**package** Clustering;

**import** java.io.IOException;

**import** java.util.\*;

**import** java.io.\*;

**import** org.apache.hadoop.conf.Configuration;

**import** org.apache.hadoop.filecache.DistributedCache;

**import** org.apache.hadoop.fs.FileSystem;

**import** org.apache.hadoop.fs.Path;

**import** org.apache.hadoop.io.\*;

**import** org.apache.hadoop.mapred.\*;

**import** org.apache.hadoop.mapred.Reducer;

//@SuppressWarnings("deprecation")

**public** **class** Clustering {

**public** **static** String *OUT* = "outfile12525";

**public** **static** String *IN* = "inputlarger";

**public** **static** String *CENTROID\_FILE\_NAME* = "/centroid.txt";

**public** **static** String *OUTPUT\_FILE\_NAME* = "/part-00000";

**public** **static** String *DATA\_FILE\_NAME* = "/data.txt";

**public** **static** String *JOB\_NAME* = "KMeans";

**public** **static** String *SPLITTER* = "\t| ";

**public** **static** List<Double> *mCenters* = **new** ArrayList<Double>();

/\*

\* In Mapper class we are overriding configure function. In this we are

\* reading file from Distributed Cache and then storing that into instance

\* variable "mCenters"

\*/

**public** **static** **class** Map **extends** MapReduceBase **implements**

Mapper<LongWritable, Text, DoubleWritable, DoubleWritable> {

@Override

**public** **void** configure(JobConf job) {

**try** {

// Fetch the file from Distributed Cache Read it and store the

// centroid in the ArrayList

Path[] cacheFiles = DistributedCache.*getLocalCacheFiles*(job);

**if** (cacheFiles != **null** && cacheFiles.length > 0) {

String line;

*mCenters*.clear();

BufferedReader cacheReader = **new** BufferedReader(

**new** FileReader(cacheFiles[0].toString()));

**try** {

// Read the file split by the splitter and store it in

// the list

**while** ((line = cacheReader.readLine()) != **null**) {

String[] temp = line.split(*SPLITTER*);

*mCenters*.add(Double.*parseDouble*(temp[0]));

}

} **finally** {

cacheReader.close();

}

}

} **catch** (IOException e) {

System.*err*.println("Exception reading DistribtuedCache: " + e);

}

}

/\*

\* Map function will find the minimum center of the point and emit it to

\* the reducer

\*/

@Override

**public** **void** map(LongWritable key, Text value,

OutputCollector<DoubleWritable, DoubleWritable> output,

Reporter reporter) **throws** IOException {

String line = value.toString();

**double** point = Double.*parseDouble*(line);

**double** min1, min2 = Double.*MAX\_VALUE*, nearest\_center = *mCenters*

.get(0);

// Find the minimum center from a point

**for** (**double** c : *mCenters*) {

min1 = c - point;

**if** (Math.*abs*(min1) < Math.*abs*(min2)) {

nearest\_center = c;

min2 = min1;

}

}

// Emit the nearest center and the point

output.collect(**new** DoubleWritable(nearest\_center),

**new** DoubleWritable(point));

}

}

**public** **static** **class** Reduce **extends** MapReduceBase **implements**

Reducer<DoubleWritable, DoubleWritable, DoubleWritable, Text> {

/\*

\* Reduce function will emit all the points to that center and calculate

\* the next center for these points

\*/

@Override

**public** **void** reduce(DoubleWritable key, Iterator<DoubleWritable> values,

OutputCollector<DoubleWritable, Text> output, Reporter reporter)

**throws** IOException {

**double** newCenter;

**double** sum = 0;

**int** no\_elements = 0;

String points = "";

**while** (values.hasNext()) {

**double** d = values.next().get();

points = points + " " + Double.*toString*(d);

sum = sum + d;

++no\_elements;

}

// We have new center now

newCenter = sum / no\_elements;

// Emit new center and point

output.collect(**new** DoubleWritable(newCenter), **new** Text(points));

}

}

**public** **static** **void** main(String[] args) **throws** Exception {

*run*(args);

}

**public** **static** **void** run(String[] args) **throws** Exception {

*IN* = args[0];

*OUT* = args[1];

String input = *IN*;

String output = *OUT* + System.*nanoTime*();

String again\_input = output;

// Reiterating till the convergence

**int** iteration = 0;

**boolean** isdone = **false**;

**while** (isdone == **false**) {

JobConf conf = **new** JobConf(Clustering.**class**);

**if** (iteration == 0) {

Path hdfsPath = **new** Path(input + *CENTROID\_FILE\_NAME*);

// upload the file to hdfs. Overwrite any existing copy.

DistributedCache.*addCacheFile*(hdfsPath.toUri(), conf);

} **else** {

Path hdfsPath = **new** Path(again\_input + *OUTPUT\_FILE\_NAME*);

// upload the file to hdfs. Overwrite any existing copy.

DistributedCache.*addCacheFile*(hdfsPath.toUri(), conf);

}

conf.setJobName(*JOB\_NAME*);

conf.setMapOutputKeyClass(DoubleWritable.**class**);

conf.setMapOutputValueClass(DoubleWritable.**class**);

conf.setOutputKeyClass(DoubleWritable.**class**);

conf.setOutputValueClass(Text.**class**);

conf.setMapperClass(Map.**class**);

conf.setReducerClass(Reduce.**class**);

conf.setInputFormat(TextInputFormat.**class**);

conf.setOutputFormat(TextOutputFormat.**class**);

FileInputFormat.*setInputPaths*(conf,

**new** Path(input + *DATA\_FILE\_NAME*));

FileOutputFormat.*setOutputPath*(conf, **new** Path(output));

JobClient.*runJob*(conf);

Path ofile = **new** Path(output + *OUTPUT\_FILE\_NAME*);

FileSystem fs = FileSystem.*get*(**new** Configuration());

BufferedReader br = **new** BufferedReader(**new** InputStreamReader(

fs.open(ofile)));

List<Double> centers\_next = **new** ArrayList<Double>();

String line = br.readLine();

**while** (line != **null**) {

String[] sp = line.split("\t| ");

**double** c = Double.*parseDouble*(sp[0]);

centers\_next.add(c);

line = br.readLine();

}

br.close();

String prev;

**if** (iteration == 0) {

prev = input + *CENTROID\_FILE\_NAME*;

} **else** {

prev = again\_input + *OUTPUT\_FILE\_NAME*;

}

Path prevfile = **new** Path(prev);

FileSystem fs1 = FileSystem.*get*(**new** Configuration());

BufferedReader br1 = **new** BufferedReader(**new** InputStreamReader(

fs1.open(prevfile)));

List<Double> centers\_prev = **new** ArrayList<Double>();

String l = br1.readLine();

**while** (l != **null**) {

String[] sp1 = l.split(*SPLITTER*);

**double** d = Double.*parseDouble*(sp1[0]);

centers\_prev.add(d);

l = br1.readLine();

}

br1.close();

// Sort the old centroid and new centroid and check for convergence

// condition

Collections.*sort*(centers\_next);

Collections.*sort*(centers\_prev);

Iterator<Double> it = centers\_prev.iterator();

**for** (**double** d : centers\_next) {

**double** temp = it.next();

**if** (Math.*abs*(temp - d) <= 0.1) {

isdone = **true**;

} **else** {

isdone = **false**;

**break**;

}

}

++iteration;

again\_input = output;

output = *OUT* + System.*nanoTime*();

}

}

}

data.txt

1

2

3

4

5

6

7

8

9

10

Centroid.txt

1

2

3

[training@localhost ~]$hadoop jar Clustering.jar Clustering.Clustering hadoop/ hadoop/ hadoop/Clustering/