2 Programming

2.1 K-means and GMM Implementation

2.1.1 K-means

For K-means, I use 4 functions to do the clustering, they are initialize_centroid, assign_centroids, assign_cluster and fit_kmeans.

```
def initialize_centroid(data, K):
    data = list(data)
    init_centroid = random.sample(data, K)
    return init_centroid
```

For initialize_centroid, I randomly select 3 samples from the whole dataset to be the initial centroid.

```
def assign_centroids(curr_cluster, K):
    centroids = []
    for idx in range(K):
        centroid = np.mean(curr_cluster[idx], axis = 0)
        centroids.append(centroid)
    return centroids
```

For assign_centroids, the current clusters generated by last loop should be the input, and we just calculate the mean value of the points in each cluster, then assign the mean value to the new centroids.

```
def assign_cluster(data, curr_centroids, K):
    curr_cluster = {}
    for i in range(K):
        curr_cluster[i] = []
    for i in data:
        clusteridx = -1
        dist = []
        for j in range(K):
            dist.append(compute_distance(i, curr_centroids[j]))
        clusteridx = dist.index(min(dist))
        curr_cluster[clusteridx].append(i)
    return curr_cluster
```

For assign_cluster, we first generate an empty newly cluster, and iterate through all of the datapoints in the original dataset, compute the distance from each datapoint to the three newly generated centroids, then we grab the index which makes the distance from the datapoint to the centroids the smallest, to assign the datapoint to the new cluster, finally return the newly generated cluster.

```
def fit_kmeans(k, tolerence):
    K = k
    tol = tolerence
    centroids = initialize_centroid(X, K)
    current_clusters = assign_cluster(X, centroids, K)
    loop = True
    while loop:
        prev = centroids
        centroids = assign_centroids(current_clusters, K)
        current_clusters = assign_cluster(X, centroids, K)
        dist_sum = 0
        for i in range(len(centroids)):
            dist_sum += compute_distance(prev[i], centroids[i])
        if dist_sum < tol:
            loop = False
    return current_clusters
```

For the fit_kmeans, we first initialize the centroids, and assign the initial clusters, and start a

loop, which stops when the distance between the centroids from the last loop and the new loop is smaller than the tolerance. Finally we output the final clusters.

2.1.2 GMM

For GMM, I use 6 main functions to do the clustering. They are initialize_para, update_gamma, update_mu, update_sigma, predict and fit_GMM.

```
def initialize_para(data, K):
    pi = [1 / K] * K
    split_data = np.array_split(data, K)
    mu = [np.mean(split, axis = 0) for split in split_data]
    sigma = [np.cov(split.T) for split in split_data]
    return pi, mu, sigma
```

For initialize_para, I set the pi to be uniformly distributed probabilities, mu to be the mean value of each split from the whole data, sigma to be the covariance matrix of each split of the whole data.

```
def update_gamma(data, K, pi, mu, sigma):
    gamma = np.zeros((len(data), K))
    for n in range(len(data)):
        for k in range(K):
            num = pi[k] * multivariate_normal.pdf(data[n], mu[k], sigma[k])
            deno = []
            for j in range(K):
                element = pi[j] * multivariate_normal.pdf(data[n], mu[j], sigma[j])
                deno.append(element)
            gamma[n][k] = num / logsumexp(deno)
    return gamma
```

For update_gamma, I use the formula $\gamma_k = \frac{\pi_k N(x \mid \mu_k,\ \Sigma_k)}{\sum_{j=1}^{K} \pi_j N(x \mid \mu_j,\ \Sigma_j)}$ to update it.

```
def update_mu(data, K, gamma, N, mu):
    for k in range(K):
        musum = 0
        for n in range(len(data)):
            musum += gamma[n][k] * data[n]
        mu[k] = musum / N[k]
    return mu
```

```
def update_sigma(data, K, mu, gamma, sigma, N):
    for k in range(K):
        musum = 0
        for n in range(len(data)):
            musum += gamma[n][k] * np.outer((data[n] - mu[k]), (data[n] - mu[k]).T)
        sigma[k] = musum / N[k]
    return sigma
```

For update_mu and update_sigma, I use the formulas below to update them.

$$\boldsymbol{\mu}_k = \frac{1}{N_k} \sum_{n=1}^{N} \gamma_k^{(n)} \mathbf{x}^{(n)}$$

$$\boldsymbol{\Sigma}_k = \frac{1}{N_k} \sum_{n=1}^{N} \gamma_k^{(n)} \left(\mathbf{x}^{(n)} - \boldsymbol{\mu}_k\right) \left(\mathbf{x}^{(n)} - \boldsymbol{\mu}_k\right)^{\top}$$

```
def predict(data, K, pi, mu, sigma):
    gamma = update_gamma(data, K, pi, mu, sigma)
    label = []
    for i in range(len(data)):
        l = np.argmax(gamma[i])
        label.append(l)
    return label
```

For predict, I choose the index which has the largest gamma value to be the cluster that it belongs to, and output the current labels.

```
def fit_GMM(data, K, pi, mu, sigma, tol):
    flag = True
    while flag:
        gamma = update_gamma(data, K, pi, mu, sigma)
        N = np.sum(gamma, axis = 0)
        mu = update_mu(data, K, gamma, N, mu)
        sigma = update_sigma(data, K, mu, gamma, sigma, N)
        prev = list(pi)
        pi = [N[k] / len(data) for k in range(K)]
        diffsum = 0
        for i in range(len(prev)):
            diffsum += (prev[i] - pi[i])**2
        if diffsum <= tol:
            flag = False
    return pi, mu, sigma
```

For fit_GMM, I just create a loop to iterate the whole process, to update the parameters each time. The stopping criterion is that, when the distance between the current π_k and π_k in the last iteration is smaller than or equal to the tolerance. Then we finally output the last value of the three parameters.

2.2 Silhouette Coefficient and Rand Index

We calculate silhouette coefficient and rand index for Kmeans clustering:

```
In [19]: curr_cluster = fit_kmeans(3, 0.000001)
         sil = silhouette_coefficient(curr_cluster, 3)
         ran = rand_index(curr_cluster, txt, 3)

In [20]: sil
Out[20]: 0.47193373191268945

In [21]: ran
Out[21]: 0.8743677375256322
```

We also calculate silhouette coefficient and rand index for GMM clustering:

```
In [25]: X_ = shuffle(X)
         pi, mu, sigma = initialize_para(X_, 3)
         pi, mu, sigma = fit_GMM(X_, 3, pi, mu, sigma, 0.000001)
         label = predict(X_, 3, pi, mu, sigma)
         current_clusters = get_curr_clusters(label, X_, 3)
         sil = silhouette_coefficient(current_clusters, 3)
         ran = rand_index(current_clusters, txt, 3)

In [26]: sil
Out[26]: 0.4442191107942514

In [27]: ran
Out[27]: 0.8769195716564138
```

From the results, we can see that for the two distinct clustering algorithms, for silhouette coefficient, Kmeans > GMM. Since silhouette coefficient represents the distance between a point to its same cluster and its nearest cluster, if the distance between the point and the points in the same cluster is larger, the performance for the clustering algorithm should be good. So the performance of Kmeans is better than GMM judged by the silhouette coefficient.

However, for the rand index, we can see that GMM > Kmeans, rand index is useful when the true label is provided, if the rand index is larger and closed to 1, we say that the clustering

algorithm generated a result that is closed to the ground truth. Thus if we judge the performance of these two algorithms by rand index, we can say that GMM has better performance than Kmeans.

2.3 The sensitivity to the initialization of Kmeans and GMM

For each algorithm, we randomly assign the initialization, and run the two algorithms for ten times each.

We calculate the standard deviation of the two evaluation metrics for Kmeans:

```
In [22]: res = performance_evaluation_Kmeans()
         sil_std = res['silhouette_coef'].std()
         ran_std = res['rand_index'].std()

In [23]: sil_std
Out[23]: 0.001832990974720474

In [24]: ran_std
Out[24]: 0.0014527695986154717
```

We calculate the standard deviation of the two evaluation metrics for GMM:

```
In [28]: result = performance_evaluation_GMM()
         sil_std = result['silhouette_coef'].std()
         ran_std = result['rand_index'].std()

In [29]: sil_std
Out[29]: 0.03991633229536218

In [30]: ran_std
Out[30]: 0.0602683539129245
```

From the above results, we can observe that the standard deviations for these two evaluation metrics are quite small, which further implies that these two clustering algorithms works properly with different initializations.