ee249
Eylül Ertay

Homework 3

CS 5785

Programming Exercises

0.1 Eigenface for Face Recognition

0.1.1 (a) Download the Data

We download the train and the test dataset as shown.

```python
train_df = pd.read_csv("faces/train.txt",header=None, low_memory=False)
test_df = pd.read_csv("faces/test.txt", header=None, low_memory=False)
train_df
```

	0
0	./faces/images/person01_63.png 1
1	./faces/images/person01_14.png 1
2	./faces/images/person01_09.png 1
3	./faces/images/person01_05.png 1
4	./faces/images/person01_01.png 1
...	...
535	./faces/images/person10_33.png 10
536	./faces/images/person10_46.png 10
537	./faces/images/person10_19.png 10
538	./faces/images/person10_64.png 10
539	./faces/images/person10_31.png 10

540 rows × 1 columns

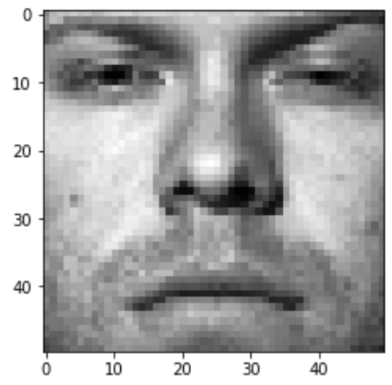
0.1.2 (b) Load the Data

Loaded the training data that consists of 540 images into a concatenated 2500-vector, and the test data into a 100 x 2500 vector. Displaying an image from both the X and X_test vectors. The code provided in the assignment description is used.

```
def plot_face(data):
  plt.imshow(data.reshape(50,50), cmap = cm.Greys_r)
  plt.show()
```

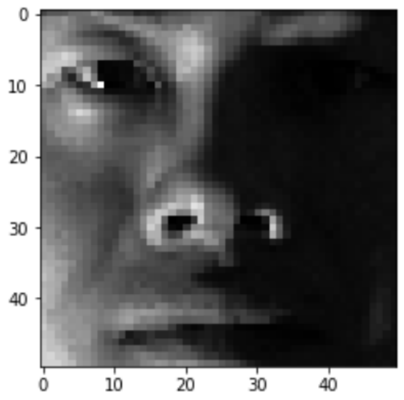
```
# Loading the training set into the matrix X

import numpy as np
from scipy import misc
from matplotlib import pylab as plt
import matplotlib.cm as cm
import imageio
%matplotlib inline
train_labels, train_data = [], []
for line in open('./faces/train.txt'):
    im = imageio.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)
print(train_data.shape, train_labels.shape)

plot_face(train_data[10, :])
```

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



```python
# Loading the test set into a matrix

test_labels, test_data = [], []
for line in open('./faces/test.txt'):
    im = imageio.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)
print(test_data.shape, test_labels.shape)

plot_face(test_data[10, :])
```

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



```python
train_data
```

```
array([[  5.,   7.,   7., ...,  13.,  20.,  31.],
       [ 65.,  74.,  74., ..., 111., 106.,  96.],
       [101., 116., 130., ...,  61.,  52.,  41.],
       ...,
       [107., 115., 108., ...,  26.,  23.,  24.],
       [255., 255., 255., ...,  22.,  22.,  22.],
       [174., 178., 202., ...,  28.,  26.,  28.]])
```

```python
test_data
```

```
array([[107., 113., 127., ...,  37.,  38.,  22.],
       [111., 138., 153., ...,  30.,  25.,  22.],
       [111., 106.,  58., ...,  12.,  15.,  11.],
       ...,
       [ 10.,  11.,  11., ..., 177., 187., 169.],
       [136., 111., 106., ...,  17.,  18.,  17.],
       [ 51.,  36.,  32., ..., 161., 138., 132.]])
```

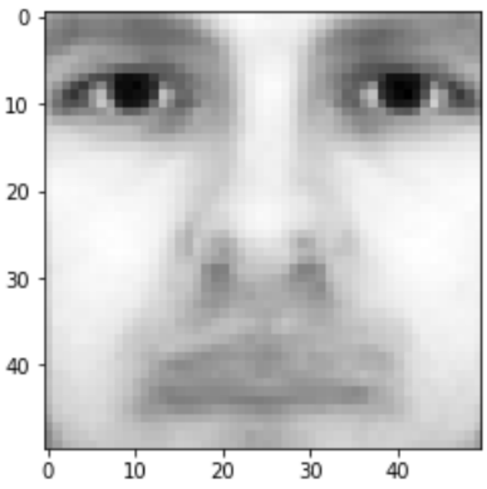
0.1.3 (c) Average Face

Computed the average face μ from the training set. The image displayed is the result of the average face computed.

```
avg = np.add.reduce(train_data)
avg_face = avg/(len(train_data))
avg_face
```

```
array([59.25185185, 56.10185185, 52.42222222, ..., 67.22222222,
       64.61851852, 59.27592593])
```

```
plot_face(avg_face)
```



```
avg_face.shape
```

```
(2500,)
```

0.1.4 (d) Mean Subtraction

Subtracting the average face from every data point in the training and test datasets, using the formula $\mathbf{x_i} := \mathbf{x_i} - \mu$.

```
def mean_subtract(data, average_face):
  return data - average_face
```

```
train_subtracted = mean_subtract(train_data, avg_face)
test_subtracted =  mean_subtract(test_data, avg_face)
```

```
train_subtracted
```

```
array([[-54.25185185, -49.10185185, -45.42222222, ..., -54.22222222,
        -44.61851852, -28.27592593],
       [  5.74814815,  17.89814815,  21.57777778, ...,  43.77777778,
         41.38148148,  36.72407407],
       [ 41.74814815,  59.89814815,  77.57777778, ...,  -6.22222222,
        -12.61851852, -18.27592593],
       ...,
       [ 47.74814815,  58.89814815,  55.57777778, ..., -41.22222222,
        -41.61851852, -35.27592593],
       [195.74814815, 198.89814815, 202.57777778, ..., -45.22222222,
        -42.61851852, -37.27592593],
       [114.74814815, 121.89814815, 149.57777778, ..., -39.22222222,
        -38.61851852, -31.27592593]])
```

```
test_subtracted
```

```
array([[ 47.74814815,  56.89814815,  74.57777778, ..., -30.22222222,
        -26.61851852, -37.27592593],
       [ 51.74814815,  81.89814815, 100.57777778, ..., -37.22222222,
        -39.61851852, -37.27592593],
       [ 51.74814815,  49.89814815,   5.57777778, ..., -55.22222222,
        -49.61851852, -48.27592593],
       ...,
       [-49.25185185, -45.10185185, -41.42222222, ..., 109.77777778,
        122.38148148, 109.72407407],
       [ 76.74814815,  54.89814815,  53.57777778, ..., -50.22222222,
        -46.61851852, -42.27592593],
       [ -8.25185185, -20.10185185, -20.42222222, ...,  93.77777778,
         73.38148148,  72.72407407]])
```

0.1.5 (e) Eigenface

```
def get_eigenvectors(X):
  covariance = np.dot(X.transpose(), X)
  U, Sigma, eigenvectors = np.linalg.svd(covariance, full_matrices=False)

  return eigenvectors
```

```
covariance = np.dot(train_subtracted.transpose(), train_subtracted)
covariance.shape
```

```
(2500, 2500)
```

```
U, Sigma, eigenvectors = np.linalg.svd(covariance, full_matrices=False)
```

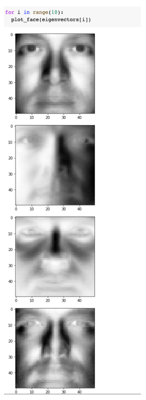
```
eigenvectors
```

```
array([[ 8.56324779e-05,  1.02866364e-03,  3.08553648e-03, ...,
         1.04812368e-02,  8.32289224e-03,  6.59057671e-03],
       [ 2.59457196e-02,  2.32848928e-02,  2.10968387e-02, ...,
        -2.43844653e-02, -2.44582450e-02, -2.32785651e-02],
       [-7.77077554e-03, -5.35923444e-03, -5.00431250e-03, ...,
         2.74064702e-02,  2.60116740e-02,  2.38793530e-02],
       ...,
       [ 0.00000000e+00, -8.99640999e-05, -3.07894529e-03, ...,
         8.64490913e-03, -7.36372725e-03,  8.51182241e-03],
       [ 0.00000000e+00, -1.97308162e-02,  3.89659516e-01, ...,
        -1.07393319e-02, -9.14118985e-04, -1.90321263e-03],
       [ 0.00000000e+00, -4.00401524e-03, -1.38652254e-01, ...,
        -1.73192290e-03, -4.03270559e-03, -2.62380042e-02]])
```

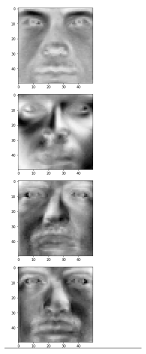
After performing eigendecomposition, $\mathbf{X}^T\mathbf{X} = \mathbf{V}\mathbf{\Lambda}\mathbf{V}^T$, we get the eigenvectors V^t, and we display the first 10 eigenfaces as shown.

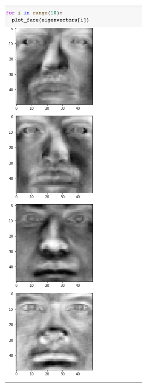
0.1.6 (f) Eigenface Feature

Generating the r-dimensional feature matrix $\mathbf{F}$ and $\mathbf{F}_{\mathbf{test}}$ using the eigenfaces computed previously.



```
for i in range(10):
  plot_face(eigenvectors[i])
```





```
for i in range(10):
  plot_face(eigenvectors[i])
```

```
def r_feature_matrix(r, X):
  Vt = get_eigenvectors(X)[:r]
  F = np.dot(X, Vt.transpose())

  return F
```

```
r_feature_matrix(3, train_subtracted).shape
```

```
(540, 3)
```

0.1.7 (g) Face Recognition

Implemented a face recognition model using scikit learn's Logistic Regression function. Extracting the training and test features for $r = 10$:

```
features_10_train = r_feature_matrix(10, train_subtracted)
features_10_test = r_feature_matrix(10, test_subtracted)
```

```
features_10_train
```

```
array([[-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]])
```

```
features_10_test
```

```
array([[ 1.85844067e+03, -8.38453574e+02, -1.57750105e+03,
        -3.19008848e+02, -1.15351155e+02, -2.06442258e+02,
        -4.87813524e+02, -5.86954275e+02, -2.64052582e+01,
         4.34697899e+02],
       [-2.00611781e+02, -1.96113891e+03,  7.85057600e+02,
         1.27523398e+02,  8.21980471e+01, -1.00032904e+01,
         2.76506683e+02,  3.58024463e+01, -2.51962820e+01,
         7.23721803e+01],
       [-2.63299407e+03,  1.20148980e+03,  3.14445763e+01,
        -4.37659065e+02,  9.09254555e+01, -6.41471441e+01,
         6.14813825e+01,  6.37357992e+01, -2.63515405e+02,
         1.47052592e+02],
       [-2.39420333e+03,  1.44002933e+03, -5.52381523e+02,
```

Now we train our logistic regression model, and test all values of r for [1,200].

```
from sklearn.linear_model import LogisticRegression

r_list = range(1,201)
accuracies = []

for r in r_list:
  F_train = r_feature_matrix(r, train_subtracted)
  F_test = r_feature_matrix(r, test_subtracted)
  log_reg = LogisticRegression(penalty='l2',
                               multi_class='ovr',
                               solver='liblinear',
                               max_iter=1000,
                               verbose=True).fit(F_train, train_labels)
  accuracy = log_reg.score(F_test, test_labels)
  accuracies.append(accuracy)
  print(accuracy)
```

```
[LibLinear]0.13
[LibLinear]0.18
[LibLinear]0.13
[LibLinear]0.13
[LibLinear]0.06
[LibLinear]0.06
[LibLinear]0.11
[LibLinear]0.08
[LibLinear]0.08
[LibLinear]0.05
[LibLinear]0.07
[LibLinear]0.09
[LibLinear]0.12
[LibLinear]0.14
[LibLinear]0.11
```

Looking at the results, the accuracies we got from the test set seems to be quite low. This is most likely due to overfitting, as our training accuracy is extremely high, whereas the test accuracies are very low.

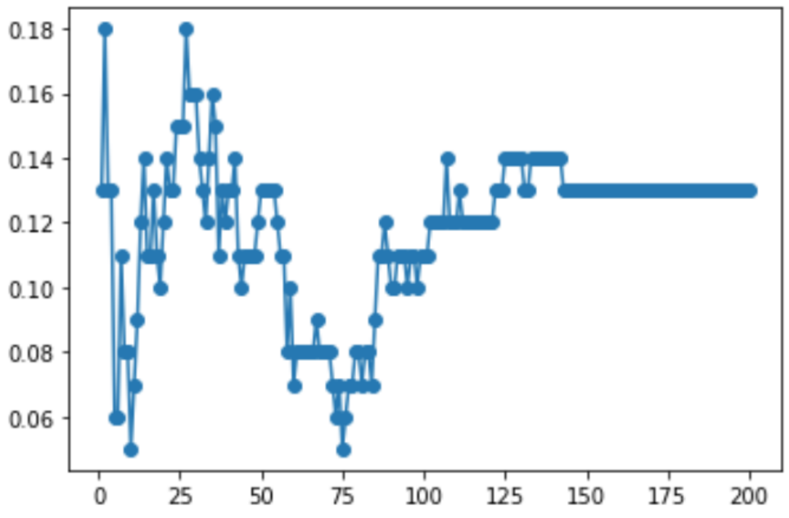
Plotting the accuracies, we get the following plot:

```
len(accuracies)
```

```
200
```

```
import matplotlib.pyplot as plt

rs = list(r_list)

plt.scatter(rs, accuracies)
plt.plot(rs, accuracies)
```

```
[<matplotlib.lines.Line2D at 0x7f03bccb6390>]
```



The plot shows that after $r = 150$, the accuracy converges to 0.13.

0.2 Implement EM Algorithm

0.2.1 Parse and Plot Data

```
geyser_df = pd.read_csv("data.csv", low_memory=False)
geyser_df
```

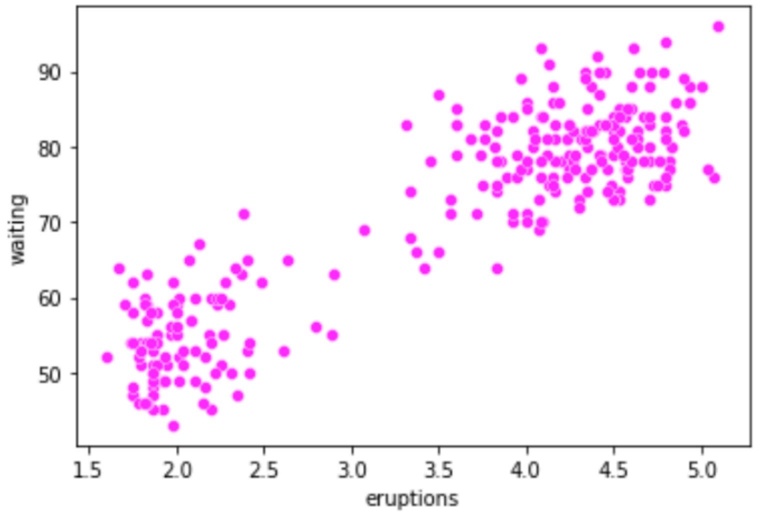
	eruptions	waiting
0	3.600	79
1	1.800	54
2	3.333	74
3	2.283	62
4	4.533	85
...	...	...
267	4.117	81
268	2.150	46
269	4.417	90
270	1.817	46
271	4.467	74

272 rows × 2 columns

Plotting the points we get the following figure:

```python
import seaborn as sns

sns.scatterplot(data=geyser_df, x='eruptions', y='waiting', color='magenta')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f03b8337c50>
```



0.2.2 E-Step

We write the expression for the posterior probability for each point $x^{(i)}$.

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

0.2.3 M-Step

The formulas for μ_k, Σ_k, and for the parameters ϕ are as follows:

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

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

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

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

0.2.4 Implement and Run the EM Algorithm

i. First, we initialise our parameters. Then, to initialise the clusters, instead of randomly assigning values, we use the K-means centroids to set our Gaussian Mixture Model.

```
k = 2
X = geyser_df.to_numpy()
attr = geyser_df.shape[1]

mu = np.mean(X, axis=0)
cov = np.dot((X - mu).T, X - mu) / (X.shape[0] - 1)
```

```
from sklearn.cluster import KMeans


def initialize_clusters(X, n_clusters):
    clusters = []
    idx = np.arange(X.shape[0])

    # Instead of randomising the centroids, we use the KMeans centroids to initialise the GMM

    kmeans = KMeans(n_clusters).fit(X)
    mu_k = kmeans.cluster_centers_

    for i in range(n_clusters):
        clusters.append({
            'p_k': 1.0 / n_clusters,
            'mu_k': mu_k[i],
            'cov_k': np.identity(X.shape[1], dtype=np.float64)
        })

    return clusters
```

```
initialize_clusters(X, k)
```

```
[{'cov_k': array([[1., 0.],
         [0., 1.]]), 'mu_k': array([ 4.29793023, 80.28488372]), 'p_k': 0.5},
 {'cov_k': array([[1., 0.],
         [0., 1.]]), 'mu_k': array([ 2.09433, 54.75    ]), 'p_k': 0.5}]
```

We define a Gaussian function to generate a normal pdf given our data X.

```
def gaussian(X, mu, cov):
    n = X.shape[1]
    diff = (X - mu).T
    return np.diagonal(1 / ((2 * np.pi) ** (n / 2) * np.linalg.det(cov) ** 0.5) * np.exp(-0.5 * np.dot(np.dot(diff.T, np.linalg.inv(cov)), diff))).reshape(-1, 1)
```

We implement the expectation and maximisation steps in two different functions.

```python
def expectation_step(X, clusters):
    totals = np.zeros((X.shape[0], 1), dtype=np.float64)

    for cluster in clusters:
        p_k = cluster['p_k']
        mu_k = cluster['mu_k']
        cov_k = cluster['cov_k']

        prob_nk = (p_k * gaussian(X, mu_k, cov_k)).astype(np.float64)

        for i in range(X.shape[0]):
            totals[i] += prob_nk[i]

        cluster['prob_nk'] = prob_nk
        cluster['totals'] = totals


    for cluster in clusters:
        cluster['prob_nk'] /= cluster['totals']
```

```
def maximization_step(X, clusters, iterations):
    N = float(X.shape[0])

    for cluster in clusters:
        prob_nk = cluster['prob_nk']
        cov_k = np.zeros((X.shape[1], X.shape[1]))

        N_k = np.sum(prob_nk, axis=0)

        p_k = N_k / N
        mu_k = np.sum(prob_nk * X, axis=0) / N_k

        for j in range(X.shape[0]):
            diff = (X[j] - mu_k).reshape(-1, 1)
            cov_k += prob_nk[j] * np.dot(diff, diff.T)

        cov_k /= N_k

        cluster['p_k'] = p_k
        cluster['mu_k'] = mu_k
        cluster['cov_k'] = cov_k

        if (iterations % 2 != 0):
          mu1_list.append(mu_k.tolist())
          iterations += 1

        else:
          mu2_list.append(mu_k.tolist())
          iterations += 1
```

Next, we write a function to get the likelihoods and train the Gaussian Mixture Model.

```
def get_likelihood(X, clusters):
    sample_likelihoods = np.log(np.array([cluster['totals'] for cluster in clusters]))
    return np.sum(sample_likelihoods), sample_likelihoods
```

```
def train_gmm(X, n_clusters, n_epochs, iterations):
    clusters = initialize_clusters(X, n_clusters)
    likelihoods = np.zeros((n_epochs, ))
    scores = np.zeros((X.shape[0], n_clusters))
    history = []

    for i in range(n_epochs):
        clusters_snapshot = []

        # This is just for our later use in the graphs
        for cluster in clusters:
            clusters_snapshot.append({
                'mu_k': cluster['mu_k'].copy(),
                'cov_k': cluster['cov_k'].copy()
            })

        history.append(clusters_snapshot)

        expectation_step(X, clusters)
        maximization_step(X, clusters, iterations)

        likelihood, sample_likelihoods = get_likelihood(X, clusters)
        likelihoods[i] = likelihood

        print('Epoch: ', i + 1, 'Likelihood: ', likelihood)

    for i, cluster in enumerate(clusters):
        scores[:, i] = np.log(cluster['prob_nk']).reshape(-1)

    print(clusters_snapshot)
    return clusters, likelihoods, scores, sample_likelihoods, history
```

Now we can test our model with the Old Faithful Geyser data.

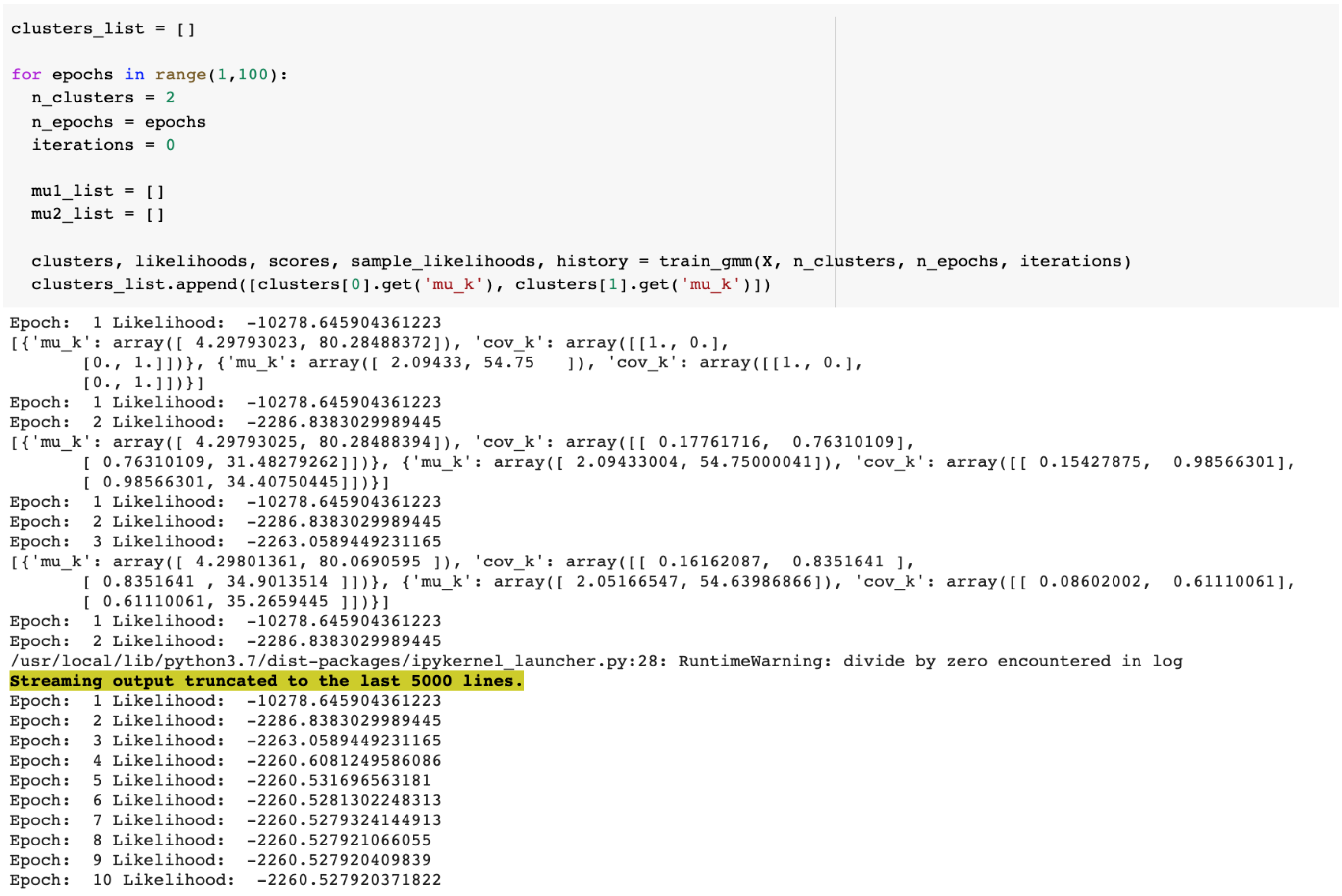
```
n_clusters = 2
n_epochs = 15
iterations = 0

mu1_list = []
mu2_list = []

clusters, likelihoods, scores, sample_likelihoods, history = train_gmm(X, n_clusters, n_epochs, iterations)
```

```
Epoch:  1 Likelihood:  -10278.645904361223
Epoch:  2 Likelihood:  -2286.8383029989445
Epoch:  3 Likelihood:  -2263.0589449231165
Epoch:  4 Likelihood:  -2260.6081249586086
Epoch:  5 Likelihood:  -2260.531696563181
Epoch:  6 Likelihood:  -2260.5281302248313
Epoch:  7 Likelihood:  -2260.5279324144913
Epoch:  8 Likelihood:  -2260.527921066055
Epoch:  9 Likelihood:  -2260.527920409839
Epoch:  10 Likelihood:  -2260.527920371822
Epoch:  11 Likelihood:  -2260.5279203696186
Epoch:  12 Likelihood:  -2260.5279203694913
Epoch:  13 Likelihood:  -2260.5279203694836
Epoch:  14 Likelihood:  -2260.5279203694836
Epoch:  15 Likelihood:  -2260.527920369483
[{'mu_k': array([ 4.28966197, 79.96811518]), 'cov_k': array([[ 0.16996844,  0.94060931],
       [ 0.94060931, 36.04621126]])}, {'mu_k': array([ 2.03638845, 54.47851638]), 'cov_k': array([[ 0.06916767,  0.43516763],
       [ 0.43516763, 33.69728209]])}]
```

```
print(clusters[0].get('mu_k'))
print(clusters[1].get('mu_k'))
```

```
[ 4.28966197 79.96811517]
[ 2.03638845 54.47851638]
```

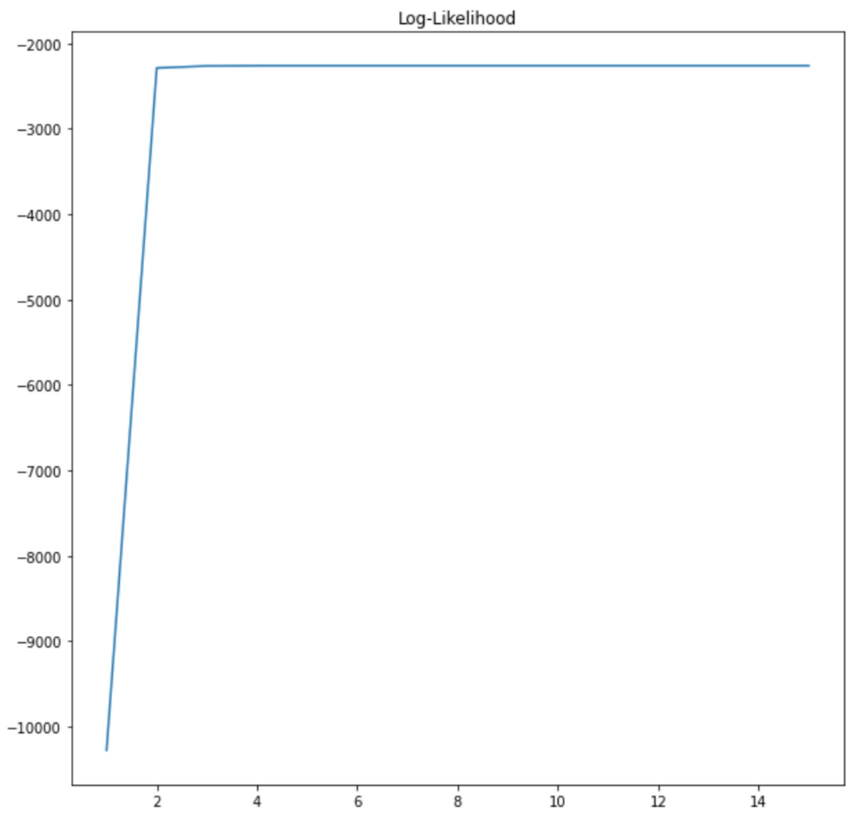
ii. A good criteria for deciding the on when to terminate the EM algorithm is looking at when the cluster centroids stop changing. Here in our implementation, we should decide on the number of epochs in order to converge the results. We can observe that when we print out the likelihood, after the 5th iteration, the likelihood changes a lot less significantly.



```
clusters_list = []

for epochs in range(1,100):
  n_clusters = 2
  n_epochs = epochs
  iterations = 0

  mu1_list = []
  mu2_list = []

  clusters, likelihoods, scores, sample_likelihoods, history = train_gmm(X, n_clusters, n_epochs, iterations)
  clusters_list.append([clusters[0].get('mu_k'), clusters[1].get('mu_k')])
```

```
Epoch:  1 Likelihood:  -10278.645904361223
[{'mu_k': array([ 4.29793023, 80.28488372]), 'cov_k': array([[1., 0.],
       [0., 1.]])}, {'mu_k': array([ 2.09433, 54.75   ]), 'cov_k': array([[1., 0.],
       [0., 1.]])}]
Epoch:  1 Likelihood:  -10278.645904361223
Epoch:  2 Likelihood:  -2286.8383029989445
[{'mu_k': array([ 4.29793025, 80.28488394]), 'cov_k': array([[ 0.17761716,  0.76310109],
       [ 0.76310109, 31.48279262]])}, {'mu_k': array([ 2.09433004, 54.75000041]), 'cov_k': array([[ 0.15427875,  0.98566301],
       [ 0.98566301, 34.40750445]])}]
Epoch:  1 Likelihood:  -10278.645904361223
Epoch:  2 Likelihood:  -2286.8383029989445
Epoch:  3 Likelihood:  -2263.0589449231165
[{'mu_k': array([ 4.29801361, 80.0690595 ]), 'cov_k': array([[ 0.16162087,  0.8351641 ],
       [ 0.8351641 , 34.9013514 ]])}, {'mu_k': array([ 2.05166547, 54.63986866]), 'cov_k': array([[ 0.08602002,  0.61110061],
       [ 0.61110061, 35.2659445 ]])}]
Epoch:  1 Likelihood:  -10278.645904361223
Epoch:  2 Likelihood:  -2286.8383029989445
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:28: RuntimeWarning: divide by zero encountered in log
Streaming output truncated to the last 5000 lines.
Epoch:  1 Likelihood:  -10278.645904361223
Epoch:  2 Likelihood:  -2286.8383029989445
Epoch:  3 Likelihood:  -2263.0589449231165
Epoch:  4 Likelihood:  -2260.6081249586086
Epoch:  5 Likelihood:  -2260.531696563181
Epoch:  6 Likelihood:  -2260.5281302248313
Epoch:  7 Likelihood:  -2260.5279324144913
Epoch:  8 Likelihood:  -2260.527921066055
Epoch:  9 Likelihood:  -2260.527920409839
Epoch:  10 Likelihood:  -2260.527920371822
```

```
[169] plt.figure(figsize=(10, 10))
      plt.title('Log-Likelihood')
      plt.plot(np.arange(1, n_epochs + 1), likelihoods)
      plt.show()
```



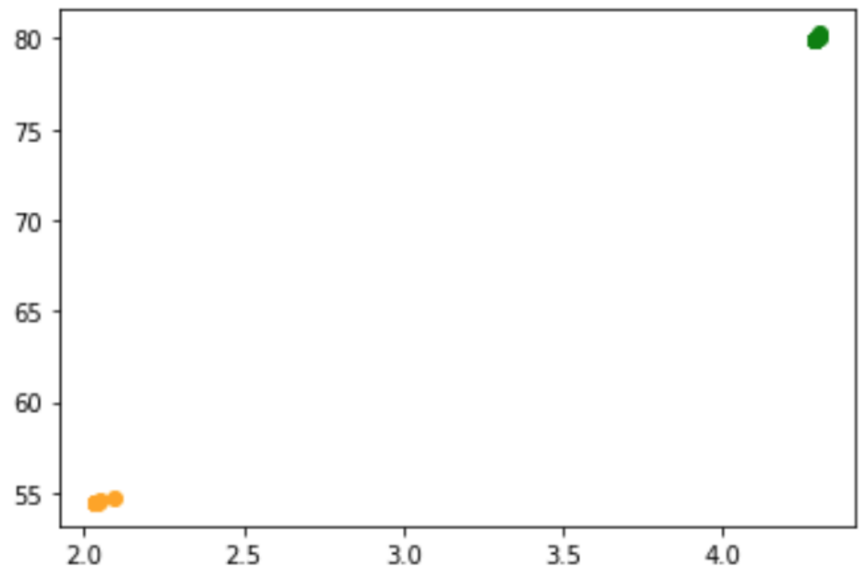
We have also observed this after running the EM algorithm multiple times up to 100 epochs, and saving the centroids μ.

```
mu1_list
```

```
[[4.2979302488235325, 80.28488394258054],
 [4.298013613252418, 80.06905949712986],
 [4.291758219582397, 79.99160735853599],
 [4.290155817819795, 79.97402058028105],
 [4.289779359310114, 79.96953203369316],
 [4.289690133455946, 79.96845562379498],
 [4.289668747050709, 79.96819709738108],
 [4.289663603677514, 79.96813489554665],
 [4.289662365663761, 79.96811992200894],
 [4.289662067611724, 79.96811631704021],
 [4.289661995852086, 79.9681154490953],
 [4.289661978574878, 79.96811524012426],
 [4.289661974415119, 79.96811518981117],
 [4.289661973413591, 79.96811517769753],
 [4.289661973172458, 79.96811517478096]]
```

iii. Plotting the trajectories of the centroids we get the following figure.

```
for i in range(len(mu1_list)):
  plt.scatter(mu1_list[i][0], mu1_list[i][1], color="green")
  plt.scatter(mu2_list[i][0], mu2_list[i][1], color="orange")
```



0.2.5 K-means

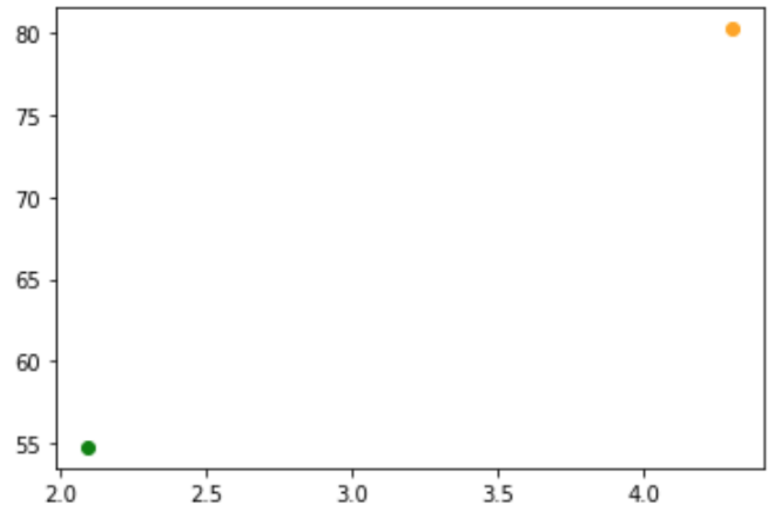
The first intuition is that the clusters obtained from K-means and EM would not be that different. Both of these algorithms converge to an optimum value after running a set amount of times. This seems aligned with the results we get after using the K-means algorithm on the dataset, even though the coordinates of the centroids slightly differ:

```
kmeans = KMeans(n_clusters=2).fit(X)
centers = kmeans.cluster_centers_

centers
```

```
array([[ 2.09433   , 54.75      ],
       [ 4.29793023, 80.28488372]])
```

```
plt.scatter(centers[0][0], centers[0][1], color="green")
plt.scatter(centers[1][0], centers[1][1], color="orange")
```

```
<matplotlib.collections.PathCollection at 0x7f03b3703c50>
```



Another reason for this intuition is the initialisation we used for the EM algorithm. In order to create more accurate results, rather than randomly setting up clusters, we initialised the clusters using K-means, then perfected the results after running the EM algorithm. Thus it is not surprising that the results of both algorithms are very similar.

Written Exercises

1. The SVD of X is given by $X = UDV^T$. If U and V are orthogonal, then we have,
$X^T X = (UDV^T)^T UDV^T$
$= VD^T U^T UDV^T$
$= VD^T DV^T$
$= VAV^T$
where we can define $A = D^T D$.
Thus, we can construct the eigendecomposition of $X^T X$ from the SVD of X.