Report

107062546 楊仲愷

Java File, Class Name: KmeansEucli01, kmeans,
KmeansEucli02, kmeans,
KmeansManh01, kmeans,
KmeansManh02, kmeans
MDA_HW03_Kmeans

- Distance

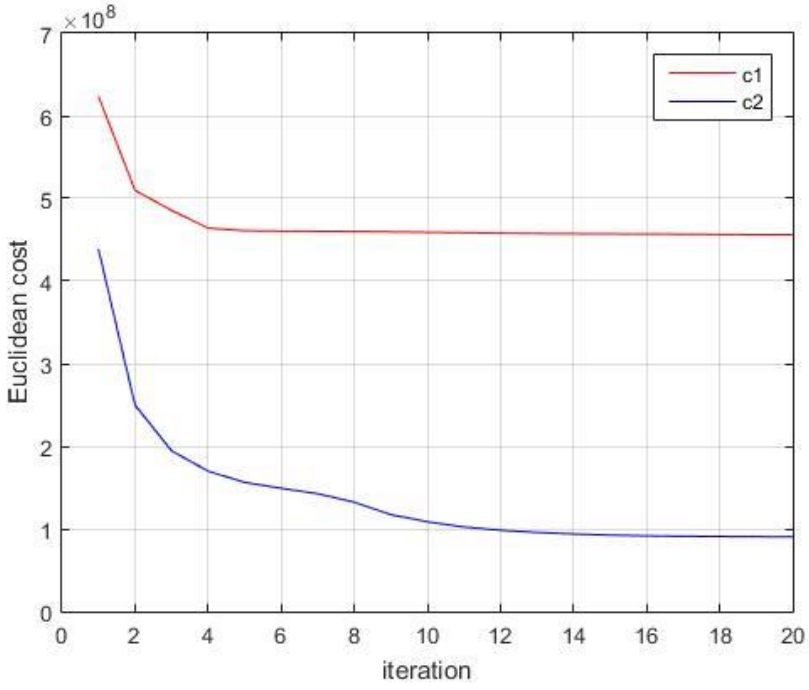
  After clustering, distances between each node

Euclidean	c1	c2	Manhattan	c1	c2
1~2	692.16	15760.00	1~2	2341.00	125.53
1~3	3490.30	14111.00	1~3	11929.00	142.13
1~4	205.75	9045.30	1~4	651.19	97.56
1~5	346.72	5567.70	1~5	496.33	74.76
1~6	512.61	1924.60	1~6	947.74	57.28
1~7	444.73	1100.90	1~7	770.74	37.82
1~8	566.20	402.89	1~8	1056.80	24.56
1~9	1282.80	2105.40	1~9	1260.50	45.85
1~10	307.67	3169.00	1~10	737.71	56.67
2~3	2798.80	11525.00	2~3	9597.40	126.50
2~4	897.66	6743.90	2~4	2778.90	84.97
2~5	1038.80	10193.00	2~5	2830.10	101.10
2~6	204.10	14455.00	2~6	3280.40	127.77
2~7	1136.30	14682.00	2~7	3104.30	120.44
2~8	1257.40	15362.00	2~8	3389.00	123.84
2~9	669.89	13675.00	2~9	2380.50	122.39
2~10	412.08	12597.00	2~10	1605.30	113.68
3~4	3695.10	9545.90	3~4	12323.00	103.39
3~5	3836.90	10883.00	3~5	12421.00	120.89
3~6	4002.70	2234.00	3~6	12871.00	132.37
3~7	3934.90	13208.00	3~7	12696.00	137.02
3~8	4056.10	13786.00	3~8	12979.00	140.01
3~9	2294.60	12509.00	3~9	10776.00	134.58
3~10	3195.90	11938.00	3~10	11197.00	130.37
4~5	142.44	3494.20	4~5	335.95	62.73

4~6	309.51	7718.20	4~6	558.47	95.48
4~7	241.73	7957.80	4~7	382.46	89.95
4~8	363.26	8644.80	4~8	667.53	94.44
4~9	1474.90	6947.80	4~9	1653.80	88.15
4~10	504.63	5876.30	4~10	1379.20	79.45
5~6	167.15	4404.60	5~6	452.86	78.17
5~7	99.55	4492.50	5~7	276.33	65.52
5~8	220.90	5169.90	5~8	561.85	71.58
5~9	1615.90	488.20	5~9	1755.10	69.06
5~10	646.93	2407.90	5~10	1226.70	52.06
6~7	67.91	1182.90	6~7	177.59	43.08
6~8	53.79	1615.80	6~8	110.22	51.79
6~9	1782.20	1313.30	6~9	2205.30	36.86
6~10	814.08	2153.80	6~10	1677.70	58.42
7~8	121.63	698.49	7~8	287.43	28.87
7~9	1715.30	1010.20	7~9	2028.90	25.98
7~10	746.34	2085.50	7~10	1501.00	42.24
8~9	1835.60	702.80	8~9	2314.70	38.74
8~10	67.8231	2768.60	8~10	1786.80	51.13
9~10	975.3204	1080.50	9~10	1006.40	45.41

- K-means with Euclidean distance:

  (1) Figure: cost vs. iteration for two initialization strategies:



(2) Analysis: percentage improvement values:

1. c1: (random initialization)
   improvement
   = (623660345.31 - 455986871.03) / 623660345.31
   = 26.89%

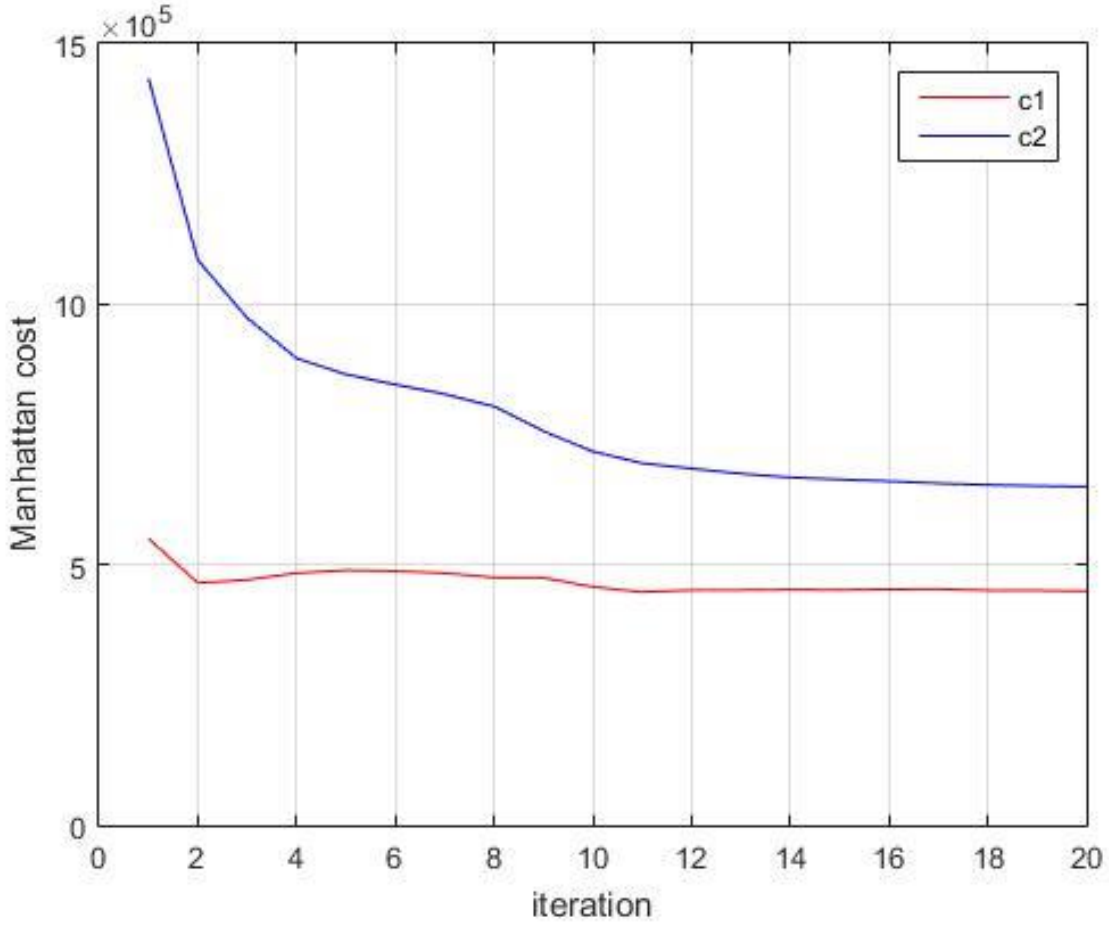
2. c2:
   improvement
   = (438747790.03 - 90216416.18) / 438747790.03
   = 79.44%

(3) Explanation:

In this case, cost of c2 is better than c1 because c2 separate initial centroids, and this will make them converge.

- K-means with Manhattan distance

(1) Figure: cost vs. iteration for two initialization strategies:



(2) Analysis: percentage improvement values:

1. c1: (random initialization)
   improvement
   = (550117.14 - 449011.36) / 550117.14
   = 18.38%

2. c2:
   improvement
   = (1433739.31 - 649689.0132) / 1433739.31
   = 54.69%

(3) Explanation:

In this case, cost of c2 is worse than cost of c1 because c1 is randomly selected; thus, maybe it is the worst case, what it chooses is the smallest cost. If we iterate more times, cost of c2 would be better.

- Program:

(1) Execute Program:

We must upload the files into the directory. File: data.txt, c1.txt, c2.txt.

a. Kmeans in Euclidean with c1 file

```
yarn jar KmeansEucli01.jar kmeans /hw03Eucli01
```

b. Kmeans in Euclidean with c2 file

```
yarn jar KmeansEucli02.jar kmeans /hw03Eucli02
```

c. Kmeans in Manhattan with c1 file

```
yarn jar KmeansManh01.jar kmeans /hw03Manh01
```

d. Kmeans in Manhattan with c2 file

```
yarn jar KmeansManh02.jar kmeans /hw03Manh02
```

(2) Code:

Taking Kmeans in Euclidean with c1 file for example:
First, I put the data of data.txt in map.

```
int num = 0;
while (br.ready()) {
    String line = br.readLine();
    map.put(num, line);
    num++;
}
```

And put the data of c1.txt in map1.

```
    while (br1.ready()) {
        String line1 = br1.readLine();
        String[] number1 = line1.split(" ");

        ArrayList<Double> array1 = new ArrayList<Double>();

        for(int i = 0; i < number1.length ; i ++) {
            array1.add(Double.parseDouble(number1[i]));
        }

        map1.put(num1, array1);
        num1 ++;
    }
```

Then, calculate the distances centroids which are in c1 file between the 10 centroids and the points in data.txt.

```
ArrayList<ArrayList<Double>> compa = new ArrayList<ArrayList<Double>>();
compa = distance(dat, map1, num1);
```

```
public static ArrayList<ArrayList<Double>> distance(ArrayList<ArrayList<Double>> dat, Map<
    ArrayList<ArrayList<Double>> finalans = new ArrayList<ArrayList<Double>>();

    for(int i = 0; i < num1; i ++) {
        ArrayList<Double> array = new ArrayList<Double>();

        for(int k = 0; k < dat.size(); k++) {
            double total = 0;
            for(int j = 0; j < dat.get(i).size(); j ++) {
                double temp = pow( dat.get(k).get(j) - map1.get(i).get(j), 2);
                total = total + temp;
            }
            array.add(total);
        }
        finalans.add(array);
    }

    return(finalans);
}
```

After that, we will get all the distances between points and centroids. We compare every distance and get the closest centroid for every node.

```
ArrayList<Integer> comp = new ArrayList<Integer>();

comp = comparison(compa);
```

```
public static ArrayList<Integer> comparison(ArrayList<ArrayList<Double>> compa) {
    ArrayList<Integer> comp = new ArrayList<Integer>();

    for(int i = 0; i < compa.get(0).size(); i ++) {
        double min = 1000000000.0;
        int mini = -1;

        for(int j = 0; j < compa.size(); j ++) {
            if(min >= compa.get(j).get(i)) {
                min = compa.get(j).get(i);
                mini = j;
            }
        }

        comp.add(mini);
    }
    return(comp);
}
```

And we can output a file.

```
0 0 0.64 0.64 0 0.32 0 0 0 0 0 0 0.64 0 0 0 0.32 0 1.29 1.93 0 0.96 0 0 0 0 0 0 0
1 0.21 0.28 0.5 0 0.14 0.28 0.21 0.07 0 0.94 0.21 0.79 0.65 0.21 0.14 0.14 0.07 0
2 0.06 0 0.71 0 1.23 0.19 0.19 0.12 0.64 0.25 0.38 0.45 0.12 0 1.75 0.06 0.06 1.0
3 0 0 0 0 0.63 0 0.31 0.63 0.31 0.63 0.31 0.31 0.31 0 0 0.31 0 0 3.18 0 0.31 0 0
4 0 0 0 0 0.63 0 0.31 0.63 0.31 0.63 0.31 0.31 0.31 0 0 0.31 0 0 3.18 0 0.31 0 0
5 0 0 0 0 1.85 0 0 1.85 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
6 0 0 0 0 1.92 0 0 0 0 0.64 0.96 1.28 0 0 0 0.96 0 0.32 3.85 0 0.64 0 0 0 0 0 0 0
7 0 0 0 0 1.88 0 0 1.88 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
8 0.15 0 0.46 0 0.61 0 0.3 0 0.92 0.76 0.76 0.92 0 0 0 0 0 0.15 1.23 3.53 2 0 0 0
9 0.06 0.12 0.77 0 0.19 0.32 0.38 0 0.06 0 0 0.64 0.25 0 0.12 0 0 0.12 1.67 0.06
7 0 0 0 0 0 0 0.96 0 0 1.92 0.96 0 0 0 0 0 0 0.96 3.84 0 0.96 0 0 0 0 0 0 0 0 0 0
3 0 0 0.25 0 0.38 0.25 0.25 0 0 0 0.12 0.12 0.12 0 0 0 0 0 1.16 0 0.77 0 0 0 0 0
0 0 0.69 0.34 0 0.34 0 0 0 0 0 0 0.69 0 0 0 0.34 0 1.39 2.09 0 1.04 0 0 0 0 0 0 0
```

The first number in every line means the centroids in these 10. And the numbers after the number of centroids are distances in every dimension.

Now, let's get into mapper. We can take advantage of the file just output and we set the centroids number with key and the string after that with value.

```
for(int i = 1; i < a.length; i ++) {
    if(i < a.length - 1) {
        string = string + a[i] + " ";
    }
    else {
        string = string + a[i];
    }
}

file1.set(a[0]);
file2.set(string);

context.write(file1, file2);
```

Therefore, we will get the file like this in every iteration.

```
0       0.11368686868686867 0.13171717171717165 0.32595959595959595 0.2176094
1       0.10509677419354843 0.13122580645161286 0.338 0.21593548387096773 0.2
2       0.2707619047619047 0.18285714285714277 0.28961904761904744 0.04504761
3       0.1113716814159292 0.16393805309734513 0.43451327433628306 0.00650442
4       0.06116666666666666 0.1556666666666667 0.2444583333333333 0.296125 0.
```

Then, we go to the reducer part. First, store the string and split it into array list.

```
for (Text val : values) {
    String num[] = val.toString().split(" ");

    ArrayList<Double> array1 = new ArrayList<Double>();

    for(int i = 0 ; i < num.length; i ++) {
        array1.add(Double.parseDouble(num[i]));
    }
    array.add(array1);
    count++;
}
```

After storing into array list, we can calculate the new centroid and update. First, we initialize the array list. And we add the values of whole points in the same centroid. Finally, we use the count and get the final centroid value.

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

for(int i = 0; i < array.get(0).size(); i ++) {
    ans.add(0.0);
}

for(int i = 0; i < count; i ++) {
    for(int j = 0; j < array.get(i).size(); j ++) {
        ans.set(j, ans.get(j) + array.get(i).get(j));
    }
}

for(int i = 0; i < ans.size(); i ++) {
    ans.set(i, ans.get(i)/count);
}

word1.set(key);
```

In the final of program, we can output a file to list the whole distance between each centroid.

```
Path out = new Path(finaloutpath + "/finalanswer.txt");
FileSystem fs0 = FileSystem.get(new Configuration());
FSDataOutputStream outputStream = fs0.create(out);

for(int i = 0; i < num1; i ++) {
    for(int j = i + 1; j < num1; j ++) {
        double dis = distance(map.get(i), map.get(j));
        String string1 = i + "-" + j + ":" + dis;
        outputStream.writeBytes(string1);
        outputStream.write('\n');
    }
}
```

```java
public static double distance(ArrayList<Double> list1, ArrayList<Double> list2) {
    double total = 0.0;

    for(int i = 0; i < list1.size(); i ++) {
        double temp = pow((list1.get(i) - list2.get(i)), 2);
        total = total + temp;
    }

    return(total);
}
```

In the final, we will get the file like this.

```
0-1:479082.5399183721
0-2:1.2181905376356192E7
0-3:42333.17724050414
0-4:120213.94189946611
0-5:262771.31547539425
0-6:197785.66284760908
0-7:320584.6960772738
0-8:1645501.0395388512
0-9:94660.2925410118
1-2:7833287.33516361
1-3:805791.6557660815
1-4:1079161.3038366504
1-5:1449804.30927412
1-6:1291239.8323014905
```