Programming Assignment 2: K-means and GMM Algorithm

Member: Luyao Wang, Di Jin, Yingqi Lin

Individual contributions:

(1) Luyao Wang:
  Implementation of K-means
  Implementation of GMM Algorithm
  The third part of report

(2) Di Jin:
  Implementation of K-means
  Implementation of GMM Algorithm
  The first part and second part of report

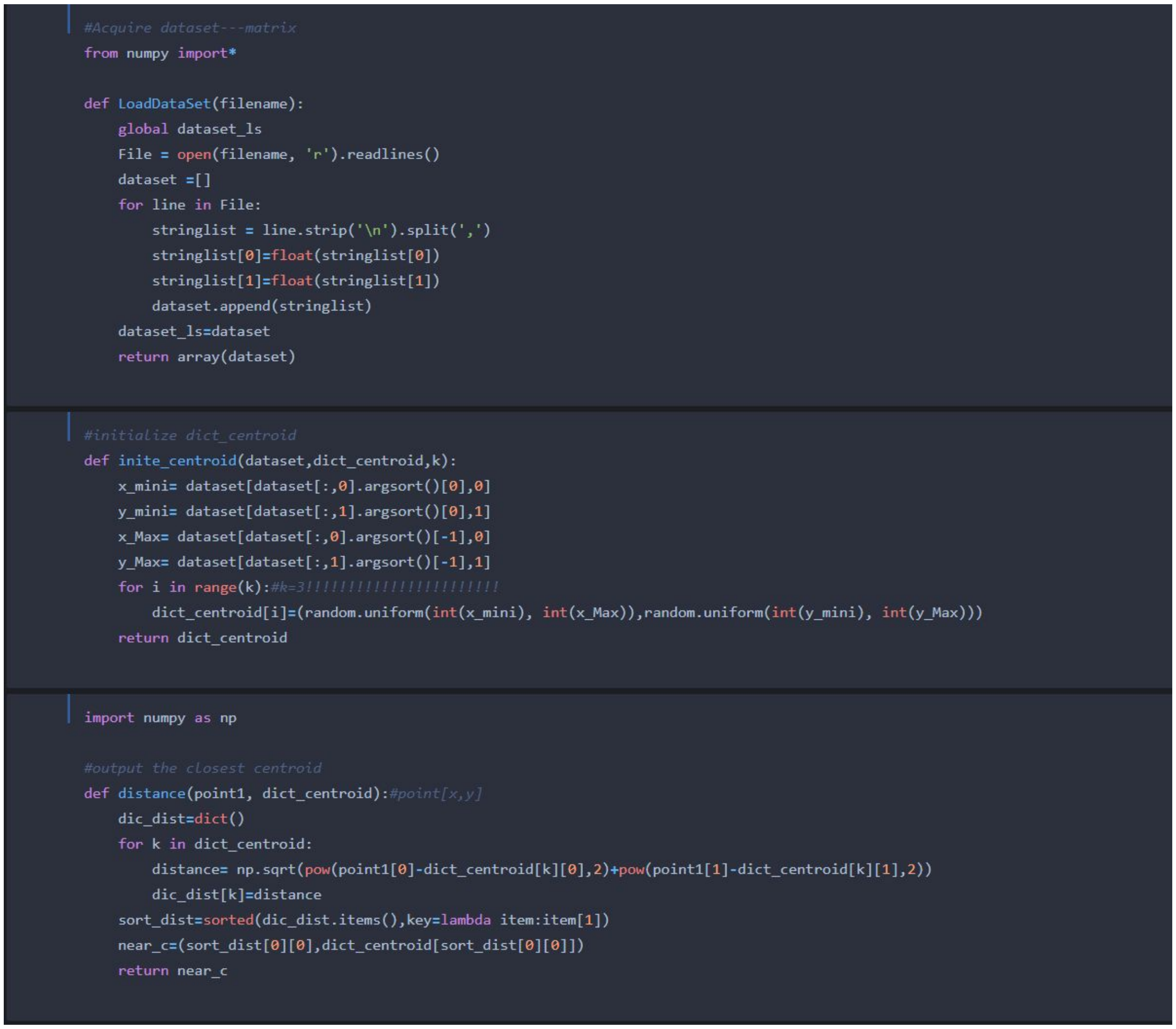
(3) Yingqi Lin:
  Discussion of K-means
  Implementation of GMM Algorithm
  The first part and second part of report

Part 1: Implementation

1. Implementation of K-means and GMM Algorithm

(1)K-means Algorithm:

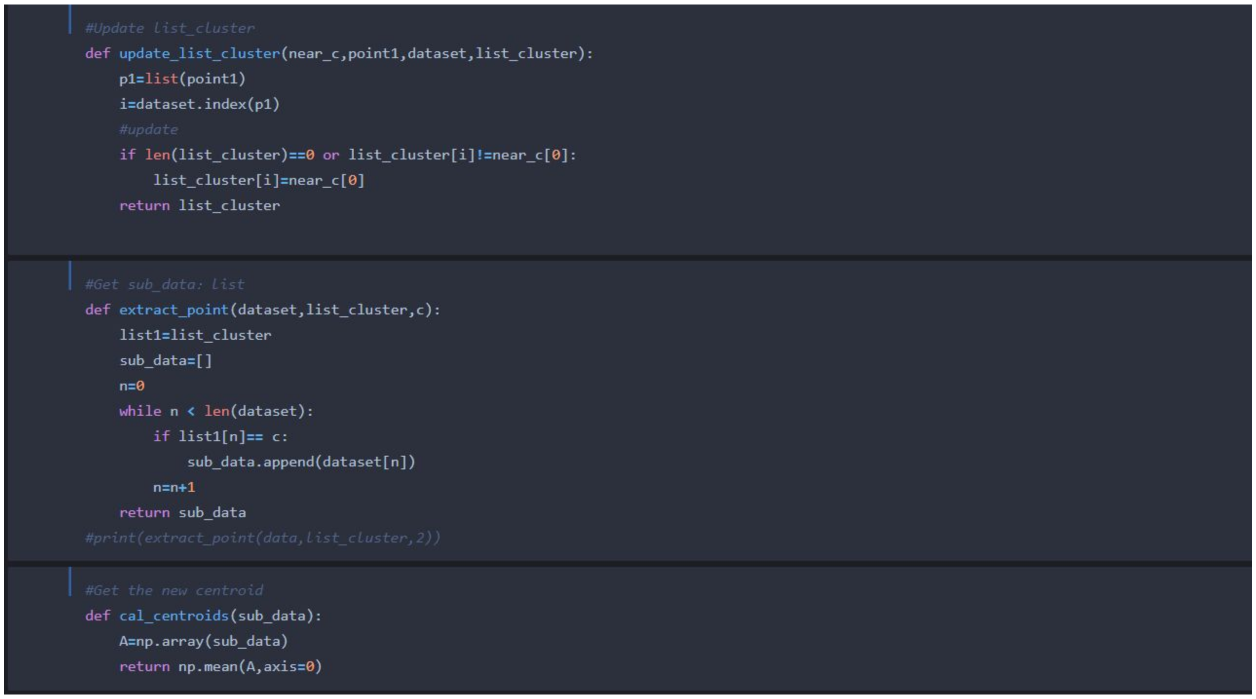
a. Implementation of K-means Algorithm



```
#Acquire dataset---matrix
from numpy import*

def LoadDataSet(filename):
    global dataset_ls
    File = open(filename, 'r').readlines()
    dataset =[]
    for line in File:
        stringlist = line.strip('\n').split(',')
        stringlist[0]=float(stringlist[0])
        stringlist[1]=float(stringlist[1])
        dataset.append(stringlist)
    dataset_ls=dataset
    return array(dataset)
```

```
#initialize dict_centroid
def inite_centroid(dataset,dict_centroid,k):
    x_mini= dataset[dataset[:,0].argsort()[0],0]
    y_mini= dataset[dataset[:,1].argsort()[0],1]
    x_Max= dataset[dataset[:,0].argsort()[-1],0]
    y_Max= dataset[dataset[:,1].argsort()[-1],1]
    for i in range(k):#k=3!!!!!!!!!!!!!!!!!!!!!!!!
        dict_centroid[i]=(random.uniform(int(x_mini), int(x_Max)),random.uniform(int(y_mini), int(y_Max)))
    return dict_centroid
```

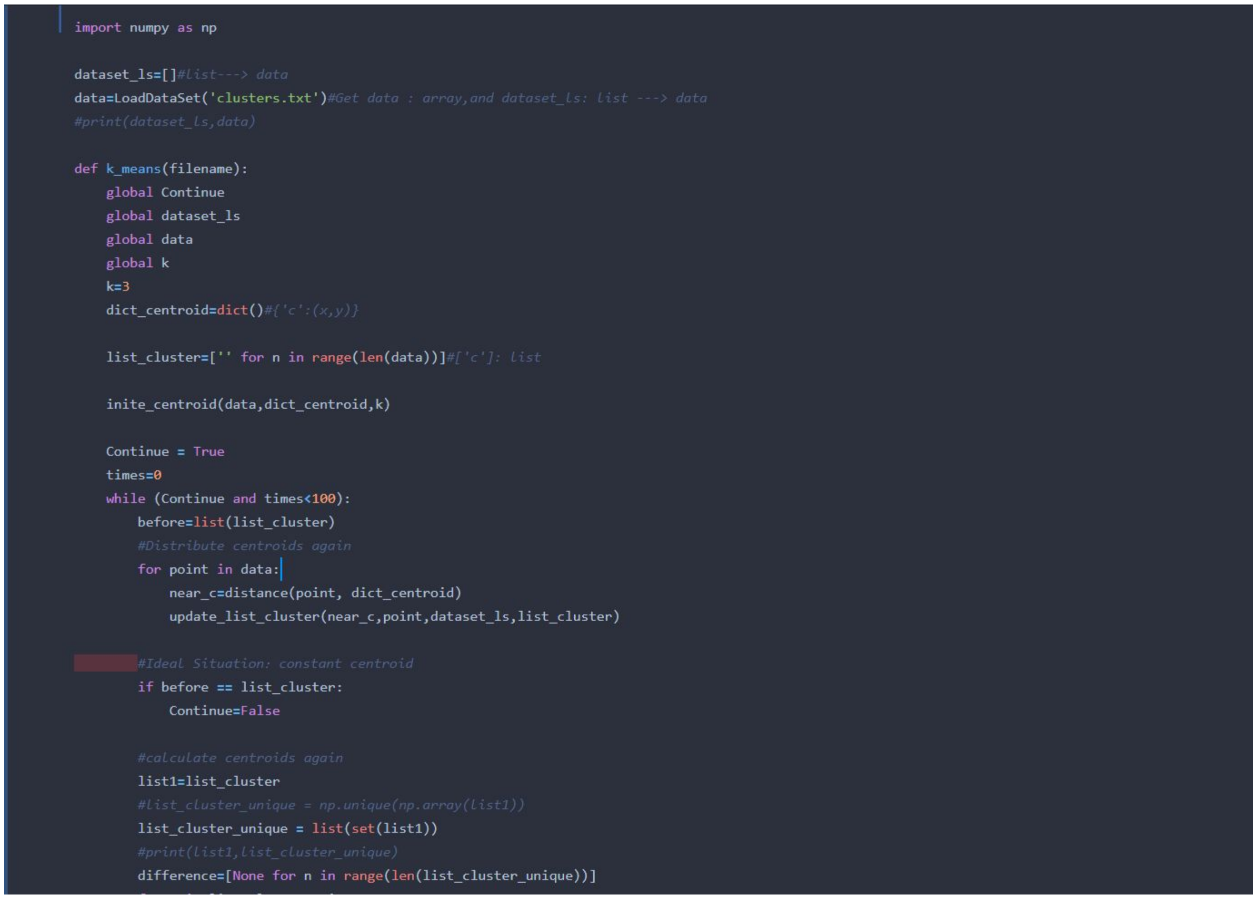
```
import numpy as np

#output the closest centroid
def distance(point1, dict_centroid):#point[x,y]
    dic_dist=dict()
    for k in dict_centroid:
        distance= np.sqrt(pow(point1[0]-dict_centroid[k][0],2)+pow(point1[1]-dict_centroid[k][1],2))
        dic_dist[k]=distance
    sort_dist=sorted(dic_dist.items(),key=lambda item:item[1])
    near_c=(sort_dist[0][0],dict_centroid[sort_dist[0][0]])
    return near_c
```



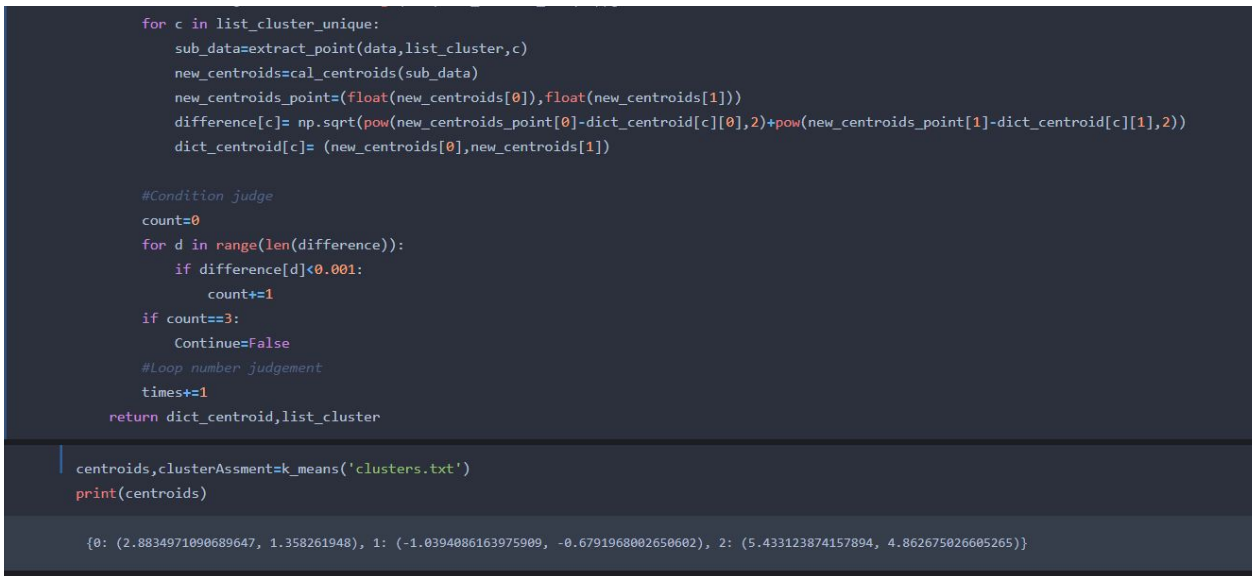
```python
#Update list_cluster
def update_list_cluster(near_c,point1,dataset,list_cluster):
    p1=list(point1)
    i=dataset.index(p1)
    #update
    if len(list_cluster)==0 or list_cluster[i]!=near_c[0]:
        list_cluster[i]=near_c[0]
    return list_cluster
```

```python
#Get sub_data: list
def extract_point(dataset,list_cluster,c):
    list1=list_cluster
    sub_data=[]
    n=0
    while n < len(dataset):
        if list1[n]== c:
            sub_data.append(dataset[n])
        n=n+1
    return sub_data
#print(extract_point(data,list_cluster,2))
```

```python
#Get the new centroid
def cal_centroids(sub_data):
    A=np.array(sub_data)
    return np.mean(A,axis=0)
```



```python
import numpy as np

dataset_ls=[]#list---> data
data=LoadDataSet('clusters.txt')#Get data : array,and dataset_ls: list ---> data
#print(dataset_ls,data)

def k_means(filename):
    global Continue
    global dataset_ls
    global data
    global k
    k=3
    dict_centroid=dict()#{'c':(x,y)}

    list_cluster=['' for n in range(len(data))]#['c']: list

    inite_centroid(data,dict_centroid,k)

    Continue = True
    times=0
    while (Continue and times<100):
        before=list(list_cluster)
        #Distribute centroids again
        for point in data:
            near_c=distance(point, dict_centroid)
            update_list_cluster(near_c,point,dataset_ls,list_cluster)

        #Ideal Situation: constant centroid
        if before == list_cluster:
            Continue=False

        #calculate centroids again
        list1=list_cluster
        #list_cluster_unique = np.unique(np.array(list1))
        list_cluster_unique = list(set(list1))
        #print(list1,list_cluster_unique)
        difference=[None for n in range(len(list_cluster_unique))]
```



```
            for c in list_cluster_unique:
                sub_data=extract_point(data,list_cluster,c)
                new_centroids=cal_centroids(sub_data)
                new_centroids_point=(float(new_centroids[0]),float(new_centroids[1]))
                difference[c]= np.sqrt(pow(new_centroids_point[0]-dict_centroid[c][0],2)+pow(new_centroids_point[1]-dict_centroid[c][1],2))
                dict_centroid[c]= (new_centroids[0],new_centroids[1])

            #Condition judge
            count=0
            for d in range(len(difference)):
                if difference[d]<0.001:
                    count+=1
            if count==3:
                Continue=False
            #Loop number judgement
            times+=1
        return dict_centroid,list_cluster
```

```
centroids,clusterAssment=k_means('clusters.txt')
print(centroids)
```

```
{0: (2.8834971090689647, 1.358261948), 1: (-1.0394086163975909, -0.6791968002650602), 2: (5.433123874157894, 4.862675026605265)}
```

b. Data Storage Structure

Our algorithm uses array type to store 150 raw data, which is convenient for us to use some array methods, such as numpy.mean, numpy.argsort, etc. At the same time, we also use the type of list to store these 150 data, which is convenient for us to use some list methods, such as list index () list append (), etc. In addition, we use the dictionary type to store the distances from all points to the centroid. The sorted () method of the dictionary can be used to find the point with the smallest distance from the centroid.
We use multiple types of storage structures to store data.

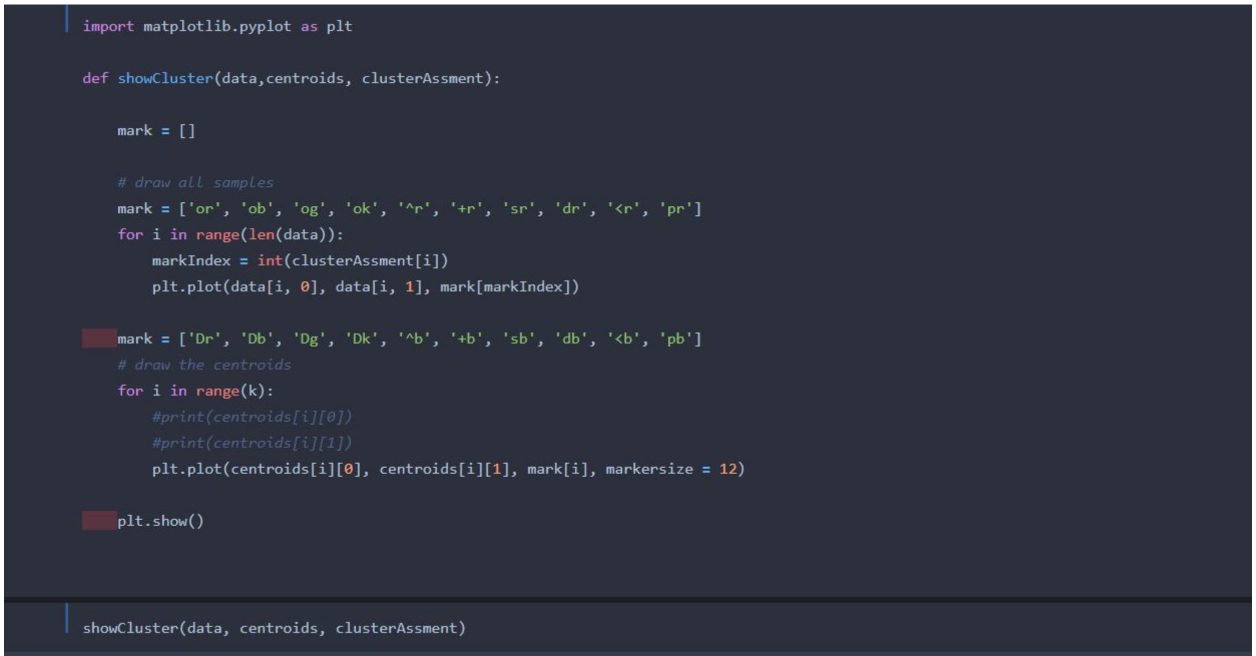
c. Code-level Optimizations

In the beginning, the convergence condition is that the three centroids don't change. However, we find that this situation is difficult to achieve and we modified the code. The convergence condition is that the changing distance of the centroid in each cluster is less than 0.001.

d. Challenges

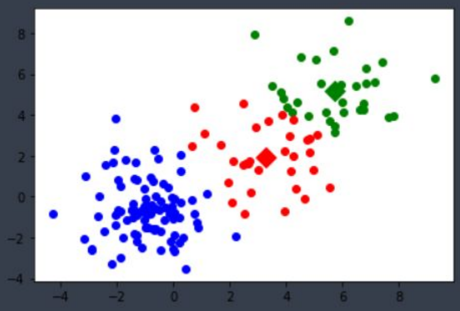
1) In the beginning, we only used the storage structure of the array to store data. Whenever we used the list method, we often encountered errors. This is due to our unfamiliarity with the list and array methods. And then, we stored the original data into two types.
2)We are not familiar with visualizing the result. Our group spent a lot of time learning how to use the matplotlib.pyplot to visualize the data.

e. Requested Output



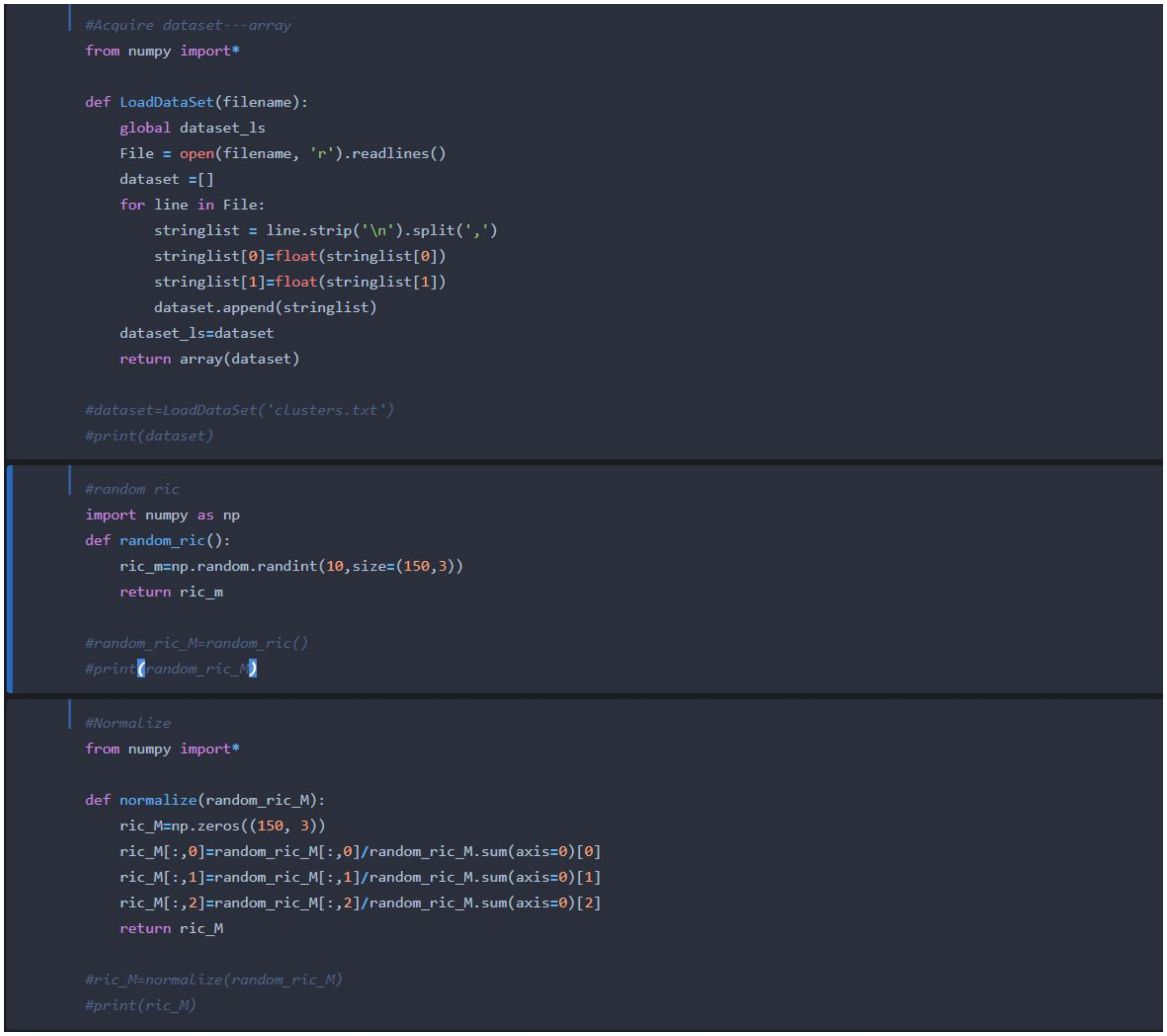
```
import matplotlib.pyplot as plt

def showCluster(data,centroids, clusterAssment):

    mark = []

    # draw all samples
    mark = ['or', 'ob', 'og', 'ok', '^r', '+r', 'sr', 'dr', '<r', 'pr']
    for i in range(len(data)):
        markIndex = int(clusterAssment[i])
        plt.plot(data[i, 0], data[i, 1], mark[markIndex])

    mark = ['Dr', 'Db', 'Dg', 'Dk', '^b', '+b', 'sb', 'db', '<b', 'pb']
    # draw the centroids
    for i in range(k):
        #print(centroids[i][0])
        #print(centroids[i][1])
        plt.plot(centroids[i][0], centroids[i][1], mark[i], markersize = 12)

    plt.show()
```

```
showCluster(data, centroids, clusterAssment)
```



(2)GMM Algorithm:

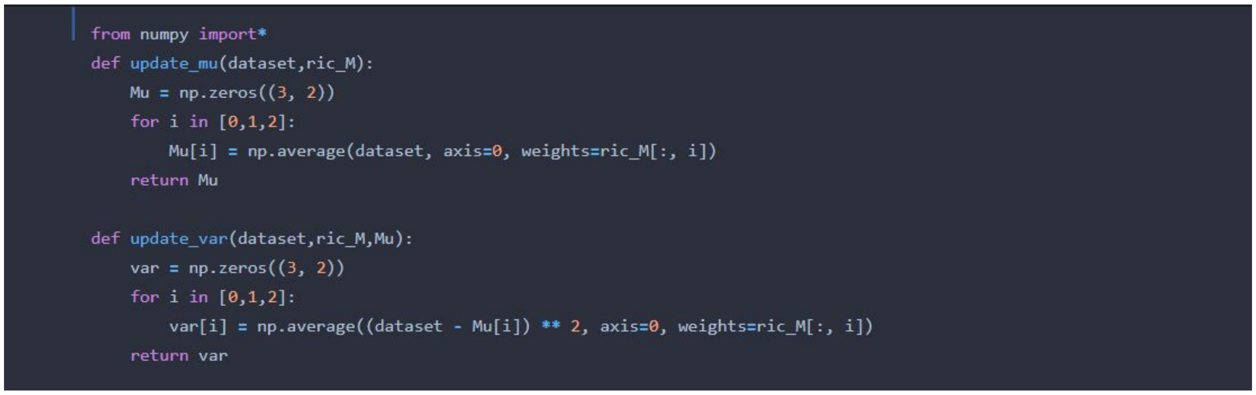
a. Implementation of GMM Algorithm



```
#Acquire dataset---array
from numpy import*

def LoadDataSet(filename):
    global dataset_ls
    File = open(filename, 'r').readlines()
    dataset =[]
    for line in File:
        stringlist = line.strip('\n').split(',')
        stringlist[0]=float(stringlist[0])
        stringlist[1]=float(stringlist[1])
        dataset.append(stringlist)
    dataset_ls=dataset
    return array(dataset)

#dataset=LoadDataSet('clusters.txt')
#print(dataset)
```

```
#random ric
import numpy as np
def random_ric():
    ric_m=np.random.randint(10,size=(150,3))
    return ric_m

#random_ric_M=random_ric()
#print(random_ric_M)
```

```
#Normalize
from numpy import*

def normalize(random_ric_M):
    ric_M=np.zeros((150, 3))
    ric_M[:,0]=random_ric_M[:,0]/random_ric_M.sum(axis=0)[0]
    ric_M[:,1]=random_ric_M[:,1]/random_ric_M.sum(axis=0)[1]
    ric_M[:,2]=random_ric_M[:,2]/random_ric_M.sum(axis=0)[2]
    return ric_M

#ric_M=normalize(random_ric_M)
#print(ric_M)
```



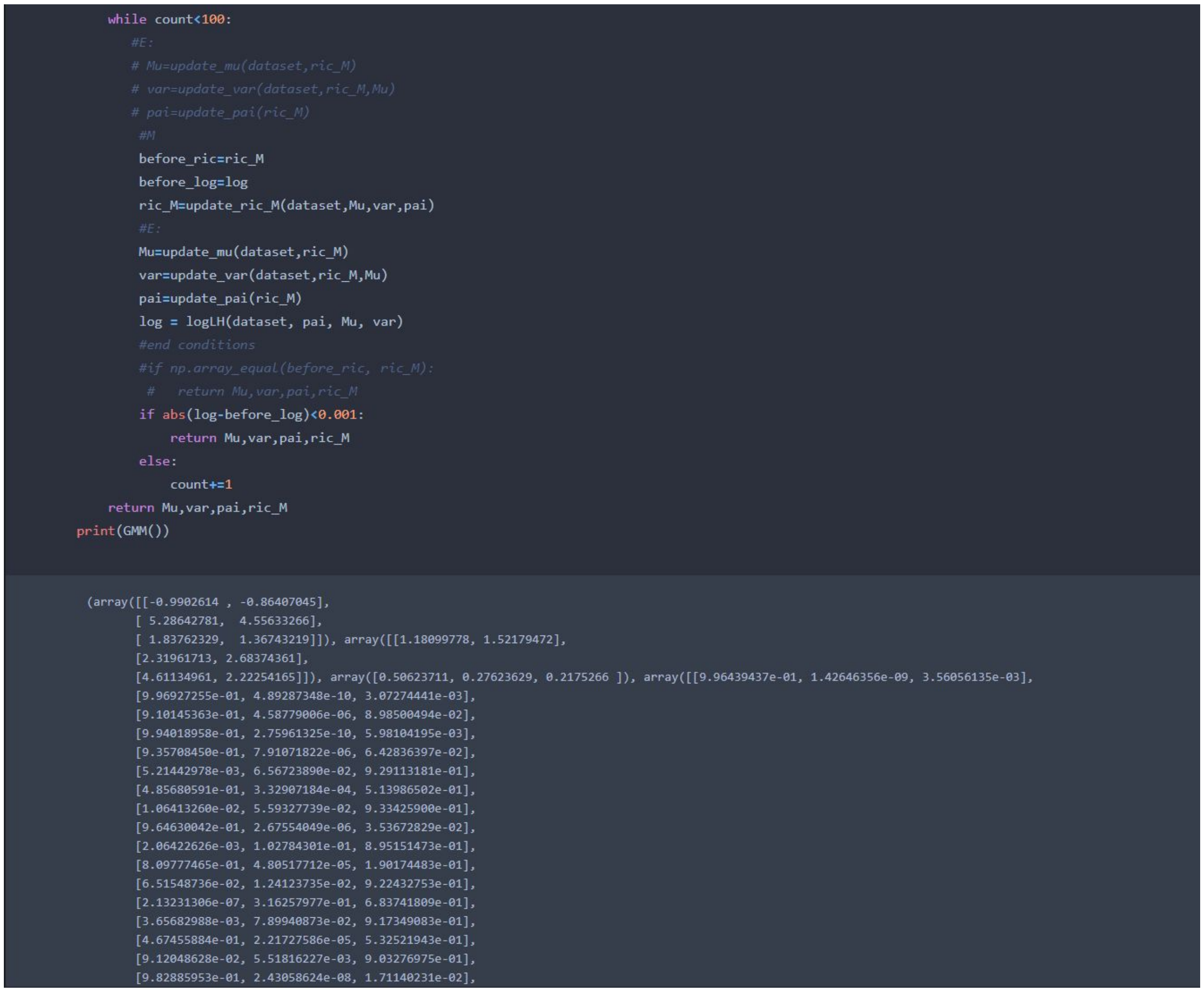
```
from numpy import*
def update_mu(dataset,ric_M):
    Mu = np.zeros((3, 2))
    for i in [0,1,2]:
        Mu[i] = np.average(dataset, axis=0, weights=ric_M[:, i])
    return Mu

def update_var(dataset,ric_M,Mu):
    var = np.zeros((3, 2))
    for i in [0,1,2]:
        var[i] = np.average((dataset - Mu[i]) ** 2, axis=0, weights=ric_M[:, i])
    return var
```

```python
#step E--->update_pai
def update_pai(ric_M):
    pai = ric_M.sum(axis=0) / ric_M.sum()
    return pai
#pai=update_pai(random_ric_M)
#print(pai)
```

```python
from scipy.stats import multivariate_normal

def update_ric_M(dataset,Mu,var,pai):
    R = np.zeros(((150, 3)))
    for i in [0,1,2]:
        R[:, i]=pai[i] * multivariate_normal.pdf(dataset, Mu[i], np.diag(var[i]))
    ric_M= R/R.sum(axis=1).reshape(-1, 1)
    return ric_M

#ric_M=update_ric_M(dataset,Mu,var,pai)
#print(ric_M)
```

```python
#calculate log
def logLH(dataset, pai, Mu, var):
    n_clusters = 3 #column
    n_points = 150 #row
    R = np.zeros(((n_points, n_clusters)))#build an empty 150*3array
    for i in range(n_clusters):
        R[:, i] = pai[i] * multivariate_normal.pdf(dataset, Mu[i], np.diag(var[i]))
    return np.mean(np.log(R.sum(axis=1)))

#log = logLH(dataset, pai, Mu, var)
#print(log)
```

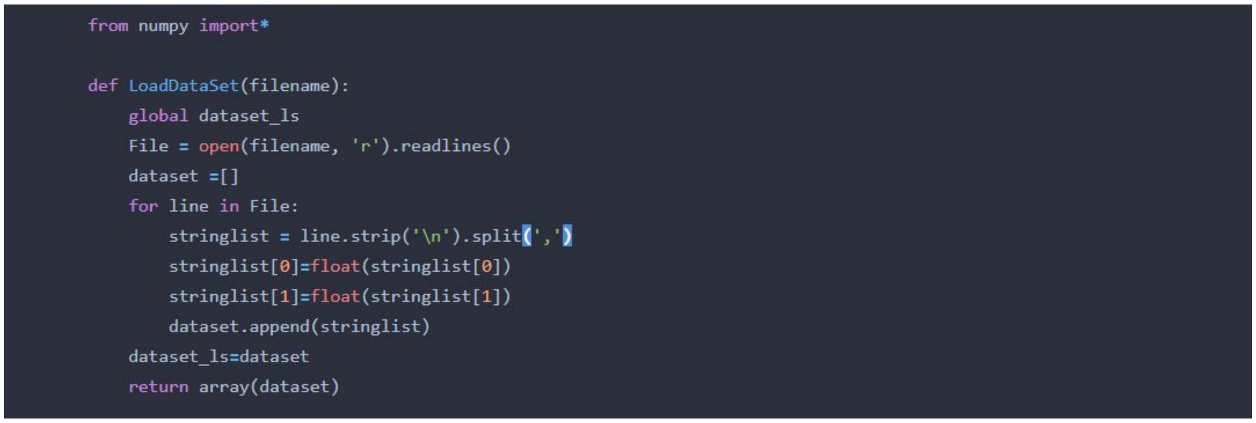
```python
#main
import numpy as np
def GMM():
    dataset=LoadDataSet('clusters.txt')
    random_ric_M=random_ric()
    ric_M=normalize(random_ric_M)
    #Mu=[[0, -1], [6, 0], [0, 9]]
    #var=[[1, 1], [1, 1], [1, 1]]
    #pai=[1 / 3] * 3
    pai=update_pai(ric_M)
    Mu=update_mu(dataset,ric_M)
    var=update_var(dataset,ric_M,Mu)


    count=0
    log=0
```



```
    while count<100:
       #E:
       # Mu=update_mu(dataset,ric_M)
       # var=update_var(dataset,ric_M,Mu)
       # pai=update_pai(ric_M)
        #M
        before_ric=ric_M
        before_log=log
        ric_M=update_ric_M(dataset,Mu,var,pai)
        #E:
        Mu=update_mu(dataset,ric_M)
        var=update_var(dataset,ric_M,Mu)
        pai=update_pai(ric_M)
        log = logLH(dataset, pai, Mu, var)
        #end conditions
        #if np.array_equal(before_ric, ric_M):
         #   return Mu,var,pai,ric_M
        if abs(log-before_log)<0.001:
            return Mu,var,pai,ric_M
        else:
            count+=1
    return Mu,var,pai,ric_M
print(GMM())
```

```
(array([[-0.9902614 , -0.86407045],
       [ 5.28642781,  4.55633266],
       [ 1.83762329,  1.36743219]]), array([[1.18099778, 1.52179472],
       [2.31961713, 2.68374361],
       [4.61134961, 2.22254165]]), array([0.50623711, 0.27623629, 0.2175266 ]), array([[9.96439437e-01, 1.42646356e-09, 3.56056135e-03],
       [9.96927255e-01, 4.89287348e-10, 3.07274441e-03],
       [9.10145363e-01, 4.58779006e-06, 8.98500494e-02],
       [9.94018958e-01, 2.75961325e-10, 5.98104195e-03],
       [9.35708450e-01, 7.91071822e-06, 6.42836397e-02],
       [5.21442978e-03, 6.56723890e-02, 9.29113181e-01],
       [4.85680591e-01, 3.32907184e-04, 5.13986502e-01],
       [1.06413260e-02, 5.59327739e-02, 9.33425900e-01],
       [9.64630042e-01, 2.67554049e-06, 3.53672829e-02],
       [2.06422626e-03, 1.02784301e-01, 8.95151473e-01],
       [8.09777465e-01, 4.80517712e-05, 1.90174483e-01],
       [6.51548736e-02, 1.24123735e-02, 9.22432753e-01],
       [2.13231306e-07, 3.16257977e-01, 6.83741809e-01],
       [3.65682988e-03, 7.89940873e-02, 9.17349083e-01],
       [4.67455884e-01, 2.21727586e-05, 5.32521943e-01],
       [9.12048628e-02, 5.51816227e-03, 9.03276975e-01],
       [9.82885953e-01, 2.43058624e-08, 1.71140231e-02],
```

b. Data Storage Structure
We store data in the type of array so we can access data randomly using the index number. For the specific content of the array, there are 3 columns that show the 3 clusters and 150 rows show the data points.

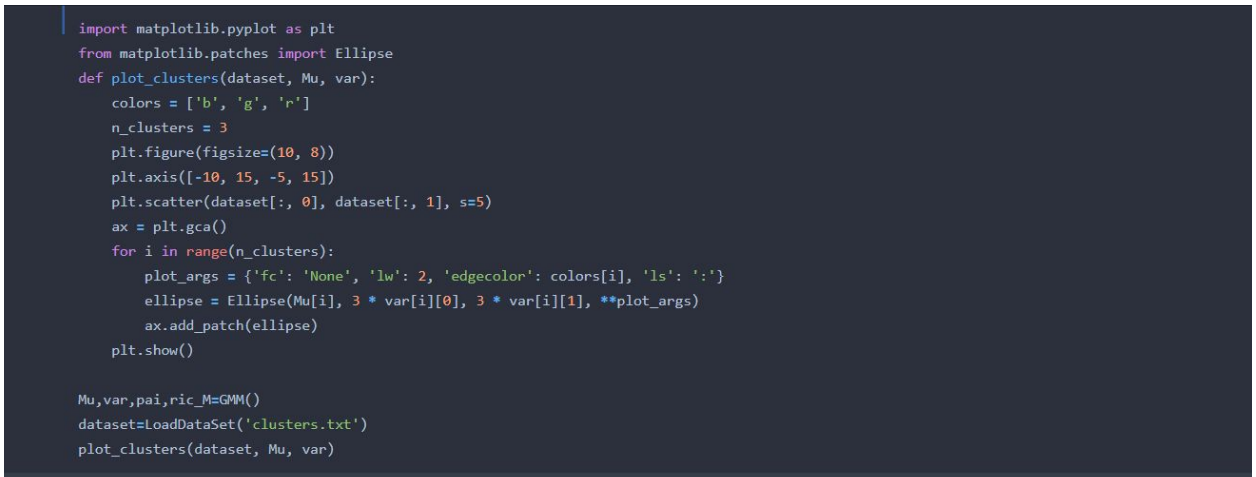


```
from numpy import*

def LoadDataSet(filename):
    global dataset_ls
    File = open(filename, 'r').readlines()
    dataset =[]
    for line in File:
        stringlist = line.strip('\n').split(',')
        stringlist[0]=float(stringlist[0])
        stringlist[1]=float(stringlist[1])
        dataset.append(stringlist)
    dataset_ls=dataset
    return array(dataset)
```

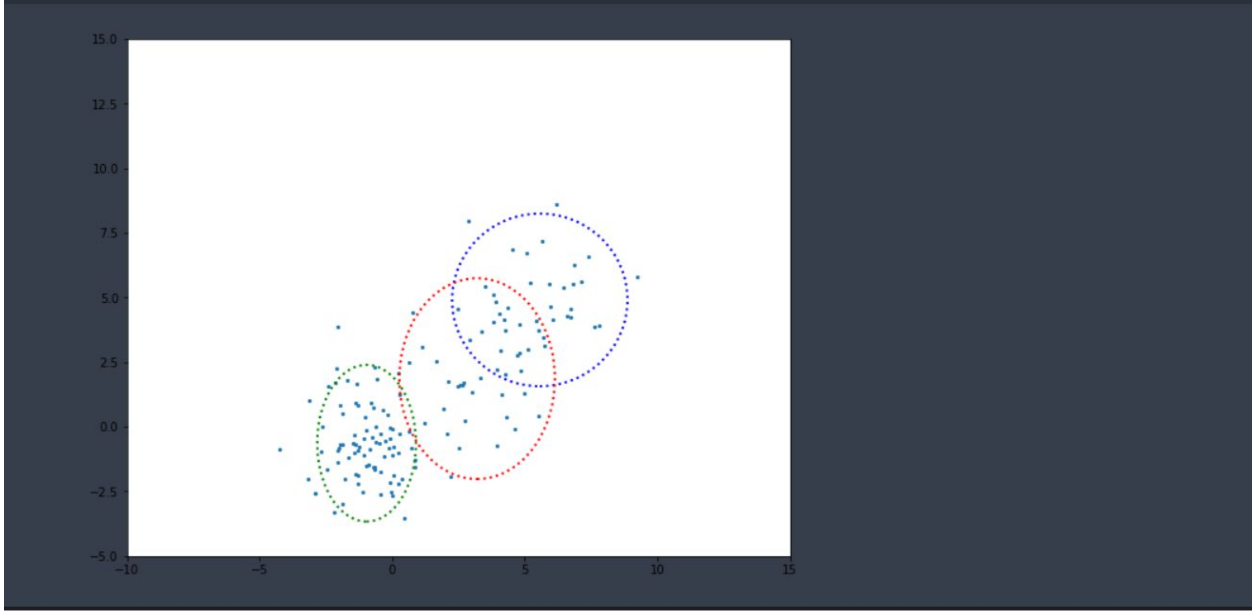
c. Code-level Optimizations

c1: Independent Functions: We extract some codes as the independent functions like some parameters' calculations before the main function so that we just call the function when we use them which can concise the code a lot.

c2: Visualization of K-means Algorithm: In order to improve the code performance, we import the matplotlib so that we can see the performance of the K-means Algorithm easily.



```
import matplotlib.pyplot as plt
from matplotlib.patches import Ellipse
def plot_clusters(dataset, Mu, var):
    colors = ['b', 'g', 'r']
    n_clusters = 3
    plt.figure(figsize=(10, 8))
    plt.axis([-10, 15, -5, 15])
    plt.scatter(dataset[:, 0], dataset[:, 1], s=5)
    ax = plt.gca()
    for i in range(n_clusters):
        plot_args = {'fc': 'None', 'lw': 2, 'edgecolor': colors[i], 'ls': ':'}
        ellipse = Ellipse(Mu[i], 3 * var[i][0], 3 * var[i][1], **plot_args)
        ax.add_patch(ellipse)
    plt.show()

Mu,var,pai,ric_M=GMM()
dataset=LoadDataSet('clusters.txt')
plot_clusters(dataset, Mu, var)
```



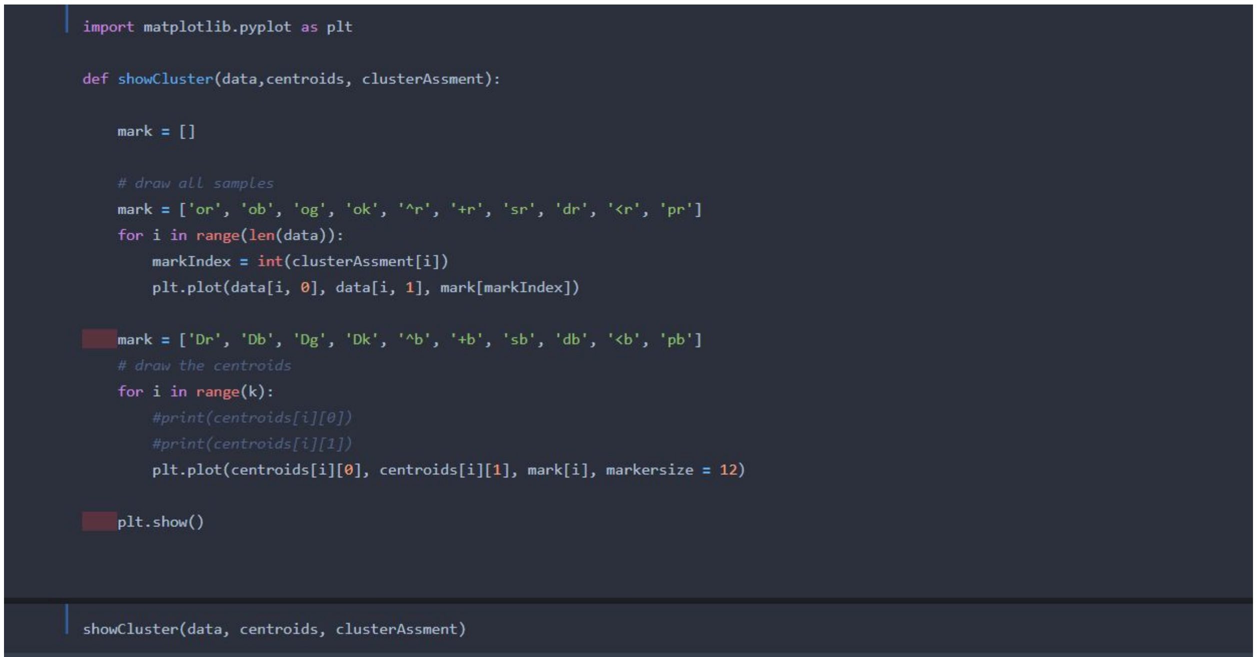
d. Challenges

d-1: Not familiar with the math knowledge in GMM Algorithm: Actually, we are not familiar with the processes to perform the GMM algorithm. The way to update the mean, covariance of each cluster is easy for us to understand but how to figure out the Gaussian Mixture distribution and likelihood function is difficult for us. So we spend a lot of time to do research on these two functions. Then we know the relationship among several parameters and know how to use the formulas in python.

d-2: Confused about the end condition: We know a new parameter called loglikelihood which can judge the end condition. If the loglikelihood parameter between two times is lower than 0.001, we judge the programming will stop and get the prediction answer.

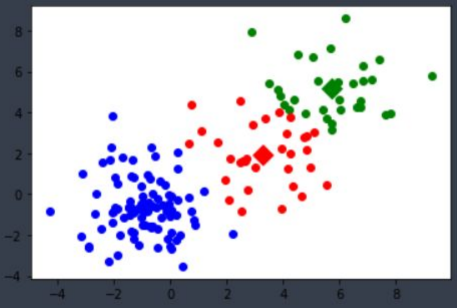
(3)Comparison between K-means and GMM Algorithm:

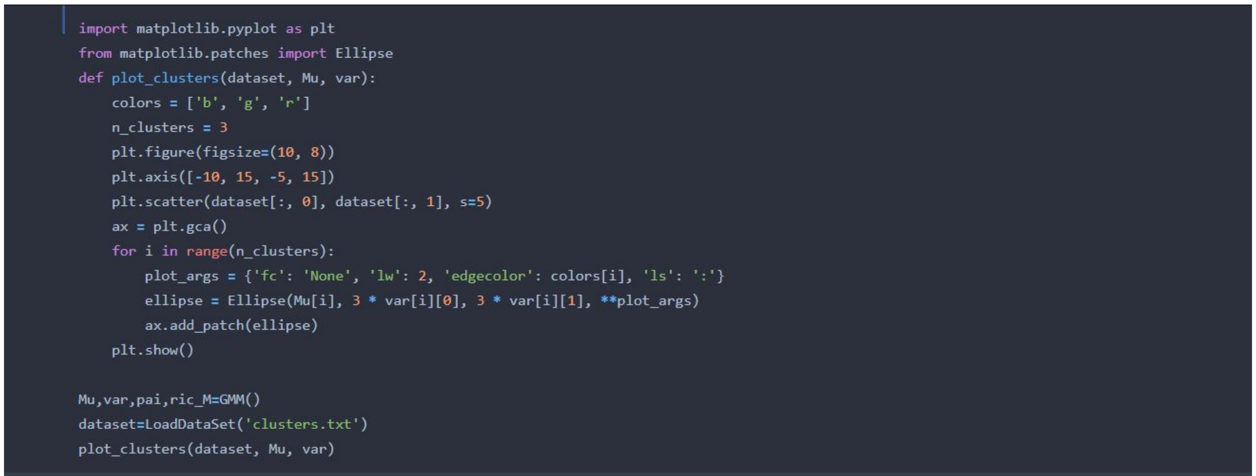
a. Result



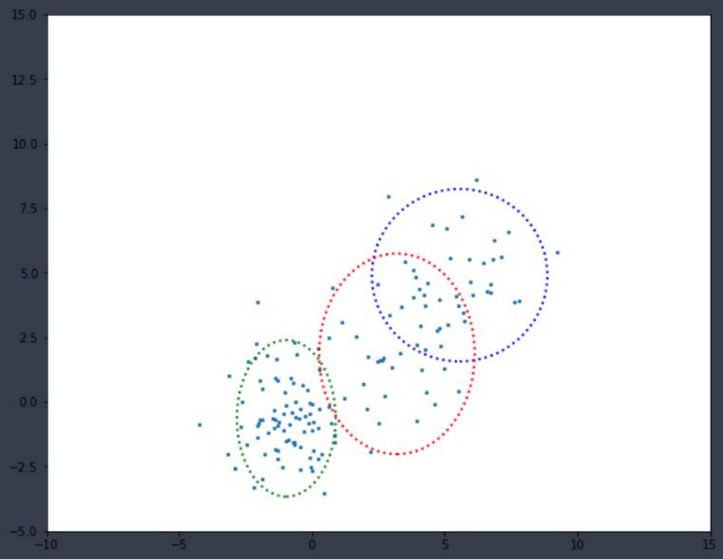
```
import matplotlib.pyplot as plt

def showCluster(data,centroids, clusterAssment):

    mark = []

    # draw all samples
    mark = ['or', 'ob', 'og', 'ok', '^r', '+r', 'sr', 'dr', '<r', 'pr']
    for i in range(len(data)):
        markIndex = int(clusterAssment[i])
        plt.plot(data[i, 0], data[i, 1], mark[markIndex])

    mark = ['Dr', 'Db', 'Dg', 'Dk', '^b', '+b', 'sb', 'db', '<b', 'pb']
    # draw the centroids
    for i in range(k):
        #print(centroids[i][0])
        #print(centroids[i][1])
        plt.plot(centroids[i][0], centroids[i][1], mark[i], markersize = 12)

    plt.show()
```

```
showCluster(data, centroids, clusterAssment)
```





```
import matplotlib.pyplot as plt
from matplotlib.patches import Ellipse
def plot_clusters(dataset, Mu, var):
    colors = ['b', 'g', 'r']
    n_clusters = 3
    plt.figure(figsize=(10, 8))
    plt.axis([-10, 15, -5, 15])
    plt.scatter(dataset[:, 0], dataset[:, 1], s=5)
    ax = plt.gca()
    for i in range(n_clusters):
        plot_args = {'fc': 'None', 'lw': 2, 'edgecolor': colors[i], 'ls': ':'}
        ellipse = Ellipse(Mu[i], 3 * var[i][0], 3 * var[i][1], **plot_args)
        ax.add_patch(ellipse)
    plt.show()

Mu,var,pai,ric_M=GMM()
dataset=LoadDataSet('clusters.txt')
plot_clusters(dataset, Mu, var)
```



According to the two plots, we can see the results between the two algorithms are similar.

b. Advantages and Disadvantages of two algorithm
Gaussian mixture models can be used to cluster unlabeled data in much the same way as k-means. However, there are some advantages to use Gaussian mixture models compared with the k-means algorithm. Firstly, k-means does not account for variance. By using the k-means algorithm, at the center of each cluster, it will be placed a circle and the radium is defined by the most distant point. When the data is circular, it is a good way to decide the clusters but when data have different shapes, we will get the similar results. Secondly, the Gaussian mixture model can deal with a very long's cluster. The most different thing between two algorithm is, the k-means algorithm performs hard classification while the GMM algorithm handles the soft classification.

Part 2: Software Familiarization

(1)K-means Algorithm

```
from sklearn.cluster import KMeans
kmeans = KMeans(n_clusters=2, random_state=0).fit(X)
kmeans.predict([[0, 0], [12, 3]])
```

The K-means algorithm is divided into three steps. In the first step, K initial centroids are randomly selected from the samples, and each sample is assigned to its nearest centroid. The second step creates a new centroid by averaging all samples assigned to each previous centroid. The third step is to calculate the difference between the new and old centroids. The algorithm repeats the last two steps until the centroids do not move obviously.
According to our k-means algorithm, our result will converge, but it may be a local minimum. It largely depends on the initialization of centroids. According to scikit-learn library function, when initializing the centroids, we try to keep them as far away from each other as possible, which leads to a better result than random initialization.

(2)GMM Algorithm

```
from sklearn import mixture
g = mixture.GMM(n_components=2)
g.fit(obs)
g.predict([[0], [2], [9], [10]])
```

The library function that offers implementation of the GMM algorithm is the GaussianMixture model. If we give the data to the model, it can assign to each sample to the Gaussian distribution. The way to use the library function about the GMM algorithm is simple. Firstly, we should from sklearn.mixture import GaussianMixture. Then we set the parameter the number of components so that we can implement the GMM algorithm. As for the way to improve our code, we can use the GaussianMixture model then set the component. Finally, we get the result. In this way, our codes will be concise a lot and the error rate of the result is low.

Part3: Applications

(1)K-means:

Dividing documents into different categories based on tags, topics, and document content is a standard K-means algorithm classification problem. First, the document needs to be initialized. Each document is represented by a vector, and the term frequency is used for document

classification. The document vectors are then clustered to identify similarities in the document group.
(2)GMM:
Clustering can help marketers improve their customer base (working in their target area) and divide customer categories based on purchase history, interest, or activity monitoring. GMM can help to find how to separate customers more accurately.