

CIS 4930 Introduction to Hadoop and Big Data

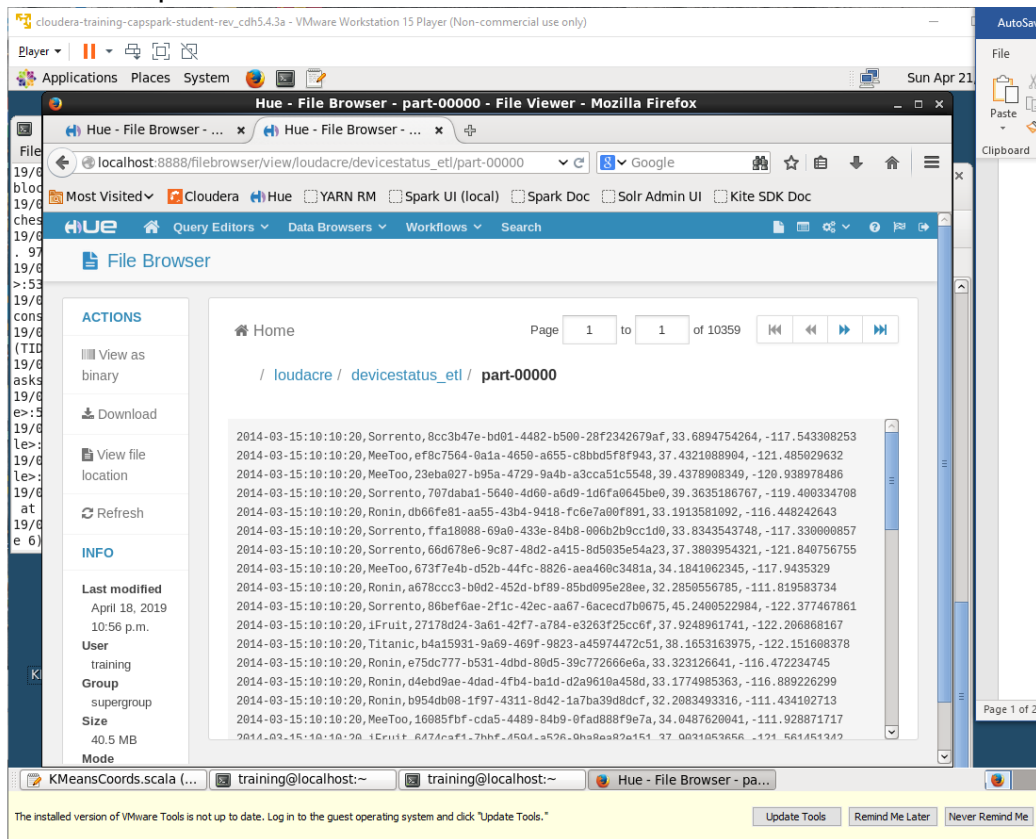
Implement an Iterative Algorithm with Spark

Julie Reyes | u76631122

Introduction:

For this project, an iterative spark algorithm was developed using the scala language. The data that was transformed was located at `/loudacre/devicestatus_etl/*` on the HDFS. This data was transformed and uploaded to the HDFS in the lab "Use RDDs to Explore and Transform a Dataset".

This is an example of the file used for this lab on the hue file browser:



Methodology:

Initially three helper functions were defined, which included *closestPoint*, *addPoints*, and *distanceSquared*. Next, the variables defined in the lab were defined in the scala file. This included $K = 5$, which is the number of means to calculate for this lab. Also, $convergeDist = 0.1$, which used to decide when the k-means calculation has been completed. This is in effect when the amount the

locations of the means changes between iterations is less than *convergeDist*. Since a perfect solution would have been 0, the approach in this lab was heuristic.

Next, the file in the directory `/loudacre/devicestatus_etl/*` was parsed. This file was split by its delimiting the character `,`, into latitude and longitude pairs, which correspond to 4th and 5th fields in each line, at index 3 and 4. While parsing, only known locations were include as latitude and longitude pairs of 0,0 were filtered out. Lastly, the resulting RDD was persisted as it was being accessed in an iterative fashion.

File being parsed with the constraints mentioned above:

```
val data = fileRdd.map(line => line.split(',')).map(pair => (pair(3).toDouble,  
pair(4).toDouble)).filter(point => !((point._1 == 0) && (point._2 ==  
0))).persist()
```

Then we started with k randomly selected points from the dataset as center points and printed out their values with the following command:

```
var kPoints = pair.takeSample(false, K, 42)  
println("Final K center points: ")  
kPoints.foreach(println)
```

To find the distance between one iteration's points and the next is less than the convergence distance of 0.5, a while loop was used. Next, an RDD of the closest point with was made using the helper function `closestPoint`. Then the RDD was reduced by key and saved as a new RDD. Then new points RDD was calculating by the average of the closest point. Then the distance between the current points and new points was calculated. Then the new points were saved to the `kPoints` array. The following displays the commands used:

```
var tempDist = Double.PositiveInfinity  
while (tempDist > convergeDist) {  
    val c = pair.map(point => (closestPoint(point, kPoints), (point, 1)))  
    //closestToKpointRdd.foreach(println)  
    val pointStats = c.reduceByKey{case ((point1,n1),(point2,n2)) =>  
(addPoints(point1,point2),n1+n2) }  
  
    // For each key (k-point index), find a new point by calculating the  
    average of each closest point  
    val newPoints = pointStats.map{case (i,(point,n)) =>  
(i,(point._1/n,point._2/n))}.collectAsMap()  
  
    // calculate the total of the distance between the current points and
```

new points

```
tempDist = 0.0
for (i <- 0 until K) {
  tempDist += distanceSquared(kPoints(i),newPoints(i))
}
println("Distance between iterations: "+tempDist)

// Copy the new points to the kPoints array for the next iteration
for (i <- 0 until K) {
  kPoints(i) = newPoints(i)
}
}
```

The final output of running the spark application is the following:

Final center points:

```
(35.08592000544959,-112.57643826547951)
(36.025400882817564,-119.86128360401455)
(44.16453784336209,-121.68421197097716)
(34.177661753708364,-117.61626933369156)
(38.49271102063178,-121.28934110975781)
```

To submit the job to the cluster the following command was used:

```
scala> :load /home/training/Desktop/KMeansCoord.scala
```

The following image depicts the job being tracked on the Hadoop cluster:

The screenshot shows a web browser window displaying the Hue interface for monitoring a Hadoop cluster. The browser address bar shows 'localhost:8088/cluster/nodes'. The interface includes a top navigation bar with links to Cloudera, Hue, YARN RM, Spark UI (local), Spark Doc, Solr Admin UI, and Kite SDK Doc. Below the navigation bar, there are two tables: 'Cluster Metrics' and 'User Metrics for dr.who'. The 'Cluster Metrics' table shows various metrics such as Apps Submitted, Apps Pending, Apps Running, Apps Completed, Containers Running, Memory Used, Memory Total, Memory Reserved, VCoers Used, VCoers Total, VCoers Reserved, and Active Nodes. The 'User Metrics for dr.who' table shows similar metrics for the user. Below these tables, there is a section for 'Nodes of the cluster' with a table showing node details. The table has columns for Node Labels, Rack, Node State, Node Address, Node HTTP Address, Last health-update, Health-report, and Containers. The table shows one entry for a node with labels '/default-rack', state 'RUNNING', address 'localhost:45198', and HTTP address 'localhost:8042'. The last health-update is 'Sun Apr 21 13:27:51 -0700 2019'. The health-report is empty. The containers column shows '0'. The interface also includes a 'Showing 1 to 1 of 1 entries' message.

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Memory Used	Memory Total	Memory Reserved	VCoers Used	VCoers Total	VCoers Reserved	Active Nodes
0	0	0	0	0	0 B	2 GB	0 B	0	8	0	1

Apps Submitted	Apps Pending	Apps Running	Apps Completed	Containers Running	Containers Pending	Containers Reserved	Memory Used	Memory Pending
0	0	0	0	0	0	0	0 B	0 B

Node Labels	Rack	Node State	Node Address	Node HTTP Address	Last health-update	Health-report	Containers
/default-rack		RUNNING	localhost:45198	localhost:8042	Sun Apr 21 13:27:51 -0700 2019		0

