

# CS5785 Applied Machine Learning Homework 3

Author: Yixuan (Rylee) Li - yl2557  
Teammate: Ningran Song - ns632

October 2021 - \*Late Submission\*

# 1. *CODING EXERCISES*

## (1) Eigenface for face recognition

### (a) Download The Face Dataset and unzip faces.zip

Completed.

### (b) Load the training set into a matrix X

Loaded data using the following code. The inspection returns (540, 2500) for train data, (540,) for train labels, (100, 2500) for test data, and (100,) for test labels.

```
# import packages
import numpy as np
# from scipy import misc
from imageio import imread
from matplotlib import pylab as plt
import matplotlib.cm as cm
from sklearn.linear_model import LogisticRegression
%matplotlib inline

# Load the training set into a matrix X
train_labels, train_data = [], []
for line in open('./faces/train.txt'):
    im = imread(line.strip().split()[0])
    train_data.append(im.reshape(2500,))
    train_labels.append(line.strip().split()[1])
train_data, train_labels = np.array(train_data, dtype=float), np.array(train_labels, dtype=int)

# Load the test set into a matrix X
test_labels, test_data = [], []
for line in open('./faces/test.txt'):
    im = imread(line.strip().split()[0])
    test_data.append(im.reshape(2500,))
    test_labels.append(line.strip().split()[1])
test_data, test_labels = np.array(test_data, dtype=float), np.array(test_labels, dtype=int)

# inspect data
print(train_data.shape, train_labels.shape)
print(test_data.shape, test_labels.shape)
```

Picked a face image from training set and display that image in grayscale:

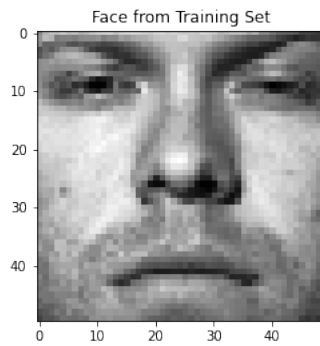


Figure 1.1: Face Image from Training Set

Picked a face image from test set and display that image in grayscale:

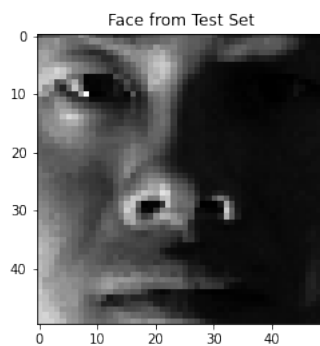


Figure 1.2: Face Image from Test Set

The size of matrix  $X_{test}$  for the test set should be  $100 \times 2500$ . The output of data inspection above was:

```
(540, 2500) (540,)
(100, 2500) (100,)
```

### (c) Average Face

```
# Compute the average face  $\mu$  from the whole training set
# by summing up every row in X then dividing by the number of faces
avg_face = np.mean(train_data, axis=0)
# print mu
# output is [59.25185185 56.10185185 52.42222222 ... 67.22222222 64.61851852 59.27592593]
print(avg_face)
# Display the average face as a grayscale image
plt.imshow(avg_face.reshape(50,50), cmap = cm.Greys_r)
plt.title('Average face from the whole training set')
plt.savefig("average_face_train_set") # save pics
plt.show()
```

Display the average face as a grayscale image:



Figure 1.3: Average face from Train Set

**(d) Mean Subtraction**

```

# Subtract average face  $\mu$  from every row in  $X$ .
train_ms = train_data - avg_face
test_ms = test_data - avg_face
# print train_ms and test_ms
print(train_ms)
print(test_ms)
# Train - Pick a face image after mean subtraction from the new X
# and display that image in grayscale
plt.imshow(train_ms[10,:].reshape(50,50), cmap = cm.Greys_r)
plt.title("Train - Face with Mean Subtraction")
plt.savefig("train_mean_sub") # save pics
plt.show()
# Test - Pick a face image after mean subtraction from the new X
# and display that image in grayscale
plt.imshow(test_ms[10,:].reshape(50,50), cmap = cm.Greys_r)
plt.title("Test - Face with Mean Subtraction")
plt.savefig("test_mean_sub") # save pics
plt.show()

```

Pick a face image after mean subtraction from the new  $X$  (train) and display that image in grayscale:

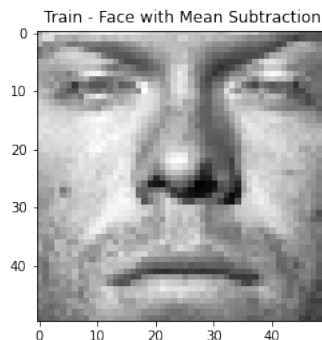


Figure 1.4: Train - Face with Mean Subtraction

Pick a face image after mean subtraction from the new X (test) and display that image in grayscale:

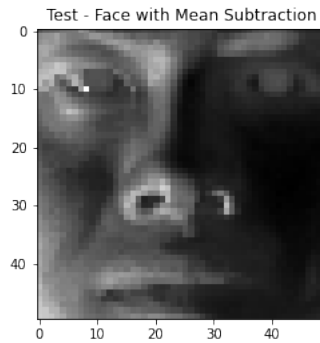


Figure 1.5: Test - Face with Mean Subtraction

### (e) Eigenface

```
# Perform eigendecomposition
from numpy.linalg import svd
U, S, V = svd(train_ms, full_matrices=False)
# Output is (540, 540) (540,) (540, 2500)
print(U.shape, S.shape, V.shape)

# Display the first 10 eigenfaces as 10 images in grayscale
plt.figure(figsize=(10, 5))
for i in range(10):
    plt.subplot(2, 5, i + 1)
    plt.imshow(V[i,:].reshape(50,50), cmap = cm.Greys_r)
    plt.title('Eigenface: #%d' % (i+1))
plt.savefig("first_10_eigenfaces") # save pics
plt.show()
```

Display the first 10 eigenfaces as 10 images in grayscale:

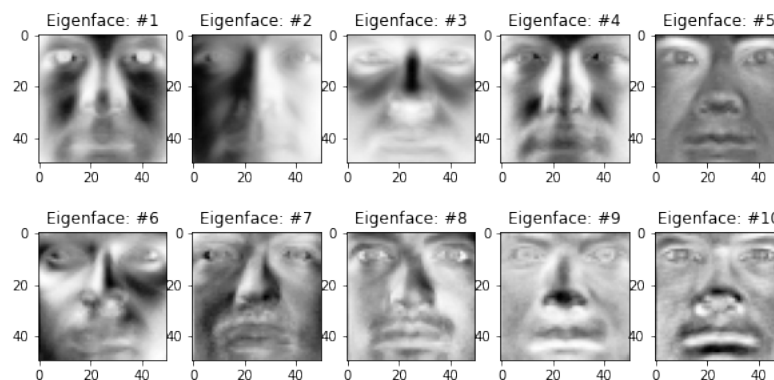


Figure 1.6: First 10 Eigenfaces

## (f) Eigenface Feature

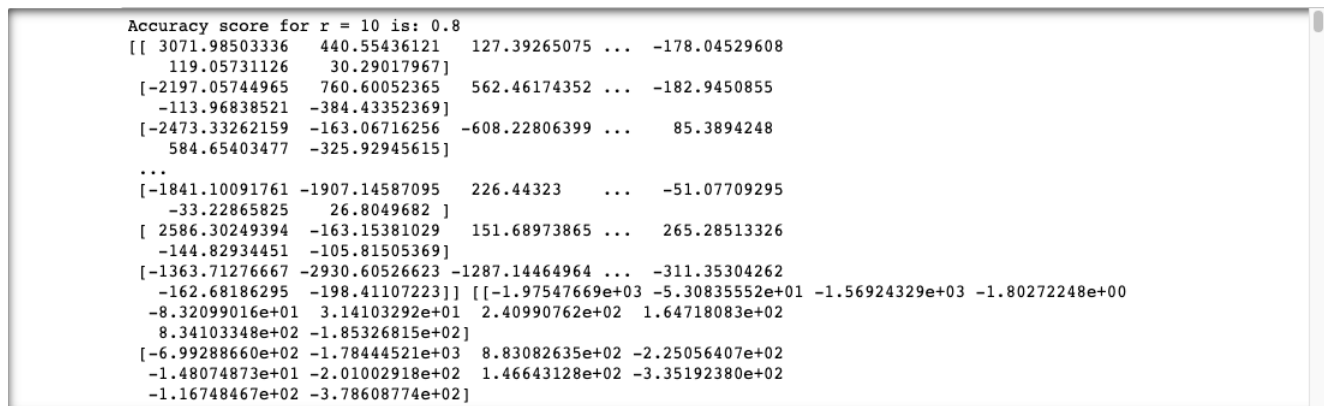
```
# Write a function to generate r-dimensional feature matrix F and Ftest
# for training images X and test images Xtest, respectively
def get_eigenface_feature(r):
    V_T_R = V[:,r,:].T
    F = np.dot(train_ms, V_T_R)
    F_test = np.dot(test_ms, V_T_R)
    return (F, F_test)
```

## (g) Face Recognition

Extracted training and test features for  $r = 10$ , and the accuracy score for  $r = 10$  was 0.8

```
# Train a Logistic Regression model using F and test on F test
from sklearn.linear_model import LogisticRegression
model = LogisticRegression(multi_class="ovr", max_iter = int(1e4))

# Extract training and test features for r = 10
F, F_test = get_eigenface_feature(10)
model.fit(F, train_labels)
print ("Accuracy score for r = 10 is: " + str(model.score(F_test, test_labels)))
print (F, F_test)
```



```
Accuracy score for r = 10 is: 0.8
[[ 3071.98503336  440.55436121  127.39265075 ... -178.04529608
  119.05731126   30.29017967]
 [-2197.05744965  760.60052365  562.46174352 ... -182.9450855
 -113.96838521 -384.43352369]
 [-2473.33262159 -163.06716256 -608.22806399 ...  85.3894248
  584.65403477 -325.92945615]
 ...
 [-1841.10091761 -1907.14587095  226.44323 ... -51.07709295
 -33.22865825  26.8049682 ]
 [ 2586.30249394 -163.15381029  151.68973865 ...  265.28513326
 -144.82934451 -105.81505369]
 [-1363.71276667 -2930.60526623 -1287.14464964 ... -311.35304262
 -162.68186295 -198.41107223]] [[-1.97547669e+03 -5.30835552e+01 -1.56924329e+03 -1.80272248e+00
 -8.32099016e+01  3.14103292e+01  2.40990762e+02  1.64718083e+02
  8.34103348e+02 -1.85326815e+02]
 [-6.99288660e+02 -1.78444521e+03  8.83082635e+02 -2.25056407e+02
 -1.48074873e+01 -2.01002918e+02  1.46643128e+02 -3.35192380e+02
 -1.16748467e+02 -3.78608774e+02]
```

Figure 1.7:  $r = 10$ : output (partial)

Train a Logistic Regression model using F and test on F test:

```
x, y = [], []
for r in range(1, 201):
    F, F_test = get_eigenface_feature(r)
    model.fit(F, train_labels)
    x.append(r)
    y.append(model.score(F_test, test_labels))
```

Report the classification accuracy on the test set:

```
# Report the classification accuracy on the test set
plt.plot(x, y)
plt.savefig("classification_accuracy") # save pics
plt.title('Classification Accuracy - Test Set')
plt.show()
```

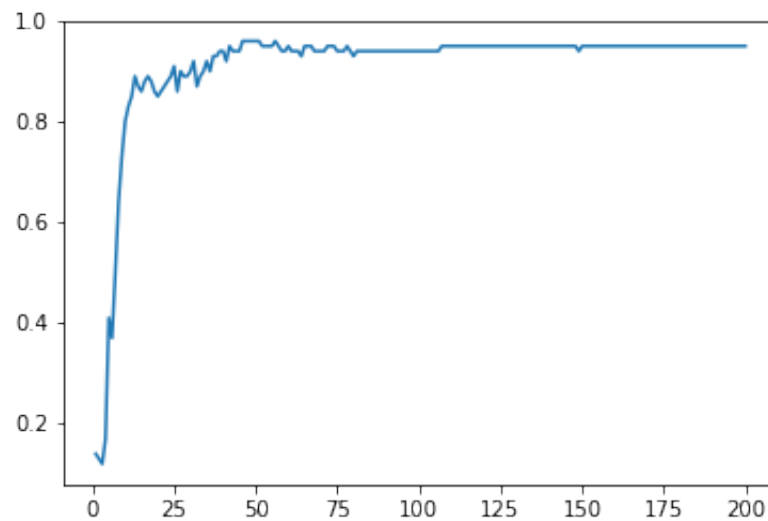


Figure 1.8: Classification Accuracy - Test Set

## (2) Implement EM algorithm

(a) Treat each data entry as a 2 dimensional feature vector. Parse and plot all data points on 2-D plane.

I parsed the dataset and treated each data entry as a 2 dimensional feature vector using the following code. And the shape of the data is (272, 2), where 272 is the number of entry, and 2 is the dimension of the feature vector.

```
# Load Data
data = []
for line in open("./old-faithful-geyser/data.txt"):
    # Treat each data entry as a 2 dimensional feature vector
    entry = []
    entry.append(line.strip().split()[1])
    entry.append(line.strip().split()[2])
    vector = np.array(entry, dtype=float)
    data.append(vector)
# Convert List to Vector
data = np.array(data)
# Inspect data
print("Shape of the data: \n" + str(data.shape) + ", where " + str(data.shape[0]) +
      " is the number of entry, and " + str(data.shape[1]) + "
      is the dimension of the feature vector.")
print("\n Inspect the First 10 elements in the data:")
print(str(data[:10]))
```

```

Shape of the data:
(272, 2), where 272 is the number of entry, and 2 is the dimension of the feature vector.

Inspect the First 10 elements in the data:
[[ 3.6  79. ]
 [ 1.8  54. ]
 [ 3.333 74. ]
 [ 2.283 62. ]
 [ 4.533 85. ]
 [ 2.883 55. ]
 [ 4.7   88. ]
 [ 3.6   85. ]
 [ 1.95  51. ]
 [ 4.35  85. ]]

```

Figure 1.9: Inspection of the Geyser Dataset

And I plotted all data points on 2-D plane using the code below:

```

# Parse and plot all data points on 2-D plane.
plt.scatter(data[:,0], data[:,1], s=10)
plt.xlabel('Eruptions')
plt.ylabel('Duration')
plt.title('Old Faithful Geyser Dataset')
plt.savefig("geyser_data_points") # save pics
plt.show()

```

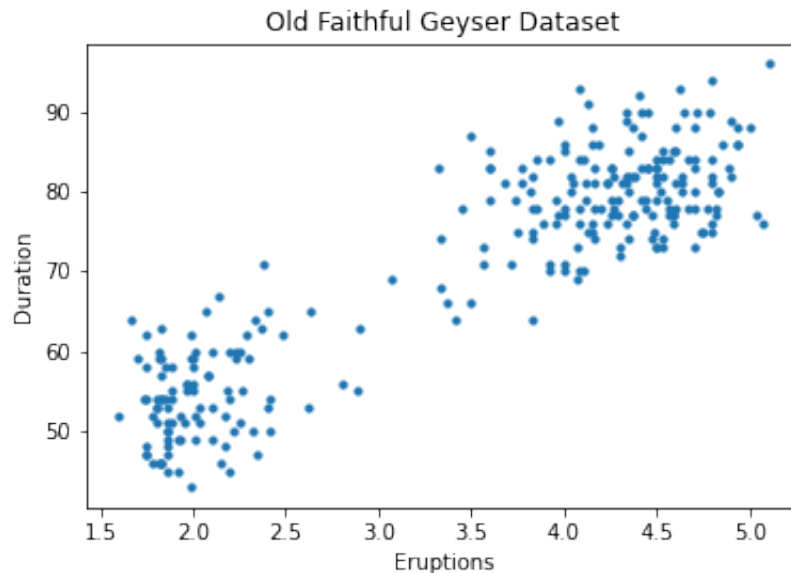


Figure 1.10: Old Faithful Geyser Dataset

### (b) E-Step: Vector of Probabilities $P_{\theta}(z = k \mid x)$

Given a trained model  $P_{\theta}(x, z) = P_{\theta}(x|z)P_{\theta}(z)$ , we can look at the posterior probability

$$P_{\theta}(z = k \mid x) = \frac{P_{\theta}(z = k, x)}{P_{\theta}(x)} = \frac{P_{\theta}(x|z = k)P_{\theta}(z = k)}{\sum_{l=1}^K P_{\theta}(x|z = l)P_{\theta}(z = l)}$$

of a point  $x$  belonging to class  $k[1]$ .



**(c) M-Step**

Define that  $n_k = |\{i : y^{(i)} = k\}|$  is the number of training targets with class  $k$ . Then we have[1]:

$$\mu_k = \frac{\sum_{i=1}^n P(z = k | x^{(i)}) x^{(i)}}{n_k}$$

$$\Sigma_k = \frac{\sum_{i=1}^n P(z = k | x^{(i)}) (x^{(i)} - \mu_k)(x^{(i)} - \mu_k)^\top}{n_k}$$

$$\phi_k = \frac{n_k}{n}$$

**(d) Implement and run the EM algorithm****i. EM implementation**

I implemented a bimodal GMM model fit using the EM algorithm[2]. And I fitted the dataset into the GMM model using EM algorithm. For this run, it took total number of 30 iterations to reach convergence. The centers are  $[[2.0363884598395923, 54.47851642946882], [4.2896619777147755, 79.96811522972116]]$ . The plots for each iteration final clustering could be found after the code.

```
import numpy as np
from scipy.stats import multivariate_normal as mv_normal
import matplotlib.pyplot as plt

# Implement a bimodal GMM model fit using the EM algorithm
class gmm:
    # Initialization: cluster, dimension, and random state
    def __init__(self, n_clusters=2, n_dimensions=2, random_state=None):
        self.n_clusters = n_clusters
        self.n_dimensions = n_dimensions
        self.rng = np.random.default_rng(seed=random_state)

    # Implement a bimodal GMM model fit using the EM algorithm
    # X: data set
    # tol: convergence threshold
    def fit(self, X, tol=1e-12):
        # n: number of data entry, d: dimension of feature vector
        n, d = X.shape
        # k: number of clusters
        k = self.n_clusters

        # E Step: Compute the posterior probability that each Gaussian generates each datapoint
        def e_step(priors, means, covars):
            resps = np.ndarray((n, k))
            for i in range(k):
                resps[:, i] = priors[i] * mv_normal.pdf(X, means[i], covars[i])
            return resps / resps.sum(axis=1)[:, np.newaxis]

        # M Step: Assuming that the data really was generated this way,
        # change the parameters of each Gaussian to maximize the probability that
        # it would generate the data it is currently responsible for.
        def m_step(resps):
            totals = resps.sum(axis=0)
            priors = totals / n
            means = (resps.T @ X) / totals[:, np.newaxis]
```

```

covars = np.ndarray(shape=(k, d, d))
for i in range(k):
    X_centered = X - means[i]
    covars[i] = (resps[:, i]*X_centered.T) @ X_centered/totals[i]
return priors, means, covars

# Initialization: random hard cluster assignments
Z = np.zeros((n, k))
Z[np.arange(n), self.rng.choice(k, n)] = 1
T = m_step(Z)

# loop until convergence:
converged = False
iteration = 0
mean_list = []

plt.figure(figsize=(40, 80))

while not converged:
    T0 = T
    Z = e_step(*T)
    T = m_step(Z)
    # get list of mu
    mean_list.append(T[1])
    # get plot of each iteration
    arr1, arr2 = [], []
    for i in range(0, 272):
        if Z[i][0] > Z[i][1]:
            arr1.append(X[i])
        else:
            arr2.append(X[i])
    arr1, arr2 = np.array(arr1), np.array(arr2)
    plt.subplot(11, 4, iteration + 1)
    plt.scatter(arr1[:, 0], arr1[:, 1], s=10, c='r', linewidth=0.1)
    plt.scatter(arr2[:, 0], arr2[:, 1], s=10, c='g', linewidth=0.1)
    # plot the center
    plt.scatter(*T[1].T, c=['c', 'm'])
    plt.title("Iteration %s" % (iteration + 1))
    iteration = iteration + 1
    # check sum of squared mean distances to detect convergence
    # where tol is the convergence threshold
    converged = np.sum(np.square(T0[1]-T[1])) < tol

plt.savefig("em_iteration.png", bbox_inches='tight', pad_inches=0) # save pics

# Plot the final clustering when reached termination criterion
fig, ax = plt.subplots()
arr1, arr2 = [], []
for i in range(0, 272):
    if Z[i][0] > Z[i][1]:
        arr1.append(X[i])
    else:
        arr2.append(X[i])

```

```

arr1, arr2 = np.array(arr1), np.array(arr2)
ax.scatter(arr1[:, 0], arr1[:, 1], s=10, c='r', linewidth=0.1)
ax.scatter(arr2[:, 0], arr2[:, 1], s=10, c='g', linewidth=0.1)
cs = ax.scatter(*T[1].T, c=['c', 'm'], s=100) # plot the center
plt.savefig("em_final_clustering") # save pics
fig.canvas.draw()
# Print out the total number of iteration
print("Total Number of Iteration: %s" % iteration)

# save list of mean through iteration
self.mean_list = mean_list
return self

# print out the center using gmm with em implementation
em_center = [[x1[len(mean_list) - 1], y1[len(mean_list) - 1]],
             [x2[len(mean_list) - 1], y2[len(mean_list) - 1]]]
# Output here is [[2.0363884598395923, 54.47851642946882], [4.2896619777147755, 79.96811522972116]]
print("GMM-EM implementation - the centers are " + str(em_center))

```

This is the plot of our final clustering:

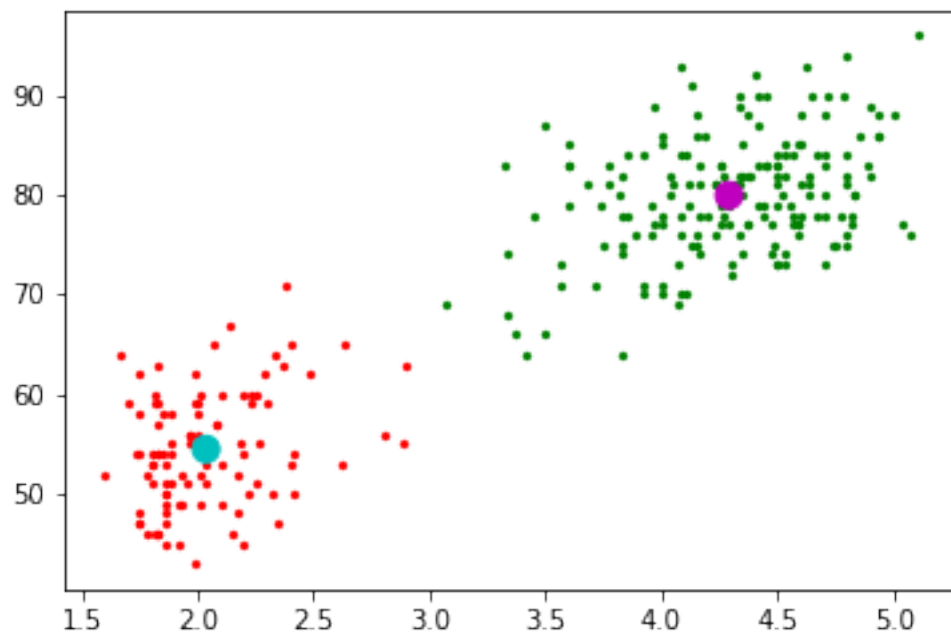


Figure 1.11: EM-Plot of Final Clustering

These are the plots for each iteration:

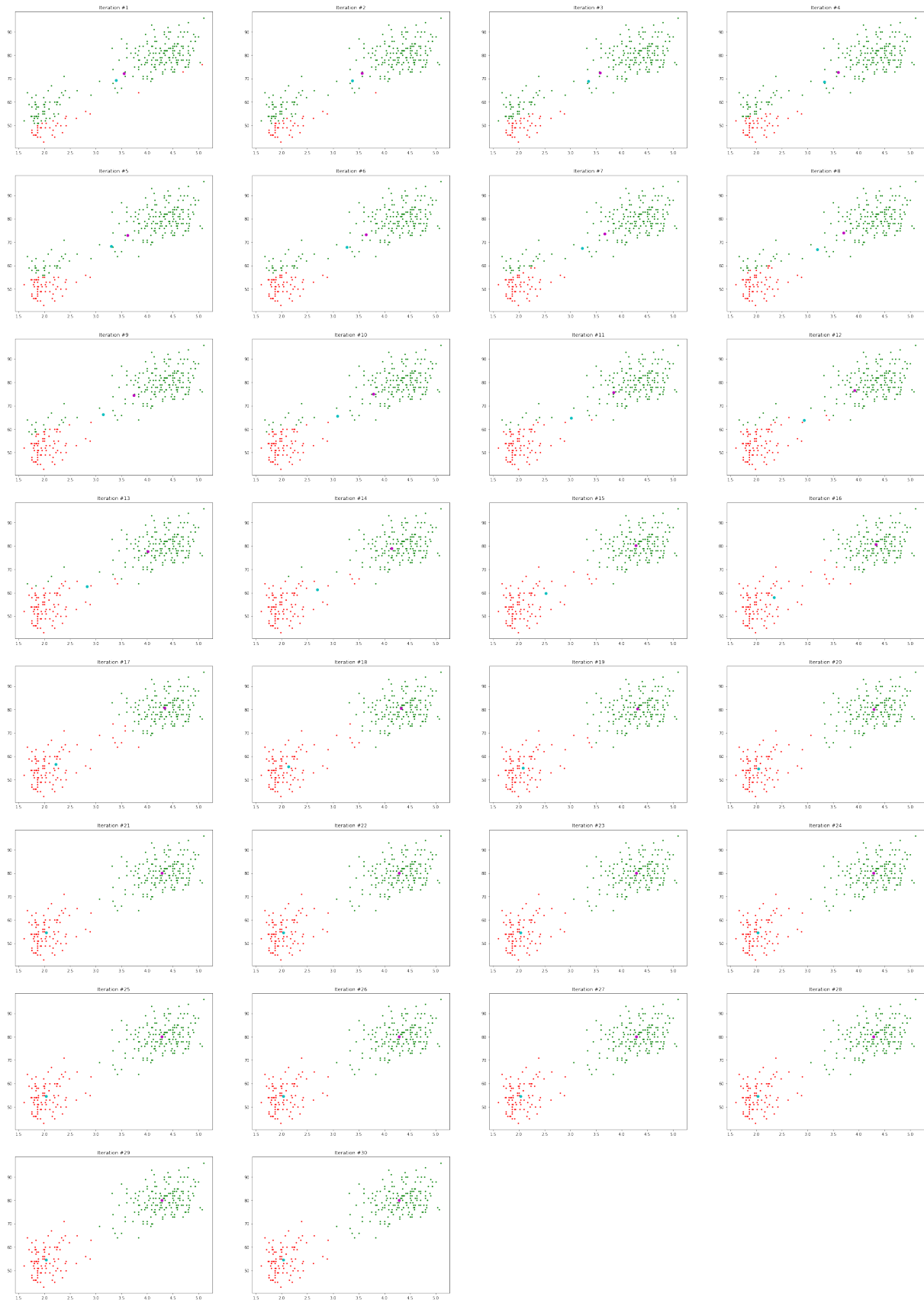


Figure 1.12: EM-Plots for Each Iteration

**ii. Termination criterion**

I chose the convergence threshold (ie.  $tol$ ) to be  $1e-12$ . My termination criterion is the sum of squared mean distance between  $\mu_i$  of this iteration and  $\mu_i$  of last iteration for each  $i$  is less or equal than the convergence threshold (ie.  $tol = 1e-12$ ). The reason is that under this criterion, the GMM parameter estimates become stable and few further improvements can be made to the likelihood value.

```
# check sum of squared mean distances to detect convergence
# where tol = 1e-12 is the convergence threshold
# T0[1] is the  $\mu$  of the last iteration
# T[1] is the  $\mu$  of this iteration
converged = np.sum(np.square(T0[1]-T[1])) < tol
```

**iii. Plot the trajectories of the two mean vectors ( $\mu_1$  and  $\mu_2$ ) in 2 dimensions to show how they change over the course of running EM.**

I completed the task using the following code. The figure could be found after the code.

```
# Obtain mu from the mean list
x1 = []
y1 = []
x2 = []
y2 = []
for i in range(len(mean_list)):
    x1.append(mean_list[i][0][0])
    y1.append(mean_list[i][0][1])
    x2.append(mean_list[i][1][0])
    y2.append(mean_list[i][1][1])

# Plot the trajectories of the two mean vectors ( $\mu_1$  and  $\mu_2$ ) in 2 dimensions
# to show how they change over the course of running EM.
plt.figure(figsize=(12, 6))
plt.plot(x1, y1, marker='o', mec='r', mfc='w', label='mu1')
plt.plot(x2, y2, marker='*', ms=10, label='mu2')
plt.legend()
plt.xlabel('Eruptions')
plt.ylabel('Duration')
plt.title("Trajectories")
plt.savefig("em_trajectories") # save pics
plt.show()
```

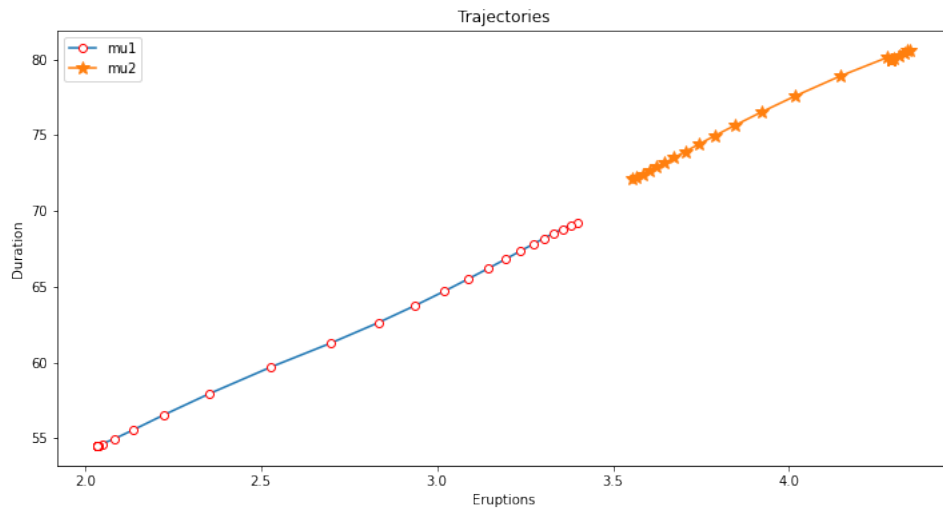


Figure 1.13: EM-Trajectories of the two mean vectors

### (e) K-means Clustering

- **THOUGHT:** I think I will get different clusters from K-means clustering compared to EM algorithm I just implemented. And my experiment with K-means clustering on the same dataset with  $K = 2$  reflects this thought (see below for details). K-means returns the centers of  $[4.29793023, 80.28488372]$ ,  $[2.09433, 54.75]$ , but GMM with EM implementation returns the centers of  $[2.0363884598395923, 54.47851642946882]$ ,  $[4.2896619777147755, 79.96811522972116]$ .
- **REASON:** Both EM and K-means allow model refining of an iterative process to find the best congestion. But K-means clustering differs in the method used for calculating the Euclidean distance while calculating the distance between each of two data items. And EM uses statistical methods. EM method is more like a soft version of K-means clustering with fixed priors and covariance. For the E-step of EM method, we do soft assignments based on the the softmax of the squared Mahalanobis distance from each point to each cluster. Each center moved by weighted means of the data, with weights given by soft assignments. But for the E-step of K-means, weights are 0 or 1 due to hard assignments.

```
# Import the sklearn package
from sklearn.cluster import KMeans
# Experiment with K-means clustering on the same dataset with K = 2
kmeans = KMeans(n_clusters=2).fit(V, 2)
# Print out the centers
# Output is [[ 4.29793023 80.28488372], [ 2.09433    54.75    ]]
print("K-means clustering - the centers are " + str(kmeans.cluster_centers_))

# Get the plot
label = kmeans.labels_
cluster1, cluster2 = [], []
# Separate data points into two clusters
for l, d in zip(label, data):
    if label[l] == 0:
        cluster1.append(d)
    else:
        cluster2.append(d)
```

```
cluster1, cluster2 = np.array(cluster1), np.array(cluster2)
# Display the scatter plot
plt.scatter(cluster1[:, 0], cluster1[:, 1], s=10, c='g', linewidth=0.1)
plt.scatter(cluster2[:, 0], cluster2[:, 1], s=10, c='r', linewidth=0.1)
plt.xlabel('eruptions')
plt.ylabel('duration')
plt.savefig('k_means_clustering') # save pics
```

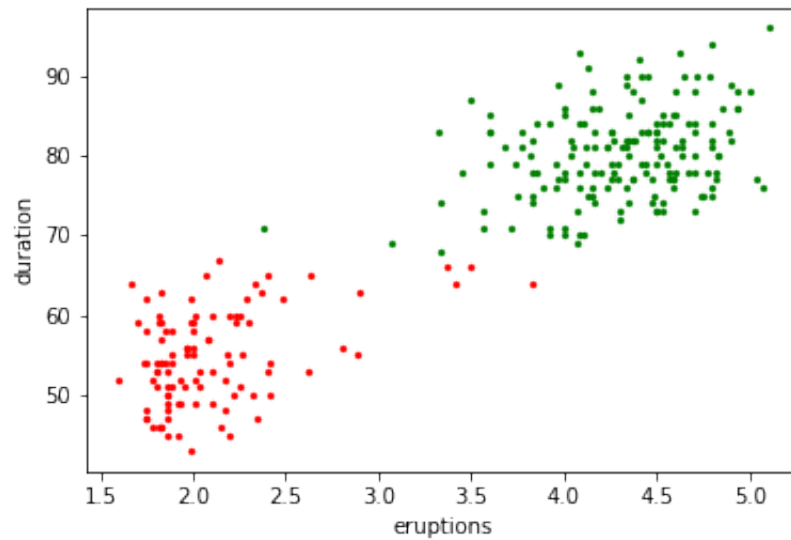


Figure 1.14: K-Means Clustering

## 2. WRITTEN EXERCISES

### Question 1 SVD and eigendecomposition

Given an  $m \times n$  matrix  $X$ , we have:

$$X^T X = (U D V^T)^T U D V^T \quad (2.1)$$

$$= V D^T U^T U D V^T \quad (2.2)$$

Since for an orthonormal matrix  $U$ ,  $U^T U = I$ , we have:

$$X^T X = V D^T D V^T \quad (2.3)$$

For  $D^T D$ ,  $D$  is a  $m \times n$  diagonal matrix with non-negative real numbers on the diagonal. We have:

$$D^T D = \begin{pmatrix} D_{11}^2 & & & \\ & D_{22}^2 & & \\ & & D_{33}^2 & \\ & & & \dots \end{pmatrix} \quad (2.4)$$

So, we have an eigen-decomposition of  $X^T X$  given by  $X^T X = V \Lambda V^T$ .  $V$  is a  $n \times n$  orthonormal matrix and  $V$  is identical to the matrix in the SVD of  $X$ .  $\Lambda$  is the diagonal matrix containing the eigenvalues  $\lambda_k := \Lambda_{kk} = D_{kk}^2$  and it's equal to the squared singular values in SVD of  $X$  (ie.  $D^2$ ).

(2.5)



## Reference

- [1] Volodymyr Kuleshov. *Gaussian Discriminant Analysis*. URL: <https://github.com/kuleshov/cornell-cs5785-2021-applied-ml/blob/main/notebooks/lecture7-gaussian-discriminant-analysis.ipynb>. (accessed: 10.16.2021).
- [2] Matt. *Expectation Maximisation to fit Gaussian Mixture Model*. URL: <https://github.com/matomatical/gmm-em-algorithm/blob/master/gmm.py>. (accessed: 10.16.2021).