2 Programming

Task: Clustering on UCI seed dataset, which can be downloaded from https://archive.ics.uci.edu/ml/datasets/seeds. The number of clusters is set as 3.

You need to:

1. Implement K-means and GMM-EM algorithms from scratch (i.e., no third-party or off-the-shelf package or library are allowed). Explain briefly your source codes in the report.

2. Implement 2 evaluation metrics including Silhouette Coefficient and Rand Index from scratch (i.e., not calling off-the-shelf package) to evaluate the performance of above clustering algorithms.

3. Analyze the sensitivity to the initialization of each algorithm (e.g., runone clustering algorithm with random initialization multiple times, and calculate the standard deviations of evaluation scores of these clustering results)

2. Programming Problems

Problem 2.1: Implement K-means and GMM-EM

2.1.1 K-Means:

The K-means algorithm is a method for clustering data points into a specified number of groups (**k**) based on their similarity. The source code provided is an implementation of the K-means algorithm. It takes in a dataset (**`data`**), the number of clusters (**`k`**), and the maximum number of iterations to run the algorithm for (**`max_iter`**).

The first step in the algorithm is to randomly choose k centroids, which will represent the center of each cluster. Then, for each data point, the Euclidean distance between the data point and each centroid is calculated through the function **`Euclidean_Distance`**. The data point is then assigned to the cluster corresponding to the centroid it is closest to.

Next, the centroids of each cluster are updated to be the mean of the data points assigned to that cluster. The algorithm then checks if the updated centroids are close enough to the previous iteration's centroids, indicating that the algorithm has converged. If the algorithm has converged, it returns the final centroids and labels for each data point indicating which cluster it belongs to. Otherwise, it continues running until the maximum number of iterations is reached.

2.1.2 GMM-EM:

The source code provided is an implementation of the Expectation-Maximization (EM) algorithm for fitting a Gaussian mixture model (GMM) to a dataset. The GMM is a probabilistic model that assumes the data points are generated from a mixture of several Gaussian distributions with different means and covariance matrices.
The EM algorithm is an iterative method for fitting the GMM to a dataset by alternating between the following two steps:

1. **E-Step:** Estimate the probabilities that each data point belongs to each cluster (i.e., to each Gaussian distribution in the mixture).

   The E step has the following simple form, which is the same for any mixture model:

   $$r_{ik} = \frac{\pi_k p(\mathbf{x}_i|\boldsymbol{\theta}_k^{(t-1)})}{\sum_{k'} \pi_{k'} p(\mathbf{x}_i|\boldsymbol{\theta}_{k'}^{(t-1)})}$$

   where $r_{ik} \triangleq p(z_i = k|\mathbf{x}_i, \boldsymbol{\theta}^{(t-1)})$ is the **responsibility** that cluster k takes for data point i.

2. **M-Step:** Update the model parameters (i.e., the means, covariance matrices, and mixture weights of the Gaussian distributions) based on the estimated probabilities from the E-step. The closed form solutions for updating model parameters are shown below:

$$\boldsymbol{\mu}_k = \frac{\sum_i r_{ik}\mathbf{x}_i}{r_k}$$
$$\boldsymbol{\Sigma}_k = \frac{\sum_i r_{ik}(\mathbf{x}_i - \boldsymbol{\mu}_k)(\mathbf{x}_i - \boldsymbol{\mu}_k)^T}{r_k}$$

In the implementation, the **GMM_EM** function takes in a dataset (**data**), the number of clusters (**num_clusters**), and optional arguments for the maximum number of iterations to run the algorithm for (**max_iter**) and the tolerance level for checking convergence (**tol**).

The function first initializes the model parameters (the means, covariance matrices, and mixture weights of the Gaussian distributions) randomly. Then, it repeatedly performs the E-step and M-step until the algorithm converges or the maximum number of iterations is reached. Finally, it **returns the predicted labels** for each data point indicating which cluster it belongs to.

In addition, since I need to calculate the matrix inverse for the probability density function (PDF) of the Gaussian Distribution every time, there might be some possible cases that the matrix is singular, so I set the methods as below:

```
# Estimate the probabilities that each data point belongs to each cluster
for k in range(num_clusters):
    mu_k = mu[k, :]
    cov_k = cov[k, :, :]
    pi_k = pi[k]
    z[:, k] = pi_k * multivariate_normal(mu_k, cov_k, allow_singular=True).pdf(data)
z /= z.sum(axis=1, keepdims=True)
```

Figure 1

```
# Compute the log-likelihood
log_likelihood_new = 0
for k in range(num_clusters):
    mu_k = mu[k, :]
    cov_k = cov[k, :, :]
    pi_k = pi[k]
    log_likelihood_new += pi_k * multivariate_normal(mu_k, cov_k, allow_singular=True).pdf(data)
log_likelihood_new = np.log(log_likelihood_new).sum()
```

Figure 2

Also, a small residual matrix is added to the covariance matrix each time in the E step to guarantee that it is **positive semi-definite**.

```
    cov[k, :, :] = (weighted_cov_sum / N_k[k]) + (1e-7) * np.eye(dims)
```

Figure 3

However, if there are still some cases that the function generates erroneous results, you may just **re-try**.

Problem 2.2: Implement Silhouette Coefficient and Rand Index

2.2.1 Silhouette Coefficient

The source code provided calculates the silhouette coefficient for a given dataset (**X**) and predicted cluster labels (**labels**). The silhouette coefficient is a metric that measures the quality of a clustering by quantifying how closely each data point is associated with its own cluster compared to other clusters.

The silhouette coefficient is calculated for each data point as follows:

1. Calculate the average distance between the data point and all other data points in the same cluster. This is referred to as the "**intra-cluster distance**" for the data point.
2. Calculate the average distance between the data point and all other data points in the nearest cluster. This is referred to as the "**nearest-cluster distance**" for the data point.
3. The silhouette coefficient for the data point is then calculated as the difference between the nearest-cluster distance and the intra-cluster distance divided by the maximum of the two distances.

The silhouette coefficient for the entire dataset is then calculated as the average silhouette coefficient for all data points. **A value close to 1** indicates that the data points **are well-separated** and correctly assigned to their respective clusters, whereas **a value close to -1** indicates that the data points are **poorly separated** and may be assigned to the wrong cluster.

Note that I maintained **a dictionary** to find the mean distance to the **nearest** cluster of each point, since there are multiple clusters.

```
outer_dist = dict()
dist_counter = dict()
for c in set(labels):
    outer_dist[c] = 0
    dist_counter[c] = 0
```

Figure 4

```
for key, values in outer_dist.items():
    if key != cluster:
        mean_cluster_distance.append(values / dist_counter[key])
b = np.amin(mean_cluster_distance)
```

Figure 5

2.2.2 Rand Index

Definition [edit]

Given a set of n elements $S = \{o_1, \ldots, o_n\}$ and two partitions of S to compare, $X = \{X_1, \ldots, X_r\}$, a partition of S into r subsets, $Y = \{Y_1, \ldots, Y_s\}$, a partition of S into s subsets, define the following:

- a, the number of pairs of elements in S that are in the **same** subset in X and in the **same** subset in Y
- b, the number of pairs of elements in S that are in **different** subsets in X and in **different** subsets in Y
- c, the number of pairs of elements in S that are in the **same** subset in X and in **different** subsets in Y
- d, the number of pairs of elements in S that are in **different** subsets in X and in the **same** subset in Y

The Rand index, R, is:[1][2]

$$R = \frac{a+b}{a+b+c+d} = \frac{a+b}{\binom{n}{2}}$$

Figure 6: Rand Index Definition

The source code you provided calculates the rand index for a given set of true cluster labels (**`labels_true`**) and predicted cluster labels (**`labels_pred`**). It does this by first computing the number of pairs of data points in the dataset. Then, it loops through each pair of data points and checks whether they are in the same cluster in both the true and predicted labels (**a**) or they are both clustered into different subsets (**b**). If they are, it increments a counter.

Finally, it calculates the rand index as the ratio of the number of pairs of data points that are in the same/different cluster in both sets of labels to the total number of pairs of data points. The function returns the calculated rand index.

2.2.3 Performance Evaluation

2.2.3.1 Simple Visualization of K-Means

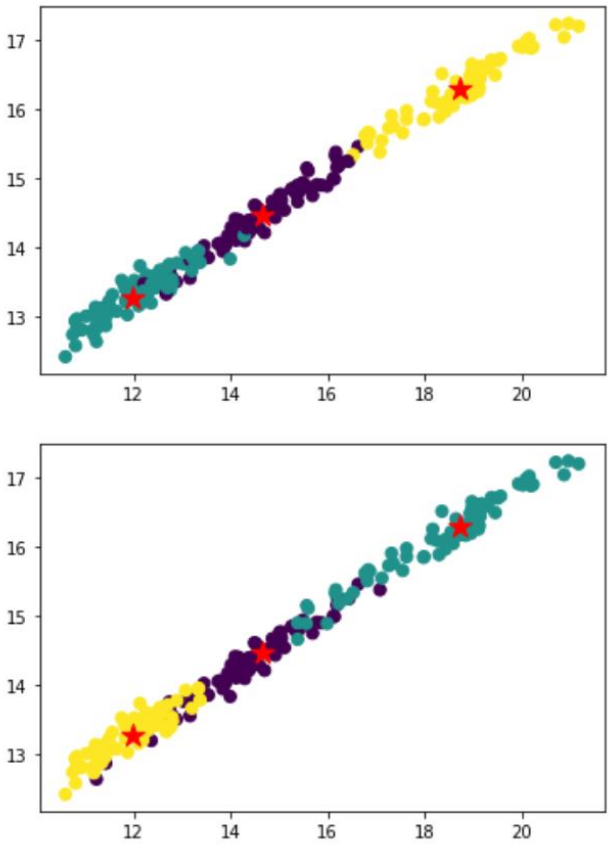

Figure 7

2.2.3.2 Silhouette Coefficient

```
# from sklearn.metrics import silhouette_score
# score = silhouette_score(X_train, em_labels)
# print(score)


print("Silhouette Coefficient for K-Means: ",silhouette_coefficient(X_train, km_labels))
print("Silhouette Coefficient for EM-GMM: ",silhouette_coefficient(X_train, em_labels))
```

executed in 73ms, finished 01:15:18 2022-12-15

```
Silhouette Coefficient for K-Means:  0.4757067358779757
Silhouette Coefficient for EM-GMM:  0.45192550844273177
```

Figure 8

2.2.3.3 Rand Index

```
print("Rand Index for K-Means: ", rand_index(km_labels, gt_label))
print("Rand Index for EM-GMM: ", rand_index(em_labels, gt_label))
```

executed in 56ms, finished 01:15:25 2022-12-15

```
Rand Index for K-Means:  0.8713602187286398
Rand Index for EM-GMM:  0.8769195716564138
```

Figure 9

Problem 2.3: Sensitivity Analysis

In this section, I analyze the sensitivity to the initialization of each algorithm by running one clustering algorithm with random initialization multiple times, and calculate the standard deviations of evaluation scores of these clustering results.

Source code:

```
for i in range (100):
    em_labels = GMM_EM(X_train, 3)
    km_centroids, km_labels = K_Means(X_train, 3)
    km_silh.append(silhouette_coefficient(X_train, km_labels))
    km_rand.append(rand_index(km_labels, gt_label))
    em_silh.append(silhouette_coefficient(X_train, em_labels))
    em_rand.append(rand_index(em_labels, gt_label))

km_silh_std = np.std(km_silh)
km_rand_std = np.std(km_rand)
em_silh_std = np.std(em_silh)
em_silh_std = np.std(em_rand)
```

Figure 9

The result is calculated based on **100 times** random initialization on the algorithm.

```
Sensitivity of K-Means to random initialization:
std_1(silhouette_coeff) = 0.0018269560468029222 std_2(rand_idx) = 0.0014733772888921263

Sensitivity of GMM-EM to random initialization.
std_1(silhouette_coeff) = 0.14107815167472268 std_2(rand_idx) = 0.14107815167472268
```

Figure 10

Hence, we can conclude that GMM EM is more sensitive to the random initialization of the model parameters.