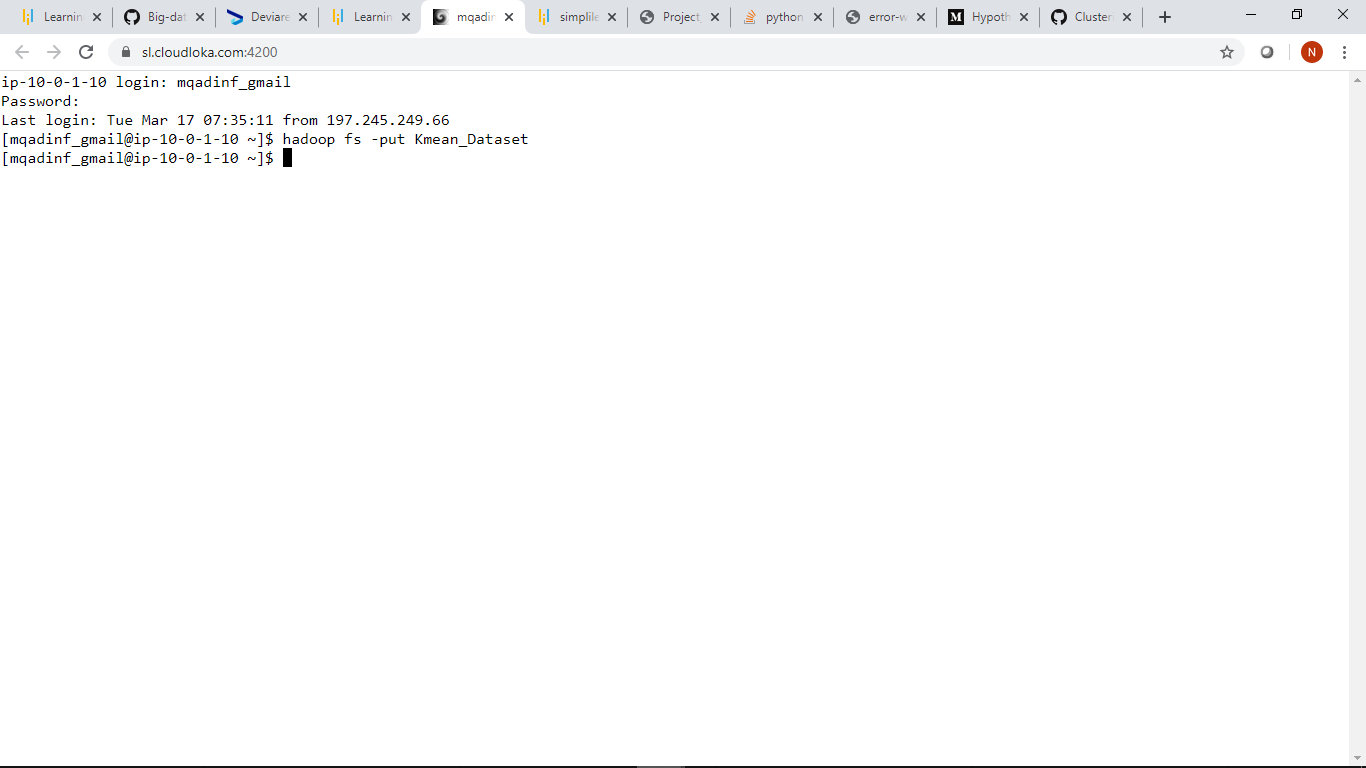
**Analysis to be done:**

**The fourth and the fifth field of the dataset has latitude and longitude of users which is an important information for the company. You have to find this information of latitude and longitude on the basis of available dataset and create three clusters of users with a k-means algorithm. This will help Loudacre maximize the coverage for its users.**

I renamed the Project2\_Dataset to Kmean\_Dataset.

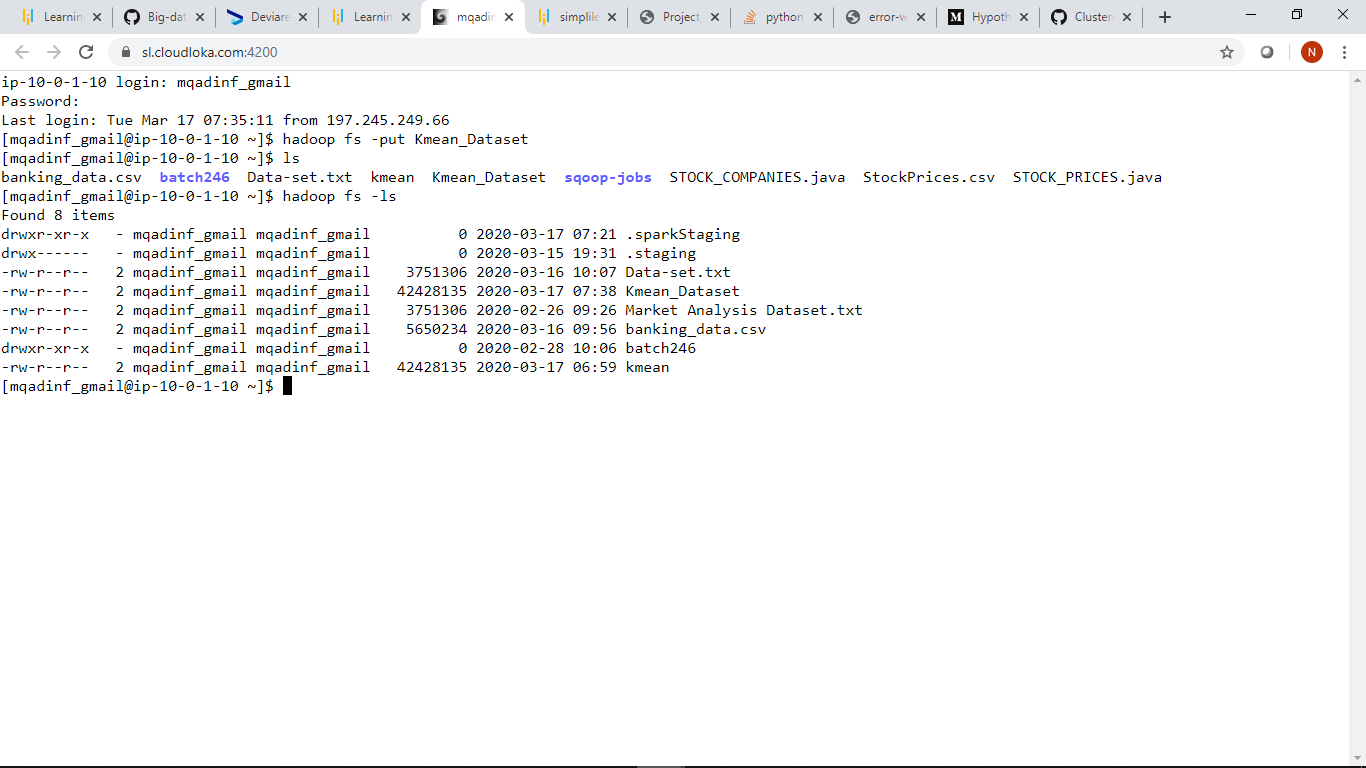
To load the Dataset to HDFS I’ve used this command:

**Hadoop fs -put Kmean\_Dataset**



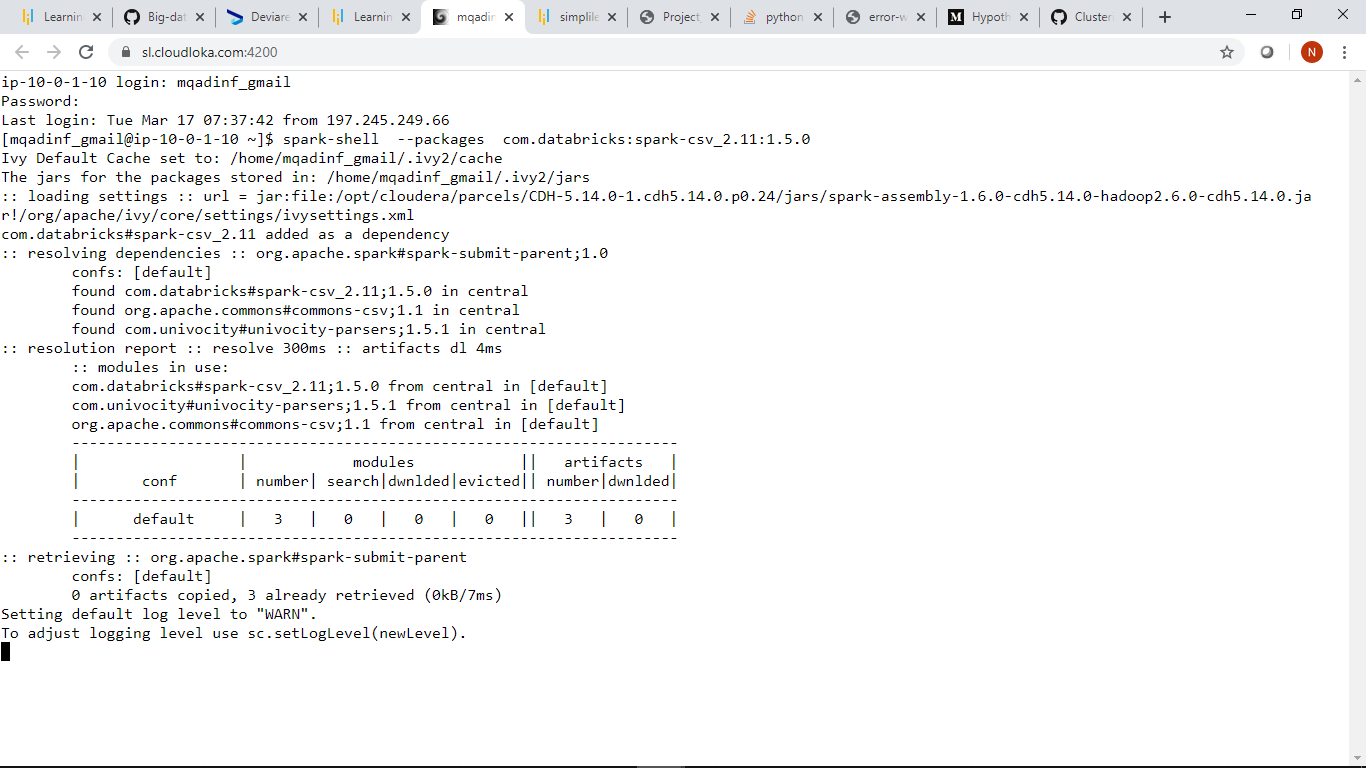
To check that I’ve loaded the Dataset to HDFS I’ve used this command **Hadoop fs -ls**,and to confirm that the dataset is available we can see that the Kmean\_Dataset

available.



We are going to use scala to do the work for us ,so we must import spark using this command:

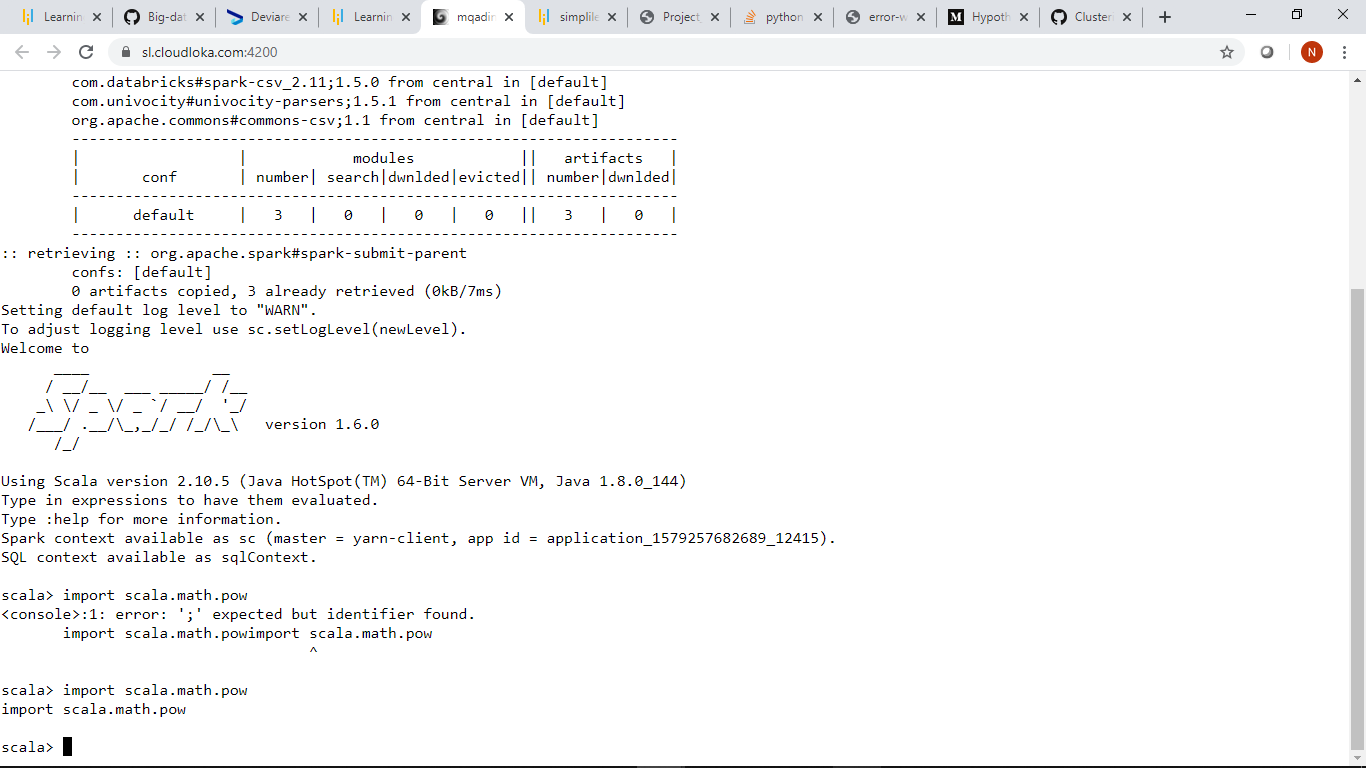
**Spark-shell --packages com.databricks:spark-csv\_2.11:1.5.0**



We must first import the library scala.math.pow to perform some mathematical computations for use:

**Code:**

**Import scala.math.pow**



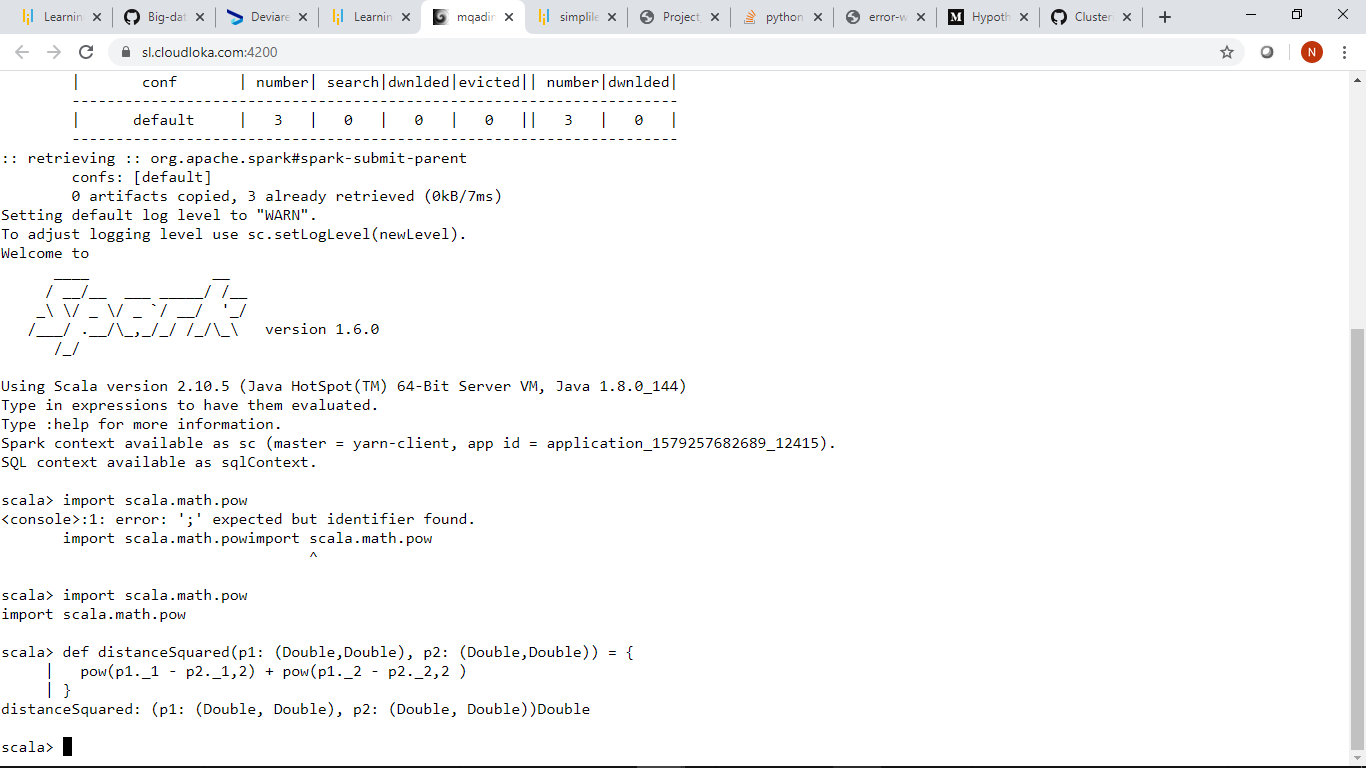
To get the squared distances between 2 points we will create a method def distanceSquared

**Code:**

**def distanceSquared(p1: (Double,Double), p2: (Double,Double)) = {**

**pow(p1.\_1 - p2.\_1,2) + pow(p1.\_2 - p2.\_2,2 )**

**}**



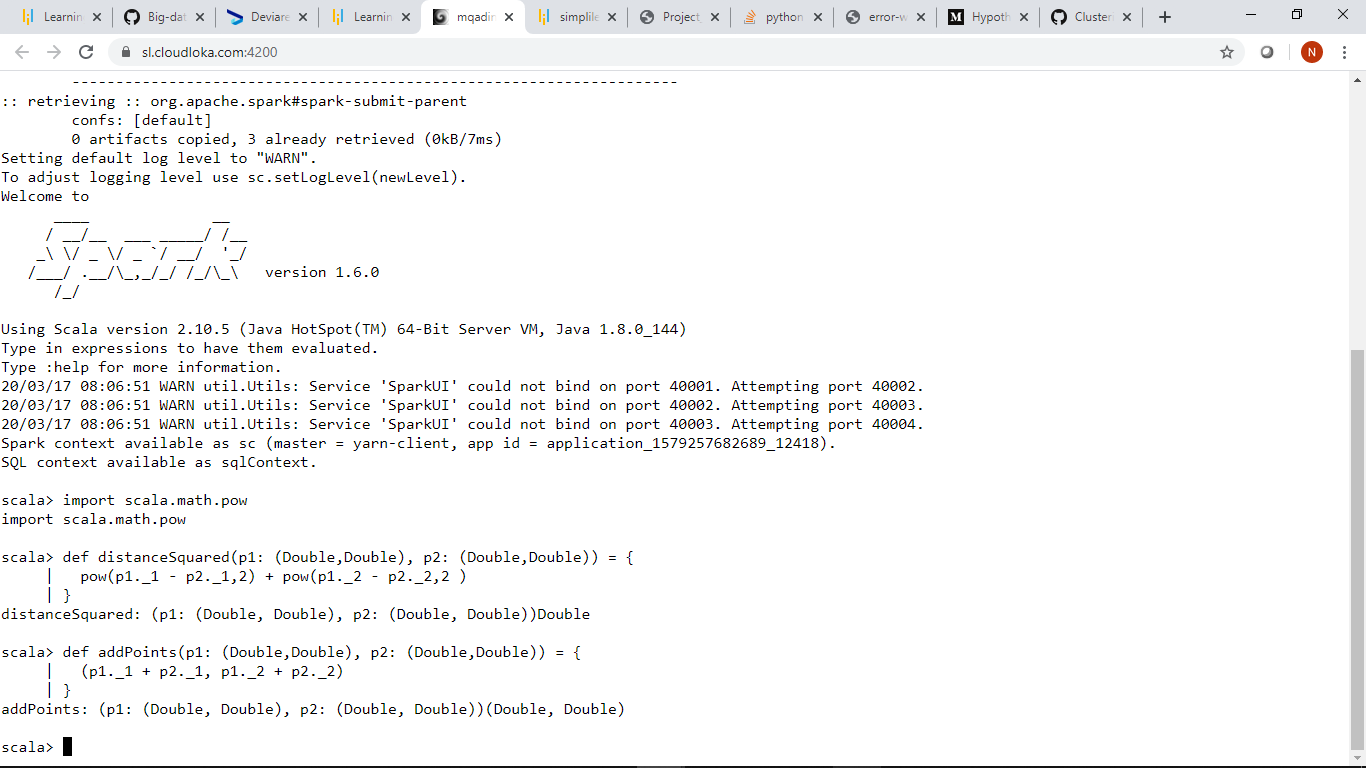
To add a sum of 2 points we create another method addPoints

**Code:**

**def addPoints(p1: (Double,Double), p2: (Double,Double)) = {**

**(p1.\_1 + p2.\_1, p1.\_2 + p2.\_2)**

**}**



For a point p and an array of points, return the index in the array of the point closest to p.

**Code:**

**def closestPoint(p: (Double,Double), points: Array[(Double,Double)]): Int = {**

**var index = 0**

**var bestIndex = 0**

**var closest = Double.PositiveInfinity**

**for (i <- 0 until points.length) {**

**val dist = distanceSquared(p,points(i))**

**if (dist < closest) {**

**closest = dist**

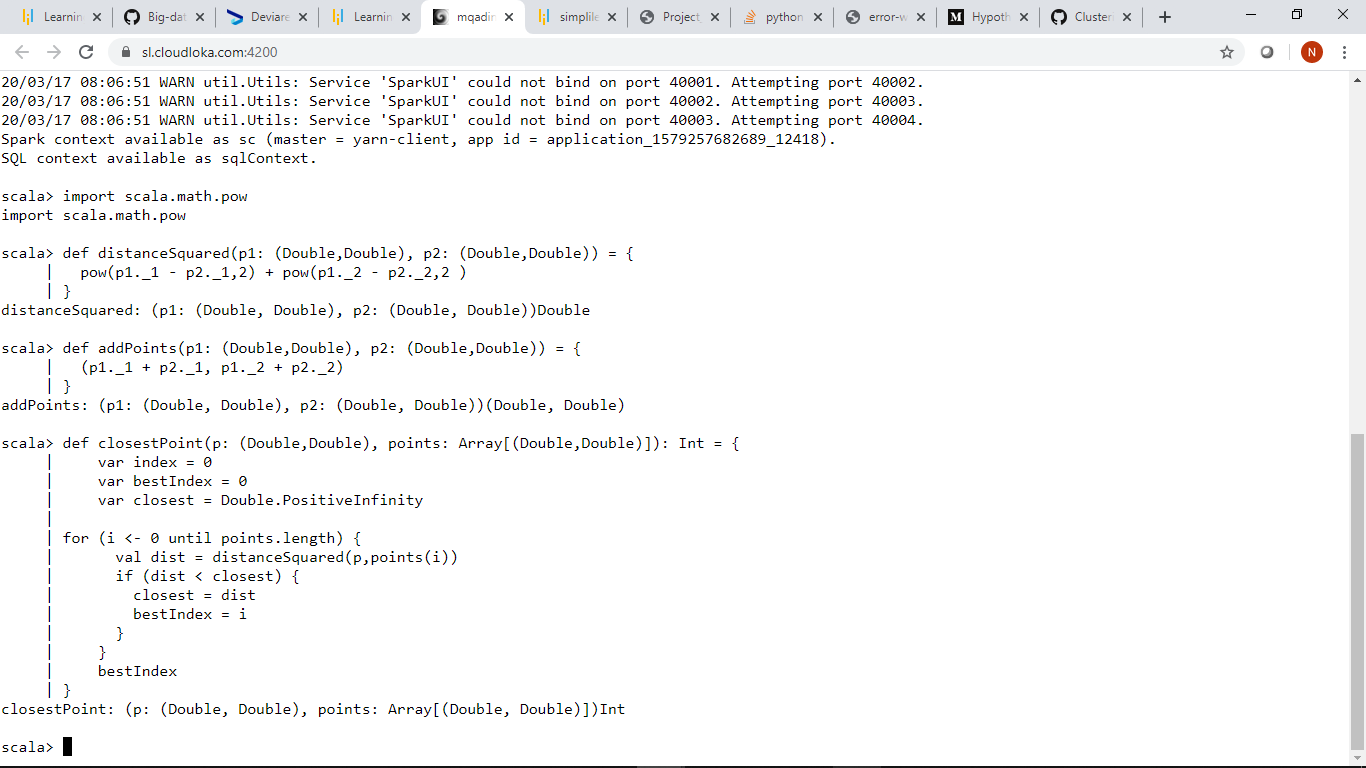
**bestIndex = i**

**}**

**}**

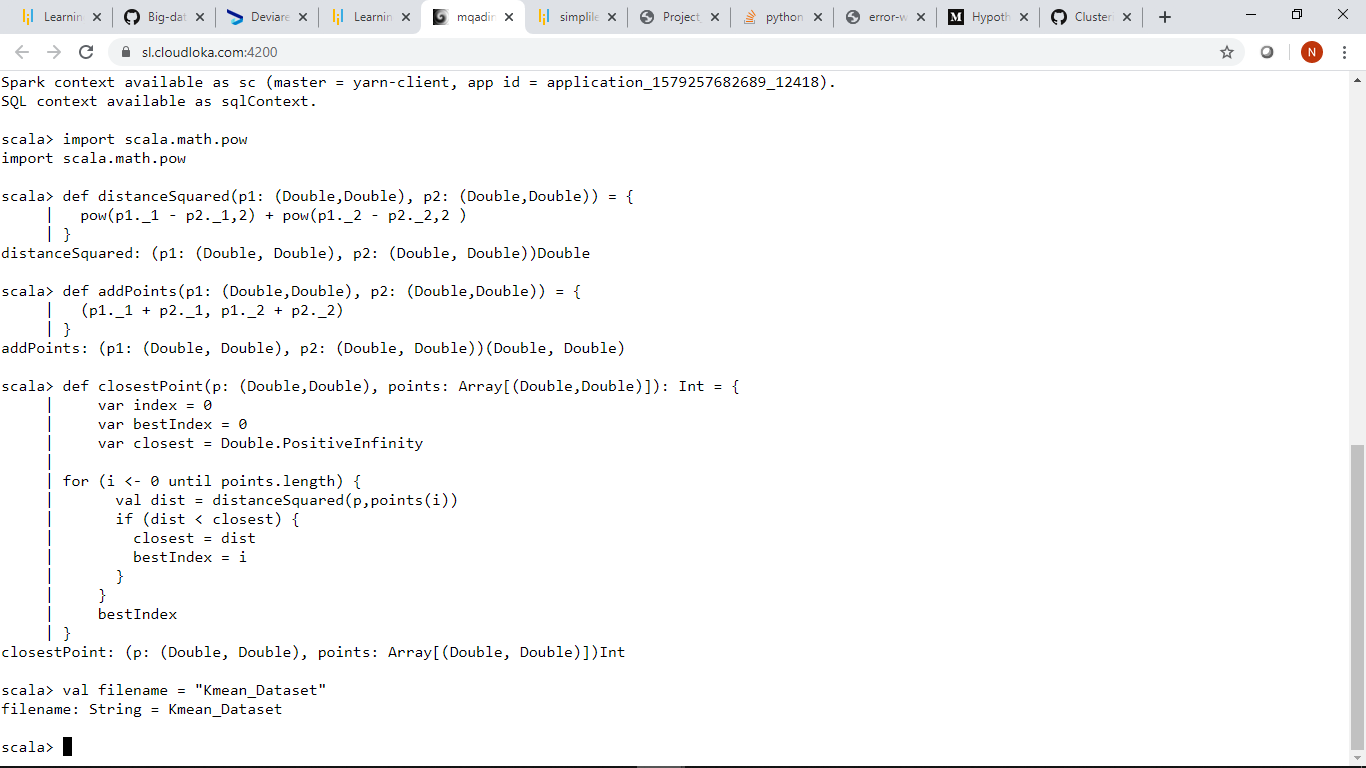
**bestIndex**

**}**

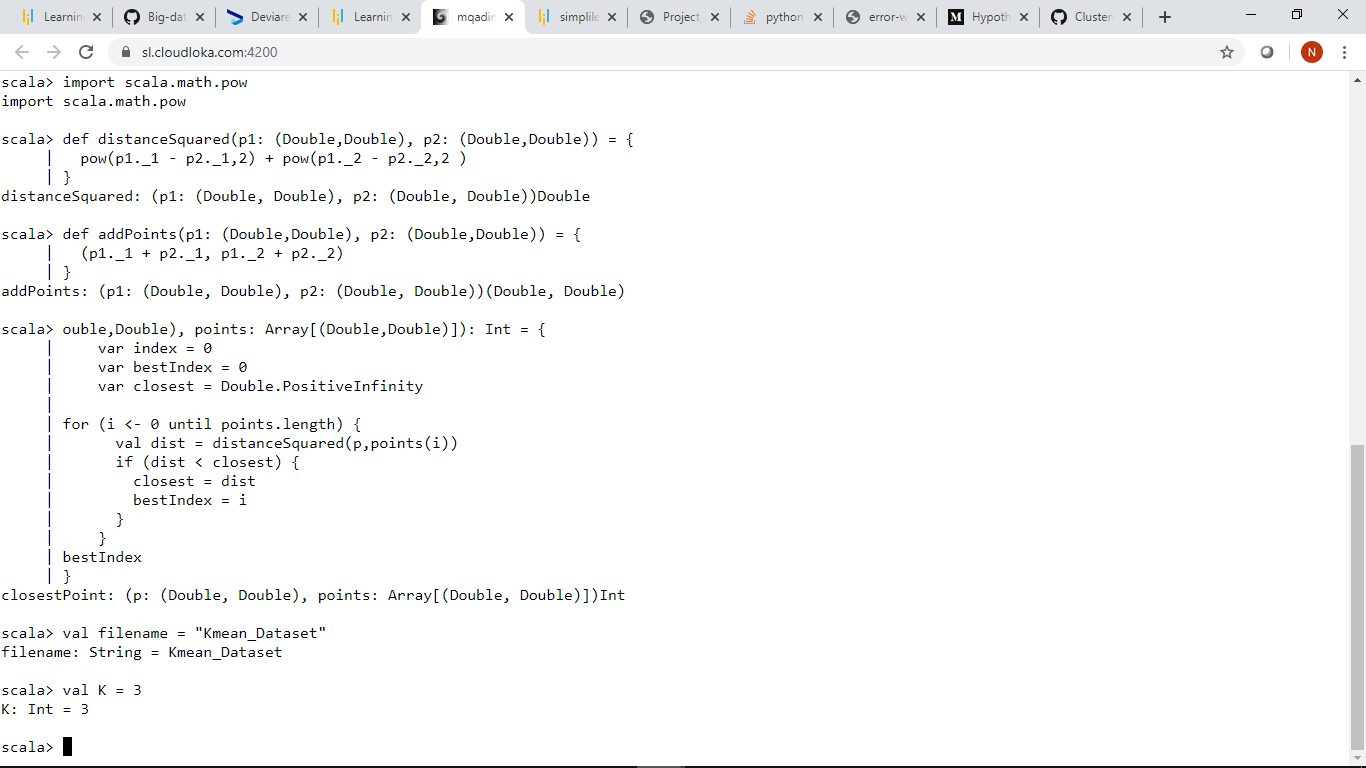


**To use the data Kmean\_Dataset**

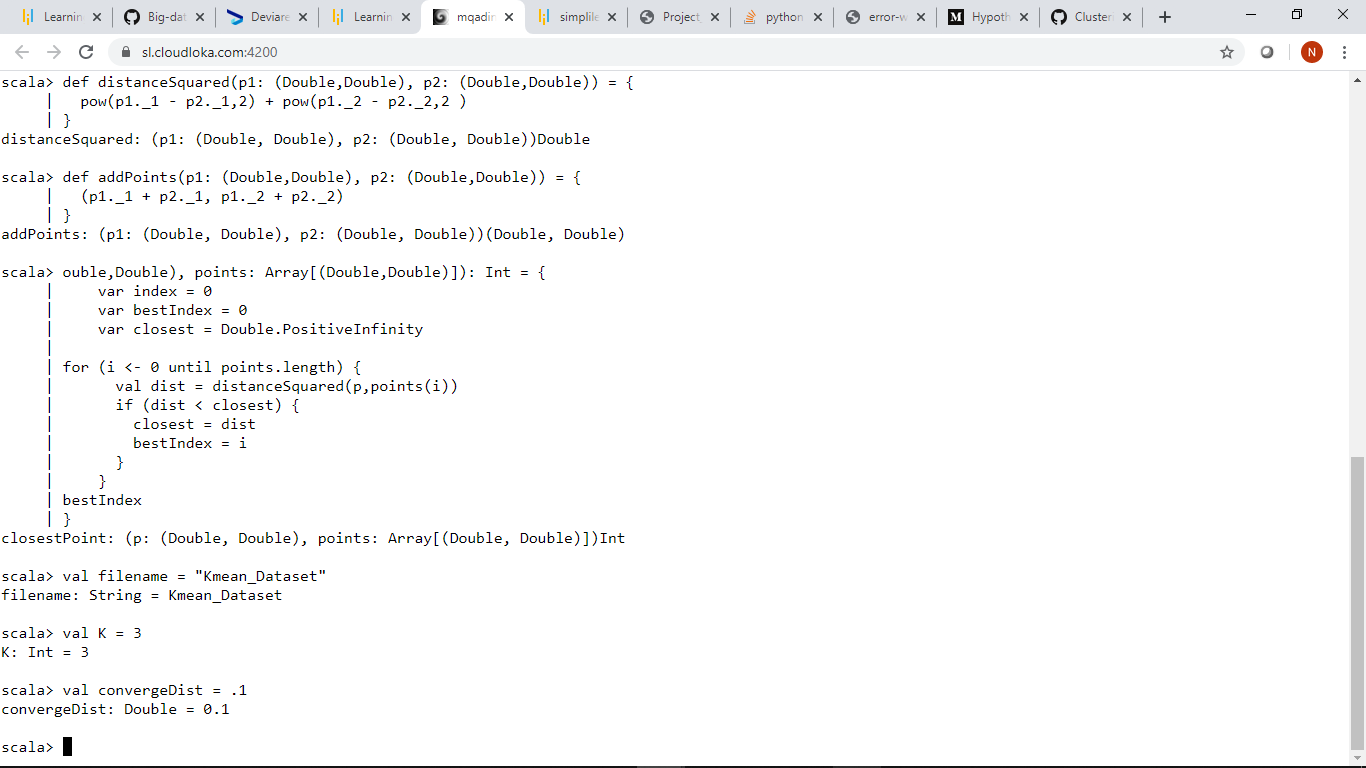
**Code: val filename = “Kmean\_Dataset”**



**K is the number of means (center points of clusters) to find.So we will set k=3**



**ConvergeDist is the threshold "distance" between iterations at which we decide we are done,so we will set ConvergeDist=.1**



Here we will Parse the device status data file,split by delimeter,Parse the longitude and latitude(4th and 5th fields into pairs) and

We will filter out records where longitude or latitude is unavailable –ie 0/0 points.

**Code:**

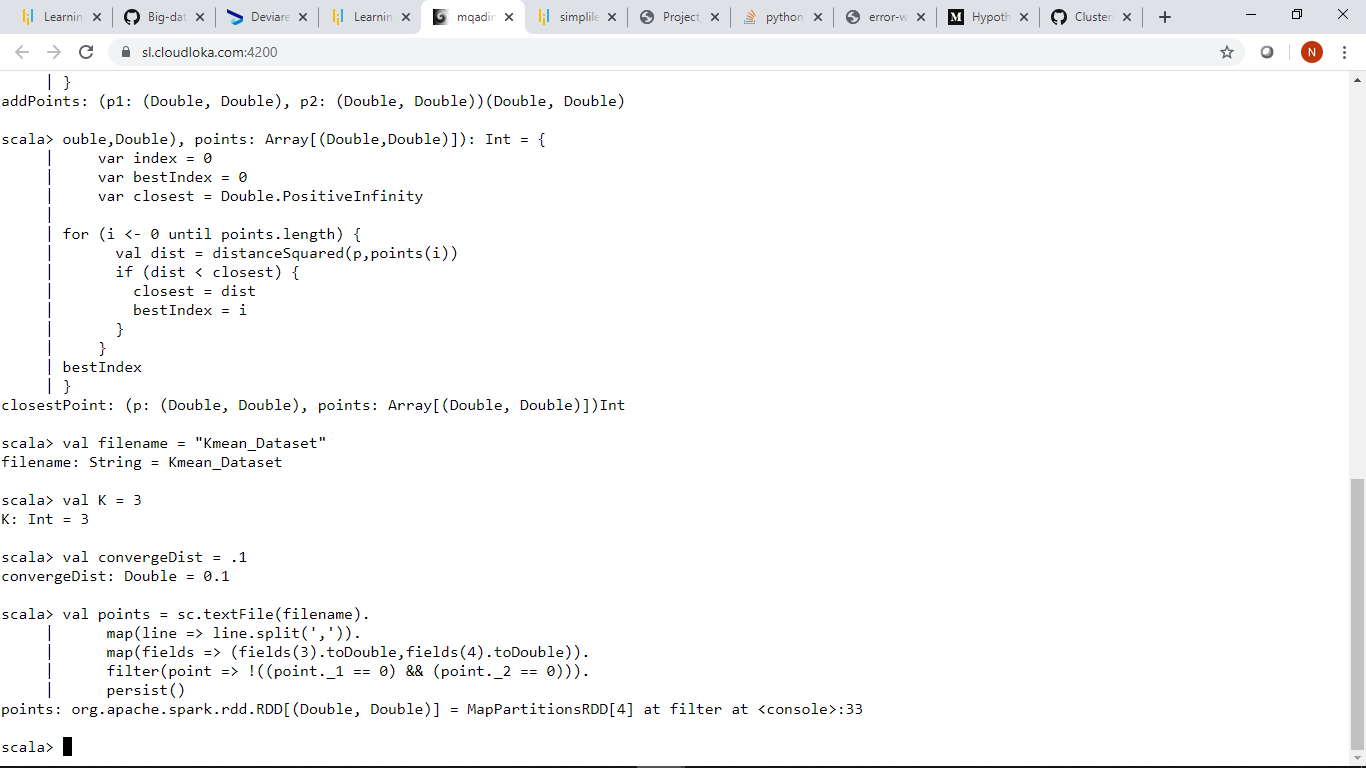
**val points = sc.textFile(filename).**

**map(line => line.split(',')).**

**map(fields => (fields(3).toDouble,fields(4).toDouble)).**

**filter(point => !((point.\_1 == 0) && (point.\_2 == 0))).**

**persist()**



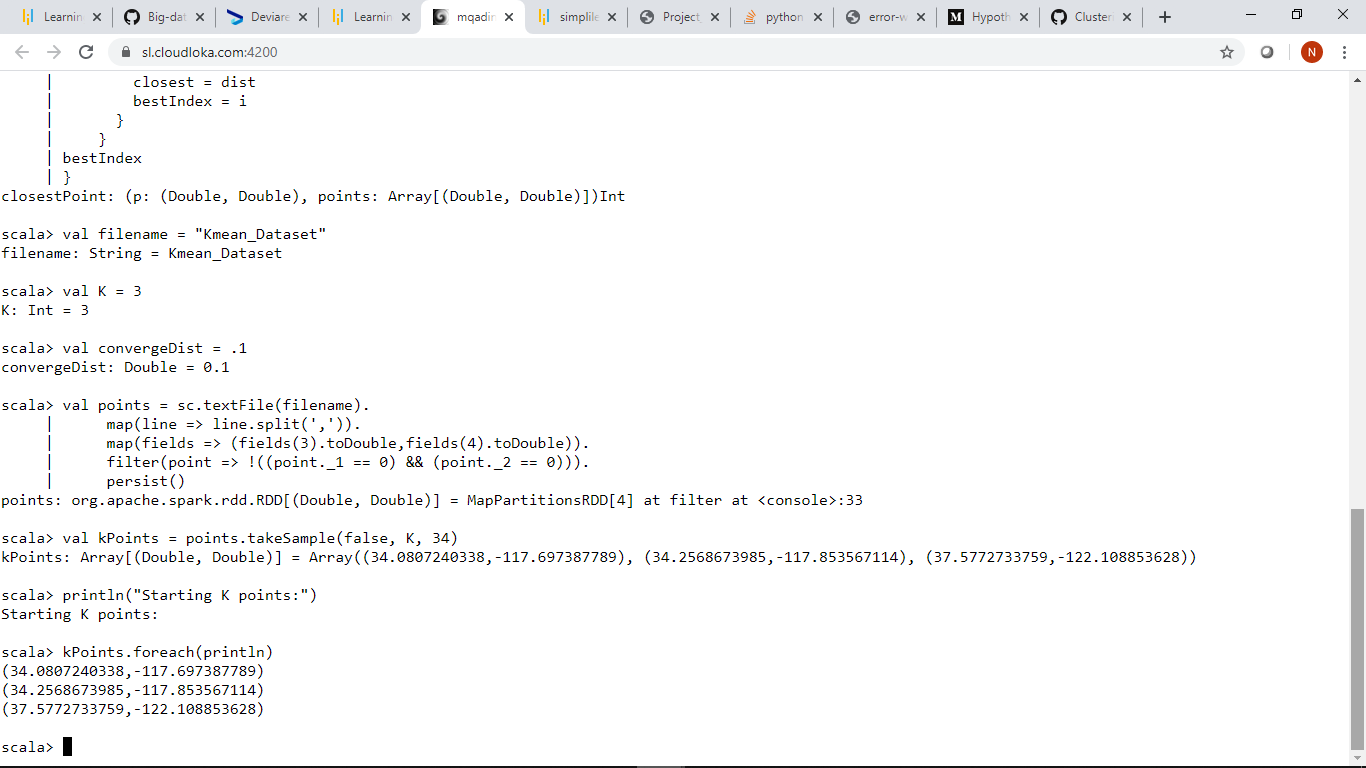
We will start with K randomly selected points from the dataset,and they are being displayed below.

**Code:**

**val kPoints = points.takeSample(false, K, 34)**

**println("Starting K points:")**

**kPoints.foreach(println)**



After that we will loop until the total distance between 1 iteration’s points and the next is less than the convergence distance specified.

For each point we are required to find the index of the closest kpoint. map to (index, (point,1)) .

For each key (k-point index),we are required to reduce by adding the coordinates and number of points.

For each key (k-point index),we are then required to find a new point by calculating the average of each closest point.

Then after that we must calculate the total of the distance between the current points and new points,and then we must copy the new

Points to the KPoints array for the next iteration.

**Code:**

**var tempDist = Double.PositiveInfinity**

**while (tempDist > convergeDist) {**

**// for each point, find the index of the closest kpoint. map to (index, (point,1))**

**val closest = points.map(p => (closestPoint(p, kPoints), (p, 1)))**

**// For each key (k-point index), reduce by adding the coordinates and number of points**

**val pointStats = closest.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)**

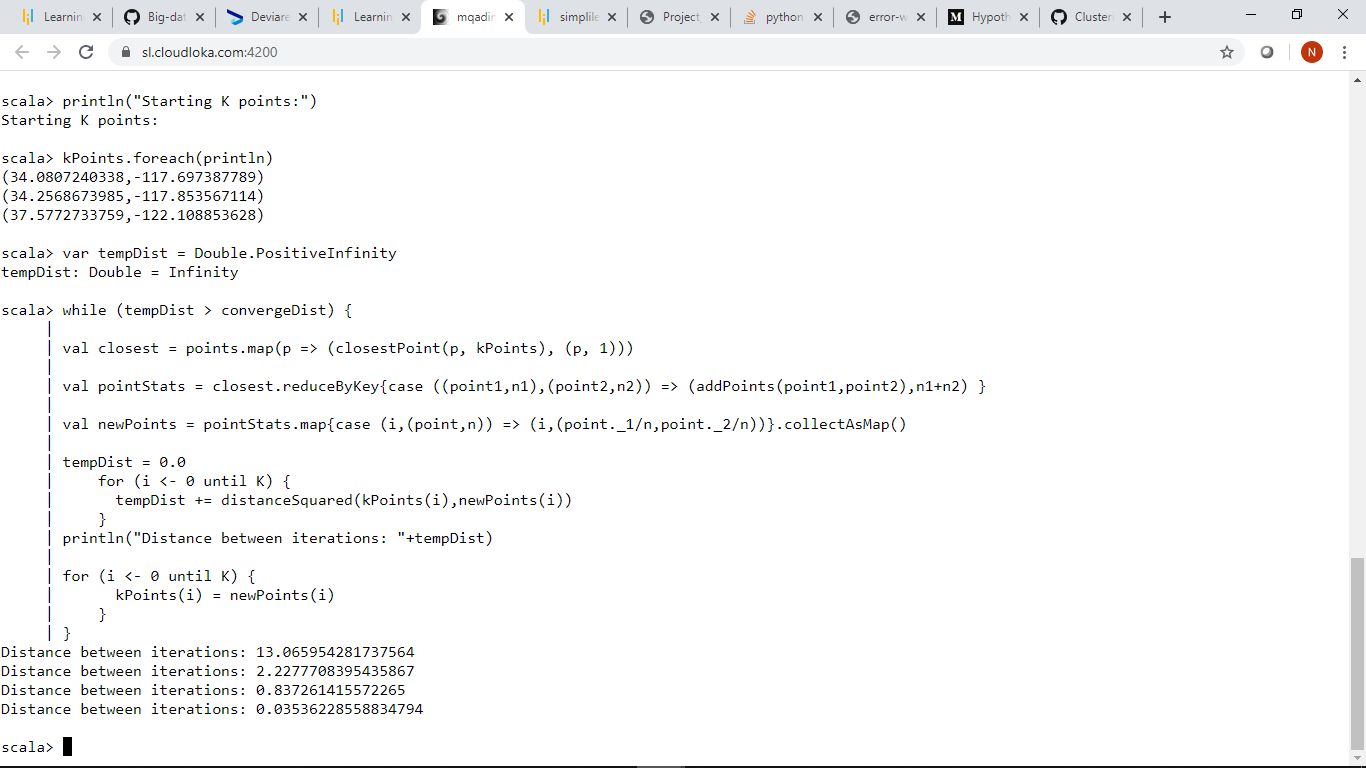
**// Copy the new points to the kPoints array for the next iteration**

**for (i <- 0 until K) {**

**kPoints(i) = newPoints(i)**

**}**

**}**



**Now finally we can display the final center points.**

**Code:**

**println("Final K points: " )**

**kPoints.foreach(println)**

