Coding part

Task I: Implement *k*-means

By running the *k*-means algorithm 100 times, we obtain the following table:

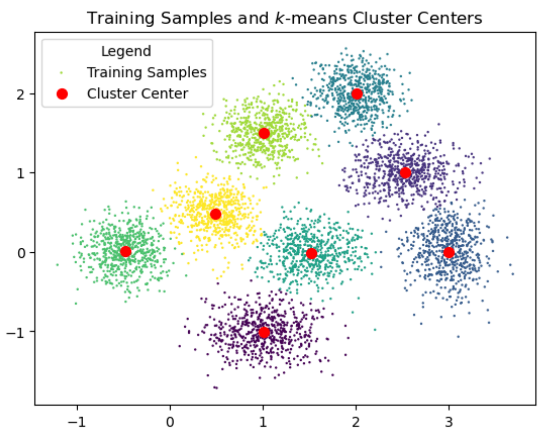
Convergence status	Number of appearances
NaN-value	58
Local minimum	39
Global minimum	3

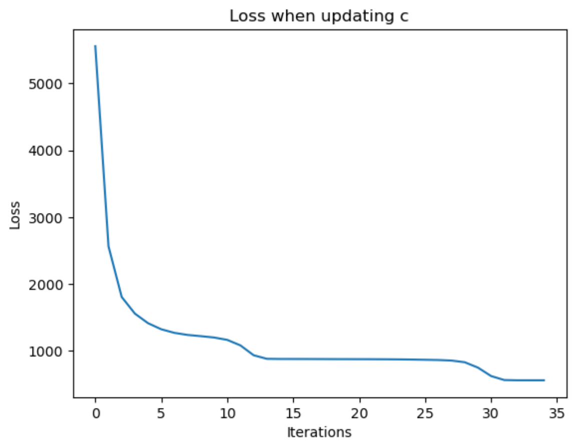
(Global minimum appears at the 53rd, the 69th, and the 91st time)

The frequency of the algorithm converging to the global minimum is **around 0.03**.

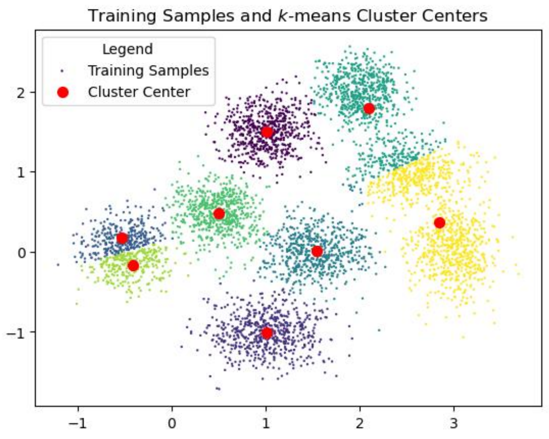
Below showing the scatter graph of cluster centers and the loss graph under different conditions:

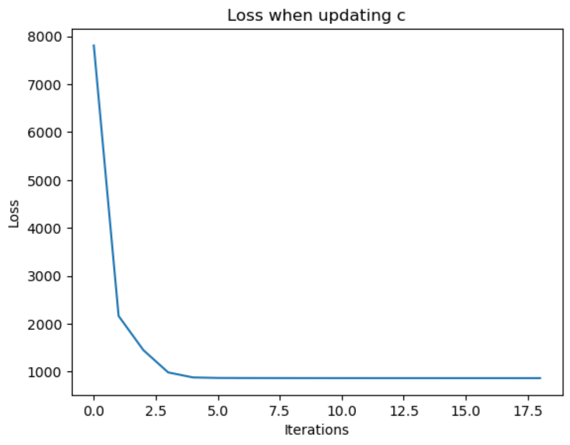
Global minimum:



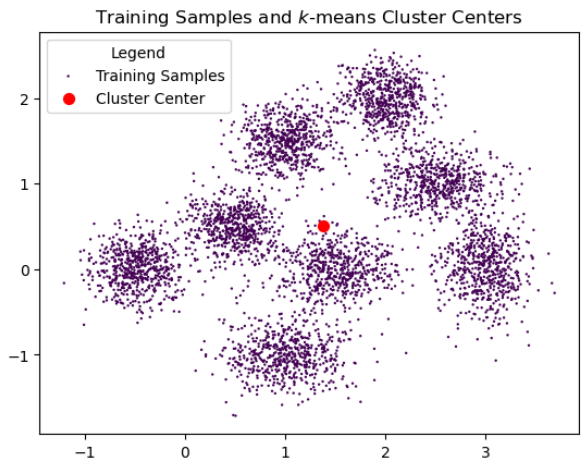


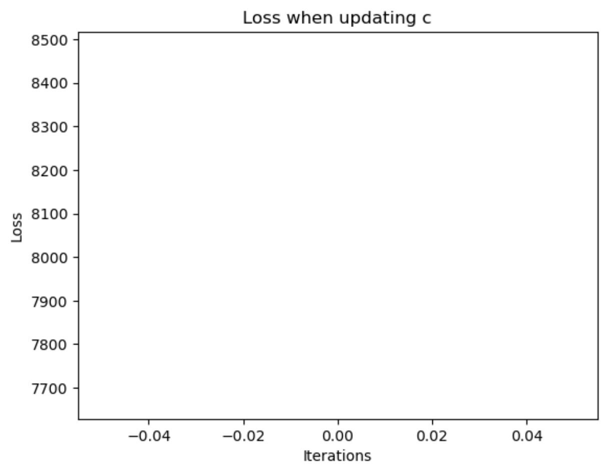
Local minimum:





NaN-value:





Besides, by repeatedly running the k-means algorithm multiple times, we obtain 10 results that converge to the global minimum, which have the iteration times of $\{25, 18, 11, 31, 33, 33, 38, 33, 25, 24\}$. The average iteration time is 27.1, which can roughly represent the convergence speed of the k-means algorithm.

Explanation for codes:

1. Key Idea:
   Iterate all centroids and calculate the distances from all points to each centroid. Obtain a (K, N) sized array (*distances*). By using argmin function along axis=0, we obtain the index array (*index*) for each data point referring to which cluster centroid it belongs to, and by using min function along axis=0, we obtain the distance array (didn't create an array to store, directly compute) from the data point to the centroid they belong. By taking the sum of square for elements in the distance array
2. Important variables: (a) *distances*: first list, then changed to ndarray, storing distances from all points to all centroids; (b) *c*, *c_new*: ndarray, sized (K, D), centroids from the last iteration and newly computed respectively; (c) *index*, ndarray, as explained above.

Task II: Implement *k*-means++

By running the k-means++ algorithm for 50 times, we obtain the following table:

Convergence status	**Number of appearances**
NaN-value	0
Local minimum	22
Global minimum	28

The frequency of the algorithm converging to the global minimum is **<u>around 0.56</u>**. Compared with the *k*-means algorithm, *k*-means++ algorithm has a higher probability to converge to a global minimum.

Besides, by repeatedly running the *k*-means++ algorithm multiple times, we obtain 10 results that converge to the global minimum, which have the iteration times of {20, 11, 5, 9, 10, 10, 7, 14, 11, 10}. The average iteration time is 10.7, which can roughly represent the convergence speed of the *k*-means++ algorithm.

Compare the convergence speed with the convergence speed of the *k*-means algorithm. We could have a directional hypothesis that the average convergence speed of the *k*-means++ algorithm is faster than *k*-means algorithm.

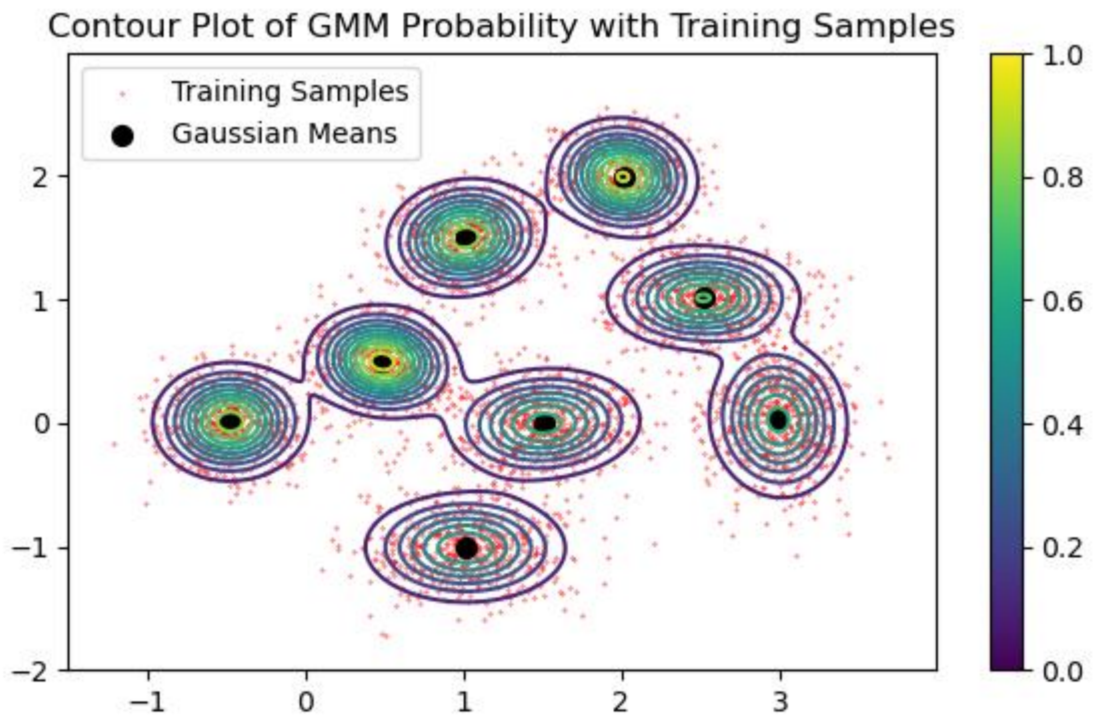
Explanation for codes:

1. Key Idea:
   The algorithm follows the instruction from the task description. First pick a sample data point into set *I*, which is the array of all centroids. Then generate the probability for each data point. In the generating process, first calculate the square distances from all data points to centroids chosen. Then, for each data point, keep the shortest distance from it to the closest given centroid. Append the data into a list (*lst_all*). Change the list into uniform data by dividing it with the sum of the whole list (and convert to a ndarray stored in *p*), we obtain the probability for all points to be the new centroid, which can be applied in the *np.random.choice()* function.
2. Important variables:
   (a) *lst_all*: as explained above; (b) *lst_distance*: list storing ndarrays that contain square distances from all data points to all centroids that have been chosen in the set *I* (centroids array).

(The manuscript of the comparison is attached at the end of the report)

Task III: Implement a Gaussian Mixture Model

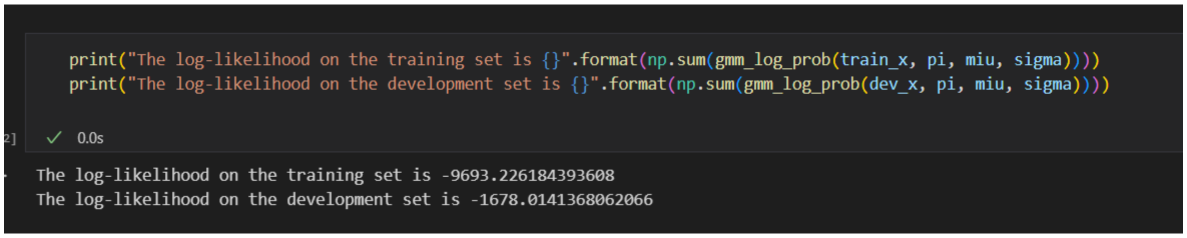
Implement a Gaussian Mixture Model using the Expectation-Maximization algorithm. The initial cluster centers generated by the *k*-means++ algorithm from task 2 are ensured to be at the local minimum. After we obtain all parameters of the Gaussian Mixture Model, we could plot the 2-d contour plot of the GMM model as follows:



It's obvious to see that the GMM fits the data well.

The log-likelihood of the GMM on the training set is -9693.226184393608.

The log-likelihood of the GMM on the development set is -1678.0141368062066.



```
print("The log-likelihood on the training set is {}".format(np.sum(gmm_log_prob(train_x, pi, miu, sigma))))
print("The log-likelihood on the development set is {}".format(np.sum(gmm_log_prob(dev_x, pi, miu, sigma))))
```

```
The log-likelihood on the training set is -9693.226184393608
The log-likelihood on the development set is -1678.0141368062066
```

Explanation for codes:

Key Idea:

a) *e_step()*: directly follow the formula given by the slide. We first calculate the numerator of the formula and then divide each with the sum according to clusters (K), which is the sum of row (axis=0) for each entry in *gama* which is a ndarray sized (N, K). Return *gama*;

b) *m_step()*: still directly follow the formula given by the slide. Enumerate from 0 to N and enumerate from 0 to K inside, we cumulatively calculate the N_k (*N_* , sum the j^th^ entry with the i, j-entry of *gama*) and μ_k (miu_new, similar but i,j-entry of *gama* times each data point). Calculate π_k. Calculate the covariance matrices after centroids since which are used in calculating those matrices. Return *pi_new*, *miu_new*, and *sigma_new*;

c) *em()*: simply initialize the *Nk*, *pi*, *miu*, and *sigma* with idea similar to *m_step()*. There's nothing important to explain. In the iteration, input the training data into *e_step()* and obtain the above 4 updated model parameters. Also compute the log-likelihood and append it in a list. The stopping criterion is to check if the difference between the new log-likelihood and the log-likelihood from last iteration is smaller than tolerance (set to be 10^{-5}), if smaller, then break, except reaching the maximum iteration times set as 1000. Finally, return *pi*, *miu*, and *sigma*, which are the GMM parameters.

d) All the important parameters are explained above, more to see code.