# IE 529 Fall 2016 Computational Assignment 2

Jifu Zhao Date: 12/15/2016

## 1 Clustering Result Comparison

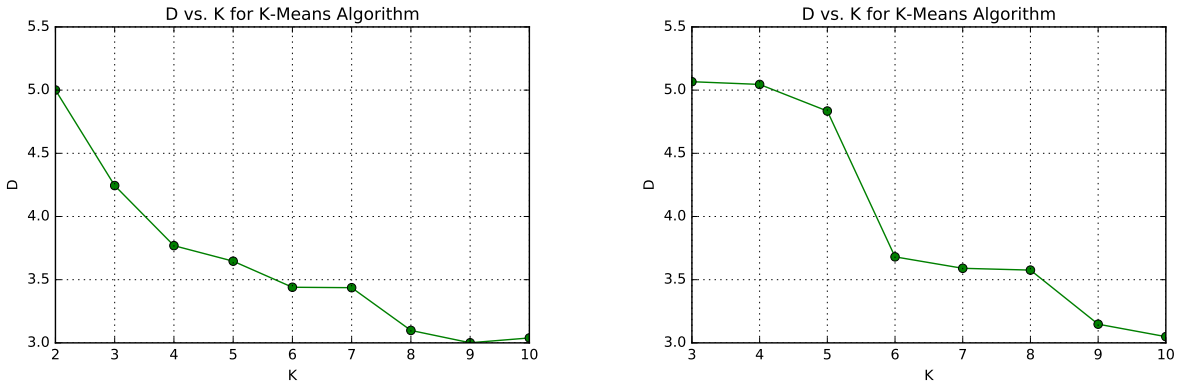
In this part, we have implemented 5 different algorithms for clustering problems: Lloyd's (K-Means) algorithm, Greedy K-Centers algorithm, Single-Swap for K-Centers algorithm, Spectral Clustering and Expectation Maximizaiton algorithm. Since all algorithm have some kind of randomness, to get the best solution that is as close to the global optimum as possiblem, in each algorithm, we change the random state of each algorithm for each run by using Python Numpy function:

$$np.random.seed()$$

In this way, for each run, we have different initializations. To find the best solution, we run each algorithm for 50 times and keep recording the lowest cost. Using the result that corresponds to the lowest cost as the final result. The details are shown below.

**(I).** Lloyd's (K-Means)Algorithm

1. In this part, with the Lloyd's algorithm for k-means clustering, we choose the best distortion D as the the objection function. The change of D versus cluster number K is shown in Fig. 1, where the result for clustering.txt is shown in Fig. 1a and the result for bigClusteringData.txt is show in Fig. 1b.




(a) clustering.txt (b) bigClusteringData.txt

Figure 1: Change of Distoration versus Cluster Number K for K-Means Algorithm

2. The scatter plot of the clustering result for clustering.txt is shown in Fig. 2 and the scatter plot of the clustering result for bigClusteringData.txt is shown in Fig. 3. The cluster centroids are clearly marked and different clusters are denoted by different colors.

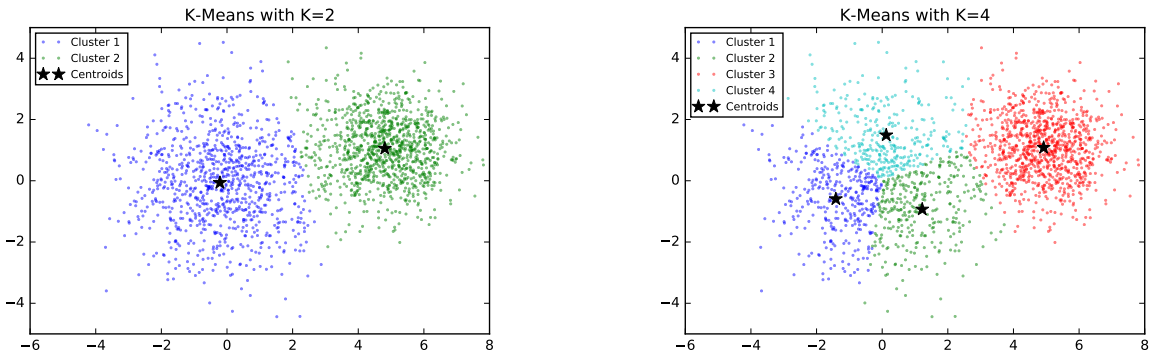



Figure 2: Clustering Result for clustering.txt with K-Means Algorithm

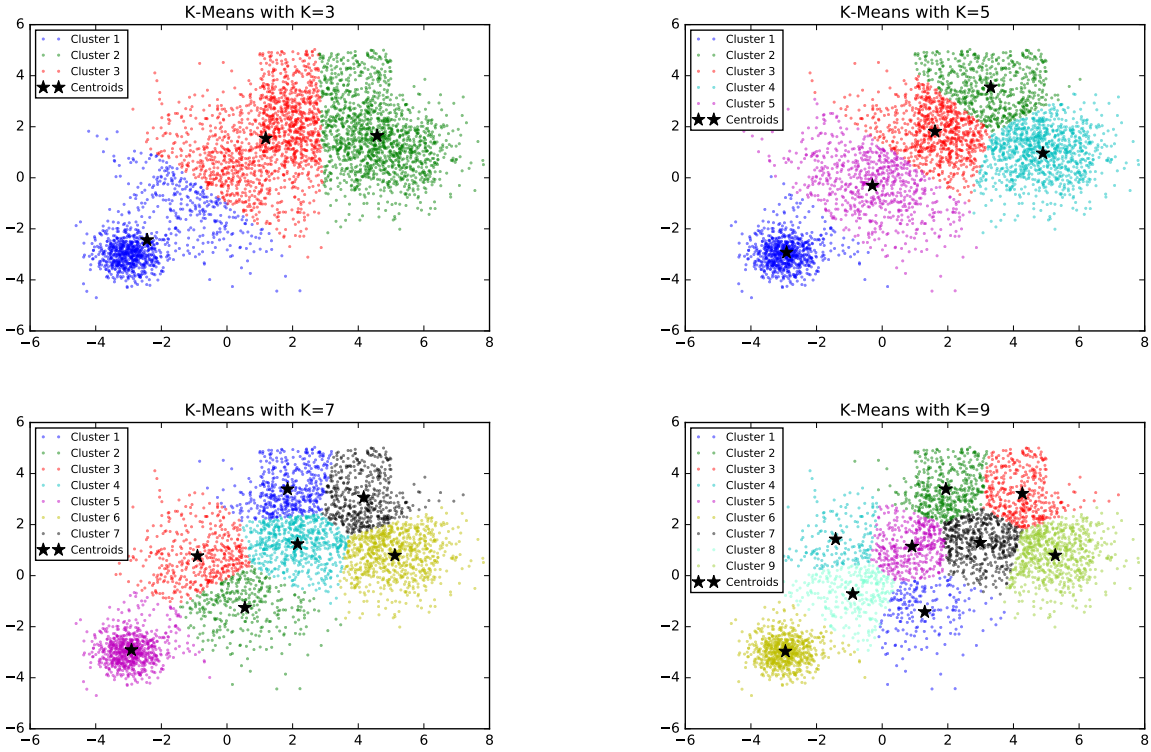



Figure 3: Clustering Result for bigClusteringData.txt with K-Means Algorithm

3. The Python code used for K-Means Algorithm is shown in Listing 1.

```python
import numpy as np
import time

def kMeans(X, K, tol=0.00001, random_state=None, verbose=True):
    """ function to implement the Lloyd's algorithm for k-means problem """
    np.random.seed(random_state)
    t0 = time.time()

    N, d = X.shape  # number of observations and dimensions
    index = np.random.choice(range(N), size=K, replace=False)
    Y = X[index, :]  # initial k centers
    C = np.zeros(N)
    D = 100
    count = 0
    diff = 100  # difference between D1 and D0

    while diff >= tol:
        D0 = D
        for i in range(N):
            # assign centers to ith data
            C[i] = np.argmin(np.sum((Y - X[i, :]) ** 2, axis=1))

        D = 0
        # re-compute the new centers
        for j in range(K):
            Y[j, :] = np.mean(X[C == j, :], axis=0)

        # compute the loss
        loss = np.zeros((N, K))
        for i in range(K):
            loss[:, i] = np.sqrt(np.sum((X - Y[i, :])**2, axis=1))
        D = np.max(np.min(loss, axis=1))
        diff = abs(D - D0)
        count += 1

    if verbose is True:
        t = np.round(time.time() - t0, 4)
        print('K-Means finished in ' + str(t) + 's, ' + str(count) + '
    iters')

    return Y, C, D
```

Listing 1: K-Means Algorithm Python Code

4. Convergence analysis.

   In Lloyd's algorithm, we choose $D = \max_{x_i \in X}(\min_{c_j \in Q}||x_i - c_j||_2)$ as the cost, the stop criterion is: $|D^{p+1} - D^p| < tol$, where tol is user-defined. A comparison of cost for clustering.txt and bigClusteringData.txt with different tols are shown in Table 1.

Table 1: Convergence with different tols for Lloyd's Algorithm

| Data / tol | 1E-7 | 1E-6 | 1E-5 | 1E-4 | 1E-3 | 1E-2 | 1E-1 | 1 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| clustering | 4.5434 | 4.3220 | 4.5206 | 4.8404 | 4.5841 | 4.9737 | 4.4175 | 5.0615 | 4.9204 |
| bigClusteringData | 5.0665 | 5.0665 | 5.1257 | 5.0665 | 5.0665 | 5.0569 | 4.8251 | 5.0097 | 5.4848 |

Note: here we choose $K = 3$.

In Table 1, we only run the code one time, the solution is not optimal. But as we increase the tol from $1E-7$ to 10, the final cost D tends to increase, which is consistent with our theoretical analysis.

5. Different Initialization Comparison.
   In Lloyd's algorithm, we run 50 times with different initialization and fix

$$tol = 1E-5$$

   the best result of D is shown in Table 2.

Table 2: Best Result (D) for Lloyd's Algorithm

| Data / K | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| clustering | 5.0008 | 4.2447 | 3.8771 | 3.6497 | 3.5388 | 3.3601 | 3.2111 | 3.0413 | 2.9365 |
| bigClusteringData | N.A. | 5.0665 | 5.0450 | 4.8319 | 3.6807 | 3.5872 | 3.5485 | 3.0802 | 3.0375 |

6. Cluster Index Comparison.
   In Lloyd's algorithm, the cluster index set C (first 20) for the best result is shown in Table 3 and Table 4.

Table 3: Index (first 20) for Lloyd's Algorithm (clustering.txt)

| K | Index |
|---|---|
| 2 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 |
| 3 | 0 0 2 0 2 2 2 0 2 0 2 0 2 2 0 2 2 0 0 2 |
| 4 | 2 3 3 3 1 1 1 2 1 2 1 2 1 1 2 1 1 2 3 1 |
| 5 | 0 3 3 3 1 1 1 0 1 0 1 0 1 4 0 1 1 0 3 1 |
| 6 | 4 4 3 4 3 5 5 4 5 5 5 2 5 5 4 5 5 2 4 5 |
| 7 | 6 6 5 2 5 6 6 2 3 6 3 6 3 3 2 3 6 0 6 6 |
| 8 | 4 6 6 2 3 5 5 4 5 5 5 4 3 5 2 5 3 4 6 5 |
| 9 | 8 8 0 8 1 8 8 7 4 8 8 8 1 4 7 1 1 3 8 4 |
| 10 | 7 7 3 7 2 0 7 5 0 7 0 7 2 1 4 0 0 5 7 0 |

Table 4: Index (first 20) for Lloyd's Algorithm (bigClusteringData.txt)

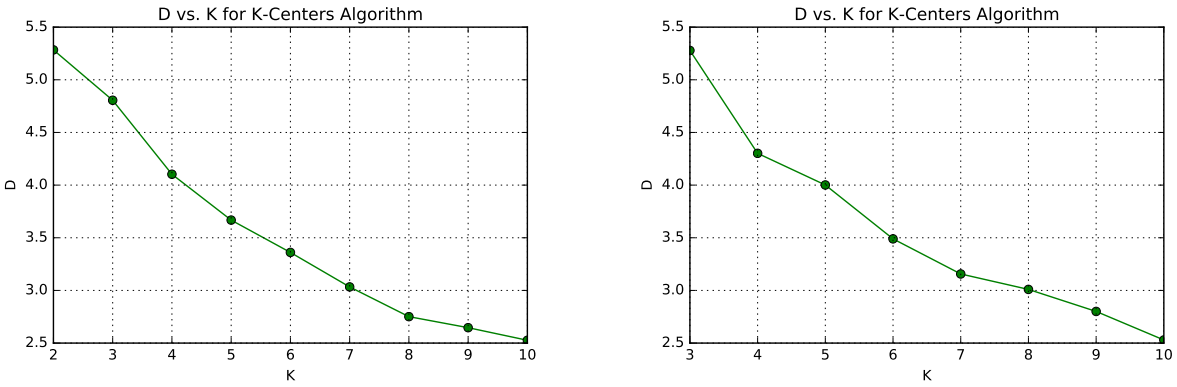
| K | Index |
|---|---|
| 3 | 2 0 1 0 0 1 0 1 0 1 0 1 2 2 0 0 2 0 0 0 |
| 4 | 0 3 2 1 1 2 1 2 1 2 3 1 0 0 1 1 3 3 3 1 |
| 5 | 3 2 4 2 2 4 2 4 2 4 2 1 3 3 2 1 0 2 0 1 |
| 6 | 3 5 1 5 2 1 4 1 5 1 5 2 3 3 5 4 0 5 0 4 |
| 7 | 5 0 3 0 4 3 0 3 0 3 0 4 6 5 0 2 6 0 1 4 |
| 8 | 7 0 4 2 2 4 6 4 0 4 0 1 5 7 0 6 5 0 3 6 |
| 9 | 6 0 1 3 3 1 4 1 3 1 0 8 2 6 3 4 2 0 5 3 |
| 10 | 6 8 0 8 8 0 1 0 8 0 8 9 2 6 8 1 4 8 3 7 |

**(II).** Greedy K-Centers Algorithm

1. In this part, with the Greedy K-Centers Algorithm, we choose

$$D = \max_{x_i \in X} (\min_{c_j \in Q} ||x_i - c_j||_2)$$

as the the objection function. The change of D versus cluster number K is shown in Fig. 4, where the result for clustering.txt is shown in Fig. 4a and the result for bigClusteringData.txt is show in Fig. 4b.




(a) clustering.txt (b) bigClusteringData.txt

Figure 4: Change of Distoration versus Cluster Number K for K-Center Algorithm

2. The scatter plot of the clustering result for clustering.txt is shown in Fig. 5 and the scatter plot of the clustering result for bigClusteringData.txt is shown in Fig. 6. The cluster centroids are clearly marked and different clusters are denoted by different colors.

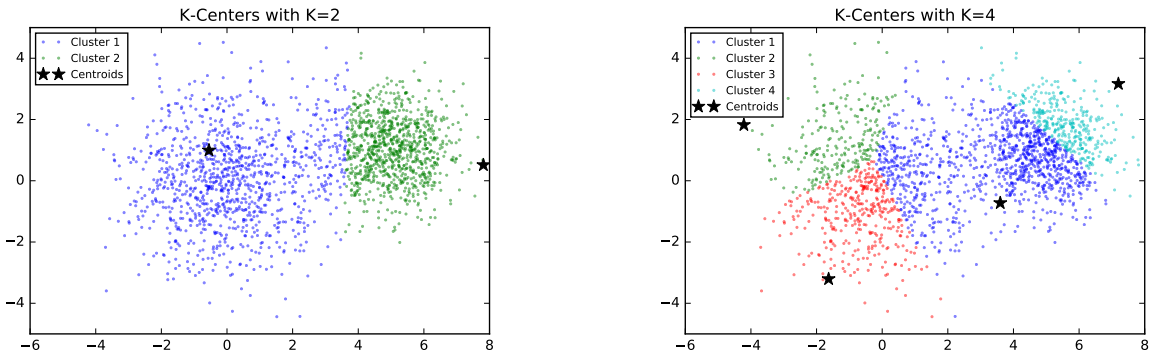



Figure 5: Clustering Result for clustering.txt with K-Center Algorithm

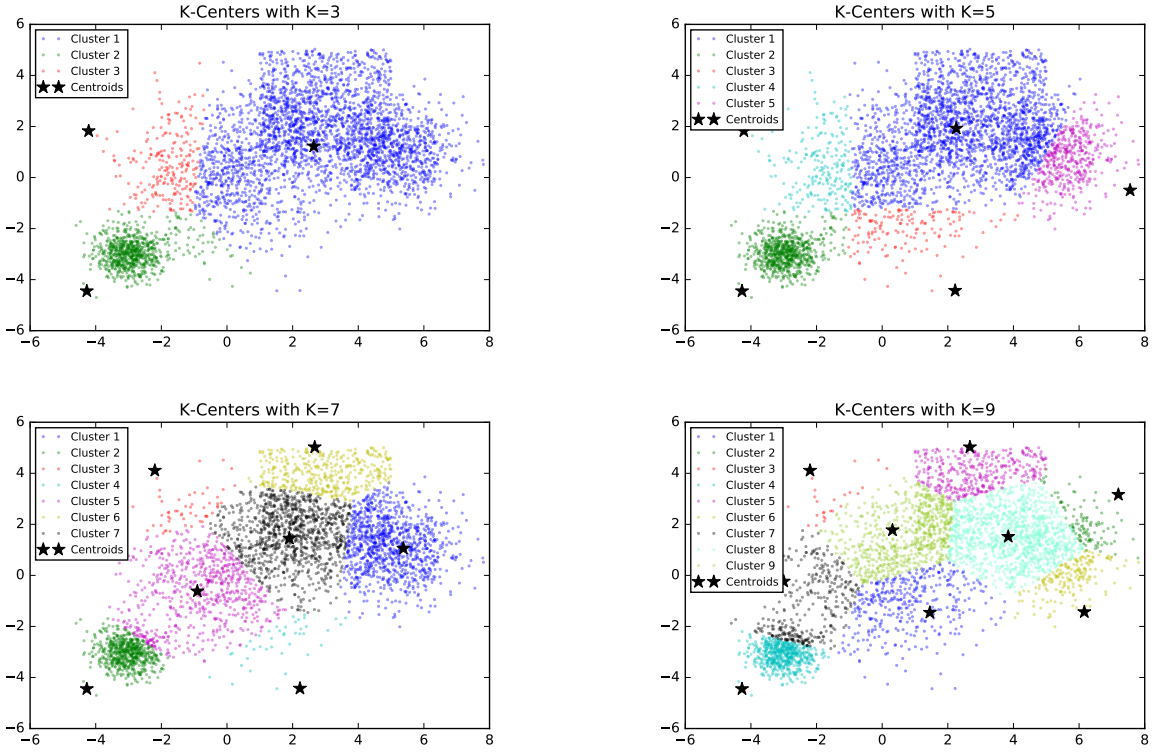



Figure 6: Clustering Result for bigClusteringData.txt with K-Center Algorithm

3. The Python code used for K-Centers Algorithm is shown in Listing 2.

```
import numpy as np
import time

def kCenters(X, K, random_state=None, verbose=True):
    """ function to implement the greedy k-centers algorithm """
    np.random.seed(random_state)
    t0 = time.time()

    N, d = X.shape
```

```
    # find the initial center
    index = np.random.choice(range(N), size=1)
    Q = np.zeros((K, d))
    Q[0, :] = X[index, :]
    idx = [index]

    i = 1
    while i < K:
        distance = np.zeros((N, i))
        for j in range(i):
            distance[:, j] = np.sum((X - Q[j, :])**2, axis=1)
        min_distance = np.min(distance, axis=1)
        new_index = np.argmax(min_distance)
        idx.append(new_index)
        Q[i, :] = X[new_index, :]
        i += 1

    loss = np.zeros((N, K))
    for i in range(K):
        loss[:, i] = np.sqrt(np.sum((X - Q[i, :])**2, axis=1))
    D = np.max(np.min(loss, axis=1))
    C = np.argmin(loss, axis=1)

    if verbose is True:
        t = np.round(time.time() - t0, 4)
        print('K-Centers is finished in ' + str(t) + 's')

    return Q, C, D, idx
```

Listing 2: K-Centers Algorithm Python Code

4. Different Initialization Comparison.

   In K-Centers algorithm, we run 50 times with different initialization, the best result of D is shown in Table 5.

Table 5: Best Result (D) for K-Centers Algorithm

| Data / K | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| clustering | 5.2838 | 4.8162 | 4.1387 | 3.7503 | 3.4691 | 3.1323 | 2.7510 | 2.7206 | 2.4634 |
| bigClusteringData | N.A. | 5.5288 | 4.1446 | 3.8318 | 3.3765 | 3.2933 | 3.0093 | 2.7431 | 2.6249 |

5. Cluster Index Comparison.

   In K-Centers algorithm, the cluster index set C (first 20) for the best result is shown in Table 6 and Table 7.

Table 6: Index (first 20) for K-Centers Algorithm (clustering.txt)

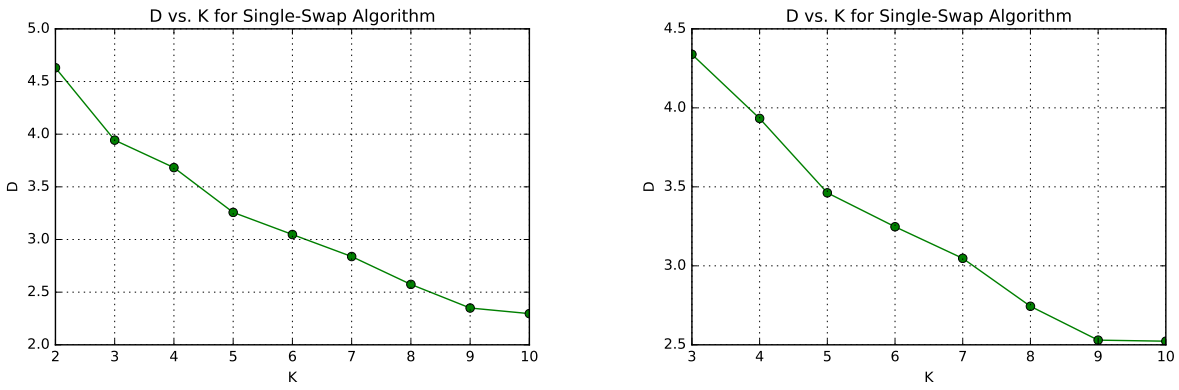
| K | Index |
|---|---|
| 2 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 |
| 3 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 |
| 4 | 2 2 1 2 1 3 3 2 3 2 3 2 3 3 2 3 3 2 2 3 |
| 5 | 4 4 4 4 4 4 4 4 4 4 4 4 4 0 4 4 4 3 4 4 |
| 6 | 5 5 5 5 5 5 5 3 5 5 5 5 5 5 3 5 5 3 5 5 |
| 7 | 4 4 6 4 6 4 4 4 4 4 4 4 4 4 1 4 4 3 4 4 |
| 8 | 6 6 6 6 0 0 0 6 0 6 0 6 0 5 6 0 0 2 6 0 |
| 9 | 6 6 6 6 0 0 6 6 0 6 0 6 0 5 6 0 0 2 6 6 |
| 10 | 6 6 6 6 6 6 6 6 6 6 6 6 5 5 0 6 6 8 6 6 |

Table 7: Index (first 20) for K-Centers Algorithm (bigClusteringData.txt)

| K | Index |
|---|---|
| 3 | 1 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 2 0 2 0 |
| 4 | 3 3 0 0 0 0 3 0 3 0 2 0 1 3 3 0 2 2 2 0 |
| 5 | 3 3 4 0 0 4 3 4 3 4 2 0 1 3 3 0 2 2 2 0 |
| 6 | 0 0 1 4 4 1 0 1 4 1 0 4 0 0 0 4 0 0 5 4 |
| 7 | 5 5 2 5 0 6 5 2 5 2 5 6 4 1 5 6 4 5 0 6 |
| 8 | 7 7 1 0 0 6 7 1 0 1 0 6 4 2 7 6 4 0 0 6 |
| 9 | 7 4 8 4 4 8 4 1 4 1 4 6 0 7 4 4 5 4 5 4 |
| 10 | 3 5 0 7 7 0 3 0 5 0 5 7 5 6 5 7 5 5 5 7 |

**(III).** Single-Swap Algorithm

1. In this part, with the Single-Swap Algorithm for K-Centers clustering, we choose the best distortion D as the the objection function. Setting $\tau = 0.05$, the change of D versus cluster number K is shown in Fig. 7, where the result for clustering.txt is shown in Fig. 7a and the result for bigClusteringData.txt is show in Fig. 7b.




(a) clustering.txt (b) bigClusteringData.txt

Figure 7: Change of Distoration versus Cluster Number K for Single-Swap Algorithm

2. The scatter plot of the clustering result for clustering.txt is shown in Fig. 8 and the scatter plot of the clustering result for bigClusteringData.txt is shown in Fig. 9. The cluster centroids are clearly marked and different clusters are denoted by different colors.

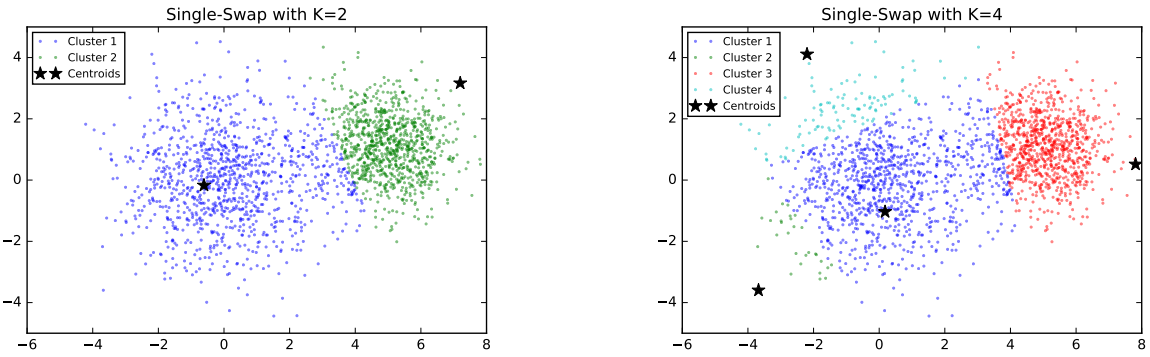



Figure 8: Clustering Result for clustering.txt with Single-Swap Algorithm

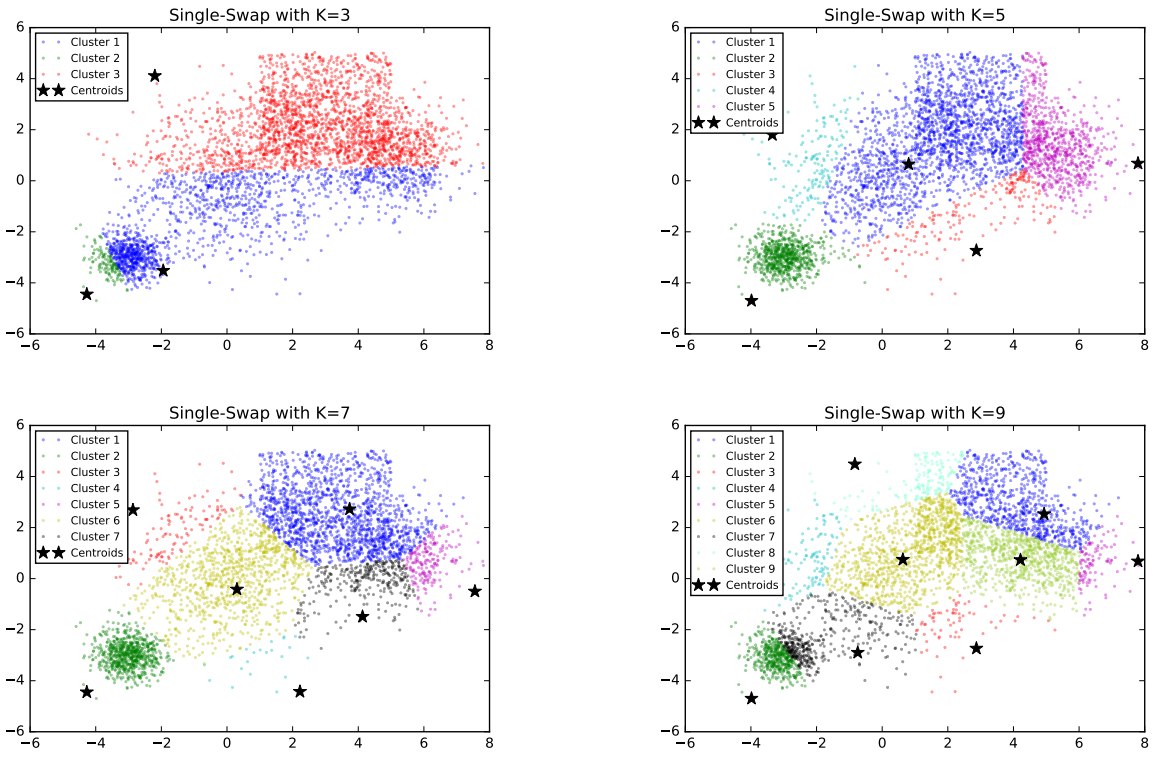



Figure 9: Clustering Result for bigClusteringData.txt with Single-Swap Algorithm

3. The Python code used for Single-Swap Algorithm is shown in Listing 3.

```
import numpy as np
import time
from k_centers import kCenters

def singleSwap(X, K, tau=0.05, random_state=None, verbose=True):
    """ function to implement the single-swap for k-centers algorithm """
    t0 = time.time()

    # calculate the initial centers
    Q, _, pre_cost, _ = kCenters(X, K, random_state=random_state,
                                 verbose=False)
    N, d = X.shape

    # compute the distance based on current centers
    distance = np.zeros((N, K))
    for idx in range(K):
        distance[:, idx] = np.sqrt(np.sum((X - Q[idx, :])**2, axis=1))
    cost = np.max(np.min(distance, axis=1))  # calculate cost

    i = 0
    while i < K:
        if i == 0:
            min_dist = np.min(distance[:, 0:], axis=1)
        elif i == (K - 1):
            min_dist = np.min(distance[:, :-1], axis=1)
        else:
```

```
            min_dist = np.minimum(np.min(distance[:, :i], axis=1),
                                  np.min(distance[:, (i + 1):], axis=1))
        swap = False  # keep recording whether or not swaped
        for j in range(N):
            tmp_dist = np.sqrt(np.sum((X - X[j, :])**2, axis=1))
            new_cost = np.max(np.minimum(min_dist, tmp_dist))
            if new_cost / cost < (1 - tau):
                Q[i, :] = X[j, :]
                distance[:, i] = tmp_dist
                swap = True
                cost = new_cost
        i += 1

        if swap is False:
            if i == K - 1:
                break
            else:
                i += 1
        elif (swap is True) and (i == K):
            i = 0

    C = np.argmin(distance, axis=1)

    if verbose is True:
        t = np.round(time.time() - t0, 4)
        print('Single-Swap is finished in ' + str(t) + 's')

    return Q, C, cost
```

Listing 3: Single-Swap Algorithm Python Code

4. Different Initialization Comparison.

   In Single-Swap algorithm, we run 50 times with different initialization, the best result of D is shown in Table 8.

Table 8: Best Result (D) for Single-Swap Algorithm

| Data / K | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| clustering | 4.6305 | 3.9619 | 3.7152 | 3.3236 | 3.0420 | 2.8479 | 2.5808 | 2.3493 | 2.3493 |
| bigClusteringData | N.A. | 4.3388 | 3.8504 | 3.4619 | 3.2156 | 2.9379 | 2.8479 | 2.6053 | 2.4973 |

5. Cluster Index Comparison.

   In Single-Swap algorithm, the cluster index set C (first 20) for the best result is shown in Table 9 and Table 10.

Table 9: Index (first 20) for Single-Swap Algorithm (clustering.txt)

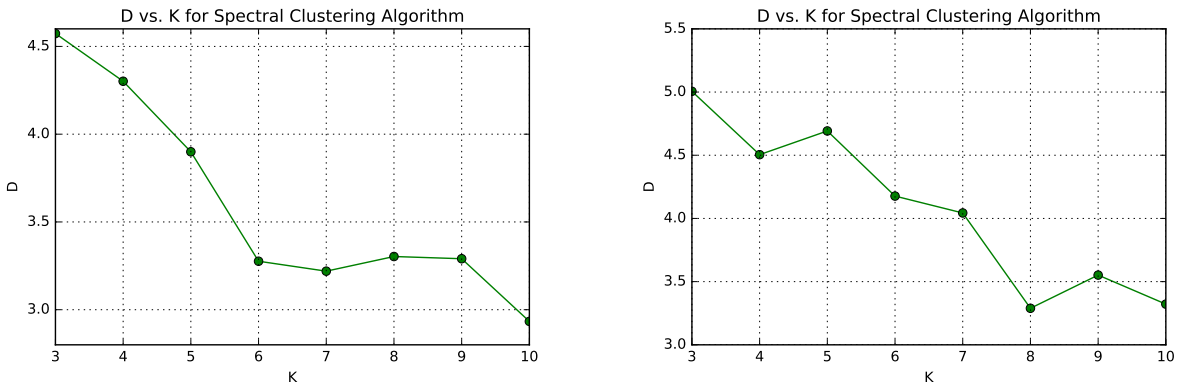
| K | Index |
|---|---|
| 2 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 |
| 3 | 0 0 0 0 2 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 |
| 4 | 0 0 0 2 0 0 0 2 0 0 0 0 0 0 2 0 0 2 0 0 |
| 5 | 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 4 3 4 4 |
| 6 | 0 0 3 0 5 0 0 0 0 0 0 0 4 0 2 0 0 2 0 0 |
| 7 | 6 6 6 6 6 6 6 6 0 6 0 6 4 0 3 6 6 6 6 0 |
| 8 | 7 7 0 7 0 7 7 7 7 7 7 7 5 5 4 7 7 4 7 7 |
| 9 | 4 4 4 4 0 4 4 4 4 4 4 4 3 8 2 4 4 2 4 4 |
| 10 | 4 4 4 4 8 4 4 4 9 4 4 4 3 9 2 4 4 2 4 4 |

Table 10: Index (first 20) for Single-Swpa Algorithm (bigClusteringData.txt)

| K | Index |
|---|---|
| 3 | 0 0 1 0 0 1 0 1 0 1 0 2 0 0 0 0 0 0 0 0 |
| 4 | 3 3 0 0 0 0 3 0 3 0 2 0 1 3 3 0 2 2 2 0 |
| 5 | 0 0 1 0 0 1 0 1 0 1 0 0 4 3 0 0 4 0 4 0 |
| 6 | 5 5 1 5 5 1 5 1 5 1 5 0 4 5 5 5 4 5 4 5 |
| 7 | 5 4 0 4 4 0 4 0 4 0 4 2 1 5 4 4 1 4 6 4 |
| 8 | 7 0 3 0 0 3 0 3 0 3 0 6 1 5 0 0 7 0 4 0 |
| 9 | 4 4 8 6 6 8 4 8 4 8 4 6 0 7 4 6 0 4 4 6 |
| 10 | 4 4 0 7 7 0 3 0 4 0 4 7 4 6 4 7 4 4 2 7 |

**(IV).** Spectral Clustering Algorithm

1. In this part, with the Spectral Clustering Algorithm, we choose the best distortion D as the the objection function and using Euclidean distance between points to construct the W matrix. Then, using the first k-eigenvectors as the input to Greedy K-Centers algortihm, the change of D versus cluster number K is shown in Fig. 10, where the result for clustering.txt is shown in Fig. 10a and the result for bigClusteringData.txt is show in Fig. 10b.




(a) clustering.txt (b) bigClusteringData.txt

Figure 10: Change of Distoration versus Cluster Number K for Spectral Clustering

Note: in above part, in order to calculate the cost $D = \max_{x_i \in X}(\min_{c_j \in Q}||x_i - c_j||_2)$, we actually implement the Greedy K-Centers algorithm, rather than K-Means algorithm. However, the result of K-Centers algorithm is very bad. In the following part, we implement K-Means algorithm for actual clustering.

2. The scatter plot of the clustering result for clustering.txt is shown in Fig. 11 and the scatter plot of the clustering result for bigClusteringData.txt is shown in Fig. 12. The cluster centroids are clearly marked and different clusters are denoted by different colors.

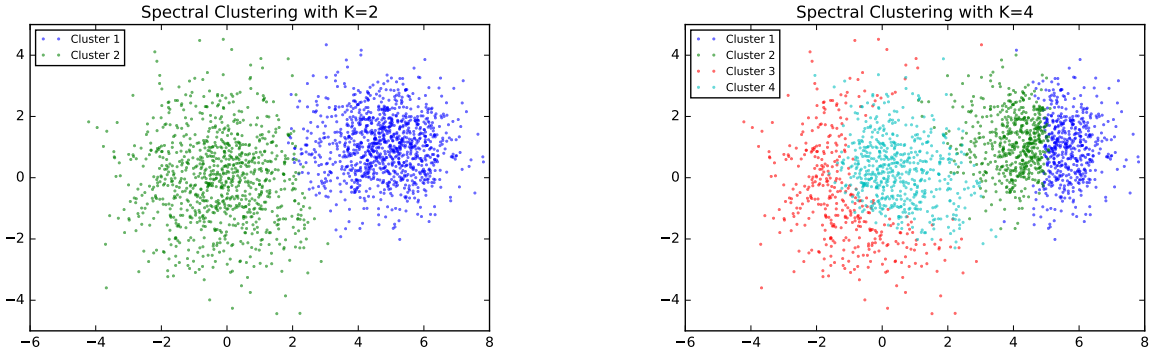



Figure 11: Clustering Result for clustering.txt with Spectral Clustering

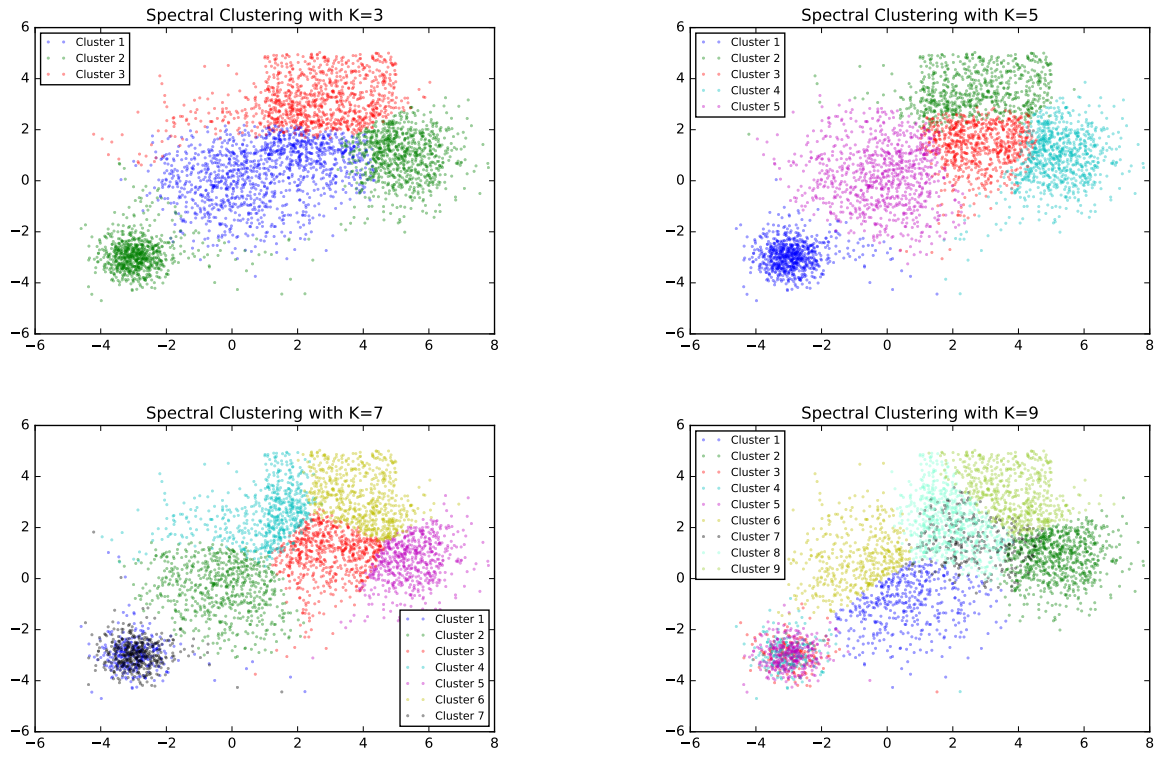



Figure 12: Clustering Result for bigClusteringData.txt with Spectral Clustering

3. The Python code used for Spectral Clustering Algorithm is shown in Listing 4.

```
import numpy as np
import time
# from k_centers import kCenters
from k_means import kMeans


def spectralClustering(X, K, random_state=None, verbose=True):
    """ function to implement the spectral clustering algorithm """
    t0 = time.time()

    N, d = X.shape
    W = np.zeros((N, N))  # adjacency matrix W
    for i in range(N):
        distance = np.sqrt(np.sum((X - X[i, :])**2, axis=1))
        W[:, i] = distance

    diag = np.sum(W, axis=1)
    D = np.diag(diag)  # diagnoal matrix D
    L = D - W  # Laplacian matrix L
    L = np.identity(N) - np.dot(np.linalg.inv(D), W)
    eigvals, U = np.linalg.eigh(L)

    U = U[:, -K:]  # first K eigenvectors

    ## call k-centers for clustering
    # _, C, _, idx = kCenters(U, K, random_state=random_state, verbose=
```

```
    False)
    #
    # Q = X[idx, :]
    # loss = np.zeros((N, K))
    # for i in range(K):
    #     loss[:, i] = np.sqrt(np.sum((X - Q[i, :])**2, axis=1))
    # D = np.max(np.min(loss, axis=1))
    #
    # if verbose is True:
    #     t = np.round(time.time() - t0, 4)
    #     print('Spectral Clustering finished in ' + str(t) + 's')
    #
    # return W, U, Q, C, D

    # call k-means for clustering
    Q, C, D = kMeans(U, K, tol=0.00001, random_state=random_state,
                     verbose=False)

    if verbose is True:
        t = np.round(time.time() - t0, 4)
        print('Spectral Clustering finished in ' + str(t) + 's')

    return W, U, Q, C, D
```

Listing 4: Spectral Clustering Algorithm Python Code

4. Different Initialization Comparison.

   In Spectral Clustering algorithm, we run 50 times with different initialization, the best result of D is shown in Table 11.

Table 11: Best Result (D) for Spectral Clustering Algorithm

| Data / K | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| clustering | 5.4780 | 4.8812 | 4.6889 | 3.7015 | 3.3656 | 3.3084 | 3.4417 | 3.1799 | 3.3345 |
| bigClusteringData | N.A. | 5.0057 | 4.7021 | 4.2090 | 4.6857 | 3.6327 | 3.3554 | 3.5176 | 3.7858 |

5. Cluster Index Comparison.

   In Spectral Clustering algorithm, the cluster index set C (first 20) for the best result is shown in Table 12 and Table 13.

Table 12: Index (first 20) for Spectral Clustering Algorithm (clustering.txt)

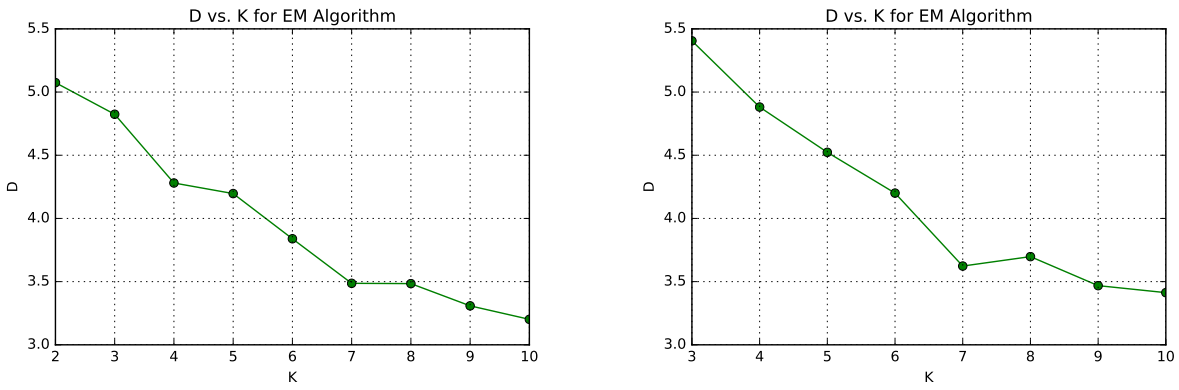
| K | Index |
|---|---|
| 2 | 1. 1. 0. 1. 0. 0. 0. 1. 0. 1. 0. 1. 0. 0. 1. 0. 0. 1. 1. 0. |
| 3 | 1. 1. 1. 1. 1. 1. 1. 1. 0. 1. 1. 0. 1. 0. 1. 1. 1. 1. 1. 0. |
| 4 | 3. 3. 3. 3. 3. 2. 2. 3. 2. 2. 2. 2. 0. 0. 3. 2. 2. 3. 2. 2. |
| 5 | 0. 2. 2. 0. 1. 1. 1. 0. 1. 0. 1. 0. 1. 1. 2. 1. 1. 2. 1. 1. |
| 6 | 0. 0. 0. 0. 4. 0. 0. 2. 4. 0. 4. 1. 4. 4. 2. 4. 4. 2. 0. 1. |
| 7 | 1. 6. 6. 6. 4. 6. 6. 1. 4. 6. 4. 1. 4. 4. 1. 4. 4. 1. 6. 6. |
| 8 | 5. 2. 2. 5. 2. 5. 5. 4. 1. 5. 1. 5. 1. 1. 4. 1. 1. 4. 5. 1. |
| 9 | 4. 5. 5. 5. 2. 2. 2. 4. 3. 4. 2. 4. 0. 3. 4. 2. 2. 4. 5. 2. |
| 10 | 3. 5. 5. 5. 4. 5. 5. 0. 4. 5. 4. 3. 4. 4. 0. 4. 4. 3. 5. 4. |

Table 13: Index (first 20) for Spectral Clustering Algorithm (bigClusteringData.txt)

| K | Index |
|---|---|
| 3 | 0. 2. 0. 2. 2. 0. 2. 0. 2. 0. 2. 2. 0. 0. 2. 2. 1. 2. 1. 2. |
| 4 | 3. 1. 2. 1. 1. 2. 1. 2. 1. 2. 1. 2. 3. 3. 1. 1. 2. 1. 2. 1. |
| 5 | 0. 1. 3. 4. 4. 3. 4. 3. 4. 3. 1. 4. 0. 0. 4. 4. 1. 1. 1. 4. |
| 6 | 4. 4. 1. 5. 5. 1. 4. 1. 4. 1. 4. 5. 2. 5. 4. 5. 2. 4. 2. 5. |
| 7 | 3. 1. 5. 4. 4. 5. 1. 5. 1. 5. 1. 4. 3. 3. 1. 4. 2. 1. 2. 4. |
| 8 | 2. 4. 7. 4. 4. 7. 1. 7. 4. 7. 4. 0. 5. 2. 4. 1. 5. 4. 4. 1. |
| 9 | 5. 4. 1. 4. 8. 1. 0. 1. 4. 1. 4. 8. 3. 5. 4. 0. 1. 4. 1. 0. |
| 10 | 2. 1. 4. 1. 3. 4. 7. 4. 1. 4. 1. 3. 6. 2. 1. 7. 6. 1. 8. 7. |

**(V).** Expectation Maximization (EM) Algorithm

1. In this part, assuming we are dealing with the Gaussian Mixture Model (GMM), then with the EM algorithm, we choose the best distortion D as the the objection function. The change of D versus cluster number K is shown in Fig. 13, where the result for clustering.txt is shown in Fig. 13a and the result for bigClusteringData.txt is show in Fig. 13b.




(a) clustering.txt (b) bigClusteringData.txt

Figure 13: Change of Distoration versus Cluster Number K for EM Algorithm

2. The scatter plot of the clustering result for clustering.txt is shown in Fig. 14 and the scatter plot of the clustering result for bigClusteringData.txt is shown in Fig. 15. The cluster centroids are clearly marked and different clusters are denoted by different colors.

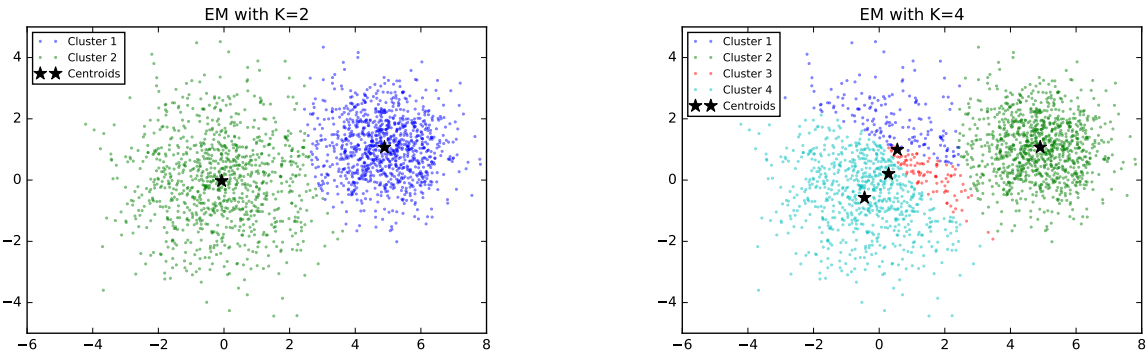



Figure 14: Clustering Result for clustering.txt with EM Algorithm

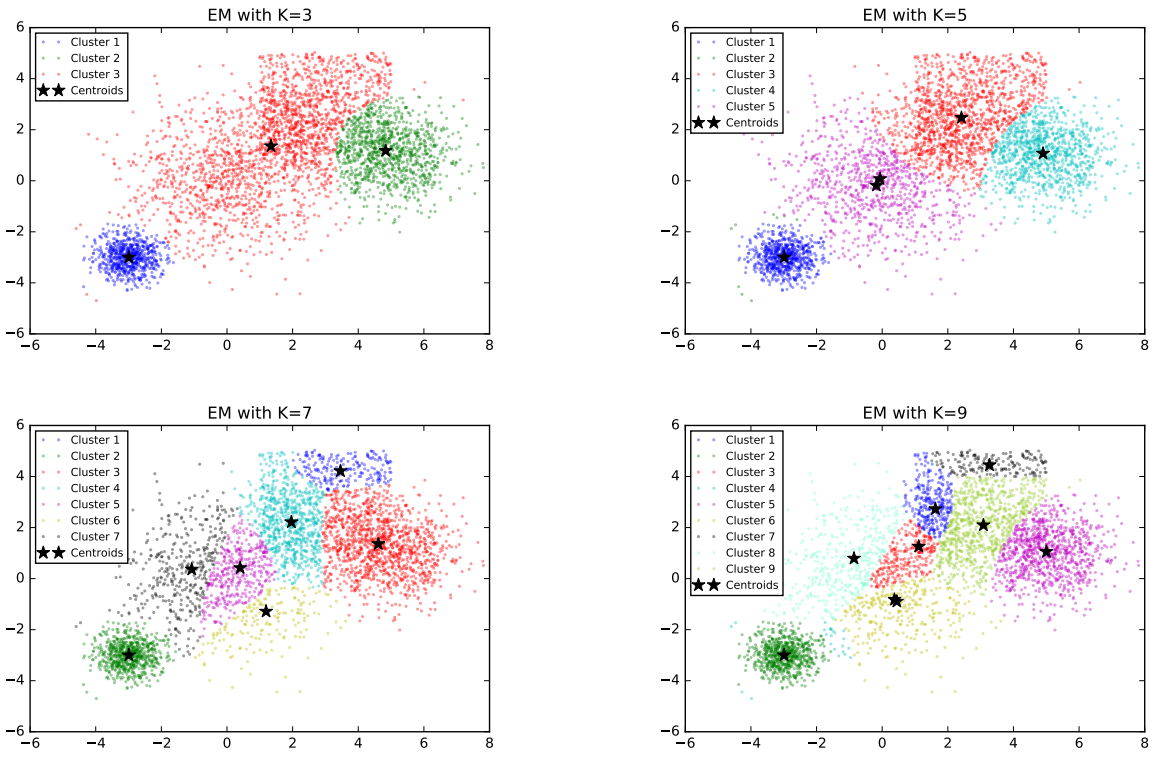



Figure 15: Clustering Result for bigClusteringData.txt with EM Algorithm

3. The Python code used for EM Algorithm is shown in Listing 5.

```
import numpy as np
import time

class EM(object):
    """ self-defined calss for EM algorithm """
    def __init__(self, m, threshold=0.01, random_state=None, maxIter=500):
        """ initialize the EM algorithm """
        self.m = m
        self.threshold = threshold
        self.random_state = random_state
        self.maxIter = maxIter
        self.w = None
        self.gamma = None
        self.mu = None
        self.sigma = None
        self.gaussianProb = None
        self.logLikelihood = None
        self.distance = None
        self.D = None

    def train(self, x, verbose=False):
        """ function to perform EM algorithm on X """
        t0 = time.time()
        np.random.seed(self.random_state)

        # initialize the mean and covariance matrix
```

```
        self.initialize(x)

        # iterate through E and M steps
        for i in range(1, self.maxIter + 1):
            self.estep(x)
            self.mstep(x)
            if abs(self.logLikelihood[-1] - self.logLikelihood[-2]) \
               / abs(self.logLikelihood[-2]) < self.threshold:
                for i in range(self.m):
                    self.distance[:, i] = np.sqrt(np.sum((x - self.mu[i])
    **2,
                                                         axis=1))
                self.D = np.max(np.min(self.distance, axis=1))
                if verbose is True:
                    t = np.round(time.time() - t0, 4)
                    print('Reach threshold at', i,
                          'th iters in ' + str(t) + 's')
                return

        for i in range(self.m):
            self.distance[:, i] = np.sqrt(np.sum((x - self.mu[i])**2, axis
    =1))
        self.D = np.max(np.min(self.distance, axis=1))
        if verbose is True:
            t = np.round(time.time() - t0, 4)
            print('Stopped, reach the maximum iteration ' + str(t) + 's')

    def initialize(self, x):
        """ function to initialize the parameters """
        n, dim = x.shape  # find the dimensions
        self.distance = np.zeros((n, self.m))
        self.w = np.ones(self.m) * (1 / self.m)
        self.gamma = np.zeros((n, self.m))
        self.gaussianProb = np.zeros((n, self.m))
        self.mu = [None] * self.m
        self.sigma = [None] * self.m
        self.logLikelihood = []

        cov = np.cov(x.T)
        mean = np.mean(x, axis=0)
        for k in range(self.m):
            self.mu[k] = mean + np.random.uniform(-0.5, 0.5, dim)
            self.sigma[k] = cov

        # update gamma
        self.gamma = self.gammaprob(x, self.w, self.mu, self.sigma)

        # calculate the expectation of log-likelihood
        self.logLikelihood.append(self.likelihood())

    def estep(self, x):
        """ function to conduct E-Step for EM algorithm """
        self.gamma = self.gammaprob(x, self.w, self.mu, self.sigma)

    def mstep(self, x):
        """ function to conduct M-Step for EM algorithm """
        n, dim = x.shape
        sumGamma = np.sum(self.gamma, axis=0)
        self.w = sumGamma / n
```

```
84
85          for k in range(self.m):
86              self.mu[k] = np.sum(x.T * self.gamma[:, k], axis=1) / sumGamma[
    k]
87              diff = x - self.mu[k]
88              weightedDiff = diff.T * self.gamma[:, k]
89              self.sigma[k] = np.dot(weightedDiff, diff) / sumGamma[k]
90              if np.linalg.matrix_rank(self.sigma[k]) != 3:
91                  randv = np.random.random(dim) / 10000
92                  self.sigma[k] = self.sigma[k] + np.diag(randv)
93
94          # calculate the expectation of log-likelihood
95          self.logLikelihood.append(self.likelihood())
96
97      def gammaprob(self, x, w, mu, sigma):
98          """ function to calculate the gamma probability """
99          for k in range(self.m):
100             self.gaussianProb[:, k] = self.gaussian(x, mu[k], sigma[k])
101
102         weightedSum = np.sum(w * self.gaussianProb, axis=1)
103         gamma = ((w * self.gaussianProb).T / weightedSum).T
104
105         return gamma
106
107     def gaussian(self, x, mu, sigma):
108         """ function to calculate the multivariate gaussian probability """
109         inversion = np.linalg.inv(sigma)
110         part1 = (-0.5 * np.sum(np.dot(x - mu, inversion) * (x - mu), axis
    =1))
111         part2 = 1 / ((2 * np.pi) ** (len(mu) / 2) *
112                      (np.linalg.det(sigma) ** 0.5))
113
114         pdf = part2 * np.exp(part1)
115
116         return pdf
117
118     def likelihood(self):
119         """ function to calculate the log likelihood """
120         log = np.log(np.sum(self.w * self.gaussianProb, axis=1))
121         logLikelihood = np.sum(log)
122
123         return logLikelihood
124
125     def get_label(self):
126         """ function to predict the classes using calculated parameters """
127         label = np.argmax(self.w * self.gaussianProb, axis=1)
128
129         return label
```

Listing 5: EM Algorithm Python Code

4. Different Initialization Comparison.

   In EM algorithm, we run 50 times with different initialization, the best result of D is shown in Table 14.

Table 14: Best Result (D) for EM Algorithm

| Data / K | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| clustering | 5.0744 | 4.8260 | 4.2467 | 4.1715 | 3.8116 | 3.7734 | 3.4768 | 3.4228 | 3.2690 |
| bigClusteringData | N.A. | 5.4054 | 4.8813 | 4.5730 | 4.0522 | 3.8400 | 3.5024 | 3.4828 | 3.4400 |

5. Cluster Index Comparison.

   In EM algorithm, the cluster index set C (first 20) for the best result is shown in Table 15 and Table 16.

Table 15: Index (first 20) for EM Algorithm (clustering.txt)

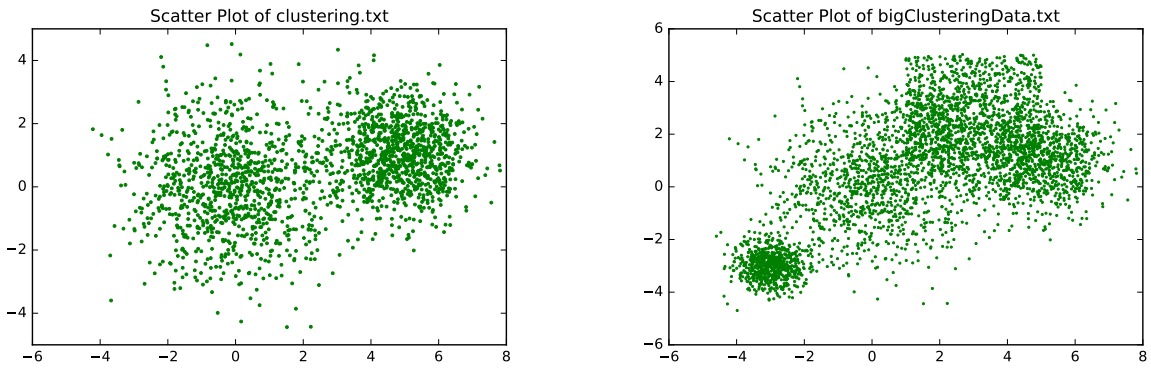
| K | Index |
|---|---|
| 2 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 |
| 3 | 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 |
| 4 | 0 3 3 0 1 3 3 0 1 3 1 0 1 1 0 1 1 0 3 3 |
| 5 | 3 0 0 3 0 1 1 3 0 1 1 3 0 4 3 0 0 4 0 1 |
| 6 | 2 0 0 0 1 2 2 5 3 2 2 2 1 3 5 2 2 3 2 2 |
| 7 | 4 4 4 4 2 0 4 3 0 4 0 3 2 0 3 0 0 3 4 0 |
| 8 | 6 6 5 6 5 6 6 0 6 6 6 6 3 4 0 6 6 0 6 6 |
| 9 | 3 3 7 3 7 3 3 0 2 3 3 3 6 2 0 3 3 0 3 3 |
| 10 | 6 6 3 6 9 8 8 6 8 6 8 3 0 8 4 8 9 7 6 8 |

Table 16: Index (first 20) for EM Algorithm (bigClusteringData.txt)

| K | Index |
|---|---|
| 3 | 0 1 2 1 1 2 1 2 1 2 1 1 0 0 1 1 1 1 1 1 |
| 4 | 2 1 3 0 0 3 0 3 1 3 1 0 2 2 1 0 1 1 1 0 |
| 5 | 0 4 1 4 4 1 3 1 4 1 4 2 0 0 4 2 4 4 4 2 |
| 6 | 2 1 3 1 1 3 5 3 1 3 1 0 2 2 1 5 4 1 4 1 |
| 7 | 0 3 2 6 6 2 4 2 6 2 3 1 0 0 3 4 0 3 3 6 |
| 8 | 7 5 0 2 2 0 4 0 5 0 5 6 7 7 5 4 7 5 5 2 |
| 9 | 2 1 8 7 7 8 1 8 1 8 1 3 2 2 1 7 2 1 1 7 |
| 10 | 4 5 1 2 2 1 9 1 5 1 5 7 4 4 5 9 8 5 5 2 |

## 2 Natural Clusters Discussion

The distribution of clustering.txt and bigClusteringData.txt are shown in Fig. 16a and Fig. 16b.




(a) clustering.txt

(b) bigClusteringData.txt

Figure 16: Scatter Plot of Original Data

From Fig. 16a and Fig. 16b, the natural clusters is: 2 clusters for clustering.txt and 4 clusters for bigClusteringData.txt. This is determined by personal observation.

# 3 K-Means Convergence Discussion

In Lloyd's algorithm, we choose

$$D = \max_{x_i \in X}(\min_{c_j \in Q}||x_i - c_j||_2)$$

as the cost, the stop criterion is:

$$|D^{p+1} - D^p| < tol$$

where tol is user-defined.

Reason: as the clustered data get close to the local optimum, the cost D will decrease slower and slower. Finally, it will reach the local minimum. So, the difference between new cost and previous cost should be smaller and smaller. So, $|D^{p+1} - D^p| < tol$ can be used as the stop criterion.

# 4 Computational Effort Discussion

In this part, we run those 5 algorithms on the same dataset, and report the total running time, which is shown in Table 17 and Table 18.

Table 17: Computation Time Comparison for clustering.txt

| K | K-Means | K-Centers | Single-Swap | Spectral Clustering | EM |
|---|---|---|---|---|---|
| 3 | 0.2673 | 0.0067 | 0.1582 | 1.5716 | 0.4412 |
| 4 | 0.2543 | 0.0010 | 0.1971 | 1.4587 | 0.5078 |
| 5 | 0.3624 | 0.0012 | 0.1895 | 1.4132 | 0.4881 |
| 6 | 0.4325 | 0.0013 | 0.2668 | 1.4840 | 0.7462 |
| 7 | 0.2837 | 0.0015 | 0.2693 | 1.4877 | 0.8227 |
| 8 | 0.1923 | 0.0019 | 0.3306 | 1.4708 | 0.9216 |
| 9 | 0.3481 | 0.0024 | 0.3012 | 1.4669 | 1.0106 |
| 10 | 0.1989 | 0.0026 | 0.3790 | 1.5053 | 1.1277 |

Note: all measured time is in unit of seconds (s)

Table 18: Computation Time Comparison for bigClusteringData.txt

| K | K-Means | K-Centers | Single-Swap | Spectral Clustering | EM |
|---|---|---|---|---|---|
| 3 | 0.4725 | 0.0020 | 0.4511 | 11.1860 | 0.3412 |
| 4 | 0.8326 | 0.0019 | 0.5692 | 10.2542 | 0.2553 |
| 5 | 0.4722 | 0.0021 | 0.6445 | 10.5166 | 0.5763 |
| 6 | 1.2580 | 0.0024 | 0.8809 | 10.2167 | 0.6196 |
| 7 | 0.9619 | 0.0025 | 0.7638 | 9.8986 | 1.1591 |
| 8 | 0.7767 | 0.0034 | 0.7815 | 9.4616 | 1.3177 |
| 9 | 0.7177 | 0.0037 | 0.9010 | 9.8733 | 1.4977 |
| 10 | 1.2369 | 0.0041 | 1.0744 | 9.8636 | 1.7560 |

Note: all measured time is in unit of seconds (s)

From Table 17 and Table 18, the K-Centers algorithm is the fastest and Spectral Clustering is the lowest. Fero the other algorithms, they are pretty similar. This is consistent with my analysis. For spectral clustering, since we need to compute the eigenvector, it will cost more time. For K-Centers algorithm, since we just just greedy methods to find k centers, it will be much faster. For the other method, their difference is not very clear since the dataset is not too big.

## 5 Single-Swap Algorithm Discussion

The final cost for Greedy K-Centers algorithm and Single-Swap algorithm are shown in Table 19 and Table 20. With Single-Swap, the improvement of cost is clear.

Table 19: Best Result (D) for K-Centers Algorithm

| Data / K | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| clustering | 5.2838 | 4.8162 | 4.1387 | 3.7503 | 3.4691 | 3.1323 | 2.7510 | 2.7206 | 2.4634 |
| bigClusteringData | N.A. | 5.5288 | 4.1446 | 3.8318 | 3.3765 | 3.2933 | 3.0093 | 2.7431 | 2.6249 |

Table 20: Best Result (D) for Single-Swap Algorithm

| Data / K | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|
| clustering | 4.6305 | 3.9619 | 3.7152 | 3.3236 | 3.0420 | 2.8479 | 2.5808 | 2.3493 | 2.3493 |
| bigClusteringData | N.A. | 4.3388 | 3.8504 | 3.4619 | 3.2156 | 2.9379 | 2.8479 | 2.6053 | 2.4973 |

To perform single-swap, we use all the points as the candidates.

In current version of Single-Swap algorithm, we use iterations to iterate all samples and all centers, this is not efficient. For better performance, for each swap, we can find the best candidate and the center that has largest radius, then we just need to swap them.

# 6 Summary

In this report, we compare the performance of different algorithms, among all these five algorithms, K-Centers algorithm performs the fastest and Spectral Clustering performs the lowest. K-Means, Single-Swap and EM algorithm is between them.

Among those five algorithms, EM algorithm performs well when the data can be modeled by GMMs. K-Means also performs well but may need a lot of different initializations. Greedy K-Centers algorithm performs the worst since it just find K centers in greedy way.

Using $D = \max_{x_i \in X}(\min_{c_j \in Q}||x_i - c_j||_2)$ as the cost, we can find that with more clusters, $D$ will be smaller.