Comparative Analysis of Two Clustering Algorithms:

K-means and FSDP (Fast Search and Find of Density Peaks)

A Thesis

Presented to

The Faculty of the Department of Computer Science

San Jose State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

By

Li Miao

Aug 2015

The Designated Thesis Committee Approves the Thesis Titled

Comparative Analysis of Two Clustering Algorithms:

K-means and FSDP (Fast Search and Find of Density Peaks)

by

Li Miao

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE

SAN JOSE STATE UNIVERSITY

Aug 2015

Dr. Sami Khuri Department of Computer Science

Dr. Teng Moh Department of Computer Science

Dr. Chris Pollett Department of Computer Science

ABSTRACT

Comparative Analysis of Two Clustering Algorithms:

K-means and FSDP (Fast Search and Find of Density Peaks)

by

Li Miao

With the overwhelming amount of data pouring into our lives, obtaining meaningful information from them is becoming a must task for people. How can people mine for "gold" in this area? Or, what tools can they use to do that? It has been proved that clustering is one of the best tools.

In this project, two clustering algorithms are studied and numerically compared with various data sets. The first one is the K-means clustering which starts with initial roughly-guessed clusters, tries to classify some data points into one cluster, and iteratively repeats until converges. The second algorithm is called Fast Search and Find of Density Peaks (FSDP), which is able to automatically detect the correct number of clusters according to the inherent property of its decision graph. It is based on the following assumptions: 1) the

cluster centers have higher density than their neighbor data points; 2) the distance between the cluster center and any data points with higher local density is relatively large. Its decision graph is a graphic and intuitive expression for the clustering. One may get more or fewer clusters, by setting smaller or larger thresholds.

The two algorithms are described in the following chapters. They are implemented in Java. To compare how well they perform on four milestone data sets, we use two metrics: Entropy and Purity. The results demonstrate that K-means clustering is faster, while the FSDP is more accurate.

Keywords: Clustering, K-means, Fast Search and Find of Density Peaks, Decision Graph, Entropy, Purity.

ACKNOWLEDGEMENTS

I would like to offer my sincere thanks to my advisor Dr. Sami Khuri. It is my pleasure to have access to his guidance. And thanks to Dr Natalia Khuri for her invaluable insights and discussions during this project. I would also like to thank my committee members Dr. Teng Moh and Dr. Chris Pollett for their invaluable supports. Also many thanks to our University, who has my beautiful memory.

I would also like to thank my family, my husband, daughter and son, for their generous supports.

TABLE OF CONTENTS

List of Figures

List of Tables

Chapter 1. Introduction

1.1 What is Clustering

Clustering, or cluster analysis, is to group, classify or categorize a set of objects into many subsets, called clusters, in such a way that the items inside one subset are more "similar" to each other, while "dissimilar" to items inside other subsets. Therefore there must be a way to distinguish between "similar" and "dissimilar" items. There are many measurements, or metrics to calculate the similarity. However, nobody knows which one is best. The research about measurement is one of the challenges. Distance is one of them. The most frequent used distance is the Euclidean distance [9]. The Euclidean distance between points x and y is the length of the line segment connecting them. In Cartesian coordinates, for two points $x = (x_1, x_2, ..., x_p)$ and $y = (y_1, y_2, ..., y_p)$, their Euclidean distance is given by $d(x,y) = \sqrt{\Sigma_{i=1}^{p}|x_i - y_i|^2}$, where p is called the dimension of a data point. Smaller distance means more similar. For example, there are 3 points $A(x_1, y_1)$, $B(x_2, y_2)$, and $C(x_3, y_3)$, if $d(A, B) < d(A, C)$, then the similarity of A and B is bigger than A and C. Clustering belongs to unsupervised learning problem. All the items do not have labels, and there is no prior knowledge on what the clusters look like.

1.2 Criteria to evaluate clustering algorithms

There are many clustering algorithms to choose from. Choosing a suitable algorithm and a suitable measurement depends on the clustering task and the features of

clustering items. Generally speaking, *internal criterion* and *external criterion* are two conventional measurements.

Internal criterion: Based on the data's intrinsic information alone.

***External criterion*:** Based on pre-known knowledge about the data.

In this article, we mainly use two external criteria, namely, purity and entropy, to evaluate the performance of the discussed clustering algorithms.

1.3 *Organizations*

The rest of this work is organized in the following fashion. Chapter two introduces the K-means algorithm. Chapter three is about FSDP clustering algorithm. Chapter four numerically compares the experiment results of the two clustering algorithms on four typical data sets. Both advantages and disadvantages of the two algorithms are briefly discussed. The last chapter gives a summary of the work including future directions.

Chapter 2. K-means clustering algorithm

2.1 Introduction to K-means clustering

K-means is a centroid based clustering algorithm. "K" represents the number of clusters, and it is also an input parameter. Each element in the data set is assigned to a cluster center with the smallest **distance** to it. There are two core steps to achieve the clustering task: 1) find the centroids, which may or may not be very accurate; 2) assign each data element to the cluster based on their distance. These two steps iterate until the algorithm converges to one optimum value(most likely, it is a local optimum).

K-means is a popular algorithm since it is very easy to implement. Provided that there is a suitable distance definition and reasonable setting of K value, it usually achieves reasonable results. Figure 1 [3] depicts the process of the standard K-means clustering algorithm where K = 3.

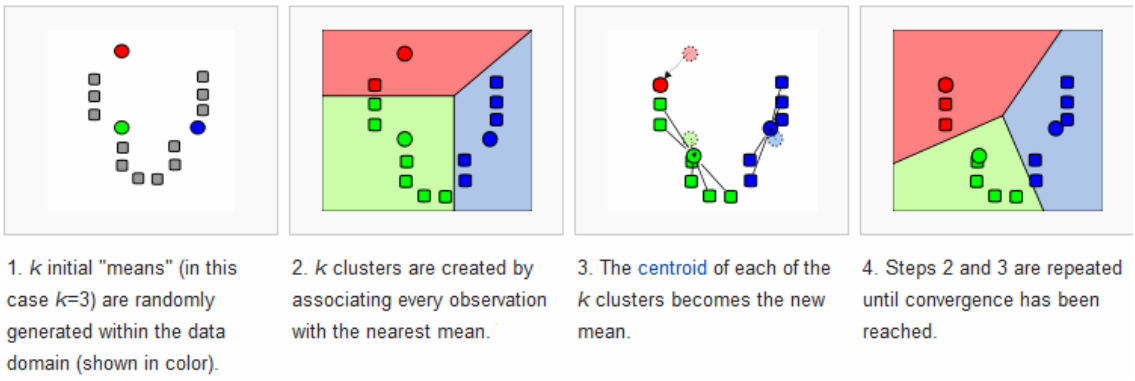

Figure 1 Demonstration of K-means clustering process

From Figure 1, one can see that the K-means algorithm is not too complicated to comprehend. The details of the algorithm are as follows.

2.2 The steps of the K-means algorithm

Given a set of n elements $\{x_1, x_2, ..., x_n\}$, where each element is a d-dimensional vector (d being one integer), our goal is to partition the set into k different clusters or groups.

The steps of the K-means clustering algorithm are:

1. Initialize k centroids $m_1, m_2, ..., m_k$. For simplicity, let's select them randomly.
2. Repeat until all centroids keep constant:

   Assign an element x_i to the cluster S_j, according to

$$S_j = \left\{ x_t : \left\| x_t - m_j \right\|^2 \leq \left\| x_t - m_t \right\|^2, \ 1 \leq t \leq k \right\}$$

   Update the center of S_j, namely, m_j.

Figure 2 is the flowchart of K-means clustering algorithm.

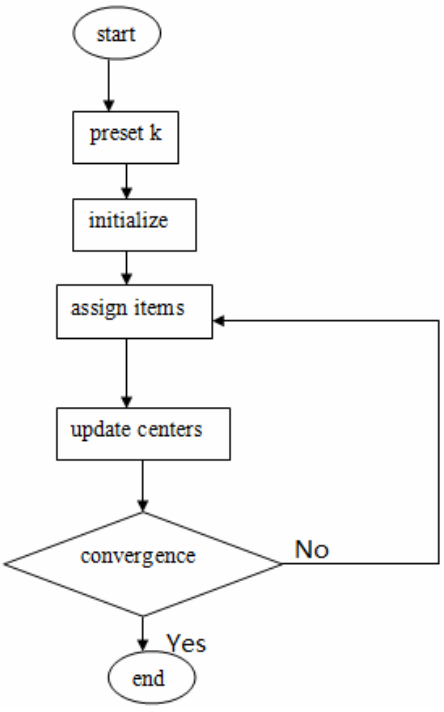

Figure 2 Flowchart of K-means clustering algorithm

2.3 Implementation

We use one small data set to show the steps of the K-means clustering algorithm.

Example 1: Given 8 data points: (1, 1.5), (1.2, 1.3), (1.5, 1.8), (5.0, 5.01), (5.1, 4.7), (5.15, 5.3), (5.75, 5.2), (2.0, 6.0). Use K-means clustering algorithm to partition the set into 2 clusters. Table 1 is the initial data set.

Table 1 Data set to be clustered

Object Number	A	B
0	1	1.5
1	1.2	1.3
2	1.5	1.8
3	5.0	5.01
4	5.1	4.7
5	5.15	5.3
6	5.75	5.2
7	2.0	6.0

The following two subsections **2.3.1** and **2.3.2** state the process step by step.

2.3.1 Iteration 1

Initialize cluster centers: $m_1 = (1.5, 1.8)$, $m_2 = (5.0, 5.01)$.

Table 2 Initial K-means cluster centers

Cluster	Object Number	Centers
Cluster 1	2	m_1: (1.5, 1.8)
Cluster 2	3	m_2: (5.0, 5.01)

Compute the Euclidean distance between all the objects and centroids. The results are given in Table 3.

Table 3 Euclidean distance from each object to centroids

Object Number	Distance to m_1	Distance to m_2
0	**0.583095**	5.321663
1	**0.583095**	5.310753
2	**0.000000**	4.749116
3	4.749116	**0.000000**
4	4.622770	**0.325730**
5	5.056926	**0.326497**
6	5.442656	**0.773692**
7	4.229657	**3.159130**

Each object is assigned to the closest cluster. Then, the set is divided into 2 clusters: cluster 1 = {0, 1, 2} and cluster 2 = {3, 4, 5, 6, 7}.

Update the two centroids m_1 and m_2:

Calculate the average of all the data points in cluster 1 to get updated m_1:

$m_1 = ((1 + 1.2 + 1.5), (1.5 + 1.3 + 1.8))/3 = (1.23333, 1.53333)$

Calculate the average of all the data points in cluster 2 to get updated m_2:

$m_2 = ((5 + 5.1 + 5.15 + 5.75 + 2), (5.01 + 4.7 + 5.3 + 5.2 + 6))/5 = (4.6, 5.242)$

2.3.2 Iteration 2

Use the updated centroids to compute the distances once more. Table 4 shows the results.

Table 4 Euclidean distances from each element to updated cluster centers

Object Number	Distance to *m1*	Distance to *m2*
0	**0.23570**	5.19255
1	**0.23570**	5.20570
2	**0.37712**	4.63221

3	5.12591	**0.46241**
4	4.99789	**0.73740**
5	5.43397	**0.55305**
6	5.81762	**1.15077**
7	4.53199	**2.70824**

Then, we put each object into the closest cluster. The set is divided into the new clusters: cluster 1 = {0, 1, 2}, and cluster 2 = {3, 4, 5, 6, 7}.

From the generated clusters of iteration 1 and iteration 2, we see that iteration 2 generates exactly the same clusters as iteration 1. Therefore the process converges and ends.

Figure 3 visualizes the result.

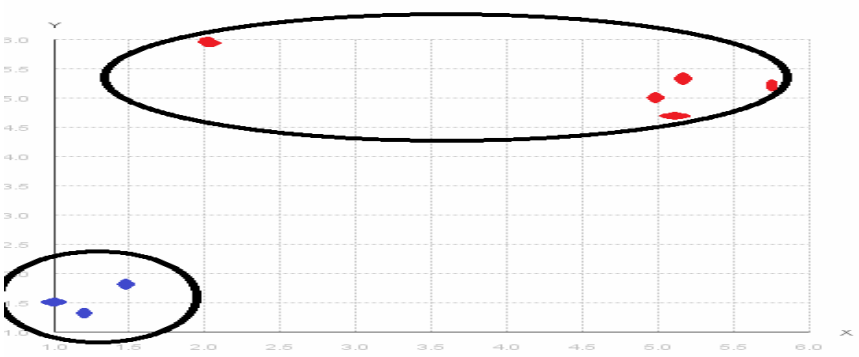

Figure 3 K-means clustering results for data set in Table 1

We can see from Figure 3 that the three blue points are relatively close to each other; similarly for the five red points. The clustering results are expected. K-means clustering works well for this set.

2.4 K-means clustering limitations

K-means clustering works well for the data set in Example 1. However, it does not do so well for all kinds of data sets. Generally, K-means clustering has some limitations.

- It highly depends on the initial cluster centroids, especially when sets do not contain many objects.
- K, the number of clusters, has to be set in advance. In fact, finding a suitable K value is one of the challenges of the K-means clustering algorithm.
- Given the same data; it may generate different results after the objects are placed in different orders. This is a disadvantage that might be amplified as the number of data points increases.
- K-means clustering algorithm cannot detect non-spherical clusters. We shall see the disadvantage of K-means clustering algorithm in Figure 9 in chapter 3.

In the next chapter, we introduce and discuss the various steps of the Fast Search and Find of Density Peaks Clustering algorithm.

Chapter 3. Fast Search and Find of Density Peaks (FSDP) Clustering

This is a very new clustering method, which was first published on *Science* in June 2014 [1]. It uses one kernel function to compute each point's **local density** and the **minimum distance** between it and other data points. We use $\rho(i)$ and $\delta(i)$ to represent the two quantities of any data point *i*. If the data set contains a few data points, it is better to use exponential kernel as described in Cheng [10]. Otherwise, use cut-off kernel. The algorithm's basic idea is that the cluster center is surrounded by points with lower density, and is far away from other points with higher density. The details of the algorithm are discussed in the next actions of this chapters.

3.1 Introduction

In addition to the basic idea, FSDP has another excellent feature, namely decision graph. Decision graph offers a visual foundation of this algorithm. One can figure out the number of the most reasonable partitions of one data set by using its related decision graph. An example of a decision graph is given in Figure 4. The horizontal axis is the set of local density ρ , and the vertical axis the set of the minimum distance δ. From Figure 4, we see 5 colored points. It is reasonable to partition this data set into 5 groups and the 5 colored points are the centers of these groups.

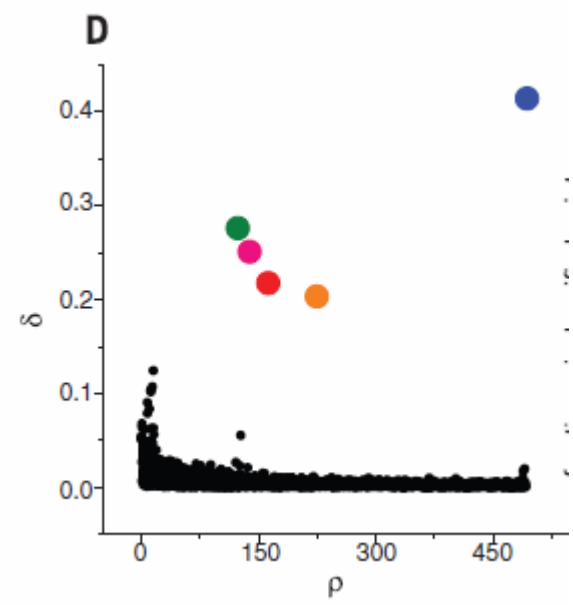

Figure 4 A decision graph

3.2 The FSDP Algorithm

Prerequisites: *dc*, $\rho_{\min}$, δ_{min}. Exponential kernel, Cut-off kernel.

dc is a pre-set numerical value, which controls how many neighbors one item can have , and also controls whether a data point belongs to a halo (to be explained in section 3.2.3) or not. It can be experimentally shown that the choice of *dc* will affect the clustering results. The rule of thumb is to set its value to allow any data point have 1% - 2% of the total number of points as neighbors.

$\rho_{\min}$ and δ_{min} are threshold values for determining clustering centers, or determining the number of clusters. If the density ρ of one point is not less than $\rho_{\min}$, and the δ of the point is not less than δ_{min}, the point is one of the centers. Setting these values requires help from decision graph.

Exponential Kernel: $f(x,m)=e^{\frac{-x^2}{m^2}}$, m $\neq 0$.

Cut-off Kernel: $\chi(x)=\begin{cases}1, x<0\\0, otherwise\end{cases}$.

3.2.1 Step 1: find centers

The first step is to find the cluster centers. To do this, we have to compute some numerical values as follows.

Suppose one data set is *S*. For each point $i \in S$, compute $\rho(i)$ and $\delta(i)$.

1. Compute density $\rho(i)$:

   For sets with few points, use Exponential Kernel.

$$\rho(i)=\sum_{i\neq j, i\in S} e^{-\frac{d_{ij}^2}{dc^2}}$$

   For sets with more points, use Cut-off Kernel.

$$\rho(i)=\sum_{j\in S, i\neq j}\chi(d_{ij}-d_c)$$

   where d_{ij} is the distance between point *i* and point *j*, and d_c is one preset value.

2. Compute $\delta(i)$:

   - if point *i* has the biggest density,

$$\delta(i)=\max_{j\in S}(d_{ij})$$

   - otherwise,

$$\delta(i)=\min_{j:\rho_j>\rho_i, j\in S}(d_{ij})$$

3. Compute the **nearest neighbor:** *nearest(i)*

If point j satisfies the formula: $d_{ij} = \min_{\substack{\rho(k)>\rho(i) \\ k,j\in S, k\neq i, j\neq i}} d_{ik}$, point j is called the **nearest neighbor** of point i and $nearest(i) = j$.

4. Plot decision graph:

Once ρ and δ are computed, one can plot the decision graph of ρ (as the horizontal axis) and δ (as the vertical axis). Based on the graph, one can set thresholds (ρ_{min}, δ_{min}) to get the set of cluster centers:.

$$centers = \{i \in S \mid \rho_i \geq \rho_{\min}, \delta_i \geq \delta_{\min}\}$$

3.2.2 Step 2: assign data points to different clusters

After the cluster centers are determined, it is straightforward to assign other data points to clusters.

1. Sort points in terms of their density values.
2. Assign data point i to the same cluster as its **nearest neighbor**.

3.2.3 (Optional) Step 3: separate the objects of any cluster into cores and halos

Considering that many data sets may be contaminated by noise, FSDP introduces the concepts of **halo** and **core**. In terms of threshold *dc***,** any cluster is separated into two sets: core and halo.

The steps to find the halo and core are as follows:

1. Find the *boarder region* for each cluster.

If points i and j belong to cluster(i) and cluster(j), and $dist(i, j) < dc$, then i belongs

to the *boarder region* of cluster(*i*), and *j* belongs to the *boarder region* of cluster(*j*).

2. Figure out the *boarder region* ρ_b for each *boarder region*.

The *boarder region* ρ_b is the largest ρ of all the data points in this border region.

3. Separate data points into halo and core based on the *boarder region* ρ_b.

If one data point has bigger density than the *boarder region* ρ_b of its cluster, the data point is assigned into the **core** of the cluster, otherwise it is assigned to the **halo**.

3.2.4 Flowchart of FSDP algorithm

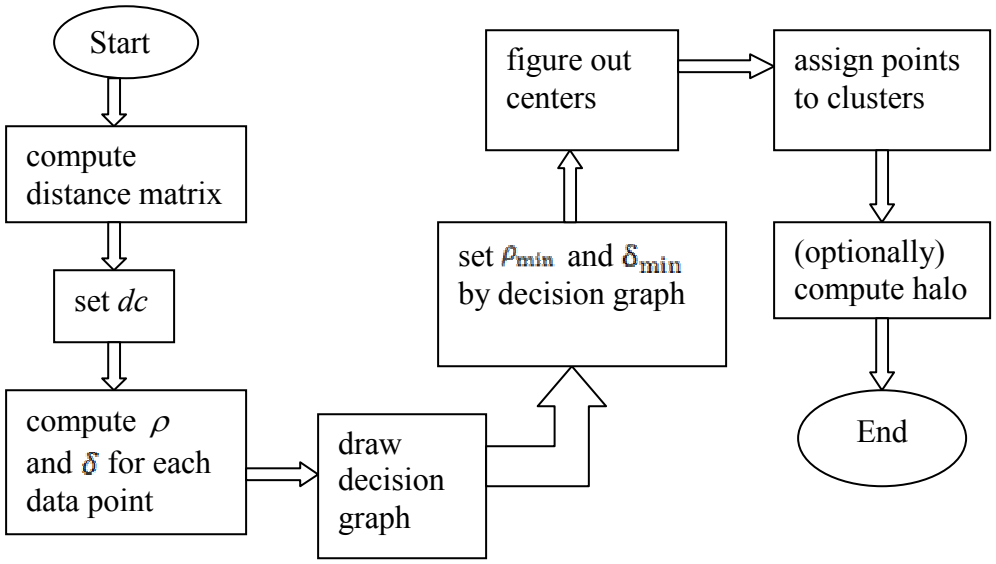

Figure 5 Flowchart of FSDP Algorithm

3.3 Implementation

Example 2: Consider the data set of Table 1.

Step 1. Calculate the distance matrix of this data set (here we use Euclidean distance). It is a symmetric matrix, so only one half of the values need to be computed, and shown in Table 5.

Table 5 Euclidean distance between data points

	0	1	2	3	4	5	6	7
0		0.2828	0.5831	5.3217	5.201	5.6269	6.0210	4.6098
1			0.5831	5.3108	5.174	5.6216	5.9927	4.7676
2				4.7491	4.6228	5.0569	5.4427	4.2297
3					0.3257	0.3265	0.7737	3.1591
4						0.6021	0.8201	3.3615
5							0.6083	3.2268
6								3.8344
7								

Step 2. Set dc = 0.384. We use **Exponential Kernel** to compute ρ and δ .

Table 6 ρ and δ for all data points of Example 2

	0	1	2	3	4	5	6	7
ρ	0	0.5813	0.1993	3.7e-67	0.4870	0.5708	0.1090	4.3e-30
δ	0.2828	6.021	0.5831	0.3257	0.6021	5.6216	0.6083	3.2268

Step 3. Plot decision graph.

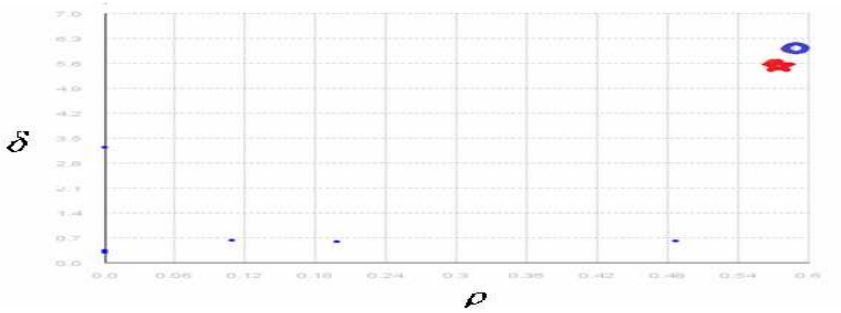

Figure 6 Decision graph for Example 2

In Figure 6, the two points, in the upper right corner, are outliers. Therefore, it is the most reasonable choice to partition this set into 2 clusters.

Step 4. According to Figure 6, set thresholds ρ_{min} to 0.54 and δ_{min} to 4.9.

Step 5. Based on Table 6, ρ_{min}, and δ_{min}, find centers are point 1 and point 5.

Step 6. Group the data set into 2 clusters: {0, 1, 2}, {3, 4, 5, 6, 7}.

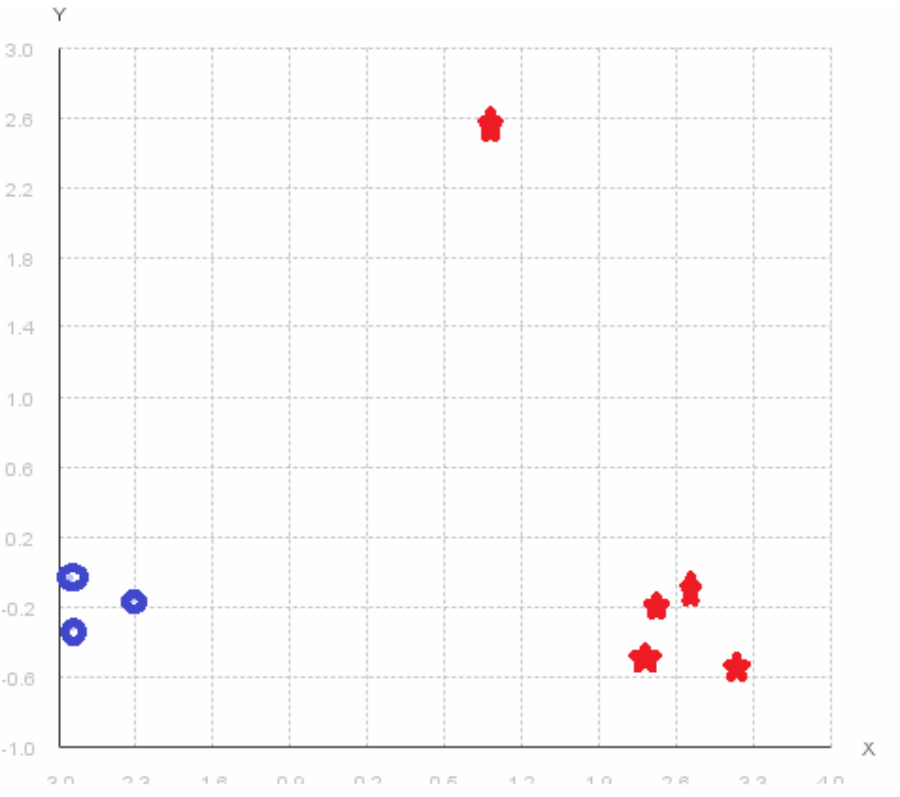

Figure 7 Two clusters for the eight data points

Here we can see, for this data set, the algorithm has the same clustering results as K-means. The advantage of FSDP is that it does not need to iterate and set the initial centers. This is due to the fact that FSDP can give out the number of centers by itself.

In the next chapter, we will compare the two clustering methods by considering various data sets.

Chapter 4. Comparative experiments on typical data sets

In previous chapters, we have discussed the K-means clustering algorithm and FSDP clustering algorithm. To demonstrate their differences, we apply them to four typical data sets. These data sets are the Flame data set, the Aggregation data set, the Iris data set and the Olivetti Face data set [5]. Their basic information is listed in Table 7. The first three data sets are from speech and image processing. The Olivetti Face Data set is from ATT, which is composed of pictures of 40 individuals. Each individual has 10 pictures. The first 100 pictures and the distance matrix [6] are adopted here.

Note that each data set has its definite clustering result, called initial clustering. In what follows, we will apply the two clustering algorithms to each set. Then we compare the generated results with the initial clustering.

Table 7 The properties of the four data sets

Data set name	Data set size	Dimension	Initial clusters
Flame	240	2	2
Iris	150	4	3
Olivetti Face	100 (pictures)	64x64 pixels	(grayscale) 10

4.1 Test with Flame data set

4.1.1 FSDP

For the Flame data set, we preset parameters: dc = 1.8, ρ_{min}= 12 and δ_{min}= 6. The left graph of Figure 8 is its decision graph, and the right graph is the clustering results.

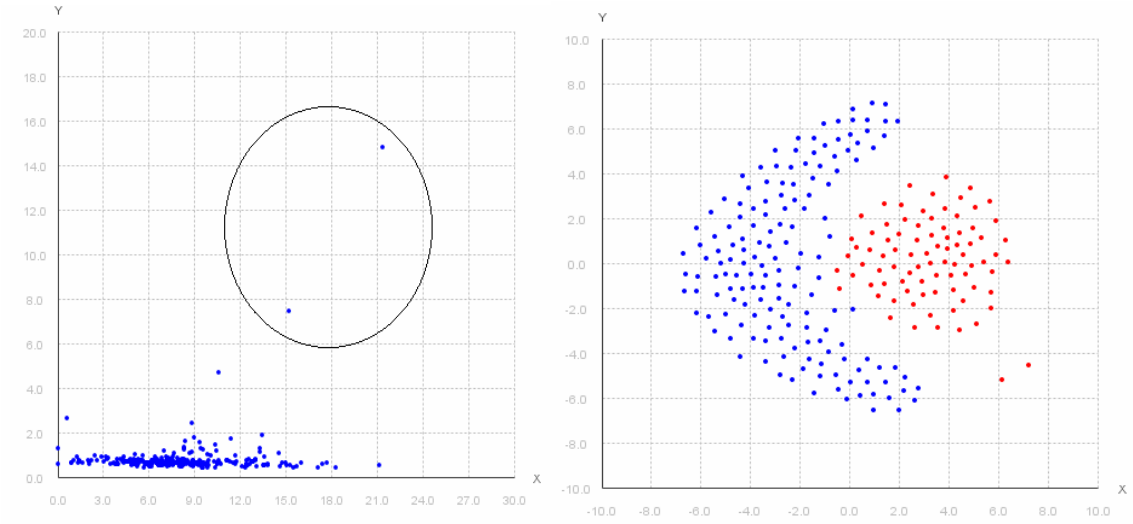

Figure 8 The decision graph (left) and clusters (right) of Flame data set by FSDP

From Figure 8, we can see that the two points in the circle have high ρ and δ values and are far from the other points. They can be set as cluster centers, and thus the two clusters are obtained, as can be seen in different colors in the right graph of Figure 8. Moreover, the data set actually contains two different partitions according to provided initial clusters. This also shows that FSDP can provide the most reasonable partitions.

The centers and data members are given in Table 8.

Table 8 Clustering result of the Flame data set by FSDP

Cluster	Center	Items
0	83	2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28, 29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52, 53,54,55,56,57,58,59,60,61,62,63,64,65,66,67,68,69,70,71,72,73,74,75,76, 77,78,79,80,81,82,83,84,85,86,87,88,89,90,91,92,93,94,95,96,97,98,99, 100,101,102,103,104,105,106,107,108,109,110,111,112,113,114,115,116, 117,118,119,120,121,122,123,124,125,126,127,128,129,130,131,132,133, 134,135,136,137,138,139,140,141,142,143,144,145,146,147,148,149,150, 151,152,153,**157,158**,163
1	236	0,1,154,155,156,159,160,161,162,164,165,166,167,168,169,170,171,172, 173,174,175,176,177,178,179,180,181,182,183,184,185,186,187,188,189, 190,191,192,193,194,195,196,197,198,199,200,201,202,203,204,205,206, 207,208,209,210,211,212,213,214,215,216,217,218,219,220,221,222,223, 224,225,226,227,228,229,230,231,232,233,234,235,236,237,238,239

In Table 8, only two points (number 157 and number 158, in red) were not assigned into their original groups. The running time is 3024 microseconds.

4.1.2 K-means

Let us compare the result to the one obtained by the K-means algorithm. Figure 9 shows its two clusters in blue and red, respectively. It can be observed that many points are assigned to the wrong cluster. Their numbers are listed (in red) in Table 9.

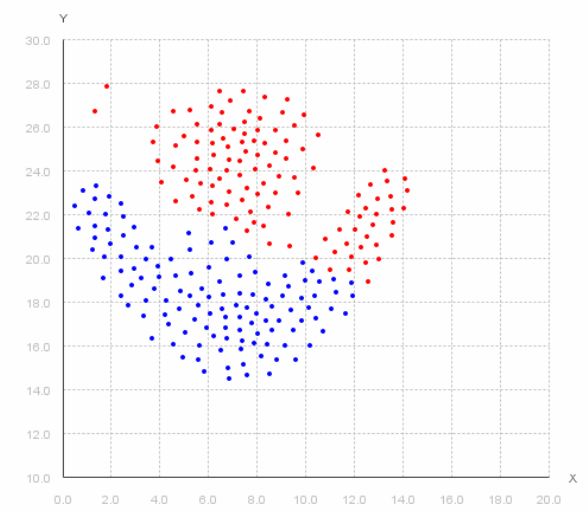

Figure 9 Clustering result of the Flame data set by K-means

This is a case of non-spherical clustering. This case reveals one of the limitations of the K-means clustering.

Table 9 Clustering result of the Flame data set by K-means

Cluster	Center	Items
0	38	32, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152, **155, 157, 158, 159, 174**
1	30	0, 1, **2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 33, 34, 35, 36, 153**, 154, 156, 160, 161, 162, 163, 164, 165, 166, 167, 168, 169, 170, 171, 172, 173, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 205, 206, 207, 208, 209, 210, 211, 212, 213, 214, 215, 216, 217, 218, 219, 220, 221, 222, 223, 224, 225, 226, 227, 228, 229, 230, 231, 232, 233, 234, 235, 236, 237, 238, 239

From Table 9, we see that 41 items are misclassified. The running time is 58.0

microseconds.

4.2 Test with Aggregation data set

The second data set is Aggregation data set.

4.2.1 FSDP

We preset *dc* = 1.5, $\rho_{\min}$= 1, and $\delta_{\min}$= 6. Figure 10 is the decision graph. Figure 11 is the clustering result.

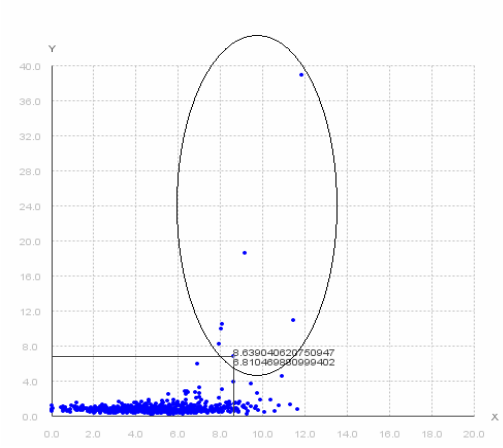

Figure 10 Decision graph of Aggregation data set

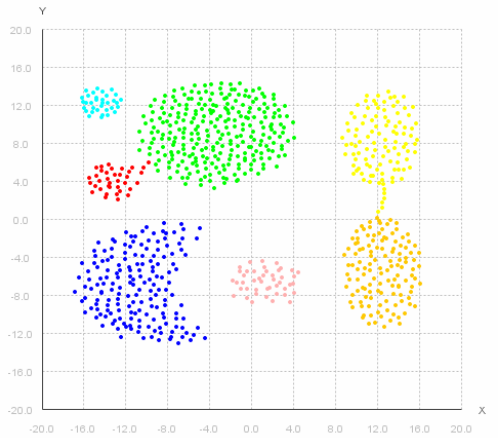

Figure 11 Clustering result of Aggregation data set by FSDP

This example shows FSDP can handle non-spherical data. Table 10 gives the detailed clustering result. However, Table 10 is too big to fit in one page.

Table 10 Clustering result of the Aggregation data set by FSDP

Cluster	Center	Items
0	126	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32, 33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49,50,51,52,53,54,55,56,57,58,59,60,61, 62,63,64,65,66,67,68,69,70,71,72,73,74,75,76,77,78,79,80,81,82,83,84,85,86,87,88,89,90, 91,92,93,94,95,96,97,98,99,100,101,102,103,104,105,106,107,108,109,110,111,112,113, 114,115,116,117,118,119,120,121,122,123,124,125,126,127,128,129,130,131,132,133, 134,135,136,137,138,139,140,141,142,143,144,145,146,147,148,149,150,151,152,153, 154,155,156,157,158,159,160,161,162,163,164,165,166,167,168,169
1	198	170,171,172,173,174,175,176,177,178,179,180,181,182,183,184,185,186,187,188,189, 190,191,192,193,194,195,196,197,198,199,200,201,202,203,**204,205,206**
2	382	207,208,209,210,211,212,213,214,215,216,217,218,219,220,221,222,223,224,225,226, 227,228,229,230,231,232,233,234,235,236,237,238,239,240,241,242,243,244,245,246, 247,248,249,250,251,252,253,254,255,256,257,258,259,260,261,262,263,264,265,266, 267,268,269,270,271,272,273,274,275,276,277,278,279,280,281,282,283,284,285,286, 287,288,289,290,291,292,293,294,295,296,297,298,299,300,301,302,303,304,305,306, 307,308,309,310,311,312,313,314,315,316,317,318,319,320,321,322,323,324,325,326, 327,328,329,330,331,332,333,334,335,336,337,338,339,340,341,342,343,344,345,346, 347,348,349,350,351,352,353,354,355,356,357,358,359,360,361,362,363,364,365,366, 367,368,369,370,371,372,373,374,375,376,377,378,379,380,381,382,383,384,385,386, 387,388,389,390,391,392,393,394,395,396,397,398,399,400,401,402,403,404,405,406, 407,408,409,410,411,412,413,414,415,416,417,418,419,420,421,422,423,424,425,426, 427,428,429,430,431,432,433,434,435,436,437,438,439,440,441,442,443,444,445,446, 447,448,449,450,451,452,453,454,455,456,457,458,459,460,461,462,463,464,465,466, 467,468,469,470,471,472,473,474,475,476
3	553	477,478,479,480,481,482,483,484,485,486,487,488,489,490,491,492,493,494,495,496, 497,498,499,500,501,502,503,504,505,506,507,508,509,510,511,512,513,514,515,516, 517,518,519,520,521,522,523,524,525,526,527,528,529,530,531,532,533,534,535,536,

		537,538,539,540,541,542,543,544,545,546,547,548,549,550,551,552,553,554,555,556, 557,558,559,560,561,562,563,564,565,566,567,568,569,570,571,572,573,574,575,576, 577,578,**579, 580,581,582**
4	658	583,584,585,586,587,588,589,590,591,592,593,594,595,596,597,598,599,600,601,602, 603,604,605,606,607,608,609,610,611,612,613,614,615,616,617,618,619,620,621,622, 623,624,625,626,627,628,629,630,631,632,633,634,635,636,637,638,639,640,641,642, 643,644,645,646,647,648,649,650,651,652,653,654,655,656,657,658,659,660,661,662, 663,664,665,666,667,668,669,670,671,672,673,674,675,676,677,678,679,680,681,682, 683,684,685,686,687,688,689,690,691,692,693,694,695,696,697,698,699,700,701,702, 703,704,705,706,707,708
5	753	709,710,711,712,713,714,715,716,717,718,719,720,721,722,723,724,725,726,727,728, 729,730,731,732,733,734,735,736,737,738,739,740,741,742,743,744,745,746,747,748, 749,750,751,752,753
6	778	754,755,756,757,758,759,760,761,762,763,764,765,766,767,768,769,770,771,772,773, 774,775,776,777,778,779,780,781,782,783,784,785,786,787

From Table 10, we can see that FSDP performs pretty well to identify these clusters. Only 7 objects (marked in red in Table 10) were misclassified. The running time is 5237 microseconds.

4.2.2 K-means

We set K to 7. Then let us see how K-means clustering algorithm works. Figure 12 shows the clustering result. Table 11 gives the detailed result. Table 11 is still divided into two parts.

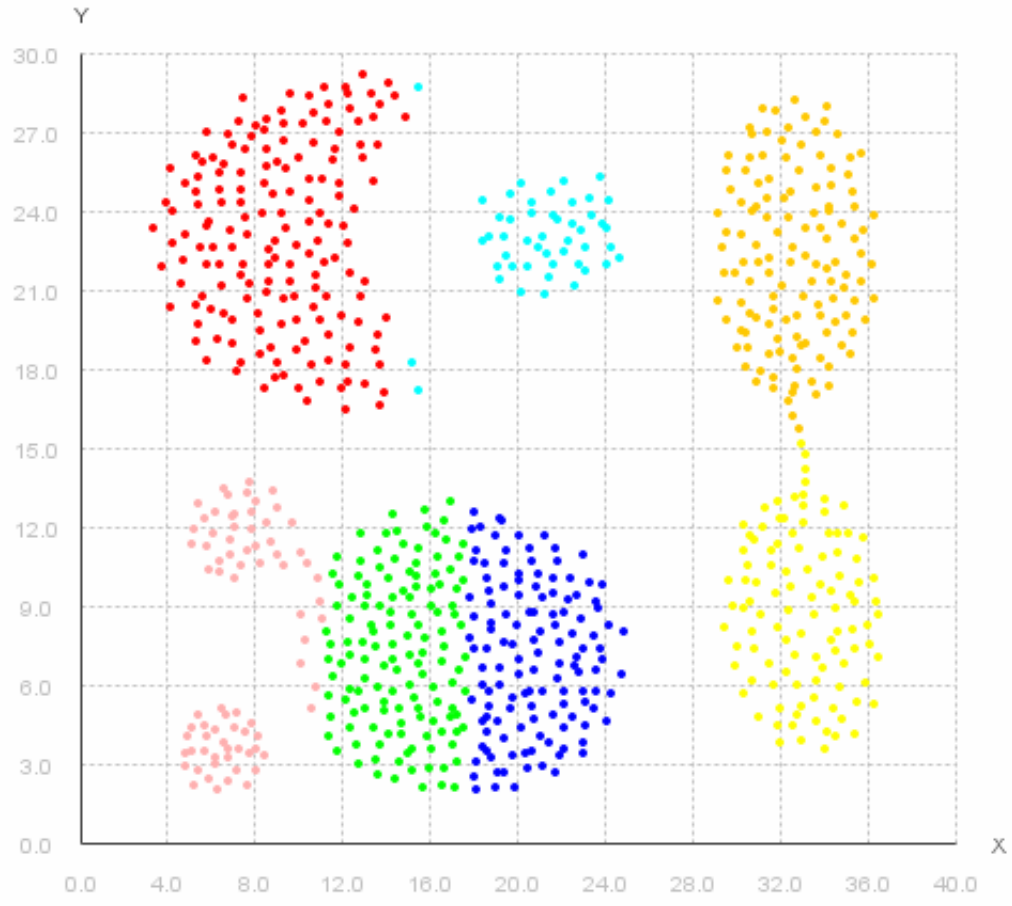

Figure 12 Clustering result of Aggregation data set by K-means

Table 11 Clustering result of Aggregation Data set by K-means

Cluster	Initial centroid	Items
0	255	327, 328, 329, 330, 331, 332, 333, 334, 335, 336, 337, 338, 339, 340, 341, 342, 343, 344, 345, 353, 354, 355, 356, 357, 358, 359, 360, 361, 362, 363, 364, 365, 366, 367, 368, 369, 370, 371, 374, 375, 376, 386, 387, 388, 389, 390, 391, 392, 393, 394, 395, 396, 397, 398, 399, 400, 401, 402, 403, 404, 405, 406, 407, 408, 409, 410, 411, 412, 413, 414, 415, 416, 419, 420, 421, 423, 424, 425, 426, 427, 428, 429, 430, 431, 432, 433, 434, 435, 436, 437, 438, 439, 440, 441, 442, 443, 444, 445, 446, 447, 448, 449, 450, 451, 452, 453, 454, 455, 456, 457, 458, 459, 460, 461, 462, 463, 464, 465, 466, 467, 468, 469, 470, 471, 472, 473, 474, 475, 476
1	140	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49, 50, 51, 52, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100, 101, 102, 103, 104, 105, 106, 107, 108, 109, 110, 111, 112, 113, 114, 115, 116, 117, 118, 119, 120, 121, 122, 123, 124, 125, 126, 127, 128, 129, 130, 131, 132, 133, 134, 135, 136, 137, 138, 139, 140, 141, 142, 143, 144, 145, 146, 147, 148, 149, 150, 151, 152, 153, 154, 155, 156, 157, 158, 159, 160, 161, 162, 163, 164, 167, 168, 169
2	286	207, 208, 211, 212, 213, 214, 215, 216, 217, 218, 219, 220, 221, 222, 223, 224, 225, 226, 227, 228, 229, 230, 234, 235, 236, 237, 238, 239, 240, 241, 242, 243, 244, 245, 246, 247, 248, 249, 250, 251, 252, 253, 254, 255, 256, 257, 258, 259, 260, 261, 262, 263, 264, 265, 266, 267, 268, 269, 272, 273, 274, 275, 276, 277, 278, 279, 280, 281, 282, 283, 284, 285, 286, 287, 288, 289, 290, 291, 292, 293, 294, 295, 296, 297, 298, 299, 300, 301, 302, 303, 304, 305, 306, 307, 308, 309, 310, 311, 312, 313, 314, 315, 316, 317, 318, 319, 320, 321, 322, 323, 324, 325, 326, 346, 347, 348, 349, 350, 351, 352, 372, 373, 377, 378, 379, 380, 381, 382, 383, 384, 385, 417, 418, 422
3	509	477, 478, 479, 480, 481, 482, 483, 484, 485, 486, 487, 488, 489, 490, 491, 492, 493, 494, 495, 496, 497, 498, 499, 500, 501, 502, 503, 504, 505, 506, 507, 508, 509, 510, 511, 512, 513, 514, 515, 516, 517, 518, 519, 520, 521, 522, 523, 524, 525, 526, 527, 528, 529, 530, 531, 532, 533, 534, 535, 536, 537, 538, 539, 540, 541, 542, 543, 544, 545, 546, 547, 548, 549, 550, 551, 552, 553, 554, 555, 556, 557, 558, 559, 560, 561, 562, 563, 564, 565, 566, 567, 568, 569, 570, 571, 572, 573, 574, 575, 576, 577, 578, 579, 580

4	156	581, 582, 583, 584, 585, 586, 587, 588, 589, 590, 591, 592, 593, 594, 595, 596, 597, 598, 599, 600, 601, 602, 603, 604, 605, 606, 607, 608, 609, 610, 611, 612, 613, 614, 615, 616, 617, 618, 619, 620, 621, 622, 623, 624, 625, 626, 627, 628, 629, 630, 631, 632, 633, 634, 635, 636, 637, 638, 639, 640, 641, 642, 643, 644, 645, 646, 647, 648, 649, 650, 651, 652, 653, 654, 655, 656, 657, 658, 659, 660, 661, 662, 663, 664, 665, 666, 667, 668, 669, 670, 671, 672, 673, 674, 675, 676, 677, 678, 679, 680, 681, 682, 683, 684, 685, 686, 687, 688, 689, 690, 691, 692, 693, 694, 695, 696, 697, 698, 699, 700, 701, 702, 703, 704, 705, 706, 707, 708
5	260	170, 171, 172, 173, 174, 175, 176, 177, 178, 179, 180, 181, 182, 183, 184, 185, 186, 187, 188, 189, 190, 191, 192, 193, 194, 195, 196, 197, 198, 199, 200, 201, 202, 203, 204, 205, 206, 209, 210, 231, 232, 233, 270, 271, 754, 755, 756, 757, 758, 759, 760, 761, 762, 763, 764, 765, 766, 767, 768, 769, 770, 771, 772, 773, 774, 775, 776, 777, 778, 779, 780, 781, 782, 783, 784, 785, 786, 787
6	38	0, 165, 166, 709, 710, 711, 712, 713, 714, 715, 716, 717, 718, 719, 720, 721, 722, 723, 724, 725, 726, 727, 728, 729, 730, 731, 732, 733, 734, 735, 736, 737, 738, 739, 740, 741, 742, 743, 744, 745, 746, 747, 748, 749, 750, 751, 752, 753

From Figure 12, we can see that the clusters of the non-spherical data set are not well captured by K-means clustering algorithm. Table 11 also shows 49 items to be misclassified (marked in red in Table 11). The running time is 86.0 microseconds.

4.3 Test with Iris Data set

4.3.1 FSDP

Preset *dc* = 1.1, ρ_{min} = 30, and δ_{min}= 0.93. FSDP also gives a reasonable partitions of this data set.

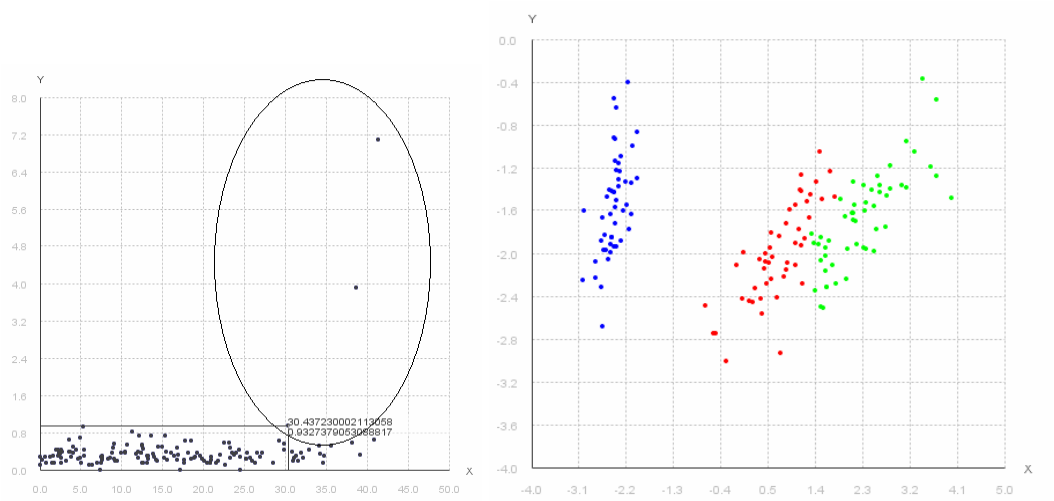

Figure 13 Decision graph (left) and cluster result (right) of Iris data set by FSDP

Table 12 Clustering result of Iris data set by FSDP

Cluster	Center	Items
0	49	0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,48,49
1	99	50,51,52,53,54,55,56,57,58,59,60,61,62,63,64,65,66,67,68,69,71,73,74,75,76,77,78,79,80,81,82,84,85,86,87,88,89,90,91,92,93,94,95,96,97,98,99,106
2	149	70,72,83,100,101,102,103,104,105,107,108,109,110,111,112,113,114,115,116,117,118,119,120,121,122,123,124,125,126,127,128,129,130,131,132,133,134,135,136,137,138,139,140,141,142,143,144,145,146,147,148,149

FSDP partitions it into three clusters. Figure 13 includes the decision graph and the plot of clustering result. Only four items (items in red in Table 12) are misclassified. The running time of the program is 2469 microseconds.

4.3.2 K-means

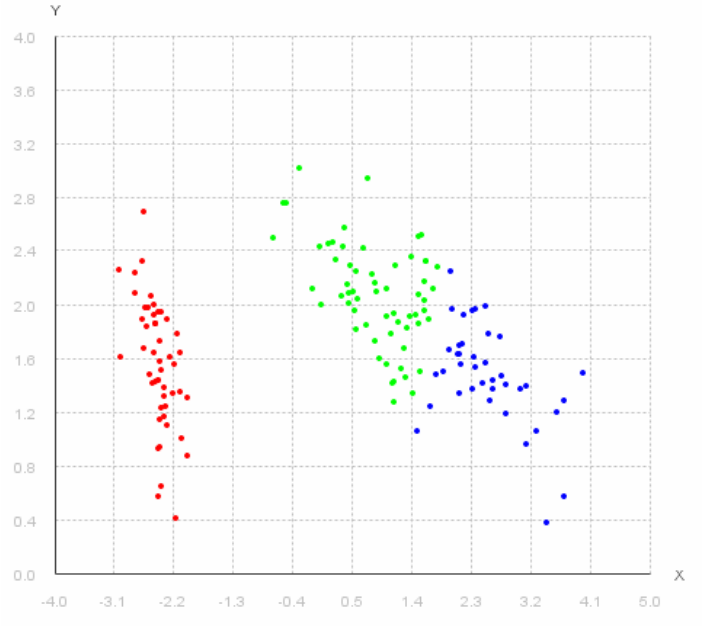

Figure 14 Clustering result of Iris data set by K-means

Table 13 Clustering result of Iris data set by K-means

Cluster	Initial center	Items
0	129	**50, 52, 77**, 100, 102, 103, 104, 105, 107, 108, 109, 110, 111, 112, 115, 116, 117, 118, 120, 122, 124, 125, 128, 129, 130, 131, 132, 134, 135, 136, 137, 139, 140, 141, 143, 144, 145, 147, 148
1	79	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45, 46, 47, 48, 49
2	55	51, 53, 54, 55, 56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 78, 79, 80, 81, 82, 83, 84, 85, 86, 87, 88, 89, 90, 91, 92, 93, 94, 95, 96, 97, 98, 99, **101, 106, 113, 114, 119, 121, 123, 126, 127, 133, 138, 142, 146, 149**

Table 13 shows that 17 items are misclassified. The running time of the program is 27 microseconds.

4.4 Olivetti Face data set

Given a set of face photos, how can we determine which photos belong to the same person? In other words, how can we partition the images such that the pictures that are in the same partition belong to the same person? This is easy for humans, but it is hard for computers. Let us test the two clustering methods on this data set. Figure 15 shows some of the photos. It includes 10 rows, and each row is composed of 10 photos of one same people.

Figure 15 First 100 faces from Olivetti Face data set

4.4.1 FSDP

The data set is special for FSDP since the number of real clusters is comparable to the number of items. This makes a reliable estimate of the densities difficult. Thus, we

assign images to a cluster following a slightly more restrictive criterion than in the preceding examples. An image is assigned to the same cluster of its nearest image with higher density only if their distance is smaller than *dc*. As a consequence, the images further than *dc* from any other image of higher density remain unassigned.

Preset *dc* = 0.07, ρ_{min} =0.6 and δ_{min}= 0.2,

Figure 16 contains the decision graph and the plot of the clustering result of Olivetti Face data set by FSDP algorithm. Figure 17 shows the clustering result.

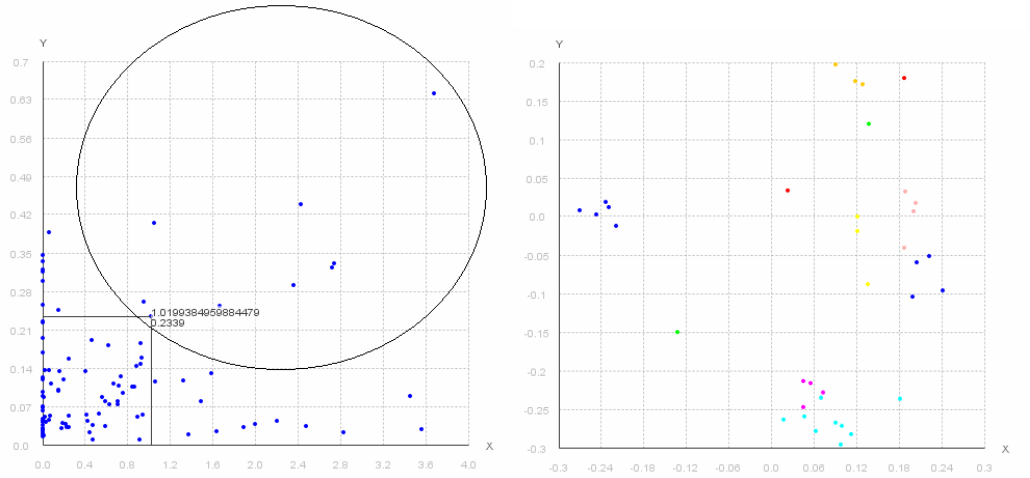

Figure 16 Decision graph of Olivetti Face data set (left), and the clusters (right) by FSDP

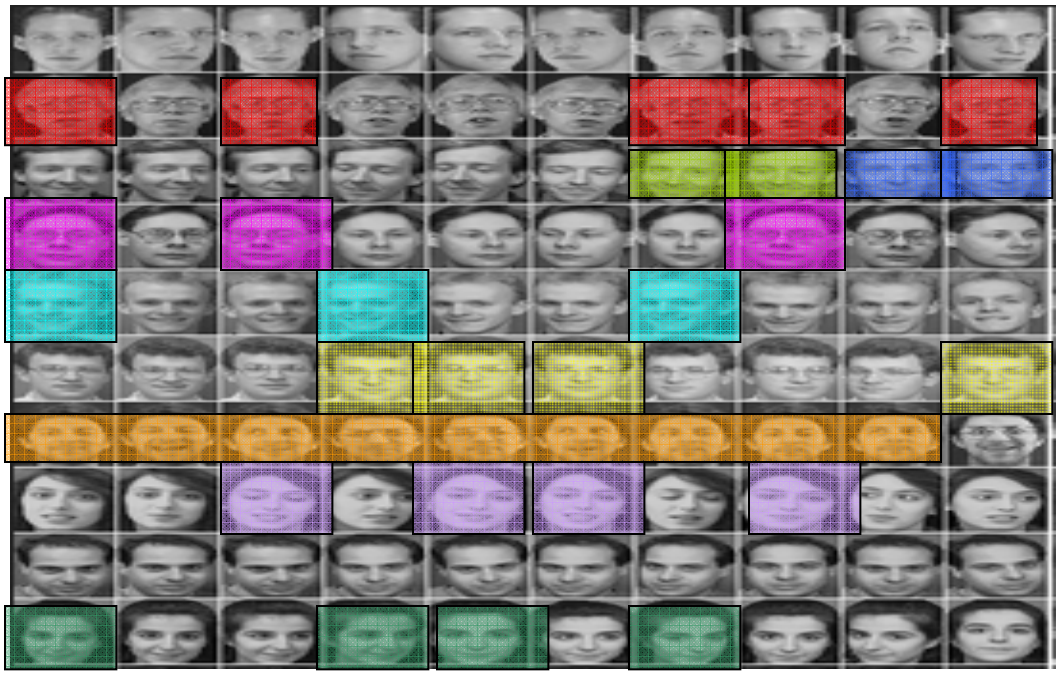

Figure 17 Clustering result of Olivetti Face data set by FSDP clustering algorithm

Table 14 lists the details.

Table 14 Cluster result of Olivetti Face data set by FSDP

Cluster	Center	Items
0	19	10,12,16,17,19
1	27	26,27
2	29	28,29
3	37	30,32,37
4	46	40,43,46
5	59	53,54,55,59
6	68	60,61,62,63,64,65,66,67,68
7	77	72,74,75,77
8	97	91,94,95,97

This algorithm assigns the photos of the same person into the same cluster. But there are some photos not assigned into any cluster due to the restrictive criterion. Finally, 36 images are correctly grouped. The running time of this program is 2217

microseconds.

4.4.2 K-means

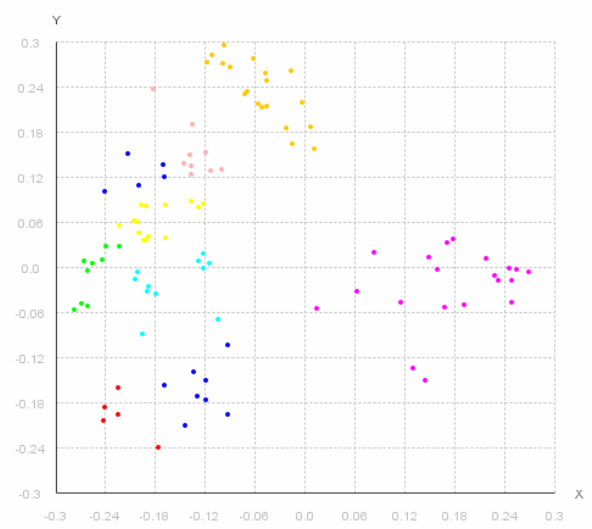

Figure 18 K-means clusters of Olivetti Face data set

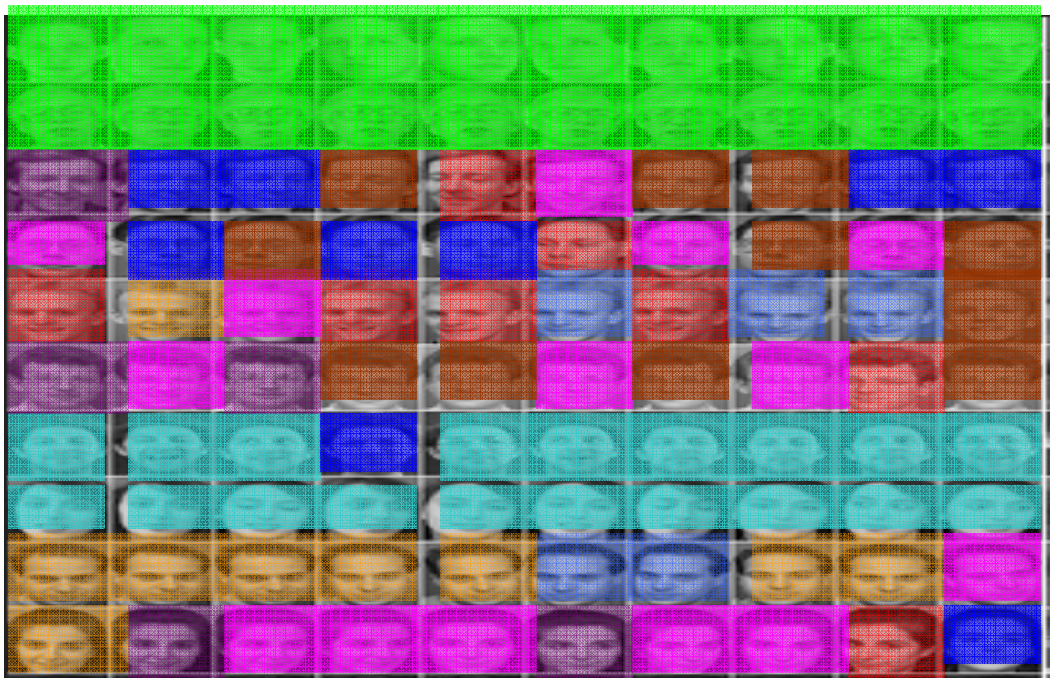

Figure 19 Clustering result of Olivetti Face data set by K-means clustering algorithm

Table 15 Cluster result of Olivetti Face data set by K-means

Cluster	Initial center	Items
0	44	24, 35, 40, 43, 44, 46, 58, 98
1	85	45, 47, 48, 85, 86
2	82	41, 80, 81, 82, 83, 84, 87, 88, 90
3	51	25, 30, 36, 38, 42, 51, 55, 57, 89, 92, 93, 94, 96, 97
4	62	60, 61, 62, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73, 74, 75, 76, 77, 78, 79
5	22	21, 22, 28, 29, 31, 33, 34, 63, 99
6	96	23, 26, 27, 32, 37, 39, 49, 53, 54, 56, 59
7	1	0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19
8	95	20, 50, 52, 91, 95

From Table 15, we can see that K-means put all items into different clusters. However, many photos of one same person are assigned into different clusters. Only cluster 8 contains photos of the same person. The running time of the program is 13.0 microseconds.

4.5 Validation

Two criteria are applied to compare the two algorithms.

The following terminology is used for both criteria.

L: the number of classes.

k: the number of clusters.

m_j: the number of items in cluster j.

m_{ij}: the number of items which belong to class j and are assigned into cluster i.

m: the total number of items.

$p_{ij} = m_{ij}/m_i$: the "probability" that one member of cluster *i* belongs to class j.

4.5.1 Entropy

The Entropy of each cluster *j* is calculated using the standard formula:

$$e_i = -\sum_{j=1}^{L} p_{ij} \log_2 p_{ij}$$

The total entropy for a set of clusters is:

$$e = \sum_{i=1}^{k} \frac{m_i}{m} e_i$$

4.5.2 Purity

The purity of cluster *j* is given by

$$p_j = \max_i p_{ij}$$

The total purity for a set of clusters is

$$\text{purity} = \sum_{i=1}^{k} \frac{m_i}{m} p_i$$

The entropy and purity of FSDP and K-means clustering algorithm for the previous four data sets are listed separately in Tables 16 to 24. Higher purity and lower entropy mean better clustering result.

By comparing the two tables, we see that FSDP works better than K-means on this data set. The values (in bold) show the results of entropy and purity.

Table 16 FSDP Entropy / Purity values of the Flame Data set

Cluster	Class 1	Class 2	Entropy	Purity
0	2	153	0.09948	0.98710
1	85	0	0.0	1.0
Total	87	153	**0.06424**	**0.99167**

Table 17 K-means Entropy / Purity values of the Flame Data set

Cluster	Class 1	Class 2	Entropy	Purity
0	5	117	0.24678	0.95902
1	83	35	0.87711	0.70339
total	88	152	**0.55669**	**0.8333**

Table 18 FSDP Entropy / Purity values of the Aggregation data set

Cluster	Class 1	Class 2	Class 3	Class 4	Class 5	Class 6	Class 7	Entropy	Purity
0	0	170	0	0	0	0	0	0.0	1.0
1	0	0	0	3	0	0	34	0.40598	0.91892
2	0	0	0	0	270	0	0	0.0	1.0
3	0	0	102	0	0	4	0	0.23181	0.96226
4	0	0	0	0	0	126	0	0.0	1.0
5	45	0	0	0	0	0	0	0.0	1.0
6	0	0	0	0	34	0	0	0.0	1.0
Total	45	170	102	3	304	130	34	**0.05024**	**0.99112**

Table 19 K-means Entropy / Purity values of the Aggregation data set

Cluster	Class 1	Class 2	Class 3	Class 4	Class 5	Class 6	Class 7	Entropy	Purity
0	0	0	0	129	0	0	0	0.0	1.0
1	0	167	0	0	0	0	0	0.0	1.0
2	0	0	0	134	0	0	0	0.0	1.0
3	0	0	102	2	0	0	0	0.13710	0.98077
4	0	0	0	0	0	128	0	0.0	1.0
5	0	0	0	10	34	0	34	1.42429	0.43590
6	45	3	0	0	0	0	0	0.33729	0.93750
Total	45	170	102	265	34	128	34	**0.17962**	**0.93782**

Table 20 FSDP Entropy / Purity values of the Iris data set

Cluster	Class 1	Class 2	Class 3	Entropy	Purity
0	50	0	0	0.0	1.0
1	0	47	1	0.14609	0.97917
2	0	3	49	0.31822	0.94231
Total	50	50	50	**0.157065**	**0.97333**

Table 21 K-means Entropy / Purity values of the Iris data set

Cluster	Class 1	Class 2	Class 3	Entropy	Purity
0	0	3	36	0.39124	0.92308
1	50	0	0	0.0	1.0
2	0	47	14	0.77715	0.77049
Total	50	50	50	**0.41777**	**0.88667**

Table 22 FSDP Entropy / Purity values of the Olivetti face data set

Cluster	Class1	Class2	C3	C4	C5	C6	C7	C8	C9	C10	Entropy	Purity
0	0	5	0	0	0	0	0	0	0	0	0.0	1.0
1	0	0	2	0	0	0	0	0	0	0	0.0	1.0
2	0	0	2	0	0	0	0	0	0	0	0.0	1.0
3	0	0	0	3	0	0	0	0	0	0	0.0	1.0
4	0	0	0	0	3	0	0	0	0	0	0.0	1.0
5	0	0	0	0	0	4	0	0	0	0	0.0	1.0
6	0	0	0	0	0	0	9	0	0	0	0.0	1.0
7	0	0	0	0	0	0	0	4	0	0	0.0	1.0
8	0	0	0	0	0	0	0	0	0	4	0.0	1.0
9	0	0	0	0	0	0	0	0	0	0	0.0	0.0
Total	0	5	4	3	3	4	9	4	0	4	0.0	1.0

Table 23 K-means Entropy / Purity values of the Olivetti data set

Cluster	Class1	Class2	C3	C4	C5	C6	C7	C8	C9	C10	Entropy	Purity
0	0	0	1	1	4	1	0	0	0	1	2.0	0.5
1	0	0	0	0	3	0	0	0	2	0	0.970950	0.6
2	0	0	0	0	1	0	0	0	7	1	0.98643	0.77778
3	0	0	1	3	1	3	0	0	1	5	2.29883	0.357143
4	0	0	0	0	0	0	9	10	0	0	0.99800	0.52632
5	0	0	4	3	0	0	1	0	0	1	1.75272	0.444444
6	0	0	3	3	1	4	0	0	0	0	1.86763	0.363636
7	10	10	0	0	0	0	0	0	0	0	1.0	0.5
8	0	0	1	0	0	2	0	0	0	2	1.52193	0.4
9	0	0	0	0	0	0	0	0	0	0	0.0	0.0
Total	10	10	10	10	13	10	17	10	12	10	1.44806	0.49

Table 24 Summary of Entropy and Purity values for FSDP and K-means

Data set	Purity		Entropy	
	FDSP	*K-means*	FDSP	*K-means*
Flame	**0.99167**	0.83333	**0.06425**	0.55669
Aggregation	**0.99111**	0.937817	**0.05025**	0.17962
Iris	**0.97333**	0.88667	**0.15706**	0.41777
Olivetti	**1.0**	0.49	**0.0**	1.44806

Table 25 shows the running time on the four data sets by the two algorithms.

Table 25 Summary of running time for FSDP and K-means

Data set	Running time (microseconds)	
	FSDP	K-means
Flame	3024	58
Aggregation	5237	86
Iris	2469	27
Olivetti	2217	13

From Tables 16 to 24, we conclude the following:

1. FSDP is more accurate than K-means on the 4 data sets. The fact is true due to the following 3 reasons:

- The values of purity by FSDP clustering algorithm are bigger than the ones by the K-means clustering algorithm. The value of purity is between 0 and 1. When the result of clustering is completely incorrect, the value of purity will be 0. Larger values of purity correlate with better accuracy. So FSDP clustering algorithm is more accurate than K-means clustering algorithm.
- The values of entropy by FSDP clustering algorithm are smaller than the ones obtained by the K-means clustering algorithm. A smaller value of entropy means better accuracy. So FSDP clustering algorithm is more accurate than K-means clustering algorithm.
- The number of misclassified items by FSDP clustering algorithm is less than the ones obtained by the K-means clustering algorithm. The fewer the

misclassified items, the more accurate the algorithm. So FSDP clustering algorithm is more accurate than K-means clustering algorithm.

2. FSDP clustering algorithm is more stable than K-means clustering algorithm.

The results of K-means clustering algorithm strongly depend on the initial centers and the order of items. The initial centers are usually chosen randomly. In addition, when the order of items is changed, it is possible that the clustering result will be different. On the other hand, FSDP clustering algorithm does not have this problem. So FSDP clustering algorithm is more stable than K-means clustering algorithm.

However, K-means clustering algorithm is faster than FSDP clustering algorithm. From Table 25, one can see that K-means was much faster than the FSDP algorithm. For example, with the Olivetti data set, K-means is 2217/13 $\approx$ 170 faster than the FSDP algorithm. The main reason behind the slow pace is the drawing of the decision graph.

In this chapter, we tested the performance of the two algorithms, the K-means and FSDP by running them on 4 data sets: Flame, Aggregation, Iris, and Olivetti. We concluded that FSDP is more accurate but much slower than the K-means algorithm.

In the next chapter, we conclude this work and suggest future extensions.

Chapter 5. Conclusions

In this work, we discussed two clustering algorithms, the K-means clustering algorithm and Fast Search and Find Density of Peaks (FSDP) clustering algorithm.

K-means clustering algorithm partitions by using multiple iterations. This algorithm is easy to understand and implement. However, its results depend on initial cluster centers and the order of items inputted. Moreover, for non-spherical data sets, K-means clustering algorithm does not work well.

The FSDP clustering algorithm is new. It is based on the observation that a cluster center has a high density of data points, but the center is far away from other centers. The FSDP clustering algorithm needs to compute the density ρ and the distance δ for all items. The δ -vs- ρ decision graph helps decide the number of clusters, and it also indicates which data point should be the cluster center.

In the two algorithms, the distance between all pairs of data points has to be computed. Choosing the distance function is one important step for solving clustering problems. However, nobody knows which distance function can work well for one data set. It is a challenging topic.

As possible future extensions of this works, one can work on the decision graph. The decision graph is the core feature of the FSDP clustering algorithm. With the decision graph, we can set the thresholds and search for the centers. However, the decision graph takes a long time to complete for large sets. Decreasing the time of getting decision graph is one of the future targets.

Another possible challenge for the future work is using the parallel computation. Both algorithms in this work are easy to be parallelized. In FSDP, for example, the computation of the distance δ and the density ρ of all data points is independent of each other. They can be distributed to many computational nodes/CPUs. In K-means, updating the cluster centers is easy by parallel computation.

References

[1] Alex Rodriguez and Alessandro Laio, "Clustering by fast search and find of density peaks," *Science*, Vol. 344, No. 6191, pp. 1492-1496, 2014.

[2] Rui Xu, Donald Wunsch, "Survey of Clustering Algorithms," IEEE Transactions on Neural Networks, Vol. 16, No. 3, pp. 645-678, 2005.

[3] https://en.wikipedia.org/wiki/K-means_clustering, last checked on 11/08/2015

[4] Yizong Cheng, "Mean shift, mode seeking and clustering," IEEE Transactions on Pattern Analysis and Machine Intelligence, Vol. 17, No. 8, pp. 790-799, 1995.

[5] Clustering data set, http://cs.joensuu.fi/sipu/data sets/, last checked on 11/08/2015

[6] http://people.sissa.it/~laio/Research/Res_clustering.php, last checked on 11/08/2015

[7] https://en.wikipedia.org/wiki/DBSCAN, last checked on 11/08/2015

[8] https://en.wikipedia.org/wiki/Apache_Hadoop, last checked on 11/08/2015

[9] https://en.wikipedia.org/wiki/Euclidean_distance, last checked on 11/08/2015

[10] Y. Cheng, "Mean shift, mode seeking, and clustering", IEEE Trans. Pattern Anal. Mach. Intell, vol. 17, no. 8, pp. 790–799, 1995.

Appendix: Source code

The following is the source codes for the both clustering algorithms in Java.

**********K-means**********

```
package Kmeans;

import java.io.BufferedReader;
import java.io.DataInputStream;
import java.io.File;
import java.io.FileInputStream;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.Random;

import javax.swing.JFrame;

import org.math.plot.Plot2DPanel;

public class Kmeans {
    private int k;//how many clusters
    private int iteration;// iteration
    private int data setLength;
    private ArrayList<double[]> data set;
    private ArrayList<double[]> center;
    //this definition of cluster fits other data sets but olivetti face
    private ArrayList<ArrayList<double[]>> cluster;
    //for olivetti face
```

```
private ArrayList<Double> offsetArray;
private Random random;



public void setData set(ArrayList<double[]> data set) {
   this.data set = data set;
}

public ArrayList<ArrayList<double[]>> getCluster() {
   return cluster;
}

public Kmeans(int k) {
   if (k <= 0) {
      k = 1;
   }
   this.k = k;
}

private void init() {
   iteration = 0;
   random = new Random();
   data setLength = data set.size();
   if (k > data setLength) {
      k = data setLength;
   }
   center = initCenters();
   cluster = initCluster();
   offsetArray = new ArrayList<Double>();
}
```

```
private ArrayList<double[]> initCenters() {
    ArrayList<double[]> center = new ArrayList<double[]>();
    int[] randoms = new int[k];
    boolean flag;
    int temp = random.nextInt(data setLength);
    randoms[0] = temp;
    for (int i = 1; i < k; i++) {
        flag = true;
        while (flag) {
            temp = random.nextInt(data setLength);
            int j = 0;
            while (j < i) {
                if (temp == randoms[j]) {
                    break;
                }
                j++;
            }
            if (j == i) {
                flag = false;
            }
        }
        randoms[i] = temp;
    }

    for (int i = 0; i < k; i++) {
        center.add(data set.get(randoms[i]));
    }

    return center;
```

```
    }

    private ArrayList<ArrayList<double[]>> initCluster() {
        ArrayList<ArrayList<double[]>> cluster = new ArrayList<ArrayList<double[]>>();
        for (int i = 0; i < k; i++) {
            cluster.add(new ArrayList<double[]>());
        }

        return cluster;
    }

    // for 4 dimensions
    /*
    private double distance(double[] point1, double[] point2) {
        double distance = 0;
        double x = point1[0] - point2[0];
        double y = point1[1] - point2[1];
       double z = point1[2] - point2[2];
        double k = point1[3] - point2[3];
        double s = x * x + y * y + z*z + k*k;
        distance = (double) Math.sqrt(s);
        return distance;
    }
   */
    // for 2 dimensions

    private double distance(double[] point1, double[] point2) {
        double distance = 0;
        double x = point1[0] - point2[0];
        double y = point1[1] - point2[1];
```

```
    double s = x * x + y * y;
    distance = (double) Math.sqrt(s);
    return distance;
}

private int minDistance(double[] distance) {
    double minDistance = distance[0];
    int minLocation = 0;
    for (int i = 1; i < distance.length; i++) {
        if (distance[i] < minDistance) {
            minDistance = distance[i];
            minLocation = i;
        } else if (distance[i] == minDistance) //
        {
            if (random.nextInt(10) < 5) {
                minLocation = i;
            }
        }
    }

    return minLocation;
}

private void clusterSet() {
    double[] distance = new double[k];
    for (int i = 0; i < data setLength; i++) {

        for (int j = 0; j < k; j++) {
            distance[j] = distance(data set.get(i), center.get(j));
            //System.out.println("distance to "+j+":"+distance[j]);
```

```
            }

            int minLocation = minDistance(distance);

            cluster.get(minLocation).add(data set.get(i));//

        }
    }

    // for 4 dimensions
    /*
    private double errorSquare(double[] element, double[] centroid) {
        double x = element[0] - centroid[0];
        double y = element[1] - centroid[1];
        double z = element[2] - centroid[2];
        double k = element[3] - centroid[3];

        double errorSquare = x * x + y * y + z*z + k*k;

        return errorSquare;
    }
*/
    // for 2 dimensions

    private double error(double[] element, double[] centroid) {
        double x = element[0] - centroid[0];
        double y = element[1] - centroid[1];
        double errorSquare = x * x + y * y;

        return errorSquare;
```

```
    }

    private void countRule() {
        double errorCluster = 0;
        for (int i = 0; i < cluster.size(); i++) {
            for (int j = 0; j < cluster.get(i).size(); j++) {
                errorCluster += error(cluster.get(i).get(j), center.get(i));


            }
        }
        offsetArray.add(errorCluster);
    }

// for 4 dimensions
    /*
    private void setNewCenter() {
        for (int i = 0; i < k; i++) {
            int n = cluster.get(i).size();
            if (n != 0) {
                double[] newCenter = { 0, 0, 0, 0, 0};
                for (int j = 0; j < n; j++) {
                    newCenter[0] += cluster.get(i).get(j)[0];
                    newCenter[1] += cluster.get(i).get(j)[1];
                    newCenter[2] += cluster.get(i).get(j)[2];
                    newCenter[3] += cluster.get(i).get(j)[3];
                    newCenter[4] += cluster.get(i).get(j)[4];
                  }

                newCenter[0] = newCenter[0] / n;
                newCenter[1] = newCenter[1] / n;
```

```
            newCenter[2] = newCenter[2] / n;
            newCenter[3] = newCenter[3] / n;
            newCenter[4] = newCenter[4] / n;
            center.set(i, newCenter);
        }
    }
}
*/
// for 2 dimensions

private void setNewCenter() {
    for (int i = 0; i < k; i++) {
        int n = cluster.get(i).size();
        if (n != 0) {
            double[] newCenter = { 0, 0, 0};
            for (int j = 0; j < n; j++) {
                newCenter[0] += cluster.get(i).get(j)[0];
                newCenter[1] += cluster.get(i).get(j)[1];
                newCenter[2] += cluster.get(i).get(j)[2];

            }
            newCenter[0] = newCenter[0] / n;
            newCenter[1] = newCenter[1] / n;
            newCenter[2] = newCenter[2] / n;
            center.set(i, newCenter);
        }
    }
}

// for 4 dimensions
```

```
    /*
    public void printDataArray(ArrayList<double[]> dataArray,
            String dataArrayName) {
        for (int i = 0; i < dataArray.size(); i++) {
            //System.out.println("print:" + dataArrayName + "[" + i + "]={"
                    //+ dataArray.get(i)[0] + "," + dataArray.get(i)[1]+"," + dataArray.get(i)[2] + "," + dataArray.get(i)[3]+
"}");
                System.out.print((int)dataArray.get(i)[4]+", ");
        }
        System.out.println("====================================");
    }
 */
    // for 2 dimensions

    public void printDataArray(ArrayList<double[]> dataArray,
            String dataArrayName) {
                System.out.println();
        for (int i = 0; i < dataArray.size(); i++) {

                System.out.print((int)dataArray.get(i)[2]+", ");
        }

    }

    private void kmeans() {
        init();


        while (true) {
            clusterSet();
```

```
            countRule();

            if (iteration != 0) {
               if (offsetArray.get(iteration) - offsetArray.get(iteration - 1) == 0) {
                  break;
               }
            }
            System.out.println("iteration:"+iteration);
            for(int i = 0; i < k; i++){
               System.out.println("center:"+center.get(i)[0]+","+center.get(i)[1] + ","+ center.get(i)[2]);
            }
            /*
            for(int j = 0; j < cluster.size(); j++){
               System.out.println("cluster:"+j);
               for(int i = 0; i < cluster.get(j).size(); i++){
                        System.out.print(cluster.get(j).get(i)[0]+","+cluster.get(j).get(i)[1]+ "," + cluster.get(j).get(i)[2] +
";");
               }
               System.out.println();
            }
            */
            setNewCenter();
            iteration++;

            cluster.clear();
            cluster = initCluster();
         }

      }
```

```
public void plot(){
        ArrayList<ArrayList<double[]>> cluster=getCluster();

   for(int i=0;i<cluster.size();i++)
   {
      printDataArray(cluster.get(i), "cluster["+i+"]");
   }
   Plot2DPanel plot = new Plot2DPanel();
   double[] z;
                double[] y;
                for(int i = 0; i < cluster.size(); i++){
                        z = new double[cluster.get(i).size()];
                        y = new double[cluster.get(i).size()];
                        for(int j = 0; j < cluster.get(i).size(); j++){
                                z[j] = cluster.get(i).get(j)[0];
                                y[j] = cluster.get(i).get(j)[1];
                        }
                        plot.addScatterPlot("clustering", z, y);
                }

            JFrame frame = new JFrame("a plot panel");
            frame.setSize(600, 600);
            frame.setContentPane(plot);
            frame.setVisible(true);

}

public void execute() {
```

```
        double startTime = System.currentTimeMillis();
        System.out.println("Kmeans starts");
        kmeans();
        double endTime = System.currentTimeMillis();
        System.out.println("kmeans running time=" + (endTime - startTime)
                + "ms");
        System.out.println("Kmeans ends");

    }
    /*
    public static void main(String[] args){

          Kmeans kmeans=new Kmeans(3);
          ArrayList<double[]> data set=new ArrayList<double[]>();
          ReadMatrix rm = new ReadMatrix("d:\\cs298\\positionMatrix_iris.txt");
          data set = rm.getData set();

         //rm.distanceMatrix();
         // System.out.println("distance matrix:");
        // rm.printDistanceMatrix();

          kmeans.setData set(data set);
          kmeans.execute();
          kmeans.plot();

    }
*/

}
```

```
class ReadMatrix{
	ArrayList<double[]> data set=new ArrayList<double[]>();
	public ReadMatrix(String dir){
		convert(dir);
	}
	void convert(String dir){
		try{
			File file = new File(dir);
			FileInputStream fis = new FileInputStream(file);
			DataInputStream dis = new DataInputStream(fis);
			InputStreamReader isr = new InputStreamReader(dis);
			BufferedReader br = new BufferedReader(isr);
			String current;
			int count = 0;
			while((current = br.readLine()) != null){
				if(current.length() == 0)
					continue;
				//if the data is seperated by ",", use the below split way
				String[] split = current.trim().split("\\,");
				//if the data is separated by space, ust the below split way
				//String[] split = current.trim().split("\\s++");
				//for 4 dimensions
				//data set.add(new double[]{Double.parseDouble(split[0]),
Double.parseDouble(split[1]), Double.parseDouble(split[2]), Double.parseDouble(split[3]), (double)count});

				data set.add(new double[]{Double.parseDouble(split[0]),
Double.parseDouble(split[1]), (double)count});
				count++;
			}
		} catch(Exception e){
```

```
                System.out.println(e);
        }

}

public ArrayList<double[]> getData set(){
        return data set;
}
//for 4 dimensions
/*
double[][] distanceMatrix(){
        double[][] distanceMatrix = new double[data set.size()][data set.size()];
        for(int i = 0; i < data set.size(); i++){
                for(int j = 0; j < data set.size(); j++){
                        double x = data set.get(i)[0] - data set.get(j)[0];
                        double y = data set.get(i)[1] - data set.get(j)[1];
                        double z = data set.get(i)[2] - data set.get(j)[2];
                        double k = data set.get(i)[3] - data set.get(j)[3];
                       // System.out.println(k);
                        distanceMatrix[i][j] = x*x + y*y + z*z + k*k;
                        distanceMatrix[j][i] = x*x + y*y + z*z + k*k;
                }
        }

        return distanceMatrix;
}
*/

// for 2 dimensions
```

```
	double[][] distanceMatrix(){
		double[][] distanceMatrix = new double[data set.size()][data set.size()];
		for(int i = 0; i < data set.size(); i++){
			for(int j = 0; j < data set.size(); j++){
				double x = data set.get(i)[0] - data set.get(j)[0];
				double y = data set.get(i)[1] - data set.get(j)[1];

				distanceMatrix[i][j] = x*x + y*y;
				distanceMatrix[j][i] = x*x + y*y;
			}
		}

		return distanceMatrix;
	}

	void printDistanceMatrix(){
		for(int i = 0; i < distanceMatrix().length; i++){
			for(int j = 0; j < distanceMatrix().length; j++){
				System.out.print(distanceMatrix()[i][j]+" ");
			}
			System.out.println();
		}
	}

}
```

**********FSDP**********

```
package cs298;
```

```
import javax.swing.*;
import calculateDistanceMatrix.*;
import mdsj.MDSJ;
import org.math.plot.*;
import java.util.ArrayList;
import java.io.*;

public class FFSP {
        double[][] distanceMatrix;
        int[] clusterArray;
        int[] centers;
        double cutoffdc;
        public static double[] rhoArray;
        public static double[] deltaArray;
        //double[] sortedDeltaArray;
        int[] originalIndexOfDelta;
        double[] sortedRhoArray;
        int[] originalIndexOfRho;
        double rhomin;
        double deltamin;
        static int clusterNumbers;
        static ArrayList<ArrayList<Integer>> group;
        ArrayList<Integer> halo;
        int[] index;
        //this constructor for files which contains the distance between points
        /*
         * The only input needed is a distance matrix file
        The format of this file should be:
        Column 1: id of element i
        Column 2: id of element j
```

```
Column 3: dist(i,j)
 * */
public FFSP(double[][] distanceMatrix, double cutoffdc, double rhomin, double deltamin){
        clusterNumbers = 0;
        clusterArray = new int[distanceMatrix[0].length];
        for(int i = 0; i < distanceMatrix[0].length; i++){
                clusterArray[i] = -2;
        }
        centers = new int[distanceMatrix[0].length];
        for(int i = 0; i < distanceMatrix[0].length; i++){
                centers[i] = -1;
        }
        this.distanceMatrix = distanceMatrix;
        this.cutoffdc = cutoffdc;
        this.rhomin = rhomin;
        this.deltamin = deltamin;
        rhoArray = new double[distanceMatrix[0].length];
        deltaArray = new double[distanceMatrix[0].length];
        group = new ArrayList<ArrayList<Integer>>();
        halo = new ArrayList<Integer>();

        originalIndexOfDelta = new int[distanceMatrix[0].length];
        index = new int[distanceMatrix[0].length];
        for(int i = 0; i < index.length; i++){
                index[i] = i;
        }
}
//this constructor for the file which contains the points
public FFSP(String dir, double cutoffdc, double rhomin, double deltamin){
        CalculateDistanceMatrix cdm = new CalculateDistanceMatrix(dir);
```

```
		distanceMatrix = cdm.distanceMatrix();


		clusterNumbers = 0;
		clusterArray = new int[distanceMatrix[0].length];
		for(int i = 0; i < distanceMatrix[0].length; i++){
			clusterArray[i] = -2;
		}
		centers = new int[distanceMatrix[0].length];
		for(int i = 0; i < distanceMatrix[0].length; i++){
			centers[i] = -1;
		}
		this.cutoffdc = cutoffdc;
		this.rhomin = rhomin;
		this.deltamin = deltamin;
		rhoArray = new double[distanceMatrix[0].length];
		deltaArray = new double[distanceMatrix[0].length];
		group = new ArrayList<ArrayList<Integer>>();
		halo = new ArrayList<Integer>();
		originalIndexOfDelta = new int[distanceMatrix[0].length];
		index = new int[distanceMatrix[0].length];
		for(int i = 0; i < index.length; i++){
			index[i] = i;
		}


}
//local density of point index: rho. here we use X(chi) function
//% "Cut off" kernel
/*
public int localDensity(int index){
		int rho = 0;
```

```
                for(int i = 0; i < distanceMatrix[0].length; i++){
                        if(distanceMatrix[index][i] < cutoffdc && i != index)
                                rho++;
                }

                return rho;
        }
        */
        // Gaussian kernel

        double localDensity(int index){
                double rho = 0;
                for(int i = 0; i < distanceMatrix[0].length && i != index; i++){
                        rho = rho + Math.exp(-
(distanceMatrix[i][index]/cutoffdc)*(distanceMatrix[i][index]/cutoffdc));
                }
                return rho;
        }

        //rhoArray and deltaArray
                void calculateRhoArrays(){
                        for(int i = 0; i < distanceMatrix[0].length; i++){
                                rhoArray[i] = localDensity(i);
                                //System.out.println("rho:"+rhoArray[i]);
                        }
                }

                //sort Arrayï¼ˆascendï¼‰ and remember the original index

                void sortRhoArray(){
```

```
            sortedRhoArray=rhoArray;
            double temp;
            int temp2;
            for(int i = 0; i <sortedRhoArray.length-1; i++){
                        for(int j = i+1; j < sortedRhoArray.length; j++){
                                    if(sortedRhoArray[i] > sortedRhoArray[j]){
                                                temp = sortedRhoArray[i];
                                                sortedRhoArray[i] = sortedRhoArray[j];
                                                sortedRhoArray[j] = temp;


                                                temp2 = index[i];


                                                index[i] = index[j];
                                                //System.out.println("**********");
                                                //System.out.print("i"+"j"+i+","+j);
                                                index[j] = temp2;
                                    }

                        }
            }

}

//calculate neighbor
int neighbor(int i){
            int neighborOfi = Integer.MAX_VALUE;
            double temp = Double.MAX_VALUE;
            for(int j = 0; j < distanceMatrix[0].length; j++){
                        if(index[j] == i){
                                    for(int k = j+1; k < distanceMatrix[0].length; k++){
```

```
                                        if(distanceMatrix[index[k]][i] < temp){
                                                temp = distanceMatrix[index[k]][i];
                                                neighborOfi = index[k];
                                        }
                                }
                                break;
                        }

                }

                if(neighborOfi == Integer.MAX_VALUE)
                        neighborOfi = i;
                return neighborOfi;
        }

//distance delta from higher points of higher density point
public double distanceFromHigherDensity(int point){
        double delta;
        double tempMaxValue = Double.MIN_VALUE;
        for(int i = 0; i < distanceMatrix[0].length; i++){
                for(int j = i+1; j < distanceMatrix[0].length; j++){
                        if(distanceMatrix[i][j] > tempMaxValue){
                                tempMaxValue = distanceMatrix[i][j];
                        }
                }
        }

        delta = tempMaxValue;

        for(int i = 0; i < distanceMatrix[0].length; i++){
```

```
                    if(index[i] == point){
                            for(int j = i+1; j < distanceMatrix[0].length; j++){
                                    if(distanceMatrix[point][index[j]] < delta){
                                            delta = distanceMatrix[point][index[j]];


                                    }
                            }
                            break;
                    }
            }
            deltaArray[point] = delta;
            return delta;


}
//calculate the deltaArray
void calculateDeltaArrays(){
            for(int i = 0; i < distanceMatrix[0].length; i++){
                    deltaArray[i] = distanceFromHigherDensity(i);
                    //System.out.println("rho:"+rhoArray[i]);
            }
}


// sort delta array
/*
void sortDeltaArray(){
            double temp;
            for(int i = 0; i <sortedRhoArray.length-1; i++){
                    for(int j = i+1; j < sortedRhoArray.length; j++){
                            if(deltaArray[i] > deltaArray[j]){
                                    temp = deltaArray[i];
```

```
                                        deltaArray[i] = deltaArray[j];
                                        deltaArray[j] = temp;

                                }

                        }
                }

        }

        */
        //assign points: points with bigger delta and rho are centers
        //all data sets except olivetti face sets

        void cluster(){

                for(int i = 0; i < distanceMatrix[0].length; i++){
                        if(localDensity(i) >= rhomin && deltaArray[i] >= deltamin){
                                System.out.println("the center point:"+i);
                                System.out.println("rho"+localDensity(i));
                                System.out.println("delta"+deltaArray[i]);
                                clusterArray[i] = clusterNumbers; // clusterNumbers: the NO. of cluster.
clusterCenters[i]: point i is the center if clusterCenters[i] != -2
                                centers[clusterNumbers] = i;
                                clusterNumbers++;


                        }
                }
```

```
                for(int j = distanceMatrix[0].length-1; j >= 0; j--){
                        if(clusterArray[index[j]] == -2){
                                clusterArray[index[j]] = clusterArray[neighbor(index[j])];


                        }
                }
                //initialize group
                for(int i = 0; i < clusterNumbers; i++){
                        group.add(new ArrayList<Integer>());
                }
                if(clusterNumbers != 0){
                        for(int k = 0; k < distanceMatrix[0].length; k++){
                                for(int i = 0; i < clusterNumbers; i++){
                                        if(clusterArray[k] == i){
                                                group.get(i).add(k);
                                        }


                                }
                        }


                }


        }


        //just for olivetti face data sets
/*
void cluster(){


                for(int i = 0; i < distanceMatrix[0].length; i++){
                        if(localDensity(i) >= rhomin && deltaArray[i] >= deltamin){
```

```
                                        System.out.println("the center point:"+i);
                                        System.out.println("rho"+localDensity(i));
                                        System.out.println("delta"+deltaArray[i]);
                                        clusterArray[i] = clusterNumbers; // clusterNumbers: the NO. of cluster.
clusterCenters[i]: point i is the center if clusterCenters[i] != -2
                                        centers[clusterNumbers] = i;
                                        clusterNumbers++;



                        }
                }

                for(int j = distanceMatrix[0].length-1; j >= 0; j--){
                        if(clusterArray[index[j]] == -2 &&
distanceMatrix[index[j]][neighbor(index[j])]<cutoffdc){
                                clusterArray[index[j]] = clusterArray[neighbor(index[j])];


                        }
                }
                //initialize group
                for(int i = 0; i < clusterNumbers; i++){
                        group.add(new ArrayList<Integer>());
                }
                if(clusterNumbers != 0){
                        for(int k = 0; k < distanceMatrix[0].length; k++){
                                for(int i = 0; i < clusterNumbers; i++){
                                        if(clusterArray[k] == i){
                                                group.get(i).add(k);
                                        }
```

```
                    }
                }
            }
        }
        */
        //halo for any data set but olivetti face data set

        void halo(){
            double[] boarderRho = new double[clusterNumbers];

          System.out.println("first halo size:"+halo.size());
            for(int i = 0; i < distanceMatrix[0].length-1; i++){
                for(int j = i + 1; j < distanceMatrix[0].length; j++){
                    if(clusterArray[i] != clusterArray[j] && distanceMatrix[i][j] < cutoffdc){
                        if(boarderRho[clusterArray[i]] < localDensity(i)){
                            //System.out.println("******cluster
"+clusterArray[i]+":"+i+":"+localDensity(i));
                            boarderRho[clusterArray[i]] = localDensity(i);
                        }
                        if(boarderRho[clusterArray[j]] < localDensity(j)){
                            //System.out.println("******cluster
"+clusterArray[j]+":"+j+":"+localDensity(j));
                            boarderRho[clusterArray[j]] = localDensity(j);
                        }

                    }
                }
```

```
        }

        for(int i = 0; i <clusterNumbers; i++){
                System.out.println("cluster boardRho"+i+":"+boarderRho[i]);
        }

        for(int j = 0; j < group.size(); j++){
                System.out.println("group "+j);
                for(int k = 0; k < group.get(j).size();k++){
                        System.out.print(localDensity(group.get(j).get(k))+"  ");
                }
                System.out.println();

        }


        System.out.println("first group size:"+group.size());
        for(int i = 0; i < group.size(); i++){
                for(int j = 0; j < group.get(i).size(); j++){

                        if(localDensity(group.get(i).get(j)) < boarderRho[i]){
                                halo.add(j);
                                //halo.get(i).add(j);
                        }

                }

    }
}
```

```
// for olivetti face set
/*
void halo(){
        double[] boarderRho = new double[clusterNumbers];


  System.out.println("first halo size:"+halo.size());
        for(int i = 0; i < distanceMatrix[0].length-1; i++){
                for(int j = i + 1; j < distanceMatrix[0].length; j++){
                        if(clusterArray[i] != clusterArray[j] && distanceMatrix[i][j] < cutoffdc){
                                if(clusterArray[i] != -2 && boarderRho[clusterArray[i]] < localDensity(i)){
                                        //System.out.println("******cluster
"+clusterArray[i]+":"+i+":"+localDensity(i));
                                        boarderRho[clusterArray[i]] = localDensity(i);
                                }
                                if(clusterArray[j] != -2 && boarderRho[clusterArray[j]] < localDensity(j)){
                                        //System.out.println("******cluster
"+clusterArray[j]+":"+j+":"+localDensity(j));
                                        boarderRho[clusterArray[j]] = localDensity(j);
                                }


                        }
                }
        }


        for(int i = 0; i <clusterNumbers; i++){
                System.out.println("cluster boardRho"+i+":"+boarderRho[i]);
        }


        for(int j = 0; j < group.size(); j++){
```

```
                System.out.println("group "+j);
                for(int k = 0; k < group.get(j).size();k++){
                        System.out.print(localDensity(group.get(j).get(k))+"  ");
                }
                System.out.println();

        }


        System.out.println("first group size:"+group.size());
        for(int i = 0; i < group.size(); i++){
                for(int j = 0; j < group.get(i).size(); j++){

                        if(localDensity(group.get(i).get(j)) < boarderRho[i]){
                                halo.add(j);
                                //halo.get(i).add(j);
                        }

                }

  }
}

*/
        void print(){

                System.out.println("distance matrix is:");
                for(int i = 0; i < distanceMatrix.length; i++){
                        for(int j = i+1; j < distanceMatrix.length; j++){
                                System.out.print(distanceMatrix[i][j]+" ");
```

```
            }
            System.out.println();
}

System.out.println("index of sorted array is:");
for(int i = 0; i < distanceMatrix[0].length; i++){
            System.out.print(index[i]);
}

System.out.println();
System.out.println("delta is:");
for(int i = 0; i < distanceMatrix[0].length; i++)
            System.out.print(deltaArray[i]+",");

System.out.println();
System.out.println("rho is:");
for(int i = 0; i < distanceMatrix[0].length; i++)
            System.out.print(localDensity(i)+",");

System.out.println();

System.out.println("neighbors:");

for(int i = 0; i < distanceMatrix[0].length; i++)
            System.out.print(i+":"+neighbor(i)+",");


System.out.println();
System.out.println("clusterNumbers:"+clusterNumbers);
System.out.println();
```

```
System.out.println("clusters:");

                for(int i = 0; i < distanceMatrix[0].length; i++)
                        System.out.print(i+":"+clusterArray[i]+",");

                System.out.println();

System.out.println("centers:");

                for(int i = 0; i < distanceMatrix[0].length; i++){
                        if(centers[i] != -1){
                                System.out.print(i+":"+centers[i]+",");
                        }

                }


                System.out.println("groups are following:");
                for(int i = 0; i < group.size(); i++){
                        System.out.println("the cluster "+i+":");
                        for(int j = 0; j < group.get(i).size(); j++){
                                System.out.print(group.get(i).get(j)+",");
                        }

                        System.out.println();

                }

        }
```

```
//centers decision graph
public void plot(){

        double[][] positionMatrix=MDSJ.classicalScaling(distanceMatrix); // apply MDS
        //decision graph
        try{
                File file = new File("d:/cs298/positionMatrix_iris.txt");
                if(!file.exists()){
                        file.createNewFile();
                }
                FileWriter fw = new FileWriter(file.getAbsoluteFile());
                BufferedWriter bw = new BufferedWriter(fw);
                for(int i = 0; i < distanceMatrix.length; i++){
                        bw.write(positionMatrix[0][i]+","+positionMatrix[1][i]);
                        bw.newLine();
                }
                bw.close();
        } catch(Exception e){
                System.out.println(e);
        }


        double[] x = new double[deltaArray.length];
   double[] y = deltaArray;
   for(int i = 0; i < deltaArray.length; i++){
        x[i] = localDensity(i);
   }
   Plot2DPanel plot1 = new Plot2DPanel();
   plot1.addScatterPlot("example_distance", x, y);
```

```
JFrame frame = new JFrame("decision graph");
frame.setSize(600, 600);
frame.setContentPane(plot1);
frame.setVisible(true);
// clusters graph
Plot2DPanel plot2 = new Plot2DPanel();
for(int i = 0; i < clusterNumbers; i++){
        double[] k = new double[group.get(i).size()];
        double[] z = new double[group.get(i).size()];
        for(int j = 0; j < group.get(i).size(); j++){
                k[j] = positionMatrix[0][group.get(i).get(j)];
                z[j] = positionMatrix[1][group.get(i).get(j)];

        }

        plot2.addScatterPlot("clustering", k, z);

}

        double[] haloK = new double[halo.size()];
        double[] haloZ = new double[halo.size()];
        for(int j = 0; j < halo.size(); j++){
                haloK[j] = positionMatrix[0][halo.get(j)];
                haloZ[j] = positionMatrix[1][halo.get(j)];
        }
        System.out.println("halo:**********");
        System.out.println("halo size:"+halo.size());
        for(int i = 0; i < halo.size(); i++){
                System.out.println(halo.get(i));
        }
```

```
    JFrame frame2 = new JFrame("clustering");
    frame2.setSize(600, 600);
    frame2.setContentPane(plot2);
    frame2.setVisible(true);
    /*
    double[] ga = new double[100];
    double[] n = new double[100];
    Plot2DPanel plot3 = new Plot2DPanel();
    for(int i = 0; i < 100; i++){
            ga[i] = localDensity(i)*distanceFromHigherDensity(i);
            n[i] = i;
    }
    plot3.addScatterPlot("ga", n, ga);
    JFrame frame3 = new JFrame("ga");
    frame3.setSize(600, 600);
    frame3.setContentPane(plot3);
    frame3.setVisible(true);
    */

    }

//border region


public static void main(String[] arg){

            long start = System.currentTimeMillis();
            //DistanceMatrix dm = new DistanceMatrix("d:/cs298/100olivetti.txt",100);
            //dm.convertFile();
```

```
                //double[][] temp = dm.getDistanceMatrix();

        //FFSP ffsp = new FFSP(dm.getDistanceMatrix(),0.07, 0.6, 0.2);

                FFSP ffsp = new FFSP("d:/cs298/iris.txt", 1.1, 30, 0.93);
                ffsp.calculateRhoArrays();

                ffsp.sortRhoArray();

                ffsp. calculateDeltaArrays();
                ffsp.cluster();

                ffsp.halo();

                ffsp.print();

                ffsp.plot();

                long end = System.currentTimeMillis();
                long runningTime = end - start;
                System.out.println("running time:"+runningTime);


        }

}

class DistanceMatrix{
        int numberOfElements;
        static double[][] distanceMatrix;
```

```
String dir;
public DistanceMatrix(String dir, int numberOfElements){
        this.numberOfElements = numberOfElements;
        distanceMatrix = new double[numberOfElements][numberOfElements];
        this.dir = dir;
                }

void convertFile(){
        File file = new File(dir);
        try{
                FileInputStream fis = new FileInputStream(file);
                DataInputStream in = new DataInputStream(fis);
                BufferedReader br = new BufferedReader(new InputStreamReader(in));
                String current;
                while((current =br.readLine()) !=null){
                        if (current.length()==0)
                                continue;
                        String[] split= current.trim().split("\\s++");
                        //System.out.println(split[0]+","+split[1]+","+split[2]);
                        distanceMatrix[Integer.parseInt(split[1])-1][Integer.parseInt(split[0])-
1]=Double.parseDouble(split[2]);
                        distanceMatrix[Integer.parseInt(split[0])-1][Integer.parseInt(split[1])-1] =
Double.parseDouble(split[2]);

                        }

                        br.close();

        } catch(Exception e){
                System.out.println(e);
```

```
                }

        }

        double[][] getDistanceMatrix(){
                return distanceMatrix;
        }

}
```

```
////////////////////////////////////////////////////////////////////////////////////////////////////////////////////////

package calculateDistanceMatrix;

import java.lang.*;

import java.io.*;

import java.util.*;


import mdsj.MDSJ;


public class CalculateDistanceMatrix {

        ArrayList<ArrayList<Double>> arr;

        File file;


        public CalculateDistanceMatrix(String dir) {

                file = new File(dir);

                arr = new ArrayList<ArrayList<Double>>();
```

```
}

public double[][] distanceMatrix() {
	try {
		FileInputStream fis = new FileInputStream(file);
		DataInputStream in = new DataInputStream(fis);
		BufferedReader br = new BufferedReader(new InputStreamReader(in));

		String current;
		while ((current = br.readLine()) != null) {
			if (current.length() == 0)
				continue;
			String[] split = current.trim().split("\\,");
			//String[] split = current.trim().split("\\s++");
			ArrayList<Double> temp = new ArrayList<Double>();
			temp.add(Double.parseDouble(split[0]));
			temp.add(Double.parseDouble(split[1]));
			temp.add(Double.parseDouble(split[2]));
			temp.add(Double.parseDouble(split[3]));
			arr.add(temp);
		}

		br.close();

	} catch (Exception e) {
		System.out.println(e);
	}

	double[][] distanceMatrix = new double[arr.size()][arr.size()];
```

```
                for (int i = 0; i < arr.size(); i++) {
                        for (int j = 0; j < arr.size(); j++) {
                                double x = arr.get(i).get(0) - arr.get(j).get(0);
                                double y = arr.get(i).get(1) - arr.get(j).get(1);
                                double z = arr.get(i).get(2) - arr.get(j).get(2);
                                double k = arr.get(i).get(3) - arr.get(j).get(3);
                                double distance = Math.sqrt(x * x + y * y + z * z + k * k);
                                //double distance = Math.sqrt(x * x + y * y);
                                distanceMatrix[i][j] = distance;
                        }
                }

                return distanceMatrix;
        }

        public static void main(String[] args) {
                CalculateDistanceMatrix iris = new CalculateDistanceMatrix("d:/cs298/aggregation_3a.txt");
                double[][] tempMatrix;
                tempMatrix = iris.distanceMatrix();
                double[][] positionMatrix=MDSJ.classicalScaling(tempMatrix); // apply MDS
                for(int i=0; i<positionMatrix[0].length; i++) {  // output all coordinates
                        System.out.println("**********************");
                           System.out.println(positionMatrix[0][i]+" "+positionMatrix[1][i]);
                        }
        }

}
```

```
*****************************Validation*****************************
package clusterValidation;

import java.io.BufferedReader;
import java.io.DataInputStream;
import java.io.FileInputStream;
import java.io.InputStreamReader;
import java.util.ArrayList;

public class ClusterValidation {
	// the input string array is {m1j, m2j, ..., mlj}
	// the sum is sum(m1j + m2j +...+mlj)

	double[] p;
	double[] e;
	ArrayList<int[]> clusterMatrix;
	double[] purity;
	double[] entropy;
	double overallEntropy;
	double overallPurity;
	int[] rowsum;

	public ClusterValidation(String file) {
		clusterMatrix = new ArrayList<int[]>();
		readFile(file);
		//readFile();
		p = new double[clusterMatrix.get(0).length];
		e = new double[clusterMatrix.get(0).length];

		purity = new double[clusterMatrix.get(0).length];
```

```
		entropy = new double[clusterMatrix.get(0).length];
		rowsum = new int[clusterMatrix.get(0).length];
	}

	public void readFile(String file) {

		try {
			FileInputStream fis = new FileInputStream(file);
			DataInputStream in = new DataInputStream(fis);
			BufferedReader br = new BufferedReader(new InputStreamReader(in));

			String current;
			while ((current = br.readLine()) != null) {
				if (current.length() == 0)
					continue;
				String[] split = current.trim().split("\\s++");
				int[] temp = new int[split.length];
				for (int i = 0; i < split.length; i++) {
					temp[i] = Integer.parseInt(split[i]);
					//System.out.println(temp[i]);
				}
				//System.out.println("*******"+clusterMatrix.size());
				clusterMatrix.add(temp);

			}

		} catch (Exception e) {
			System.out.println(e);
		}
//System.out.println(clusterMatrix.get(0));
```

```
        }

/*
public void readFile(){
int[] temp1 = {2, 153};
int[] temp2 = {85, 0};
clusterMatrix.add(temp1);
clusterMatrix.add(temp2);

}
*/
        void execute() {


                //double maxP = 0;
                for (int i = 0; i < clusterMatrix.size(); i++) {
                        int tempsum = 0;
                        for (int j = 0; j < clusterMatrix.get(i).length; j++) {
                                // System.out.println(args[i]);
                                tempsum += clusterMatrix.get(i)[j];
                        }
                        rowsum[i] = tempsum;
                        //System.out.println("sum:" + rowsum);

                        for (int j = 0; j < clusterMatrix.get(i).length; j++) {
                                p[j] = (double) (clusterMatrix.get(i)[j]) / rowsum[i];
                                //System.out.println("p[j]:" + p[j]);

                        }
    double maxP = 0;
```

```
                for (int j = 0; j < clusterMatrix.get(0).length; j++) {
                        if(p[j] != 0)
                                //p[j]*Math.log10(p[j]) / Math.log10(2) = 0;
                                entropy[i] += 0 - p[j] * Math.log10(p[j]) / Math.log10(2);
                        if (p[j] > maxP) {
                                maxP = p[j];
                        }
                }

                purity[i] = maxP;

        }

}

void overall() {
        int totalSum = 0;
        for(int i = 0; i < clusterMatrix.size(); i++){
                totalSum += rowsum[i];
        }

        for(int j = 0; j < clusterMatrix.size(); j++){
                overallEntropy = overallEntropy + (double) rowsum[j]/totalSum *entropy[j];
                overallPurity = overallPurity + ((double)rowsum[j]/totalSum)*purity[j];
        }


}

void print(){
```

```
		//System.out.println("overallPurity"+overallPurity);
		for(int i = 0; i < purity.length; i++){
			System.out.println("purity:"+purity[i]+","+"entropy:" +entropy[i]);
		}
		System.out.println("overallpurity"+":"+overallPurity+","+"overallentropy"+":"+overallEntropy);
	}

	public static void main(String[] args) {
		ClusterValidation cv = new ClusterValidation("d:/cs298/validation/olivetti_Kmeans.txt");
		cv.execute();
		cv.overall();
		cv.print();


	}

}
```