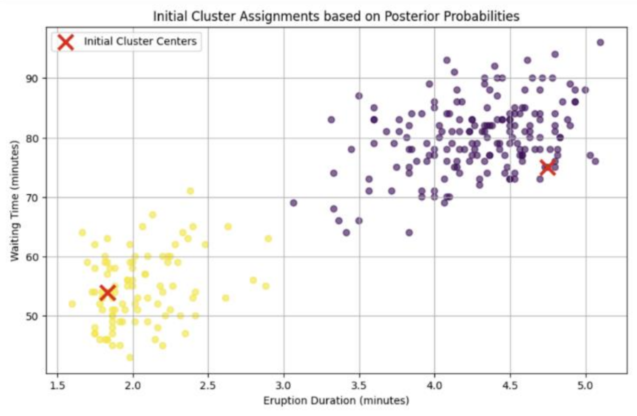
```
x = feature_vectors[:, 0]
y = feature_vectors[:, 1]

plt.figure(figsize = (10, 6))
plt.scatter(x, y, alpha = 0.6, c = 'blue')

plt.xlabel("Eruption duration")
plt.ylabel("Waiting Time")
plt.title("Eruption Scatter")
plt.show()
```



Part b

```python
def calculate_posterior_probability(X, means, covariances, weights):
    n_samples = X.shape[0]
    n_components = len(means)

    #array to store likelihood
    likelihoods = np.zeros((n_samples, n_components))

    #calculating likelihood for each cluster
    for k in range(n_components):
        gaussian = multivariate_normal(mean=means[k], cov=covariances[k])
        likelihoods[:, k] = gaussian.pdf(X) * weights[k]

    #calculate sum over all clusters
    denominator = likelihoods.sum(axis=1, keepdims=True)

    posteriors = likelihoods / denominator

    return posteriors

n_components = 2
n_features = feature_vectors.shape[1]

random_idx = np.random.choice(len(feature_vectors), n_components, replace=False)
means = feature_vectors[random_idx]

covariances = [np.diag(np.var(feature_vectors, axis=0)) for _ in range(n_components)]

#initializing weights
weights = np.ones(n_components) / n_components

#calculating posterior probabilities
posteriors = calculate_posterior_probability(feature_vectors, means, covariances, weights)
```

```python
#plotting results
plt.figure(figsize=(10, 6))

# Color points by their highest probability cluster
cluster_assignments = np.argmax(posteriors, axis=1)
plt.scatter(feature_vectors[:, 0], feature_vectors[:, 1],
           c=cluster_assignments, cmap='viridis', alpha=0.6)

# Plot cluster means
plt.scatter(means[:, 0], means[:, 1], c='red', marker='x', s=200,
           linewidth=3, label='Initial Cluster Centers')

plt.xlabel('Eruption Duration (minutes)')
plt.ylabel('Waiting Time (minutes)')
plt.title('Initial Cluster Assignments based on Posterior Probabilities')
plt.legend()
plt.grid(True)
plt.show()
```



Part c

```python
def update_means(X, posteriors):
    #num is sum of weighted points
    weighted_sum = np.dot(posteriors.T, X)
    #denom is sum of weights
    sum_posteriors = posteriors.sum(axis=0)

    new_means = weighted_sum / sum_posteriors[:, np.newaxis]
    return new_means

def update_covariances(X, means, posteriors):
    n_components = means.shape[0]
    n_features = X.shape[1]
    new_covs = []

    for k in range(n_components):
        diff = X - means[k]
        weighted_diff = diff * np.sqrt(posteriors[:, k:k+1])
        cov = np.dot(weighted_diff.T, weighted_diff) / posteriors[:, k].sum()
        new_covs.append(np.diag(np.diag(cov)))

    return new_covs

def update_weights(posteriors):
    N = posteriors.shape[0]
    new_weights = posteriors.sum(axis=0) / N
    return new_weights

def m_step(X, posteriors):
    #update means
    new_means = update_means(X, posteriors)

    #update covariaces
    new_covs = update_covariances(X, new_means, posteriors)

    #update weights
    new_weights = update_weights(posteriors)

    return new_means, new_covs, new_weights
```

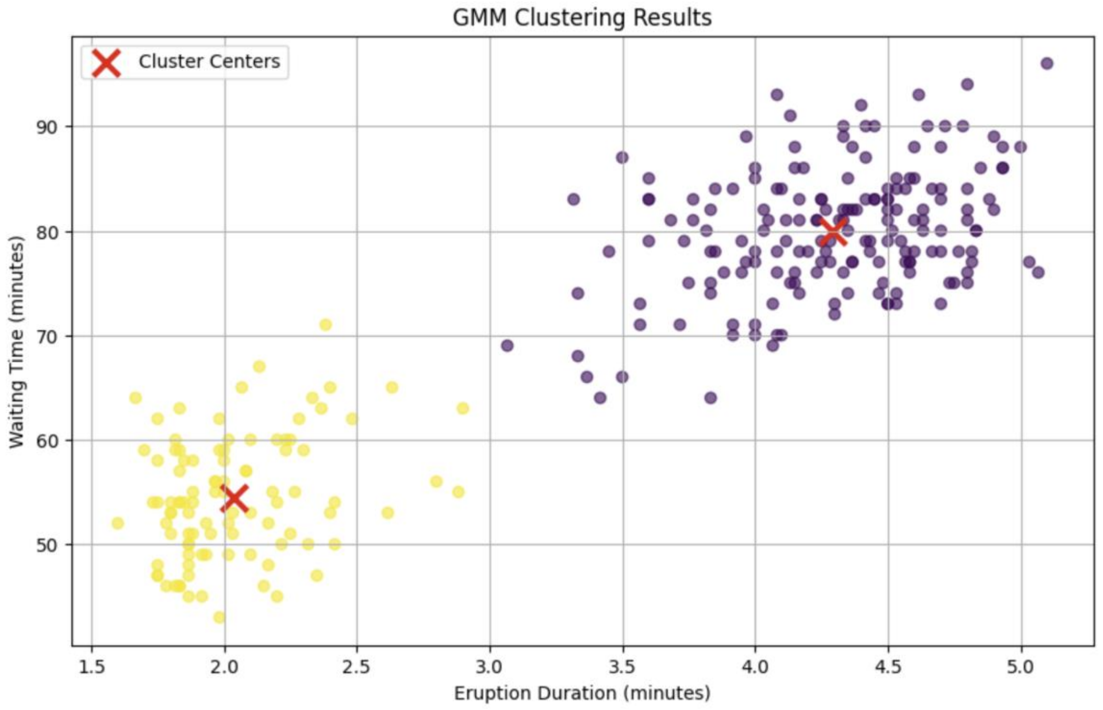
Part di

```python
class GMM:
    def __init__(self, n_components=2, max_iter=100, tol=1e-4):
        self.n_components = n_components
        self.max_iter = max_iter
        self.tol = tol

    def initialize_parameters(self, X):
        n_samples, n_features = X.shape
        random_idx = np.random.choice(n_samples, self.n_components, replace=False)
        self.means = X[random_idx]
        self.covariances = [np.diag(np.var(X, axis=0)) for _ in range(self.n_components)]
        self.weights = np.ones(self.n_components) / self.n_components

    def e_step(self, X):
        n_samples = X.shape[0]

        #likelihood of each cluster
        likelihoods = np.zeros((n_samples, self.n_components))

        for k in range(self.n_components):
            gaussian = multivariate_normal(mean=self.means[k], cov=self.covariances[k])
            likelihoods[:, k] = self.weights[k] * gaussian.pdf(X)

        posteriors = likelihoods / (likelihoods.sum(axis=1, keepdims=True) + 1e-10)

        return posteriors
    def m_step(self, X, posteriors):
        n_samples = X.shape[0]

        #update weights
        self.weights = posteriors.sum(axis=0) / n_samples

        #update means
        for k in range(self.n_components):
            self.means[k] = np.sum(X * posteriors[:, k:k+1], axis=0) / (posteriors[:, k].sum() + 1e-10)

        #update covariances
        for k in range(self.n_components):
            diff = X - self.means[k]
            weighted_diff = diff * np.sqrt(posteriors[:, k:k+1])
            self.covariances[k] = np.diag(
                np.diag(np.dot(weighted_diff.T, weighted_diff) /
                (posteriors[:, k].sum() + 1e-10))
            )
```

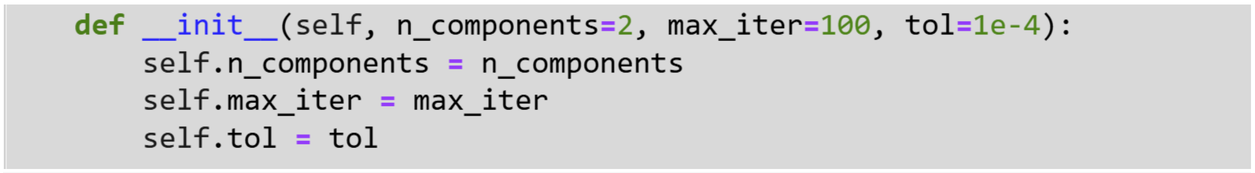
```
    def fit(self, X):
        self.initialize_parameters(X)
        prev_log_likelihood = float('-inf')
        self.mean_trajectories = [[] for _ in range(self.n_components)]

        for iteration in range(self.max_iter):
            for k in range(self.n_components):
                self.mean_trajectories[k].append(self.means[k].copy())

            #e step
            posteriors = self.e_step(X)

            #m step
            self.m_step(X, posteriors)

            #log likelihood
            log_likelihood = self.calculate_log_likelihood(X)

            #checking for convergence
            if abs(log_likelihood - prev_log_likelihood) < self.tol:
                break

            prev_log_likelihood = log_likelihood

        return self
```

```
    def calculate_log_likelihood(self, X):
        n_samples = X.shape[0]
        likelihoods = np.zeros((n_samples, self.n_components))

        for k in range(self.n_components):
            gaussian = multivariate_normal(mean=self.means[k], cov=self.covariances[k])
            likelihoods[:, k] = self.weights[k] * gaussian.pdf(X)

        return np.sum(np.log(np.sum(likelihoods, axis=1) + 1e-10))

    def predict(self, X):
        posteriors = self.e_step(X)
        return np.argmax(posteriors, axis=1)
```

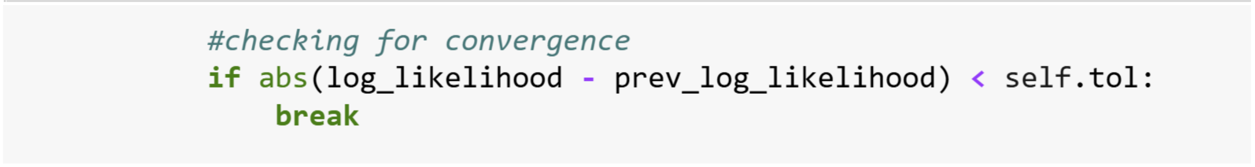
```
gmm = GMM(n_components=2, max_iter=100, tol=1e-4)
gmm.fit(feature_vectors)

#get clusters
cluster_assignments = gmm.predict(feature_vectors)

plt.figure(figsize=(10, 6))

#plot data by clusters
plt.scatter(feature_vectors[:, 0], feature_vectors[:, 1],
           c=cluster_assignments, cmap='viridis', alpha=0.6)

#plot clusters
plt.scatter(gmm.means[:, 0], gmm.means[:, 1],
           c='red', marker='x', s=200, linewidth=3,
           label='Cluster Centers')

plt.xlabel('Eruption Duration (minutes)')
plt.ylabel('Waiting Time (minutes)')
plt.title('GMM Clustering Results')
plt.legend()
plt.grid(True)
plt.show()
```



Part dii



```
    def __init__(self, n_components=2, max_iter=100, tol=1e-4):
        self.n_components = n_components
        self.max_iter = max_iter
        self.tol = tol
```

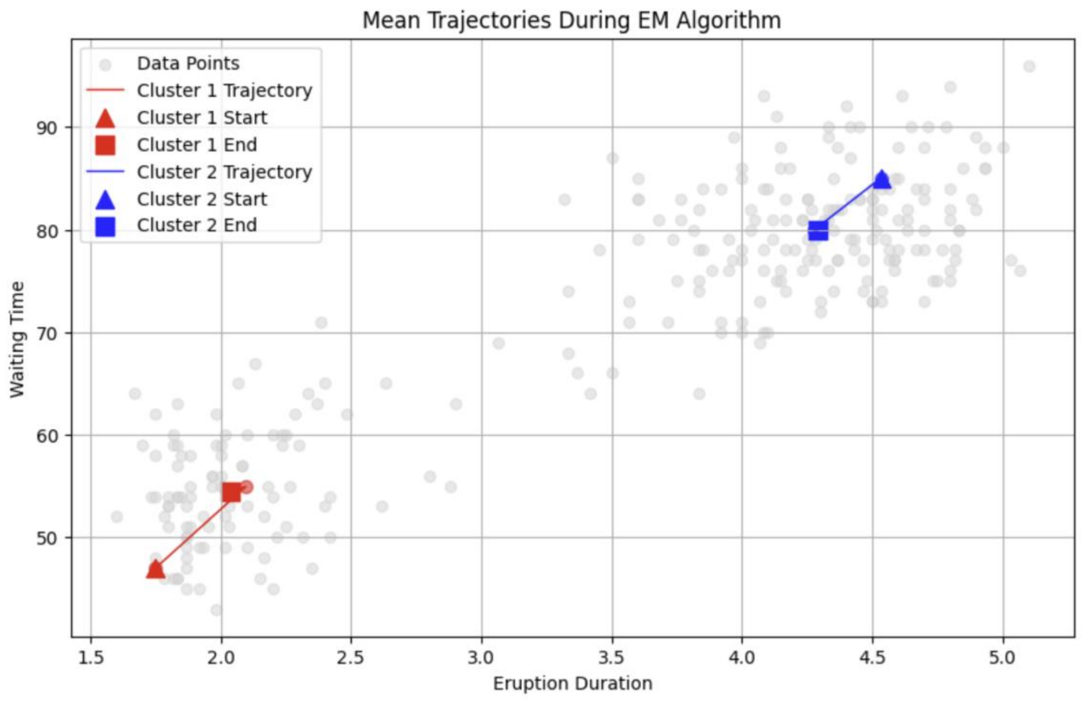


```
            #checking for convergence
            if abs(log_likelihood - prev_log_likelihood) < self.tol:
                break
```

These are the termination criterion that I chose. The first one ensures that the algorithm will stop after 100 iterations even if convergence isn't reached to not have infinte loops and just in case the algorithm does not reach convergence. The second criteria determines how well the model fits the data by checking how small the change is. If the change is less than 0.01% then it means the model has reached convergence.

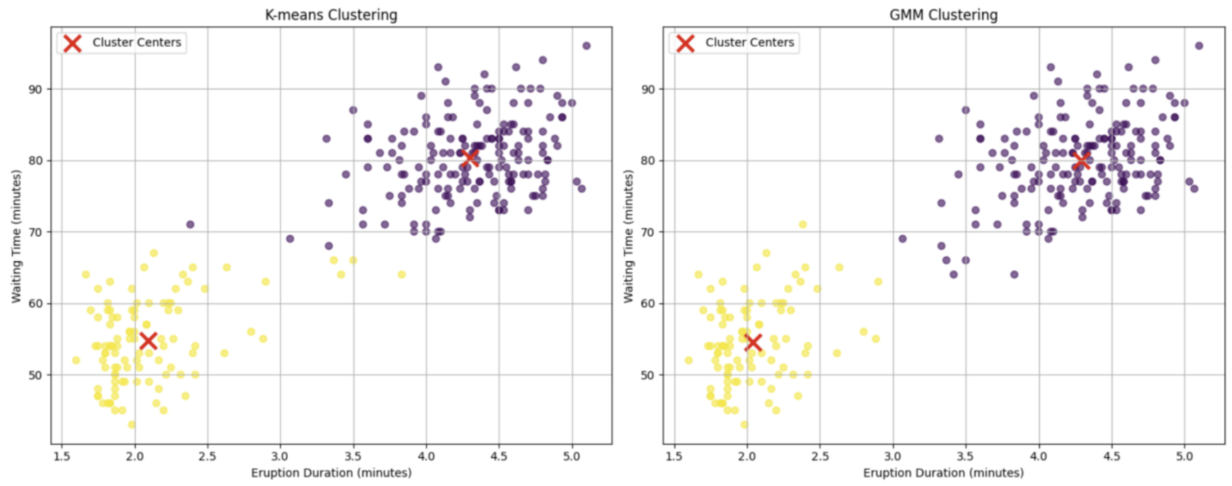
Part diii

```python
def plot_mean_trajectories(gmm, feature_vectors):
    plt.figure(figsize=(10, 6))

    #plot original points
    plt.scatter(feature_vectors[:, 0], feature_vectors[:, 1],
                color='lightgray', alpha=0.5, label='Data Points')

    trajectories = np.array(gmm.mean_trajectories)

    #plot trajectory for 2 clusters
    colors = ['red', 'blue']
    for k in range(gmm.n_components):
        trajectory = np.array([point for point in trajectories[k]])

        #trajectory line
        plt.plot(trajectory[:, 0], trajectory[:, 1],
                 color=colors[k], linestyle='-', linewidth=1,
                 label=f'Cluster {k+1} Trajectory')

        plt.scatter(trajectory[:, 0], trajectory[:, 1],
                    color=colors[k], marker='o', s=50, alpha=0.5)

        #Start an dend points
        plt.scatter(trajectory[0, 0], trajectory[0, 1],
                    color=colors[k], marker='^', s=100,
                    label=f'Cluster {k+1} Start')
        plt.scatter(trajectory[-1, 0], trajectory[-1, 1],
                    color=colors[k], marker='s', s=100,
                    label=f'Cluster {k+1} End')

    plt.xlabel('Eruption Duration')
    plt.ylabel('Waiting Time')
    plt.title('Mean Trajectories During EM Algorithm')
    plt.legend()
    plt.grid(True)
    plt.show()

gmm = GMM(n_components=2)
gmm.fit(feature_vectors)

# Plot the trajectories
plot_mean_trajectories(gmm, feature_vectors)
```



Part e

```python
def compare_kmeans_gmm(feature_vectors, gmm):
    kmeans = KMeans(n_clusters=2, random_state=42)
    kmeans_labels = kmeans.fit_predict(feature_vectors)

    gmm_labels = gmm.predict(feature_vectors)
    fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(15, 6))

    ax1.scatter(feature_vectors[:, 0], feature_vectors[:, 1],
                c=kmeans_labels, cmap='viridis', alpha=0.6)
    ax1.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1],
                c='red', marker='x', s=200, linewidth=3, label='Cluster Centers')
    ax1.set_title('K-means Clustering')
    ax1.set_xlabel('Eruption Duration (minutes)')
    ax1.set_ylabel('Waiting Time (minutes)')
    ax1.legend()
    ax1.grid(True)

    ax2.scatter(feature_vectors[:, 0], feature_vectors[:, 1],
                c=gmm_labels, cmap='viridis', alpha=0.6)
    ax2.scatter(gmm.means[:, 0], gmm.means[:, 1],
                c='red', marker='x', s=200, linewidth=3, label='Cluster Centers')
    ax2.set_title('GMM Clustering')
    ax2.set_xlabel('Eruption Duration (minutes)')
    ax2.set_ylabel('Waiting Time (minutes)')
    ax2.legend()
    ax2.grid(True)

    plt.tight_layout()
    plt.show()

#comparisons
gmm = GMM(n_components=2)
gmm.fit(feature_vectors)
compare_kmeans_gmm(feature_vectors, gmm)
```



The clusters are ever so slightly different, but still pretty similar other than a few points. A possible reason as to why there's a differenc is because K-Means clusters in linear boundaries while GMM can have curved boundaries. In this case, that could explain why there's a few points in the duration around 3.3-3.7 minutes that K-Means classified in the yellow group. Ultimately, GMM has a better performance.

Question 2

Part a

This part consisted of just downloading the face dataset onto my personal computer.

Part b

```python
TRAIN_PATH = r"C:\Users\Sofia Beyerlein\Desktop\Cornell Graduate\Applied Machine Learning\hw4\faces\train.txt"
TEST_PATH = r"C:\Users\Sofia Beyerlein\Desktop\Cornell Graduate\Applied Machine Learning\hw4\faces\test.txt"
IMAGE_DIR = r"C:\Users\Sofia Beyerlein\Desktop\Cornell Graduate\Applied Machine Learning\hw4\faces\images"
```

```python
def load_face_data(file_path, is_training=True):
    labels, data = [], []

    #grab each image
    for line in open(file_path):
        #image into path and label
        parts = line.strip().split()
        image_path = IMAGE_DIR + "\\" + parts[0].split("/")[-1]
        label = parts[1]

        im = imageio.imread(image_path)
        data.append(im.reshape(2500,))
        labels.append(label)

    data_matrix = np.array(data, dtype=float)
    labels = np.array(labels, dtype=int)

    return data_matrix, labels

def display_face(image_vector, title="Face Image"):
    #reshape 2500-dim vector to 50x50 image
    face_image = image_vector.reshape(50, 50)

    plt.figure(figsize=(5, 5))
    plt.imshow(face_image, cmap=cm.Greys_r)
    plt.title(title)
    plt.axis('off')
    plt.show()
```

```python
#load training data and testing data
X_train, y_train = load_face_data(TRAIN_PATH, is_training=True)
X_test, y_test = load_face_data(TEST_PATH, is_training=False)

#display example images from both sets
print("\nDisplaying sample images...")
print("Sample training image (index 10):")
display_face(X_train[10], "Sample Training Face")

print("\nSample test image (index 5):")
display_face(X_test[5], "Sample Test Face")

#check dimensions
print(f"Training set (X_train): {X_train.shape} (should be 540x2500)")
print(f"Test set (X_test): {X_test.shape} (should be 100x2500)")
```

```
Displaying sample images...
Sample training image (index 10):
```

Sample Training Face

```
Sample test image (index 5):
```

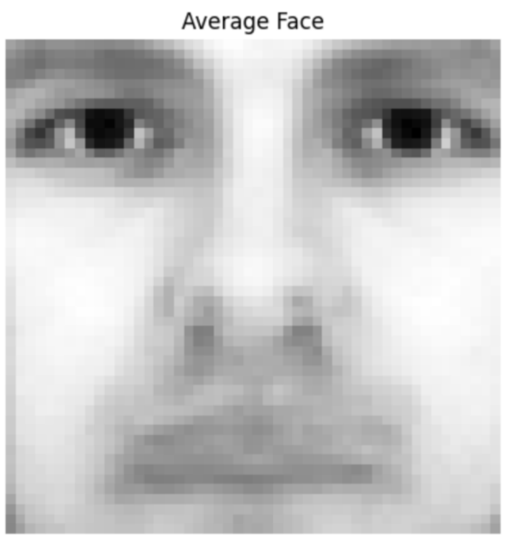
Sample Test Face

```
Training set (X_train): (540, 2500) (should be 540x2500)
Test set (X_test): (100, 2500) (should be 100x2500)
```

Part c

```
def compute_average_face(X_train):
    mu = np.mean(X_train, axis=0)
    return mu

def display_face(image_vector, title="Face Image"):
    face_image = image_vector.reshape(50, 50)

    plt.figure(figsize=(5, 5))
    plt.imshow(face_image, cmap=cm.Greys_r)
    plt.title(title)
    plt.axis('off')
    plt.show()

average_face = compute_average_face(X_train)

#display the average face
print("Displaying average face computed from training set:")
display_face(average_face, "Average Face")
```

```
Displaying average face computed from training set:
```

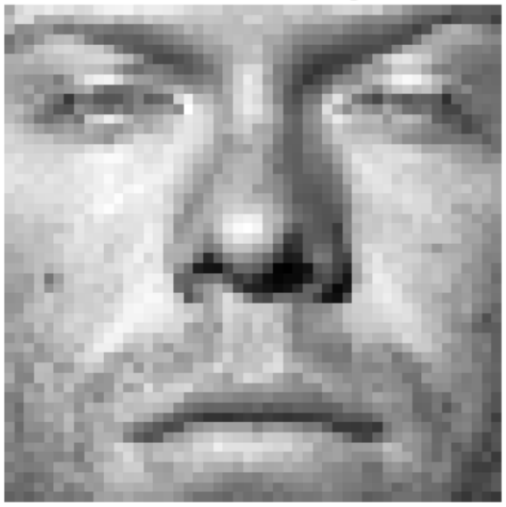


Part d

```python
def subtract_mean(X, mu):
    return X - mu

def display_face(image_vector, title="Face Image"):
    face_image = image_vector.reshape(50, 50)
    plt.figure(figsize=(5, 5))
    plt.imshow(face_image, cmap=cm.Greys_r)
    plt.title(title)
    plt.axis('off')
    plt.show()

X_train_centered = subtract_mean(X_train, average_face)
X_test_centered = subtract_mean(X_test, average_face)

print("Mean-subtracted training face:")
display_face(X_train_centered[10], "Mean-subtracted Training Face")

print("\nMean-subtracted test face:")
display_face(X_test_centered[5], "Mean-subtracted Test Face")
```

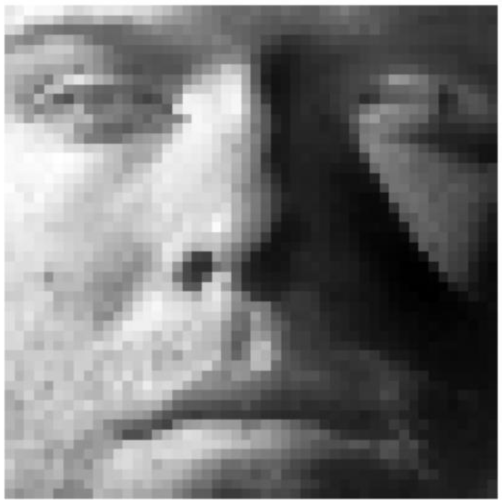
```
Mean-subtracted training face:
```

Mean-subtracted Training Face



```
Mean-subtracted test face:
```

Mean-subtracted Test Face

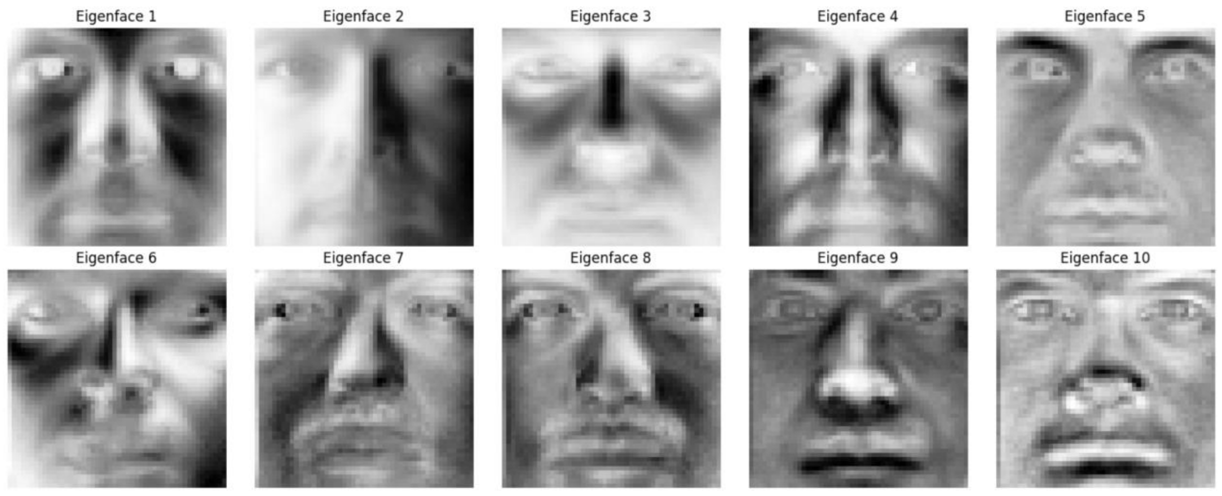


Part e

```
def compute_eigenfaces(X_centered):
    #compute X^T * X
    covariance_matrix = np.dot(X_centered.T, X_centered)

    #eigendecomposition
    eigenvalues, eigenvectors = np.linalg.eigh(covariance_matrix)

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

    #transpose to get V^T
    V_t = eigenvectors.T

    return eigenvalues, V_t

def display_eigenfaces(eigenfaces, n_components=10):
    fig, axes = plt.subplots(2, 5, figsize=(15, 6))
    axes = axes.ravel()

    for i in range(n_components):
        eigenface = eigenfaces[i].reshape(50, 50)
        axes[i].imshow(eigenface, cmap=cm.Greys_r)
        axes[i].axis('off')
        axes[i].set_title(f'Eigenface {i+1}')

    plt.tight_layout()
    plt.show()
```

```
#compute eigenfaces
eigenvalues, V_t = compute_eigenfaces(X_train_centered)

#display the first 10 eigenfaces
print("\nDisplaying first 10 eigenfaces:")
display_eigenfaces(V_t, n_components=10)
```

```
Displaying first 10 eigenfaces:
```



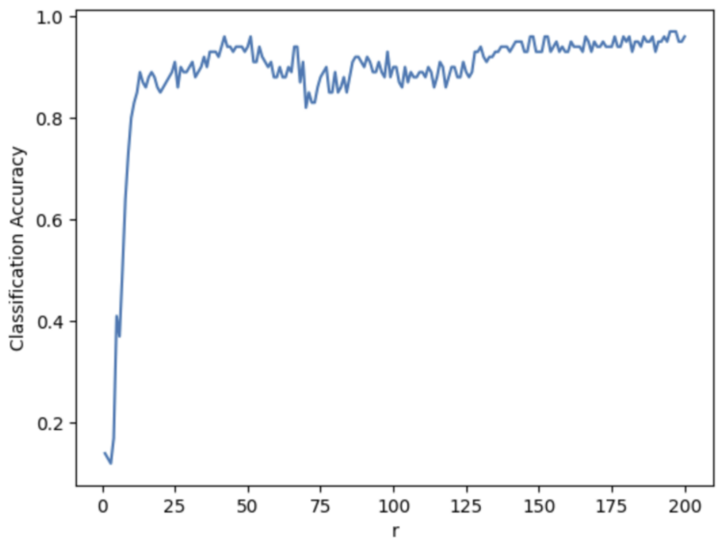
Part f

```
def compute_eigenface_features(X, V_t, r):
    V_t_r = V_t[:r, :]
    F = np.dot(X, V_t_r.T)

    return F
```

Part g

```
r = 10
F_train_10 = compute_eigenface_features(X_train_centered, V_t, r)
F_test_10 = compute_eigenface_features(X_test_centered, V_t, r)

#train and test logistic regression
clf = LogisticRegression(multi_class='ovr', max_iter=1000)
clf.fit(F_train_10, y_train)
accuracy_10 = clf.score(F_test_10, y_test)
print(f"Classification accuracy on test set (r=10): {accuracy_10:.4f}")

#test accuracy for r = 1 to 200
r_values = range(1, 201)
accuracies = []

for r in r_values:
    F_train_r = compute_eigenface_features(X_train_centered, V_t, r)
    F_test_r = compute_eigenface_features(X_test_centered, V_t, r)

    clf = LogisticRegression(multi_class='ovr', max_iter=1000)
    clf.fit(F_train_r, y_train)
    accuracy = clf.score(F_test_r, y_test)
    accuracies.append(accuracy)

#plot accuracy vs number of eigenfaces
plt.plot(r_values, accuracies)
plt.xlabel('r')
plt.ylabel('Classification Accuracy')
plt.show()
```

```
Classification accuracy on test set (r=10): 0.8000
```



Written Exercises

Question 1

<u>Question 1</u>

- $X = UDV^T$
- U and V are orthonormel matrices
- D is diagonal matrix with non-negative values

Compute $X^T X$:

$X^T X = (UDV^T)^T (UDV^T)$

Apply Transpose:

$X^T X = (VD^T U^T)(UDV^T)$

Simplify:

$X^T X = VD^T (U^T U) DV^T$

$X^T X = VD^T I DV^T$

$X^T X = VD^T DV^T$

$D^T D = \Sigma^2$

Substitute:

$X^T X = V\Sigma^2 V^T$

Conclusion:

$X = UDV^T$ where V has eigenvectors $X^T X$ and diagonal elements of Σ^2 which are eigenvalues of $X^T X$

Question 2

Part a

```
import numpy as np
```

Part a

```
M = np.array([
    [1, 0, 3],
    [3, 7, 2],
    [2, -2, 8],
    [0, -1, 1],
    [5, 8, 7]
])
```

```
#M^TM
M_transpose = M.T
MT_M = np.dot(M_transpose, M)
print("M^T M = \n", MT_M)
```

```
M^T M =
 [[ 39  57  60]
 [ 57 118  53]
 [ 60  53 127]]
```

```
#MM^T
M_MT = np.dot(M, M_transpose)
print("MM^T = \n", M_MT)
```

```
MM^T =
 [[ 10   9  26   3  26]
 [  9  62   8  -5  85]
 [ 26   8  72  10  50]
 [  3  -5  10   2  -1]
 [ 26  85  50  -1 138]]
```

Part b

Part b

```
eigenvals_MT_M = np.linalg.eigvals(MT_M)
eigenvals_MT_M = np.sort(eigenvals_MT_M)[::-1]
print("Eigenvalues of M^T M:")
print(eigenvals_MT_M)
```

```
Eigenvalues of M^T M:
[ 2.14670489e+02  6.93295108e+01 -4.65183174e-15]
```

```
eigenvals_M_MT = np.linalg.eigvals(M_MT)
eigenvals_MT_M = np.sort(eigenvals_M_MT)[::-1]
print("\nEigenvalues of MM^T:")
print(eigenvals_M_MT)
```

```
Eigenvalues of MM^T:
[ 2.14670489e+02 -1.98884150e-14  6.93295108e+01 -1.02373098e-15
  2.14922736e-15]
```

Part c

Part c

```
eigenvals_MT_M, eigenvecs_MT_M = np.linalg.eigh(MT_M)

idx_MT_M = eigenvals_MT_M.argsort()[::-1]
eigenvals_MT_M = eigenvals_MT_M[idx_MT_M]
eigenvecs_MT_M = eigenvecs_MT_M[:, idx_MT_M]

eigenvals_M_MT, eigenvecs_M_MT = np.linalg.eigh(M_MT)

idx_M_MT = eigenvals_M_MT.argsort()[::-1]
eigenvals_M_MT = eigenvals_M_MT[idx_M_MT]
eigenvecs_M_MT = eigenvecs_M_MT[:, idx_M_MT]

print("\nEigenvectors of M^T M (columns):")
print(eigenvecs_MT_M)

print("\nEigenvectors of MM^T (columns):")
print(eigenvecs_M_MT)
```

```
Eigenvectors of M^T M (columns):
[[ 0.42615127  0.01460404  0.90453403]
 [ 0.61500884  0.72859799 -0.30151134]
 [ 0.66344497 -0.68478587 -0.30151134]]

Eigenvectors of MM^T (columns):
[[-0.16492942  0.24497323  0.0696646   0.04259611  0.95190273]
 [-0.47164732 -0.45330644  0.71521789  0.24438161 -0.02833866]
 [-0.33647055  0.82943965  0.32430825 -0.09375112 -0.29129423]
 [-0.00330585  0.16974659 -0.22438689  0.95699161 -0.07065942]
 [-0.79820031 -0.13310656 -0.57278695 -0.11764774 -0.05685965]]
```

Part d

Part d

```
U, s, VT = np.linalg.svd(M)
U = U[:, :2]
s = s[:2]
VT = VT[:2, :]

Sigma = np.zeros((2, 3))
Sigma[0, 0] = s[0]
Sigma[1, 1] = s[1]

print("U (5x2):")
print(U)
print("\nΣ (2x3):")
print(Sigma)
print("\nV^T (2x3):")
print(VT)
```

```
U (5x2):
[[-0.16492942 -0.24497323]
 [-0.47164732  0.45330644]
 [-0.33647055 -0.82943965]
 [-0.00330585 -0.16974659]
 [-0.79820031  0.13310656]]

Σ (2x3):
[[14.65163776  0.          0.        ]
 [ 0.          8.32643446  0.        ]]

V^T (2x3):
[[-0.42615127 -0.61500884 -0.66344497]
 [ 0.01460404  0.72859799 -0.68478587]]
```

Part e

Part e

```
U, s, VT = np.linalg.svd(M)
s[1:] = 0
M_approx = np.dot(U[:, :1] * s[0], VT[0:1, :])

print("1D approximation of M:")
print(M_approx)
```

```
1D approximation of M:
[[1.02978864 1.48616035 1.60320558]
 [2.94487812 4.24996055 4.58467382]
 [2.10085952 3.031898   3.27068057]
 [0.02064112 0.02978864 0.0321347 ]
 [4.9838143  7.19249261 7.75895028]]
```