Programming Questions

In this problem, the task is do the clustering on UCI seed dataset. The number of clusters is set to be 3.

Dataset

The seed data is from UCI. It includes 7 attributes and the original labels. We drop the labels and use 7 attributes to fit our clustering models. The original labels will be used when evaluating the models by Rand Index.

* Some indention errors in the seeds_dataset.txt file are fixed by hand

	Area	Perimeter	Compactness	Length of Kernel	Width of Kernel	Asymmetry coefficient	Length of kernel groove	Label
0	15.26	14.84	0.8710	5.763	3.312	2.221	5.220	1
1	14.88	14.57	0.8811	5.554	3.333	1.018	4.956	1
2	14.29	14.09	0.9050	5.291	3.337	2.699	4.825	1
3	13.84	13.94	0.8955	5.324	3.379	2.259	4.805	1
4	16.14	14.99	0.9034	5.658	3.562	1.355	5.175	1
...	...	...	...	...	...	...	...	...
205	12.19	13.20	0.8783	5.137	2.981	3.631	4.870	3
206	11.23	12.88	0.8511	5.140	2.795	4.325	5.003	3
207	13.20	13.66	0.8883	5.236	3.232	8.315	5.056	3
208	11.84	13.21	0.8521	5.175	2.836	3.598	5.044	3
209	12.30	13.34	0.8684	5.243	2.974	5.637	5.063	3

210 rows × 8 columns

1 Implement 3 clustering algorithms from scratch

1.1 K-Means

1.1.1 Main functions

Two functions are defined to implement K-Means.

cal_euclidean(xi,xj): calculate the Euclidean distance between xi and xj.
k_means(data, k): main function to implement K-Means, with input data and k, and output as centers, clusters, labels_pred, iter

- **data**: X
- **k**: number of clusters that we define
- **centers**: a dictionary with cluster indices as keys, and centroids of the clusters as values
- **clusters**: a dictionary with cluster indices as keys, and the corresponding data as values

- **labels_pred**: a list stores the predicted cluster numbers of data
- **iter**: the total number of iterations

1.1.2 Complete algorithm

Initialization: Randomly pick k points as the centers

Repeat until convergence:
1) Assignment: Given cluster centers, we update the assignments by assigning the points to the nearest cluster. The distance between points is calculated by the Euclidean distance.

2) Refitting: Given the assignment, we update the cluster center by calculating the mean of data within that cluster.

3) Check convergence: If the cluster centers do not change comparing to the last iteration, we set $is\_converge = True$.

1.1.3 Sample output

We set the random seed as 0 and run the K-Means.

```
np.random.seed(0)
k_centers, k_clusters, k_labels_pred, k_iter = k_means(X, 3)
print("K-Means")
print('centers:\n', k_centers)
print('predicted labels:\n',k_labels_pred)
```

```
K-Means
centers:
 {0: array([11.96441558, 13.27480519,  0.8522    ,  5.22928571,  2.87292208,
        4.75974026,  5.08851948]), 1: array([18.72180328, 16.29737705,  0.88508689,  6.20893443,  3.72267213,
        3.60359016,  6.06609836]), 2: array([14.64847222, 14.46041667,  0.87916667,  5.56377778,  3.27790278,
        2.64893333,  5.19231944])}
predicted labels:
 [2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0, 2, 2, 0, 2, 2, 2, 2, 2, 2, 0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 1,
2, 0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0, 0, 0, 0, 2, 2, 2, 2, 2, 2, 0, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 2, 1, 2, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 1, 2, 2, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0]
```

1.2 Accelerated K-Means with Triangle-Inequality

1.2.1 Main functions

One functions is defined to implement accelerated K-Means with triangle-inequality.

acc_k_means(data, k): main function to implement accelerated K-Means (the input and output arguments are the same as **k_means(data, k)** function)

1.2.2 Complete algorithm

Initialization: Randomly pick k points as the centers. Set the lower bound $l(x, c) = 0$ for each point x and center c. We use a N*k matrix **lower** to store

the lower bounds.

Compute $d(c, c')$: Compute the distances between the centers and store it in a k*k matrix called **cen_distance**.

First assignment: Assign each x to its closest initial center $c(x) = argmin_c d(x, c)$. In this step, we use Lemma 1 to avoid redundant distance calculations. Specifically, we compute $d(x, c')$ only if $d(c, c') < 2d(x, c)$. Also update $l(x, c) = d(x, c)$ and $u(x) = min_c d(x, c)$.

Set $r(x) = False$ for all x, and repeat until convergence:

1): Compute the distances between the centers and store it in a k*k matrix called **cen_distance**. For all centers c, compute $s(c) = 1/2 * min_{c' \neq c} d(c, c')$.

2): Get the indices of the points s.t. $u(x) > s(c(x))$.

3): For the points in 2) and all centers c s.t.

- (i) $c \neq c(x)$ and
- (ii) $u(x) > l(x, c)$ and
- (iii) $u(x) > 1/2 * d(c(x), c)$

3a) If $r(x)$, compute $d(x, c(x))$, update $u(x) = d(x, c(x))$, and assign $r(x) = False$. Otherwise, $d(x, c(x)) = u(x)$.

3b) If $d(x, c(x)) > l(x, c)$ or $d(x, c(x)) > 1/2 * d(c(x), c)$, compute $d(x, c)$, update $l(x, c) = d(x, c)$, and if $d(x, c) < d(x, c(x)$ then assign $c(x) = c$.

4): Add each point x to its corresponding cluster $c(x)$. Compute and update the center $m(c)$ as the mean of data within that cluster.

5): Compute the distances between the new centers and old centers and store them into a k*1 list **c_mc_distance**. For each x, assign $l(x, c) = max\{l(x, c) - d(c, m(c)), 0\}$.

6): For each x, assign $u(x) = u(x) + d(m(c(x)), c(x))$ and $r(x) = True$.

7): Check convergence. If the cluster centers do not change comparing to the last iteration, we set $is\_converge = True$.

1.2.3 Sample output

We set the random seed as 0 and run the accelerated K-Means.

```
np.random.seed(0)
acc_centers, acc_clusters, acc_labels_pred, acc_iter= acc_k_means(X, 3)
print("Accelerated K-Means")
print('centers:\n', acc_centers)
print('predicted labels:\n',acc_labels_pred)
```

```
Accelerated K-Means
centers:
 {0: array([11.96441558, 13.27480519,  0.8522    ,  5.22928571,  2.87292208,
        4.75974026,  5.08851948]), 1: array([18.72180328, 16.29737705,  0.88508689,  6.208934
43,  3.72267213,
        3.60359016,  6.06609836]), 2: array([14.64847222, 14.46041667,  0.87916667,  5.563777
78,  3.27790278,
        2.64893333,  5.19231944])}
predicted labels:
 [2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0, 2, 2, 0, 2, 2, 2, 2, 2, 2, 0, 2, 2, 2,
2, 2, 2, 2, 2, 2, 2, 1, 2, 0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 0,
0, 0, 0, 2, 2, 2, 2, 2, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,
1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2,
1, 2, 1, 1, 1, 1, 1, 1, 1, 2, 2, 2, 2, 1, 2, 2, 2, 2, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 2, 0, 0, 0, 0, 0, 0, 0, 0]
```

Note that the results from K-Means and accelerated K-Means are exactly the same, since the basic logic behind the algorithm does not change. Accelerated K-Means algorithm only reduces the unnecessary computations to speed up the original K-Means.

```
[acc_centers[i] == k_centers[i] for i in range(len(acc_centers))]
```

```
[array([ True,  True,  True,  True,  True,  True,  True]),
 array([ True,  True,  True,  True,  True,  True,  True]),
 array([ True,  True,  True,  True,  True,  True,  True])]
```

1.3 GMM-EM

1.3.1 Main class and functions

Class **GMM** is defined to implement GMM-EM with the following inner functions:

- **initialize(self,X)**: initialize the means mu, covariances sigma, and mixing coefficients pi by randomly pick k points as centers and setting the probabilities to be uniform.
- **cal_gamma(self,X)**: calculate gamma
- **e_step(self,X)**: evaluate the responsibilities given mu, sigma, pi by calling the **cal_gamma(self,X)** function
- **m_step(self,X)**: re-estimate mu, sigma, pi given gamma
- **get_likelihood(self,X)**: compute log likelihood $ln(p(X|pi, mu, sigma))$
- **fit(self,X)**: repeat e-step and m-step until convergence (i.e. the log likelihood does not change comparing to the last iteration)

- **get_labels(self,X)**: predict the label k of data x as the one with highest probability $p(z = k|x)$
- **get_iter(self)**: return the number of total iterations

1.3.2 Complete algorithm

Specifically, the main **fit(self,X)** function does the following:

Initialization: Randomly initialize the means mu $\boldsymbol{\mu}_k$, covariances sigma $\boldsymbol{\Sigma}_k$, and mixing coefficients pi π_k.

Repeat until convergence:
E-step: Given current parameters, calculate gamma:

$$\gamma_k^{(n)} = p\left(z^{(n)} = k \mid \mathbf{x}^{(n)}; \Theta\right) = \frac{\pi_k \mathcal{N}\left(\mathbf{x}^{(n)} \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k\right)}{\sum_{j=1}^{K} \pi_j \mathcal{N}\left(\mathbf{x}^{(n)} \mid \boldsymbol{\mu}_j, \boldsymbol{\Sigma}_j\right)}$$

M-step: Given gamma, update the parameters:

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

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

$$\pi_k = \frac{N_k}{N}, \text{ with } N_k = \sum_{n=1}^{N} \gamma_k^{(n)}$$

Check convergence: If the log likelihood does not change comparing to the last iteration, jump out of the loop. The log likelihood is computed as:

$$\ln p(\mathbf{X} \mid \boldsymbol{\pi}, \boldsymbol{\mu}, \boldsymbol{\Sigma}) = \sum_{n=1}^{N} \ln \left(\sum_{k=1}^{K} \pi_k \mathcal{N}\left(\mathbf{x}^{(n)} \mid \boldsymbol{\mu}_k, \boldsymbol{\Sigma}_k\right) \right)$$

1.3.3 Sample output

We set the random seed as 0 and run the GMM-EM.

```
np.random.seed(0)
gmm = GMM(k=3)
gmm.fit(X)
gmm_labels_pred = gmm.get_labels(X)
gmm_labels_pred
```

```
array([1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 1, 1,
       1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 2, 1, 1, 1, 1, 1, 1,
       1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 0, 0, 0, 1, 1, 1,
       1, 1, 1, 0, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2, 2,
       1, 2, 1, 1, 2, 1, 1, 2, 0, 1, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1,
       0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 0, 0])
```

2 Implement Silhouette Coefficient and Rand Index from scratch

2.1 Silhouette Coefficient

2.1.1 Implementation

silhouette_coef(data, result_list) function receives the data and a list of predicted clusters, then output the silhouette coefficient for samples.

Formally,

- a: mean distance between a point and all other points in the same cluster
- b: mean distance between a point and all other points in the next nearest cluster

Silhouette Coefficient s for a data point:

$$s = \frac{b - a}{\max(a, b)}$$

Finally, the take mean of s for each sample, and get our final Silhouette Coefficient.

2.1.2 Evaluate the above 3 models

```
print('Silhouette Coefficient for model:')
print('K-Means', silhouette_coef(X,k_labels_pred))
print('Accelerated K-Means',silhouette_coef(X,acc_labels_pred))
print('GMM-EM',silhouette_coef(X,gmm_labels_pred))
```

```
Silhouette Coefficient for model:
K-Means 0.4719337319126895
Accelerated K-Means 0.4719337319126895
GMM-EM 0.4376823626336355
```

Since a larger s indicates a better clustering performance, K-Means and Accelerated K-Means are better than GMM_EM in terms of Silhouette Coefficient. Notice that K-Means and Accelerated K-Means have the same performance since they give the same clustering result.

2.2 Rand Index

2.2.1 Implementation

rand_index(labels_true, labels_pred) function receives the labels from 2 clusterings X and Y, then output the rand index for samples.

Formally, given a set of n samples S

- a: number of pairs of elements in S that are in the same subset in X and in the same subset in Y
- b: number of pairs of elements in S that are in the different subset in X and in the different subset in Y

$$\mathrm{RI} = \frac{a+b}{\frac{n(n-1)}{2}}$$

2.2.2 Evaluate the above 3 models

```
print('Rand Index for model:')
print('K-Means', rand_index(y, k_labels_pred))
print('Accelerated K-Means', rand_index(y, acc_labels_pred))
print('GMM-EM', rand_index(y, gmm_labels_pred))
```

```
Rand Index for model:
K-Means 0.8743677375256322
Accelerated K-Means 0.8743677375256322
GMM-EM 0.8997038049669629
```

Since a larger RI indicates a higher similarity and here we are comparing our labels with the true labels, GMM_EM is better than K-Means and Accelerated K-Means in terms of Rand Index. Notice that K-Means and Accelerated K-Means have the same performance since they give the same clustering result.

3 Optional

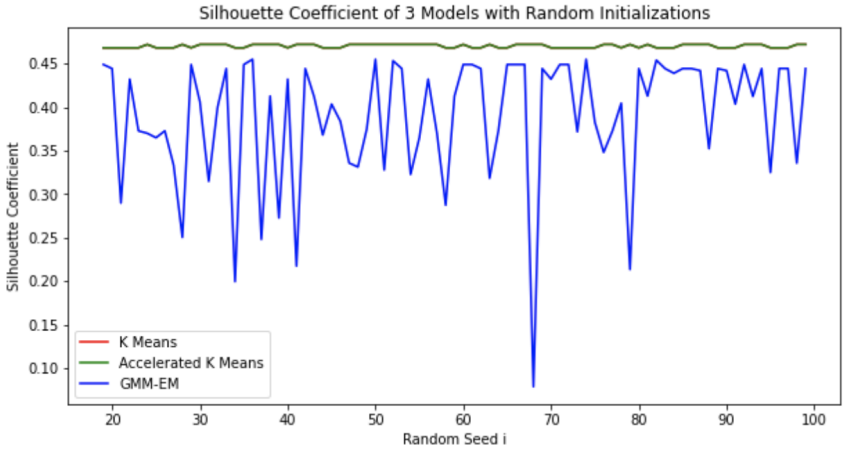
4 Analyze the sensitivity to clustering initialization

We run the 3 models with 80 random initializations by setting different seeds.

Using **Silhouette Coefficient** to evaluate, we have

```
variance of Silhouette Coefficient using
K-Means:  3.585003031722762e-06
Accelerated K-Means:  3.585003031722762e-06
GMM-EM:  0.005348251630649848

<Figure size 432x288 with 0 Axes>
```

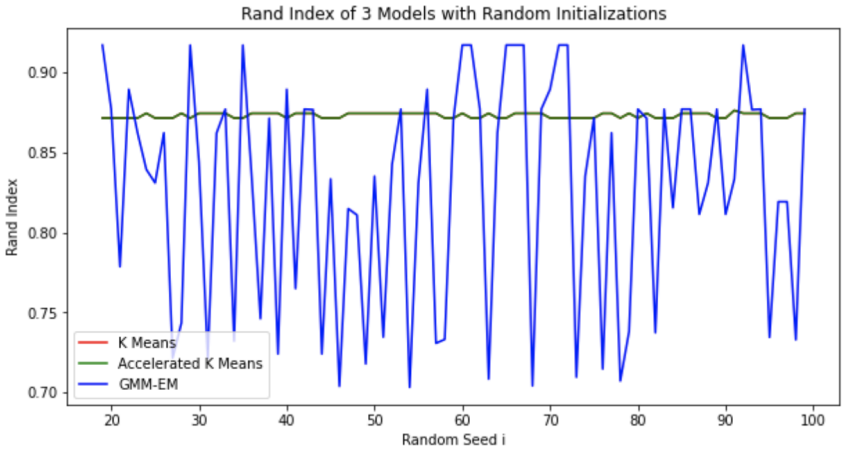


Evaluated by Silhouette Coefficient, the K-Means and Accelerated K-Means are always better than GMM_EM and less sensitive, since they have a lower variance of Silhouette Coefficient. GMM_EM model is very sensitive to the initializations. We can see that the performance of GMM_EM model varies much. Notice that K-Means and Accelerated K-Means have the same sensitivity since they give the same clustering result.

Using **Rand Index** to evaluate, we have

```
variance of Rand Index using
K-Means:  2.34674407587078e-06
Accelerated K-Means:  2.34674407587078e-06
GMM-EM:  0.004967464153572386

<Figure size 432x288 with 0 Axes>
```



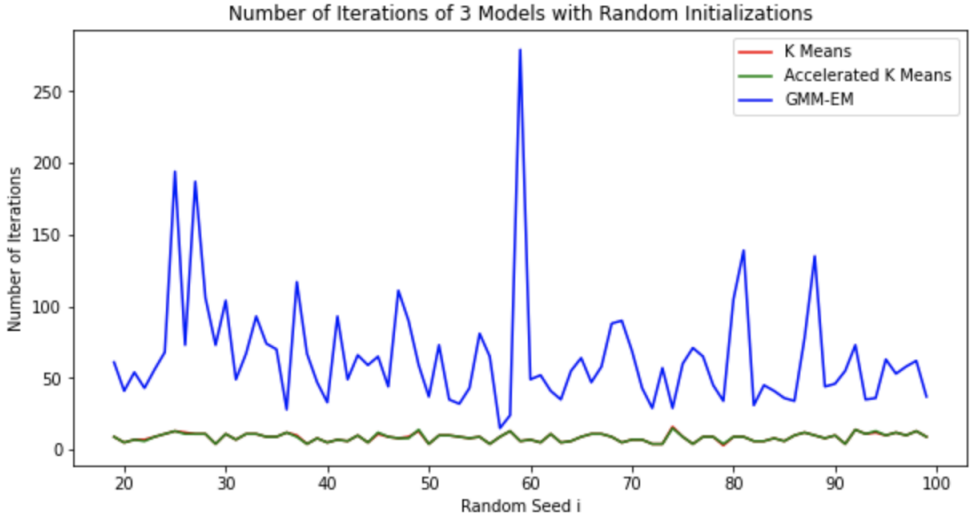
Evaluated by Rand Index, the K-Means and Accelerated K-Means are less sensitive, since they have a lower variance of Rand Index. GMM_EM model is very sensitive to the initializations. We can see that the performance of GMM_EM model varies much. Sometimes it is better than the K-Means and Accelerated K-Means, and sometimes it could have really bad results. Notice that K-Means and Accelerated K-Means have the same sensitivity since they give the same clustering result.

5 Report number of iterations and required time

5.1 Number of iterations of each algorithm

```
average number of iterations using
K-Means:  8.592592592592593
Accelerated K-Means:  8.580246913580247
GMM-EM:  65.58024691358025

<Figure size 432x288 with 0 Axes>
```

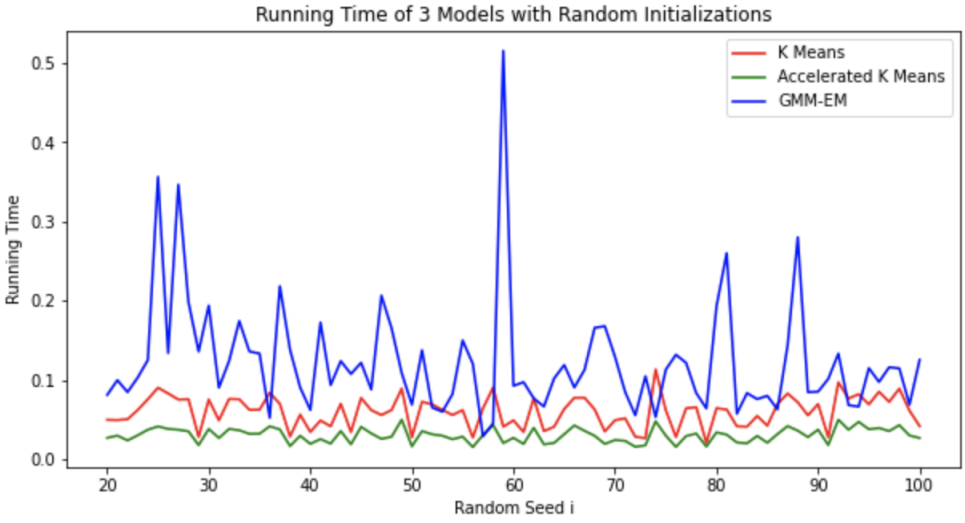


We can see that the number of iterations of K-Means and Accelerated K-Means are much lower and more stable than GMM_EM. The number of iterations are almost the same for K-Means and Accelerated K-Means.

5.2 Required time of each algorithm

```
average running time using
K-Means:  0.05996787695237148
Accelerated K-Means:  0.030372707932083694
GMM-EM:  0.12339384467513473

<Figure size 432x288 with 0 Axes>
```



We can see that the average running time of K-Means and Accelerated K-Means are lower and more stable than GMM_EM. The running time of Accelerated K-Means is lower than K-Means, since we reduce some unnecessary computations by using triangle-inequality.