INF552: Programming Assignment 2

Yi-Li Chen

Part 1: Implementation

- All imported library

```
1. import csv
2. import numpy as np
3. from math import *
4. import pandas as pd
5. import matplotlib.pyplot as plt
6. from scipy.stats import multivariate_normal
```

- Read text file into a list

```
1. data = []
2. with open('clusters.txt') as file:
3.     reader = csv.reader(file)
4.     for line in reader:
5.         data.append([float(line[0]),float(line[1])])
```

K-means

- Initialize k centroids which is picking the points randomly among the dataset.

```
1. def initial_centroids(data, k):
2.     num_samples, dim = np.array(data).shape
3.     centroids = np.zeros((k, dim))
4.     idx = np.random.randint(len(data), size=3)
5.     for i in range(k):
6.         centroids[i] = data[idx[i]]
7.     return centroids
```

- Calculate the Euclidean Distance between two data points

```
1. def min_distance(p1, p2):
2.     return sqrt((p1[0]-p2[0])**2+(p1[1]-p2[1])**2)
```

- Find out each data point belongs to which cluster by the distance between centroids

```
1. def update2centroids(datapoint, centroids):
2.     k_num_distance = []
3.     for c in centroids:
4.         d = min_distance(c, datapoint)
5.         k_num_distance.append(d)
6.     return np.argmin(k_num_distance)
```

- Classify the data points into k cluster by the index we have found in previous function. Then, the centroids will be updated by the mean values of data in each cluster. In addition, store the difference between the old and new centroids, so that we can when it should be stopped.

```
def clasify2cluster(data, k, centroids, cluster):
    cluster=[[] for i in range(k)]
    for p in data:
        c_idx = update2centroids(p, centroids)
        cluster[c_idx].append(p)
    # find new centroids from the clusters
    num_samples, dim = np.array(data).shape
    new_centroids = np.zeros((k, dim))
    for i in range(k):
        new_x = np.array(cluster[i])[:,0].mean()
        new_y = np.array(cluster[i])[:,1].mean()
        new_centroids[i] = [new_x,new_y]
        difference = new_centroids - centroids
    return (difference, new_centroids, cluster)
```

- Firstly, initialize the centroids. Secondly, classify the data repeatedly with terminated condition which is when there is no difference between the old and updated centroids. In ddition, create a dict() to store the centroids for each cluster of every iteration.

```
def kmeans(data, k):
    mark = ['or', 'ob', 'og', 'ok', '^r', '+r', 'sr', 'dr', '<r', 'pr']
    centroids_dict = {'cluster1':[],'cluster2':[],'cluster3':[]}
    cluster=[[] for i in range(k)]
    centroids = initial_centroids(data, k)
    centroids_dict['cluster1'].append(tuple(centroids[0]))
    centroids_dict['cluster2'].append(tuple(centroids[1]))
    centroids_dict['cluster3'].append(tuple(centroids[2]))
    difference, new_centroids, cluster = clasify2cluster(data, k, centroids, cluster)
    while np.any(difference != 0):
        cluster=[[] for i in range(k)]
        difference, new_centroids, cluster = clasify2cluster(data, k, new_centroids, cluster)
        for i in range(3):
            plt.plot(np.array(cluster[i])[:,0],np.array(cluster[i])[:,1],mark[i])
        plt.show()
        centroids_dict['cluster1'].append(tuple(new_centroids[0]))
```

```
        centroids_dict['cluster2'].append(tuple(new_centroids[1]))
        centroids_dict['cluster3'].append(tuple(new_centroids[2]))
    return (difference, new_centroids, cluster, centroids_dict)
```

- The final result of k-means clustering:

```
difference, new_centroids, cluster, centroids_dict = kmeans(data, 3)
```

	Cluster1	Cluster2	Cluster3
Centroid	5.738495346032258, 5.164838081193549	-0.960652907023256, -0.652218412860465	3.2888485605151514, 1.9326883657575757

GMM

- Initialize the ric of each data points randomly, but the sum of ric for each data point needs to be 1

```
def initial_ric(data, k):
    initial_ric = np.zeros((len(data), k))
    random_int = np.random.randint(low=1,high=10000,size=(len(data), k))
    for row in range(len(random_int)):
        initial_ric[row] = random_int[row]/sum(random_int[row])
    return initial_ric
```

- Update the mean for each Gaussian distribution from the ric and data points

```
def caluculate_new_mu(k, ric, data):
    new_mu = np.zeros((k, 2))
    # calculate the mu of each cluster
    for c_index in range(k):
        # loop all datapoints and their corresponding ric
        for r, p in zip(ric[:,c_index], np.array(data)):
            new_mu[c_index] += r*p
        new_mu[c_index] = new_mu[c_index] / sum(ric[:,c_index])
    return new_mu
```

- Update the covariance for each Gaussian distribution from the ric, data points and the mean we just got.

```
def caluculate_new_sigma(k, ric, data, mu):
    new_sigma = np.array([[[0, 0], [0, 0]] for i in range(0,k)], dtype='float64')
    for c_index in range(k):
        # loop all datapoints and their corresponding ric
```

```
        for r, p in zip(ric[:,c_index], np.array(data)):
            new_sigma[c_index] += r*np.outer(p-mu[c_index],p-mu[c_index])
        new_sigma[c_index] = new_sigma[c_index] / sum(ric[:,c_index])
    return new_sigma
```

- Update the pi for each Gaussian distribution from the ric, data points.

```
def caluculate_new_pi(k, ric, data):
    new_pi = np.zeros(k)
    for idx in range(k):
        new_pi[idx] = sum(ric[:,idx]) / len(data)
    return new_pi
```

- Calculate the new ric from these updated parameters for each data points based on Bayes rule where π is the prior weights and the likelihood is normal

```
def update_ric(k, data, mu, sigma, pi):
    pdfs = np.zeros(((len(data), k)))
    for i in range(k):
        pdfs[:, i] = pi[i] * multivariate_normal.pdf(data, mu[i], sigma[i])
    ric = pdfs / pdfs.sum(axis=1).reshape(-1, 1)
    return ric
```

- In sum, firstly, initialize the ric randomly. Secondly, update the μ, covariance and pi step by step for each Gaussian distribution. Then, update the ric from these updated parameters. Afterwards, go back to the previous function so that parameters can be updated again. Finally, the two steps of E-step and M-step are then repeated until convergence. We have set a condition for while loop which is it will terminate if the difference between the updated ric and old one less than 0.005

```
def GMM(data, k):
    param_dict = {'mu':[],'covariance':[],'pi':[]}
    diff = float('Inf')
    ric = initial_ric(data, k)
    while np.any(abs(diff) > 0.005):
        mu = caluculate_new_mu(k, ric, data)
        sigma = caluculate_new_sigma(k, ric, data, mu)
        pi = caluculate_new_pi(k, ric, data)
        new_ric = update_ric(k, data, mu, sigma, pi)
        # store all updated parameters for each iteration of GMM
        param_dict['mu'].append(mu)
```

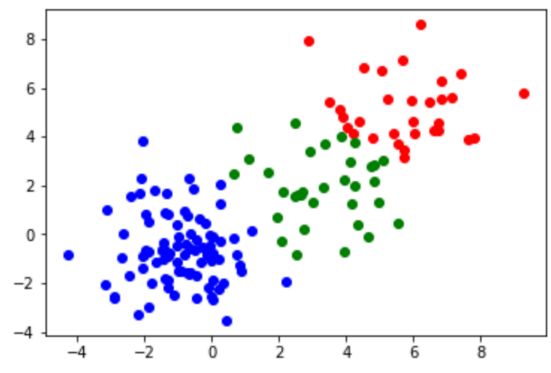
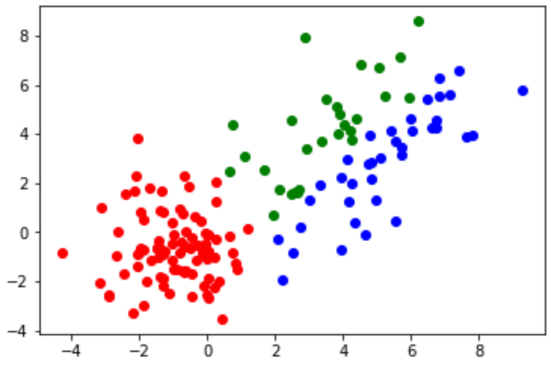
```
        param_dict['covariance'].append(sigma)
        param_dict['pi'].append(pi)
        diff = new_ric - ric
        ric = new_ric
        # classify and store the data points into list
        cluster=[[] for i in range(k)]
        for idx in range(len(ric)):
            c_idx = np.argmax(ric[idx])
            cluster[c_idx].append(data[idx])
    # plot the cluster result
    for i in range(3):
        mark = ['or', 'ob', 'og', 'ok', '^r', '+r', 'sr', 'dr', '<r', 'pr']
        plt.plot(np.array(cluster[i])[:,0],np.array(cluster[i])[:,1],mark[i])
    plt.show()
    return (mu, sigma, pi, ric, param_dact)
```

- The final result of GMM clustering:

```
mu, sigma, pi, ric, param_dict = GMM(data, 3)
```

	Cluster1	Cluster2	Cluster3
Parameter	μ: [-0.994437 0.63388089] Σ:[[1.18193522 -0.0816692] [-0.0816692 2.02109235]] π: 0.55921906	μ: [4.92603572 2.94483694] Σ:[[3.50297936 2.97191914] [2.97191914 4.33618297]] π: 0.20585555	μ: [3.83434248 3.69019296] Σ:[[3.44038384 2.08056859] [2.08056859 5.46032776]] π: 0.23492538

The comparison of the results of k-means and GMM:

k-means	GMM
	

From the above figures of both results, we can easily find out that the clusters of k-means are tend to be spherical. However, the cluster of GMM are not limited to this.

In this assignment, the most challenging I encountered is how to update all parameters simultaneously. I really learn a lot of methods of data manipulation for NumPy array. Especially, when setting the stopping criteria, I transferred two for loop into one row command.