Project 2 Documentation

Emily Hao, eyhao@wpi.edu
Daojun Liang, dliang2@wpi.edu

Environment: IntelliJ + Maven
Note: All the file paths are from personal computer, if you want to run these jave files, you may need to change the path of the files.

Question 1:
Input: datasetPoint.txt, datasetRec.txt

The main idea of solving problem1:
Because the combinations of points and rectangles are huge. Thus, we first pass a window value (In our test, the window value is “5, 20, 100, 100”) in map functions to filter points and rectangles outside the window. And in the reduce phase, we perform spatial join for the points and rectangles that have been filtered before.

Note: In main function, we use conf.set( ) to set the value of the window, and in map functions, we use conf.get( ) to get the value of the window.

In our solution, we use two map functions and one reduce function.

Map functions:

(1) pointMapper: deal with the points dataset
 The input of pointMapper: points we have created before
 The output key of pointMapper: “1”
 The output value of pointMapper: “point” + “,” + x_axis of one point, y_axis of one point.
 For example, one record of the pointMapper’s output can be: 1 point, 50, 75
(2) recMapper: deal with the rectangles dataset
 The input of recMapper: rectangles we have created before
 The output key of recMapper: “1”
 The output value of recMapper: “rec” + “,” + “bottomLeft_x, bottomLeft_y, h, w”
 For example, one record of the recMapper’s output can be : 1 rec, 42, 43, 6, 4

Reduce function:

The input of reduce function is the output of pointMapper and recMapper. In our case, the input key of reduce function is “1” because we want to join points and rectangles.

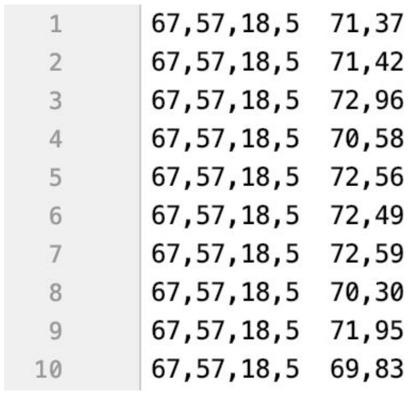
The main idea of reduce function:
1. We have two arraylist, one is for storing points, one is for storing rectangles.

2. After getting all the points and all the rectangles, we started to loop all the points inside each rectangle to see if the points are in the rectangle. If yes, we output the rectangle and corresponding points.

The output key of reduce function: rectangles
The output value of reduce function: points

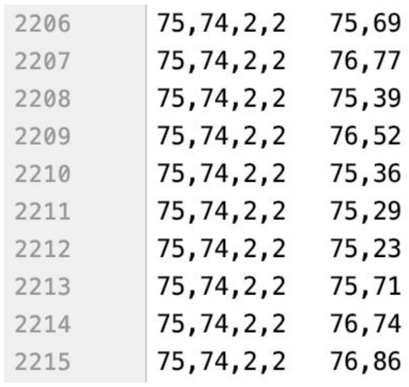
The screenshots of the result:

(The first 10 lines)



```
67,57,18,5  71,37
67,57,18,5  71,42
67,57,18,5  72,96
67,57,18,5  70,58
67,57,18,5  72,56
67,57,18,5  72,49
67,57,18,5  72,59
67,57,18,5  70,30
67,57,18,5  71,95
67,57,18,5  69,83
```

(The last 10 lines)

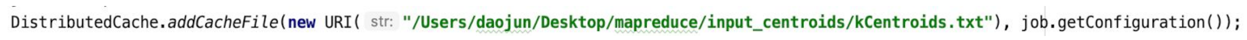


```
75,74,2,2   75,69
75,74,2,2   76,77
75,74,2,2   75,39
75,74,2,2   76,52
75,74,2,2   75,36
75,74,2,2   75,29
75,74,2,2   75,23
75,74,2,2   75,71
75,74,2,2   76,74
75,74,2,2   76,86
```

Question 2:
Input: datasetPoint.txt

In our first iteration:

1. We add the file “kCentroids.txt” into DistributedCache.



```
DistributedCache.addCacheFile(new URI( str: "/Users/daojun/Desktop/mapreduce/input_centroids/kCentroids.txt"), job.getConfiguration());
```

2. In MyMapper function:
   The setup procedure: get [the order of the initial kCentroids, the x-axis of one centroid, the y-axis of one centroid]. (using a hashmap data structure)
   For example, [1, 4, 5] means the first input centroid and its x-axis is 4, its y-axis is 5. [2, 5,9] means the second input centroid and its x-axis is 5, its y-axis is 9.

The input value is points.
In the map, we first get how many centroids we have.
For every point, we calculate its distance to every centroid and we find the nearest one.
The output key of map function: the x-axis and y-axis of centroids
The output value of map function: the points that are nearest to the corresponding centroids

3. In MyReducer function:
   Count the x-axis and y-axis of new centroids.

How to do iterations?
There are two problems we need to solve:
1. We need to compare the new centroids with the centroids in the second last iteration.
2. In the new iteration, we need to cache the centroids in the last iteration and we need to know how to get the new centroids in map function.

For problem one:
We write two classes:

```java
public boolean isSameCentroids(String initlCentroids, String newCentroids) throws IOException {
```

```java
public boolean isSameCentroidsFS(String oldCentroids, String newCentroids) throws IOException {
```

The difference between the two classes is the first class is used after the first iterations to compare a .txt file (the initial centroids) and a hdfs file. The first class is used for only once because the output of later iterations are all hdfs files. So for later iteration, we write the second class to compare two hdfs files.

Note: the input parameter are the file path of old and new centroids.

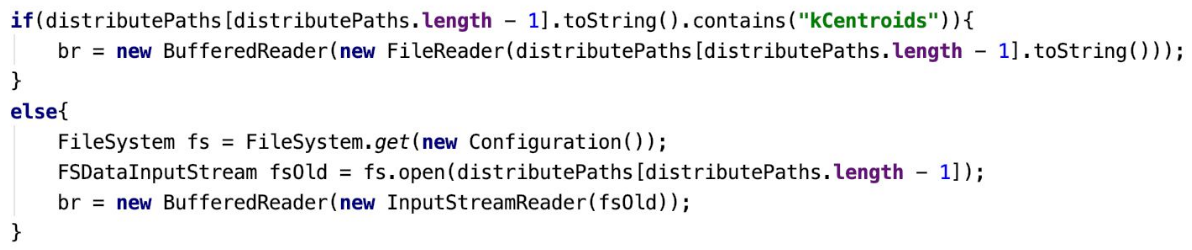
For problem two:
After each iteration, we cache the output of new centroids

```java
DistributedCache.addCacheFile(new URI("/Users/daojun/Desktop/mapreduce/outputKMeans"+ iter +"/part-r-00000"), job1.getConfiguration());
```

In map function, we use the code below to get the most recent cached file.

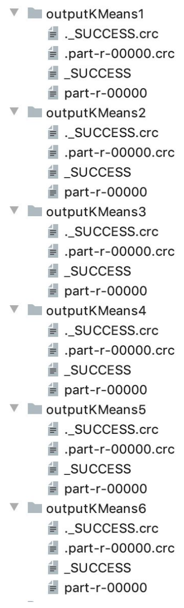
```java
distributePaths[distributePaths.length - 1]
```

But we need to know, the first time we cache a .txt file and then later we all cache hdfs file. For the two types of file, we need to differentiate them and read the contents of the file in different ways.

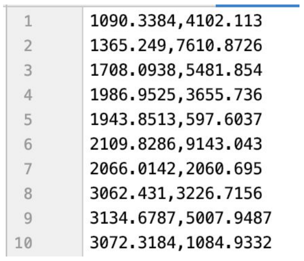

```
if(distributePaths[distributePaths.length - 1].toString().contains("kCentroids")){
    br = new BufferedReader(new FileReader(distributePaths[distributePaths.length - 1].toString()));
}
else{
    FileSystem fs = FileSystem.get(new Configuration());
    FSDataInputStream fsOld = fs.open(distributePaths[distributePaths.length - 1]);
    br = new BufferedReader(new InputStreamReader(fsOld));
}
```

The **if part** get the initial centroids and the **else part** get the latest centroids in hdfs files.

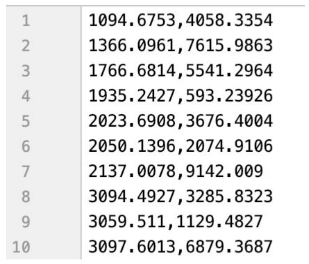
Screenshots of output:
We output six new centroids files after six iterations.


```
▼ outputKMeans1
    ._SUCCESS.crc
    .part-r-00000.crc
    _SUCCESS
    part-r-00000
▼ outputKMeans2
    ._SUCCESS.crc
    .part-r-00000.crc
    _SUCCESS
    part-r-00000
▼ outputKMeans3
    ._SUCCESS.crc
    .part-r-00000.crc
    _SUCCESS
    part-r-00000
▼ outputKMeans4
    ._SUCCESS.crc
    .part-r-00000.crc
    _SUCCESS
    part-r-00000
▼ outputKMeans5
    ._SUCCESS.crc
    .part-r-00000.crc
    _SUCCESS
    part-r-00000
▼ outputKMeans6
    ._SUCCESS.crc
    .part-r-00000.crc
    _SUCCESS
    part-r-00000
```

The fifth output file (the first 10 lines):


```
1090.3384,4102.113
1365.249,7610.8726
1708.0938,5481.854
1986.9525,3655.736
1943.8513,597.6037
2109.8286,9143.043
2066.0142,2060.695
3062.431,3226.7156
3134.6787,5007.9487
3072.3184,1084.9332
```

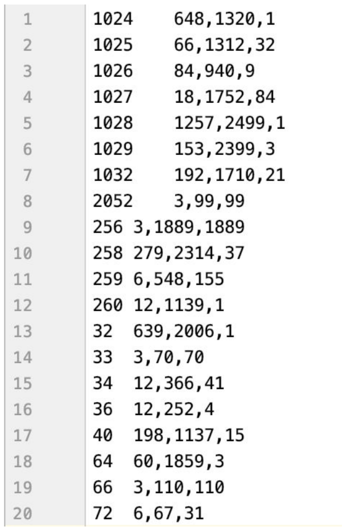
The 6th output file (the first 10 lines):


```
1094.6753,4058.3354
1366.0961,7615.9863
1766.6814,5541.2964
1935.2427,593.23926
2023.6908,3676.4004
2050.1396,2074.9106
2137.0078,9142.009
3094.4927,3285.8323
3059.511,1129.4827
3097.6013,6879.3687
```

Question 3:
This question takes the input: airfield.json that is given to us
The final output is flag, max elevation, min elevation.

I looked over FileInputFormat, NLineInputFormat and RecordReader classes to learn how to write my own jsonInputFormat class.
I extended FileInputFormat class and RecordReader class for the two classes I created. I override getSplits function from FileInputFormat class to the logic of this question by deleting some unnecessary code and calculated the number of blocks for each split.
I override nextKeyValue function from RecordReader class by calling a helper function that I created. I didn’t change much of the logic for the other functions in the RecordReader class.

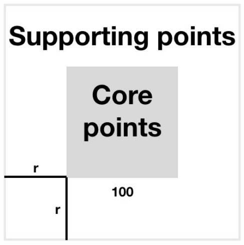
Screenshot of output:



```
1024	648,1320,1
1025	66,1312,32
1026	84,940,9
1027	18,1752,84
1028	1257,2499,1
1029	153,2399,3
1032	192,1710,21
2052	3,99,99
256	3,1889,1889
258	279,2314,37
259	6,548,155
260	12,1139,1
32	639,2006,1
33	3,70,70
34	12,366,41
36	12,252,4
40	198,1137,15
64	60,1859,3
66	3,110,110
72	6,67,31
```

Question 4:
In this question, we divide the x-y plane into different boxes. Each box’s size is 100 * 100. For box (i, j), its range of x is [100 * i + 1, 100 * (i+1)], its range of y is [100 * j + 1, 100 * (j+1)].
In this problem, we divide the x-y plane into 100 boxes.

In map function:
For every box, we calculate the core points and supporting points. The core points are inside each box, but the supporting points are outside the box.



For each box, we use tag “c” to denote the points are core points and we use tag “s” to denote the points are supporting points.

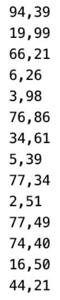
The output key of map function: box (i, j)
The output value of map function: “c,” + core points or “s,” + supporting points

In reduce function:

For every box, we loop for every core point inside the box and calculate how many points are in its circle using both core points and supporting points. If the number < k, then the point is an outlier.

In our test, we set r equals to 10, and k equals to 50.

The screenshot of part of the output:



```
94,39
19,99
66,21
6,26
3,98
76,86
34,61
5,39
77,34
2,51
77,49
74,40
16,50
44,21
```

Note:

1. In this problem, each box’s size is 100 * 100, and it’s likely that r will be larger than 100. In this case, our map function won’t work. So the box size is better associated with r value, for example, the box size can be 10r * 10r. For a simple solution, we just assume that the value of r will be less than 100. Anyway, our code implemented the high level idea of this problem.
2. The size of the box is a key factor influencing the speed of the mapreduce job. At first, we set the size of the box as 1000 * 1000 while keeping r as 10 and k as 50. The execution speed is quite low. After we set the size of the box as 100 * 100, the execution speed is obviously faster.