**DS503/CS585 - Big Data Management**

**Project 2 Report**

team 6: Kavin Chandrasekaran

Jonathan Wang

Jiaxing Zhang

**Problem 1(Spatial Join)**

**Step 1: Create Dataset\_P and Dataset\_R**

Dataset\_P:

generate\_coordinate() method will create random number between 1 and 10000 every time you call it.

The coordinates for Dataset\_P are 10000000 points in total. The size of file is 112MB

Dataset\_R:

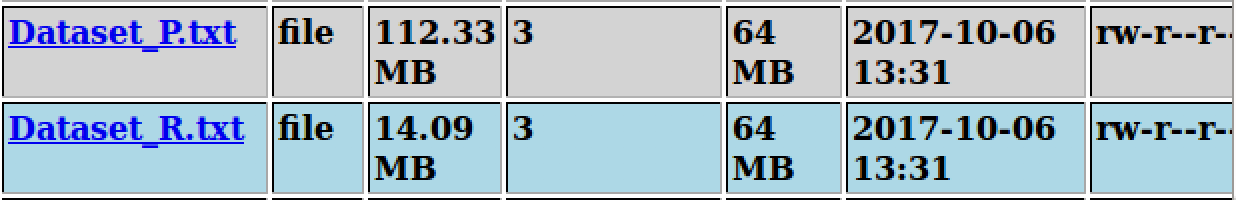
generate\_bottom\_left\_coordinate() will generate the bottom left coordinate first, and generate\_bottom\_right() and generate\_top\_left() could take the x and y value from generate\_bottom\_left(). Since we have generated three points of rectangle, we could know the fourth point.

The coordinates for Dataset\_P are 500000 points in total. The size of file is 14MB

Upload command:

**hadoop fs –put ~<your dataset path> /user/hadoop/input/Project2/Dataset/Dataset\_P.txt**

**/user/hadoop/input/Project2/Dataset/Dataset\_R.txt**



**Step 2: Join the points from Dataset\_P and the rectangles from Dataset\_R.**

User optional input:

Setting the input using configuration.set():

Screen Shot 2017-10-06 at 1.15.11 PM.png

Retrieving user input:

Screen Shot 2017-10-06 at 1.15.25 PM.png

Spatial\_Join.java

Before starting the map-reduce, we use set\_up function to load in the small file (in our case is Dataset\_R) into memory, and add all the rectangle into a java hashmap. The key for map is ri , and value is string contains left bottom and top right coordinate of the rectangle.

Map phase:

We read the Dataset\_P in, and simply write out key-value pairs. The key is x coordinate, and value is y coordinate, both are in Text type.

Reduce phase:

Use a for loop to see if the points from the mapper is contained inside any rectangle.

IsContained:

We write our own function to determine whether the points from Dataset\_P has any spatial join with all the rectangles in the hashmap. If the points in located inside the rectangle, then write out to context.

Run command:

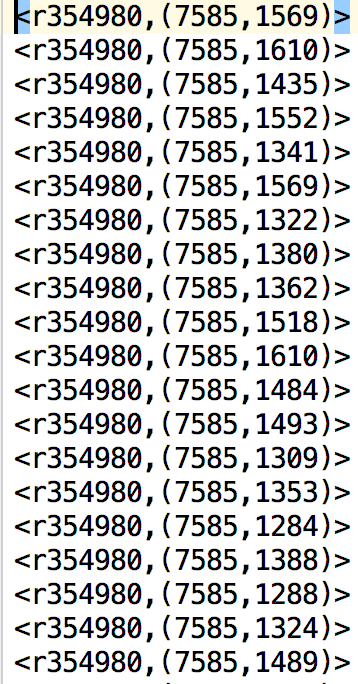
**mkdir spatial\_join**

**javac -classpath /usr/share/hadoop/hadoop-core-1.2.1.jar -d spatial\_join ./\*.java**

**jar –cvf ./Spatial\_Join.jar –C spatial\_join/ .**

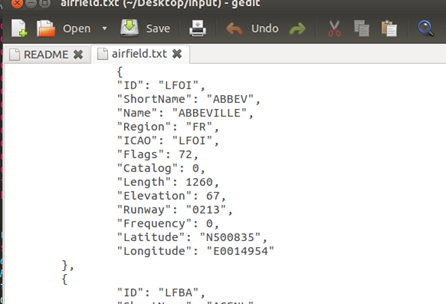
**hadoop jar Spatial\_Join.jar Job2 <Your input path> <Your output path>**

Screenshot of result



**Problem 2(Customer Input Format)**

**Step 1 Download the JSON dataset and upload it to HDFS**



Upload command:

**hadoop fs –put ~<your dataset path> /user/hadoop/input/Project2/Dataset/airfield.txt**

**Step 2 (Map job with a Customer input format)**

File: JSONReader.java

Job2.java

JSONReader includes JSONInputFormat and JSONRecordReader class. By overriding several methods, we can directly read data from the JSON file.

In job2, the mapper will group data based on the corresponding value of “Elevation” records, and the reducer will do the counting job.

Run command:

Go the directory where the job2 is located

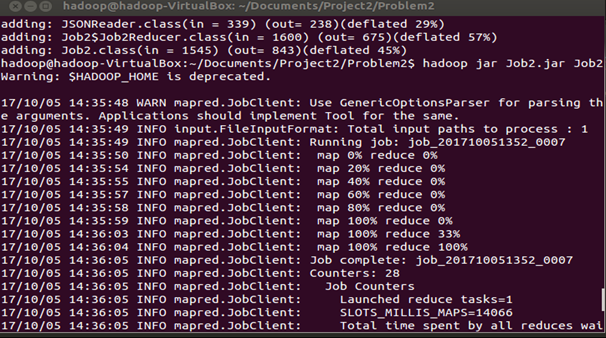
**mkdir Job2\_classes**

**javac -classpath /usr/share/hadoop/hadoop-core-1.2.1.jar -d Job2\_classes ./\*.java**

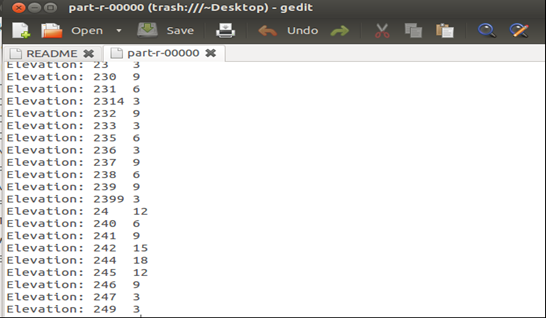
**jar –cvf ./Job2.jar –C Job2\_classes/ .**

**hadoop jar Job2.jar Job2 <Your output path>**

Screenshot of Mapper Reduce

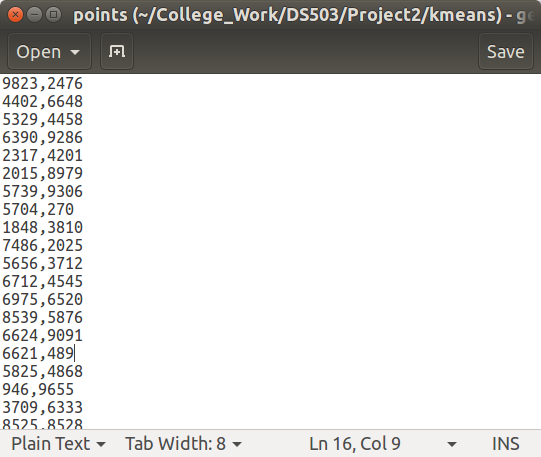


Screenshot of result



**Problem 3:**

For the problem3 we implemented kmeans to cluster a set of random points into different clusters. To generate the data set we created a program to generate point objects and write it to file until the size of the file was more than 100MB. Point object is user-defined, which has variables like the co-ordinates and has a distance function which calculates the distance to another given point. The below image shows a sample of the points file, where each line denotes a point.



There are 6 different parts that were mentioned in the question about implementing the algorithm. The code is in kmeans.jar file. Each part can be run from the same file(kmeans.jar), by specifying the corresponding class names. The classes for each part is named in the pattern kmeans\_#. For example, to run the first part, the below command can be issued.

**>hadoop jar College\_Work/DS503/Project2/kmeans.jar kmeans\_1**

**Enter input parameters \*input file\* \*output folder\* \*number of clusters\* [\*combiner\*]**

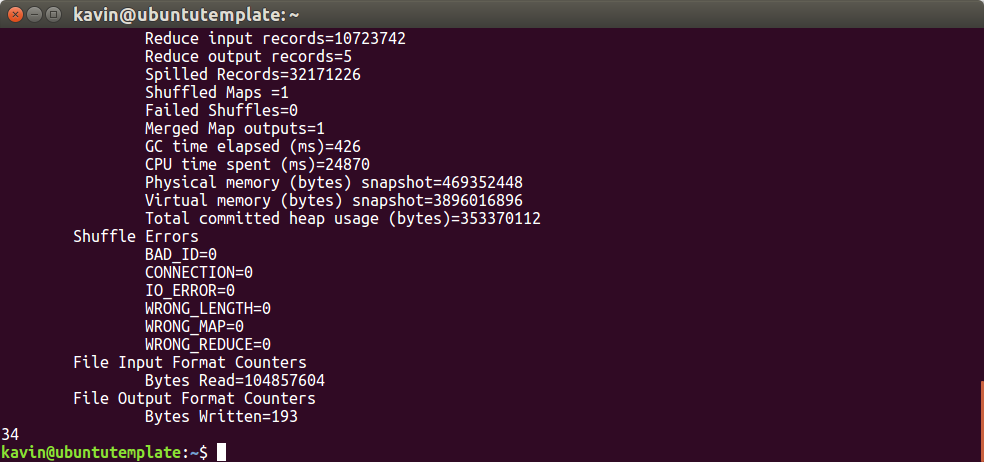
The program mentions missing inputs and the order it should be entered.

**>hadoop jar College\_Work/DS503/Project2/kmeans.jar kmeans\_1 input/project2/kmeans/points output/kmeans/ 5**

The output centroids are written in a file named “centroids” in the specified output directory.

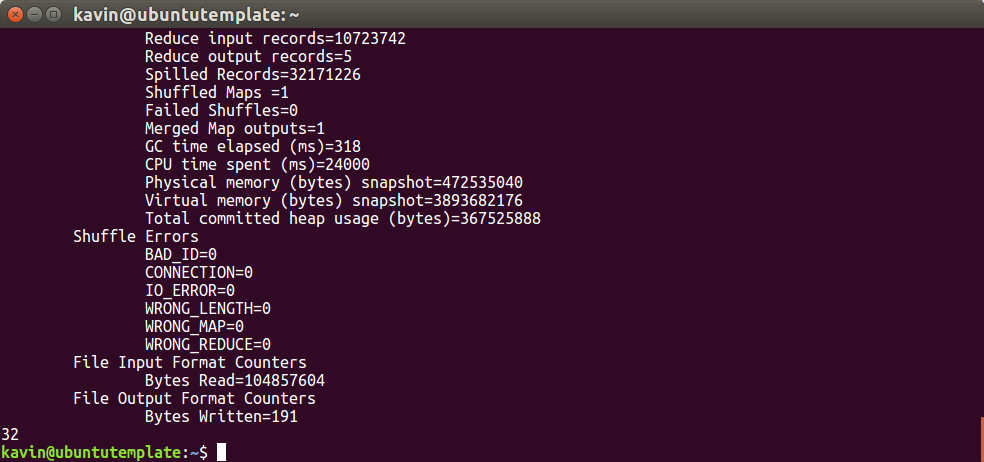
1) In the first part we implemented a k-means algorithm which only ran for one iteration. The number of clusters, mentioned in the parameters, was used to create the corresponding number of initial random centroids for the clusters. The centroids are created such that the centroid is associated with a cluster number which they belong to, and it is also stored in the file along with the centroid. The file containing the cluster centers are added to the cache, so that it can be accessed by multiple mappers. Since the size of the centroids file will be small, we used the cache. During the setup phase, the file from cache will be read by the mapper and the values will be stored in a hashmap. Based on the random centroids, each point in the points file is used to calculate centroid they are closest to and are assigned to that cluster. The map function reads in each point from the input file and creates a Point object. Each point from the hashmap containing the centroids is retrieved and the distance between all the centroids and the point is calculated. Then the id of the closest centroid is used as the key of the output from the map function and the value is the point itself. Then based on all the points in a cluster, a new centroid is calculated by taking the average of all the points. The new centroids are written out; and this marks the end of one iteration. At the end of the program, the time taken for the entire map-reduce process is displayed out in seconds, which will be used to compare the performance.

**hadoop jar College\_Work/DS503/Project2/kmeans.jar kmeans\_1 input/project2/kmeans/points output/project2/kmeans/ 5**

****

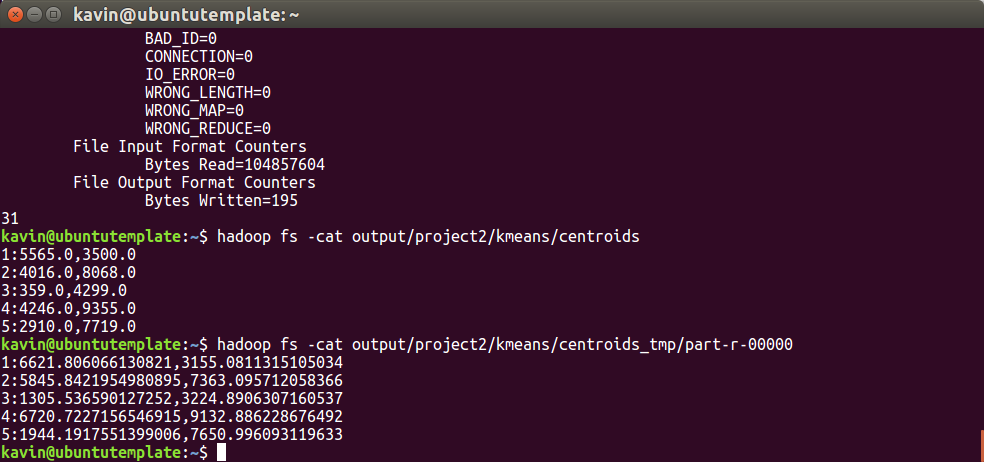
Combiner can be used to improve performance when available. To enable the use of the combiner, the word combiner should be mentioned as the last argument after all the required parameters. The format is same for all the following steps. If the use of the combiner s applicable, it will be used and if not, it will be ignored.

**hadoop jar College\_Work/DS503/Project2/kmeans.jar kmeans\_1 input/project2/kmeans/points output/project2/kmeans/ 5 Combiner**

****

The temporary files are stored in folder “centroids\_tmp” in the mentioned output folder.

Only in this step, to show the difference between the initial centroids and the centroids calculated after one iteration, the cleanup is not performed and the output is written in centroids\_tmp in the output folder.



The output from centroids file shows the initial centroids and the result in centroids\_tmp shows the new centroids. The number before the “**:**” indicates the cluster number.

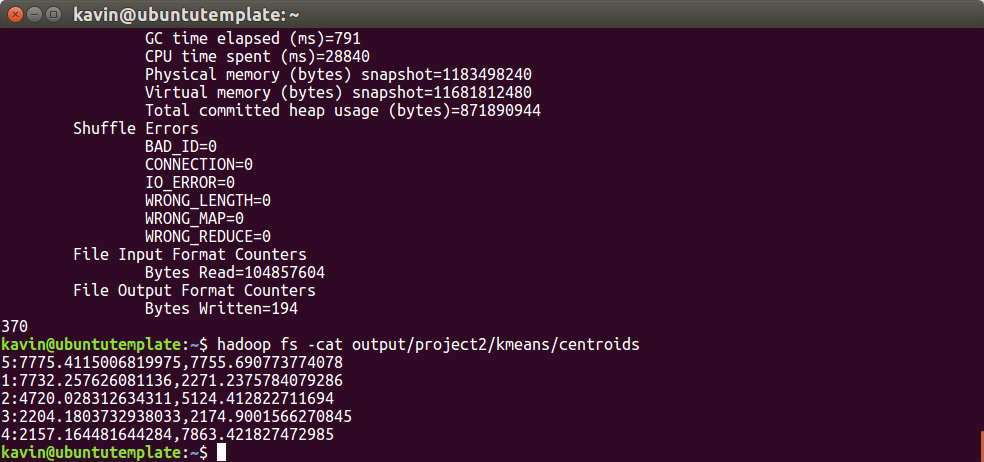
If the execution is interrupted the temporary folder might not be cleaned, in that case the following command needs to be executed, before running any other step.

**Hadoop fs -rm -r <output folder>/centroids\_tmp**

The cleanup step is the same every step here follows.

2) In the second step we run the algorithm in step 1, for a specific number of times, which is specified by the user. Since we are running the same job multiple times, we write the intermediary output to a temporary folder, which gets cleaned after each iteration. To accommodate the use of multiple reducers, after each iteration, we parse through all the files in the temporary output directory and create a new centroids file in the output directory and then remove the temporary directory. This way, the job in the next iteration will have the new centroids file in its cache.

**hadoop jar College\_Work/DS503/Project2/kmeans.jar kmeans\_2 input/project2/kmeans/points output/project2/kmeans/ 5 10 Combiner**

****

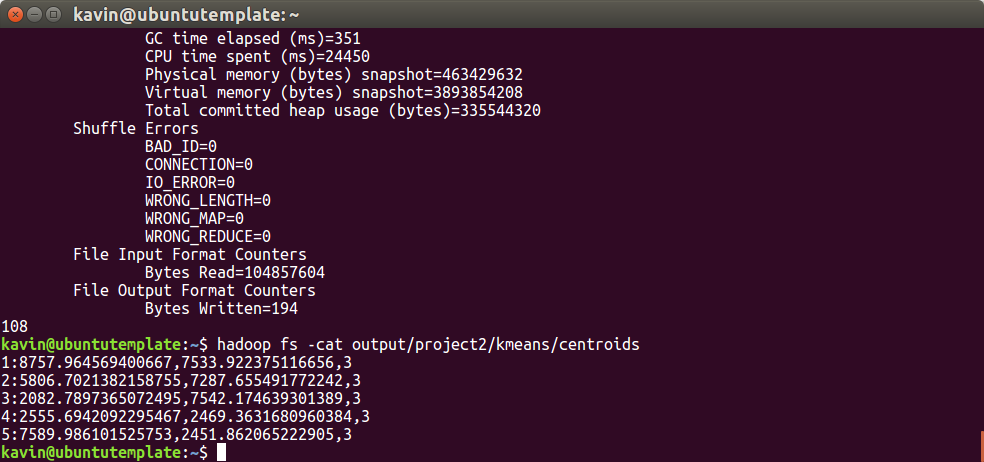
This shows the resulting centroids from the kmeans algorithm after 10 iterations.

3) For the third step we used a user specified threshold for the variability of the centroids between each iteration. If the distance between the old centroid and the new centroid is within the given threshold, then the execution is stopped. This algorithm also takes the maximum number of iterations to run as a parameter from the user. If convergence is not reached based on the threshold, then it is iterated for the maximum number of times.

To accomplish this, we used the same code from previous step and added a part where the distance between old and new centroids are calculated. Both the old and new centroids are read in from the HDFS and stored in hashmaps and then the corresponding distances are calculated. Detecting the convergence is done in the main part of the program. Convergence is considered to be reached only when all the new centroids are within the threshold from the old centroids. Once convergence is reached, the new centroids are written to the centroids file and the temporary files are cleaned.

**hadoop jar College\_Work/DS503/Project2/kmeans.jar kmeans\_3 input/project2/kmeans/points output/project2/kmeans/ 5 10 500**

The threshold here is set intentionally higher to show the convergence. The output centroids from this process will also include the iteration it stopped to show early termination based on the convergence.



Since the convergence threshold was set to 500, convergence was reached after the third iteration and it is shown in the output for demonstration purposes. It also shows the time taken was much lower since only 3 iterations were performed.

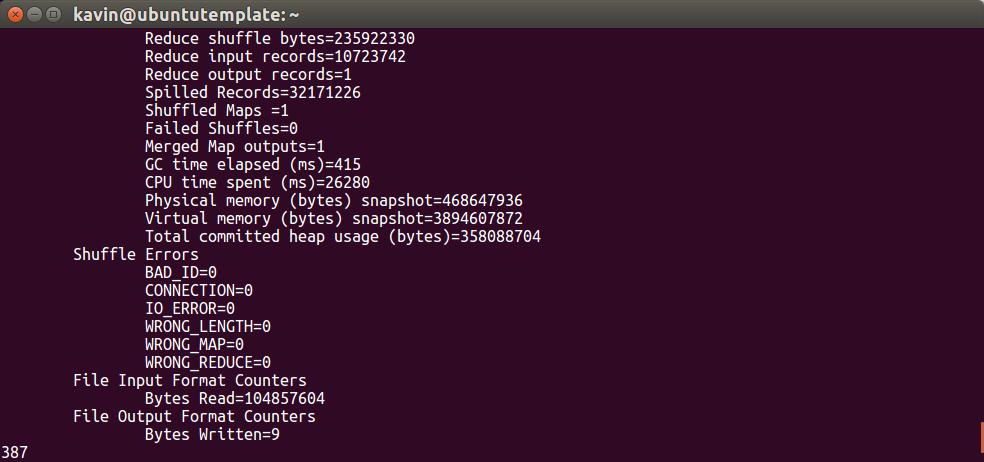
4) For optimization, we tried using the different methods like, using a combiner, using a single reducer, and loop control from different places. The use of the combiner was done with the code from the previous step with not many changes. Using single reducer with a combiner was faster than using multiple reducers, with calculating for convergence still in the main. Please note the number of reducers were controlled by giving it as a parameter for kmeans\_3.

**>hadoop jar College\_Work/DS503/Project2/kmeans.jar**

**Enter input parameters \*input file\* \*output folder\* \*number of clusters\* \*maximum number of iterations\* \*threshold for convergence\* [\*combiner\*] [\*number of reducers\*]**

We also tried to check for convergence in the cleanup section in the reducer, so that the amount of data coming from the reducer might be less. The reducer cleanup process checks convergence, updates the centroids and writes to the temporary output whether convergence is reached or not(true of false). In the main program the iteration is continued if the output is False. But this process took more time than checking for convergence in the main program. We couldn’t use a combiner since the reducer had to be modified to check for convergence.

**hadoop jar College\_Work/DS503/Project2/kmeans.jar kmeans\_4 input/project2/kmeans/points output/project2/kmeans/ 10 10 100**

****

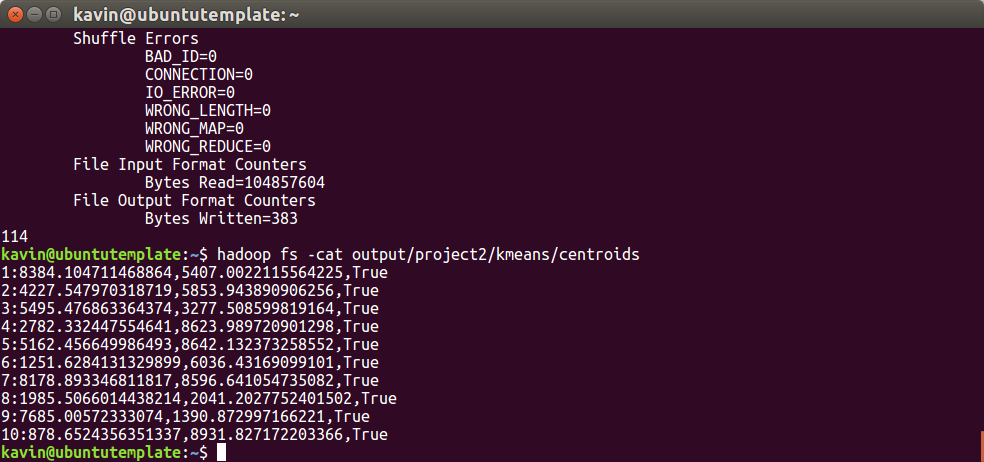
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | 1  (k=5)    (R=1) | 2  (k=5)    (R=10) | 3    (k=10)(R=10)    (Reducers=5) | 3    (k=10)(R=10)    (Reducers=1) | 5a    (k=10)(R=10)    (Reducers=1) | 5b    (k=10)(R=10)    (Reducers=1) |
| w/o combiner | 38 | 369 | 111 | 113 | 114 | 180 |
| W combiner | 32 | 370 | 150 | 111 | 112 | 181 |

\*All reported times are Seconds

Many of performance times were similar to each other but some where not and this could be due the initial set of random points. Since the initial points change for each run, the performance is not repeatable in many cases.

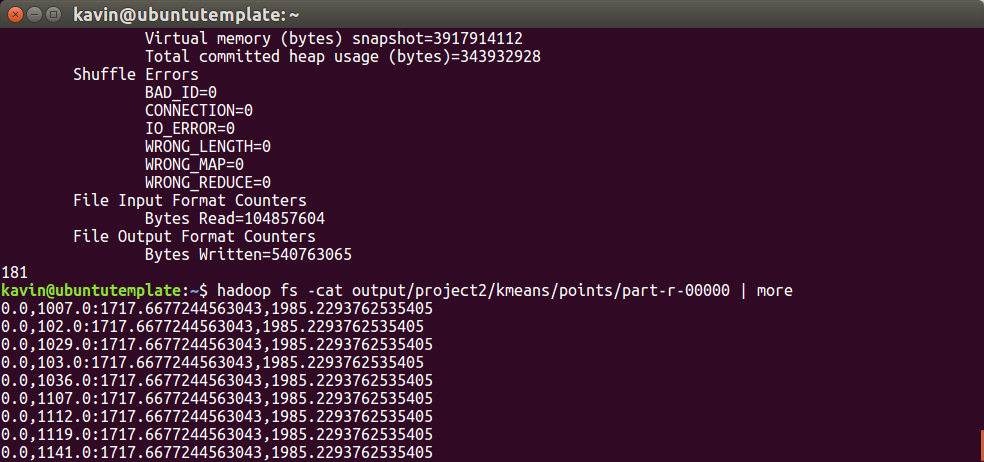
5) For step 5, we used the techniques which were faster in our analysis from different optimization techniques. We used combiners and convergence checking was done in the main and a single reducer by default. But the use of the combiner and the reducer could be changed by the user by specifying the parameters. There were two output formats requested in this step.

a) The first one is to display the centroids and indicate whether convergence was reached or not. To handle this we used the same code from part 3, and added a section in the main function where convergence is checked. Here we checked, for each centroid, if the previous centroid was with in threshold. We changed the structure a little to accommodate storing the convergence status of each centroid, instead of all the centroids. This way, if the maximum number of iterations was reached and not all centroids have converged, this output will specify which ones have converged and which centroids have not.

 The value True at the end of the centroid indicates the reaching of the convergence.

b) The second output was to show the centroid each point in the input file is associated with, after satisfying the above terminating conditions. For this step we kept the map-reduce part for calculating the centroid and then added another mapper to assign centroids for each input point. The loop control in the main program was set such that, the second map job would wait till the first job is completed meeting the requirements. The output points, along with the centroid is written out in a folder called “points” inside the specified output path. If running multiple times, this folder will have to be manually cleaned with **hadoop fs -rm -r <output path>/points**

**hadoop jar College\_Work/DS503/Project2/kmeans.jar kmeans\_5b input/project2/kmeans/points output/project2/kmeans/ 10 10 500 Combiner**

**** The coordinate before : is the point from the points file and the coordinate after “:” is the centroid that is associated with the cluster they are in.

**Collaboration:**

We coordinated initial meeting and decided that each person could try to solve all three problems and but should have two problems completed each, so that there would at-least be one other person in the team who finished it. After our initial attempt, we discussed problems with our code and where we needed help. We were successfully above to complete multiple parts with varying levels of performing code. Then finally we decided to pick the best performing code,when applicable, for each part and submit for the final submission. We wrote the final report corresponding to which code we used for each part.