Project 7

Wenjun Li

April 8, 2020

Question 1

Implement a random number generator for a random vector $X = [X_1, X_2, X_3]^T$ having multivariate Gaussian distribution with

$$\mu = \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix}, \Sigma = \begin{bmatrix} 3 & -1 & 1 \\ -1 & 5 & 3 \\ 1 & 3 & 4 \end{bmatrix}.$$

Results and Analysis

To generate a multivariate Gaussian random variables, we can first generate a standard normal random vector Z and then have $X = AZ^T + \mu$, where A can be obtained by decomposing Σ. In my simulation, I decompose Σ by cholesky method. The code is as below. Here is an example of my simulation results.

$$X = \begin{bmatrix} 1.95863891 & 2.98940629 & 3.37835383 \\ 2.95863891 & 3.98940629 & 4.37835383 \\ 3.95863891 & 4.98940629 & 5.37835383 \end{bmatrix}.$$

Python Code

Listing 1: Code of Question 1

```
import numpy as np


mu = np.array([[1], [2], [3]])
sigma = np.array([[3, −1, 1], [−1, 5, 3], [1, 3, 4]])

Z = np.random.randn(3)
A = np.linalg.cholesky(sigma)
X = mu + np.dot(A, Z.T)

print(X)
```

Question 2

Implement a random number generator for a random variable with the following mixture distribution: $f(x) = 0.4N(-1, 1) + 0.6N(1, 1)$. Generate a histogram and overlay the theoretical p.d.f. of the random variable.

Results and Analysis

In this question, we need to generate a mixture distribution, which is a random variable that is derived from a collection of other variables as follows: 1. a random variable is selected by chance from the collection according to given probabilities of selection; 2. the value of the selected random variable is realized.

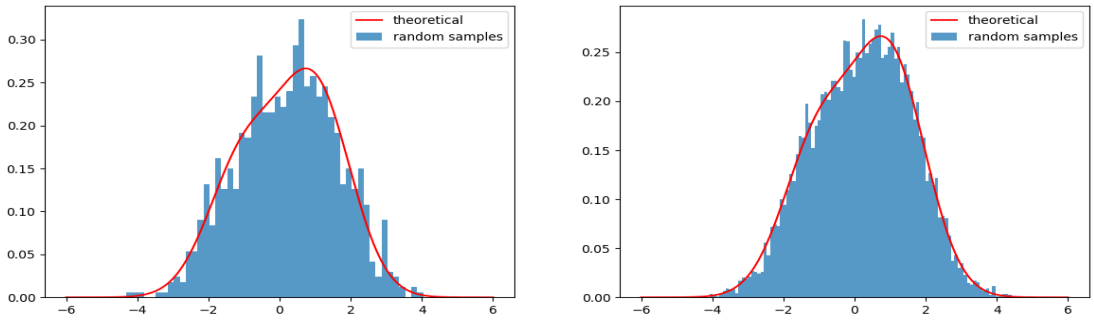
The individual distributions that are combined to form the mixture distribution are called the **mixture components**, and the probabilities (or weights) associated with each component are called the **mixture weights**.

The number of components in mixture distribution is often restricted to being finite, although in some cases the components may be countably infinite. We can write the formulation as: $f(x) = p_1 g_1(x) + p_2 g_2(x) + p_i g_i(x) + ... + p_N g_N(x)$, where p_i is the density of distribution $g_i(x)$. Note that the sum of all density coefficients equals to one, i.e. $\sum_{i=1}^{N} p_i = 1$.

Take mixture Gaussian distribution $f(x) = p * N(m1, v1) + (1-p) * N(m2, v2)$ as an example. We can generate $f(x)$ by following steps:

1. generate a random number u in U(0, 1);
2. if u $<$ p, append a random number x from N(m1, v1) to f(x);
3. if u $\geq p, append a random number x from N(m1, v1) to f(x)$;
4. repeat step1 - step3.

In my simulation, I generate N=1,000 and N=10,000 samples and compared their p.d.f to theoretical p.d.f. From Figure 2., we can conclude the sample distribution is closer to theoretical p.d.f with an increasing sample number. Also, the distribution of this mixture Gaussian random variable is not simply the sum of two Gaussian random variable.



(a) N=1,000 (b) N=10,000

Figure 1: Sample Histogram with Theoretical p.d.f Overlaid

Python Code

Listing 2: Code of Question 2

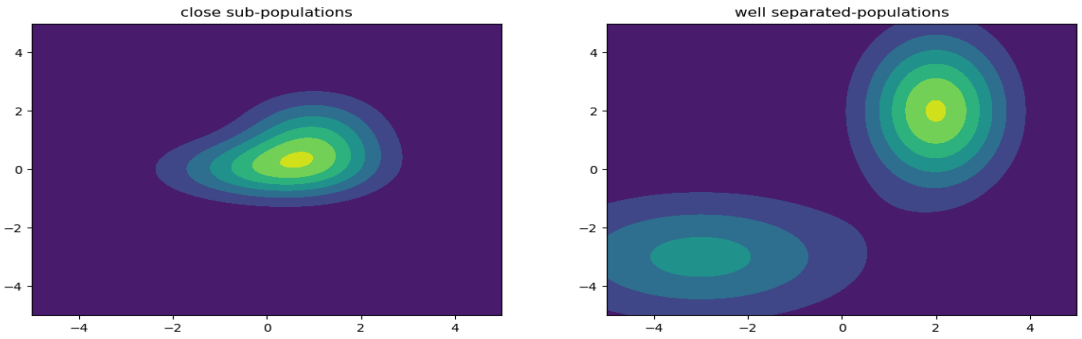
```
import numpy as np
from scipy.stats import norm
import matplotlib.pyplot as plt

# x range
xrange = np.linspace(-6, 6, 1000)

# theoretical p.d.f.
theo = 0.4*np.array(norm(-1, 1).pdf(xrange)) + 0.6*np.array(norm(1, 1).pdf(xrange)
    )

# sampling
p = 0.4
data = []
N = 1000
for i in range(N):
    if np.random.rand() < 0.4:
        data.append(np.random.normal(-1, 1))
    else:
        data.append(np.random.normal(1, 1))


# plot
plt.figure()
plt.hist(data, bins=50, density=1, label='random samples', alpha=0.75)
plt.plot(xrange, theo.tolist(), label='theoretical', color='r')
plt.legend()
plt.savefig('Q2_{}.png'.format(N))
plt.show()
```

Question 3

Implement a 2-dimensional random number generator for a Gaussian mixture model (GMM) pdf with 2 subpopulations. Use the expectation maximization (EM) algorithm to estimate the pdf parameters of the 2-D GMM from samples. Compare the quality and speed of your GMM-EM estimates using 300 samples from different GMM distributions (e.g. spherical vs ellipsoidal covariance, close vs well-separated subpopulations, etc.).

Results and Analysis

To do the comparison, I first generate two Gaussian mixture models: one is well-separated sub-populations and the other one is close sub-populations. Their contour plots are shown in Figure 3.



(a) close sub-populations (b) well-separated sub-populations

Figure 2: Two Gaussian Mixture Models

For each GMM distribution, I generate 150 samples and then put them together into a 300 long sequence. To evaluate the prediction performance, I create a label list ([1, 1, ..., 0, 0, ...]). To compare the quality and speed of my GMM-EM esitmates with this 300 samples, I compute the weight, mean, covariance, predict accuracy as well as run time. Here, I use two covariance types: **spherical** and **diag**. The comparison results are as below:

covariance type: spherical

run time: 0.0060 weight: 0.3690

mean: $\begin{bmatrix} -0.5749 & -0.0034 \\ 1.1916 & 0.6687 \end{bmatrix}$ *covariance* : $\begin{bmatrix} 0.7744 & 0.9079 \end{bmatrix}$

accuracy: 0.72

covariance type: diag

run time: 0.0024 weight: 0.5039

mean: $\begin{bmatrix} -2.6704 & -3.1117 \\ 1.9341 & 2.1566 \end{bmatrix}$ *covariance* : $\begin{bmatrix} 4.8192 & 1.9647 \\ 0.7414 & 2.8653 \end{bmatrix}$

accuracy: 0.99

From the above comparison, we can conclude that the quality is better when using **diag** as covariance type. Not only the prediction accuracy is much higher and run time is shorter, but also the mean and covariance matrices are closer to samples.

Python Code

Listing 3: Code of Question 3

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.mixture import GaussianMixture
from scipy.stats import multivariate_normal
import time


# define hyper-parameters
N = 300
cov_type = ['spherical', 'diag', 'full', 'tied']
label_1 = [1]*150 + [0]*150
label_2 = [0]*150 + [1]*150

# generate grid for plots
x, y = np.mgrid[-5:5:.01, -5:5:.01]
pos = np.empty(x.shape + (2,))
pos[:, :, 0] = x; pos[:, :, 1] = y


# close sub-populations
mu1 = [0, 0]; cov1 = [[2, 0], [0, 0.5]]
mu2 = [1, 1]; cov2 = [[1, 0], [0, 1]]
rv1 = multivariate_normal(mu1, cov1)
rv2 = multivariate_normal(mu2, cov2)

plt.figure()
plt.contourf(x, y, rv2.pdf(pos)+rv1.pdf(pos))
plt.title('close sub-populations')
plt.savefig('Q3_close_sub_populations.png')
plt.show()


# well-separated populations
mu3 = [-3, -3]; cov3 = [[5, 0], [0, 2]]
mu4 = [2, 2]; cov4 = [[1, 0], [0, 3]]
rv3 = multivariate_normal(mu3, cov3)
rv4 = multivariate_normal(mu4, cov4)

plt.figure()
plt.contourf(x, y, rv4.pdf(pos)+rv3.pdf(pos))
plt.title('well separated-populations')
plt.savefig('Q3_well_separated_populations.png')
plt.show()
```

```
44
45
46  # GMM
47  X1 = np.zeros((300, 2))
48  X1[:150, :] = np.random.multivariate_normal(mu1, cov1, size=150)
49  X1[150:, :] = np.random.multivariate_normal(mu2, cov2, size=150)
50
51  X2 = np.zeros((300, 2))
52  X2[:150, :] = np.random.multivariate_normal(mu3, cov3, size=150)
53  X2[150:, :] = np.random.multivariate_normal(mu4, cov4, size=150)
54
55
56  # spherical
57  start_time = time.time()
58  gmm = GaussianMixture(n_components=2, covariance_type='spherical')
59  gmm.fit(X1)
60  labels = gmm.predict(X1)
61  end_time = time.time()
62
63  plt.figure()
64  print('covariance type: ', cov_type[0])
65  print('run time:', end_time - start_time)
66  print('weight: ', gmm.weights_[0])
67  print('mean: \n', gmm.means_)
68  print('covariance: \n', gmm.covariances_)
69  print('accuracy:', max(np.mean(label_1 == labels), np.mean(label_2 == labels)))
70
71
72  # diag
73  start_time = time.time()
74  gmm = GaussianMixture(n_components=2, covariance_type='diag')
75  gmm.fit(X2)
76  labels = gmm.predict(X2)
77  end_time = time.time()
78
79  plt.figure()
80  print('\n ---------------------------------')
81  print('covariance type: ', cov_type[1])
82  print('run time:', end_time - start_time)
83  print('weight: ', gmm.weights_[0])
84  print('mean: \n', gmm.means_)
85  print('covariance: \n', gmm.covariances_)
86  print('accuracy:', max(np.mean(label_1 == labels), np.mean(label_2 == labels)))
```

Question 4

A geyser is a hot spring characterized by an intermittent discharge of water and steam. Old Faithful is a famous cone geyser in Yellowstone National Park, Wyoming. It has a predictable geothermal discharge and since 2000 it has erupted every 44 to 125 minutes. Refer to the addendum data file that contains waiting times and the durations for 272 eruptions.

a Generate a 2-D scatter plot of the data. Run a k-means clustering routine on the data for $k = 2$. Show the two clusters on a scatterplot.

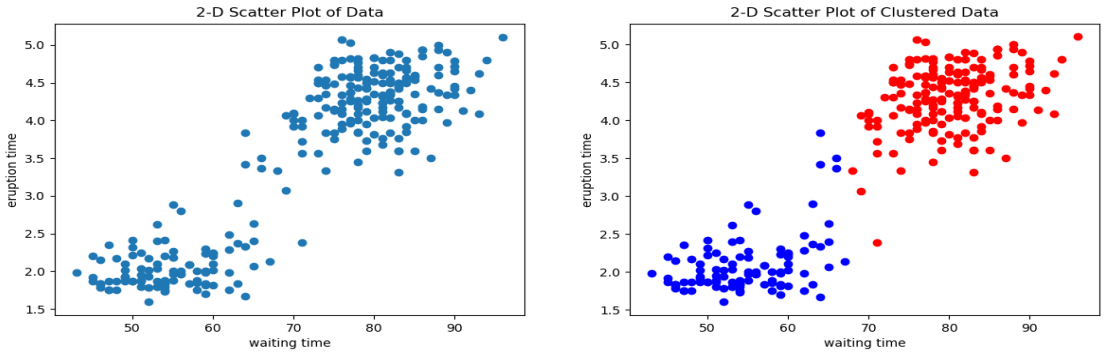
b Use a GMM-EM algorithm to fit the dataset to a GMM pdf. Draw a contour plot of your final GMM pdf. Overlay the contour plot with a scatterplot of the data set. How can you use the GMM pdf estimates to cluster the data?

Results and Analysis

Part (a)

In part (a), we need to first load data from 'faithful.txt' and process the data properly. In my code, I store eruption time and waiting time in two arrays, so that I can plot a scatter plot of the data points. The scatter plot of original data points is shown in Figure 3.

After loading and plotting original data points, we need to apply k-means clustering to cluster them. In python, we can simply import KMeans method from sklearn.cluster. After clustering the data, I plot points with two different color according to their labels, i.e. the points with same color belong to the same group. The scatter plot of clustered points is also displayed in Figure 3.



(a) scatter plot of original data (b) scatter plot of clustered data

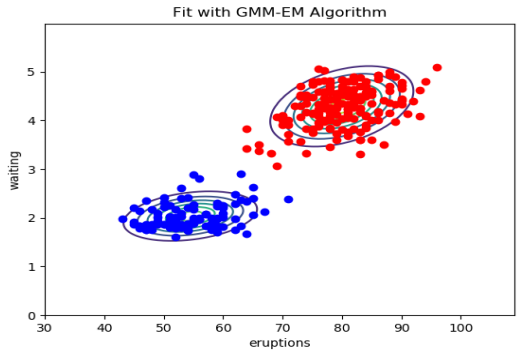
Figure 3: Scatter Plot of Data Points

As we can see from the clustering result, we can conclude that the distribution of waiting

time and eruption time can be well separated into two clusters and k-means can separate these points very well.

Part (b)

In part (b), we need to use GMM-EM algorithm to fit the dataset to a GMM p.d.f. Similarly, we can use the GaussianMixture method from sklearn.mixture to cluster data points. After fitting the data points, I plot contours of the two group and plot points with different colors. The result is shown in Figure 4.



(a)

Figure 4: Use GMM-EM to Fit Data

Also we know, GMM-EM is a soft classifier, which means the classification is based on the probability of points belonging to different groups. So, when we compare the clustering results of k-means and GMM-EM, we can see a small difference. The points in the middle of two groups are labeled differently by k-means and GMM-EM.

Python Code

Listing 4: Code of Question 4

```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.mixture import GaussianMixture
from scipy.stats import multivariate_normal


# part (a)
# load data
data = np.loadtxt('faithful.dat.txt', skiprows=26)
```

```
eruption = data[:, 1]
waiting = data[:, 2]
original_data = np.array([eruption, waiting]).T


# scatter plot
plt.figure(1)
plt.scatter(waiting, eruption)
plt.xlabel('waiting time')
plt.ylabel('eruption time')
plt.title('2-D Scatter Plot of Data')
plt.savefig('Q4_a_original_data.png')
plt.show()


# run k-means clustering
clusters = KMeans(n_clusters=2)
clusters.fit(original_data)


# scatter plot of clustered data
plt.figure(3)
colors = ['b', 'r']
for i, j in enumerate(clusters.labels_):
    plt.plot(waiting[i], eruption[i], color=colors[j], marker='o', ls='None')
plt.xlabel('waiting time')
plt.ylabel('eruption time')
plt.title('2-D Scatter Plot of Clustered Data')
plt.savefig('Q4_a_clustered_data.png')
plt.show()


# part (b)
# use GMM to fit original data
gmm = GaussianMixture(n_components=2, covariance_type='full')
gmm.fit(original_data)
label = gmm.predict(original_data)

mean = gmm.means_
cov = gmm.covariances_
weight = gmm.weights_


# plot contour
```

```
plt.figure()
x, y = np.mgrid[0:6:0.01, 30:110:1]
pos = np.empty(x.shape+(2,))
pos[:, :, 0] = x; pos[:, :, 1] = y

plt.contour(y, x, weight[0]*multivariate_normal.pdf(pos,mean[0,:],cov[0,:,:])
            + weight[1]*multivariate_normal.pdf(pos,mean[1,:],cov[1,:,:]))


# plot data points
for i in range(272):
    if label[i] == 0:
        plt.plot(waiting[i], eruption[i], color='r', marker='o')
    else:
        plt.plot(waiting[i], eruption[i], color='b', marker='o')

plt.title('Fit with GMM-EM Algorithm')
plt.xlabel('eruptions')
plt.ylabel('waiting')
plt.savefig('Q4_GMM.png')
plt.show()
```