Washington University in St. Louis

SCHOOL OF ENGINEERING & APPLIED SCIENCE

CSE 427s – Cloud Computing with Big Data Applications Final Group Project:

K-Means for Geo-Location Clustering in Spark

April 25, 2017

Hanming Li[1], Hao Zhang[2], Jiayang Tian[3], Heqing Yu[4], Washington University in St. Louis

[1]*email address: hanming@wustl.edu;* [2]*email address: h.zhang633@wustl.edu;*

[3]*jiayang@wustl.edu;* [4]*heqingyu@wustl.edu*

Introduction

The report is divided into four sections: first, we will talk about our motivations about why we chose project three. It will be followed by three milestone subsections: milestone I and II will focus on problem 1 and 2 in the project description, respectively; milestone III will discuss problem 3 and 4. Besides some attached screenshots that help the reader keep track of our process, we have submitted all the work up onto the SVN.

Motivation

Our team was initially motivated by the superior running speed, ease of use, and generality of Spark. Knowing some of the limitations that Hadoop MapReduce are facing yet can be overcome by Apache Spark, we wanted to dig in and study further into the implementation of such powerful engine. Moreover, each of us is familiar with k-means algorithm, and we all have had experiences with it in different contexts; hence, we are interested in how it will affect geo-location clustering problems and how the final results will look like as we are quite new to the geo-location topic. Last but not least, our team believes project three is readily applicable to real-world questions such as finding the best warehouse locations (also mentioned in the project description), transportation routes optimization, and many real estate related challenges.

Milestone I

As instructed, we explored the Hue File Browser and put some data files into HDFS for later use in problem 1. Also, we reviewed the Spark Programming Guide and Python API Docs as we would be using Python throughout the Spark applications. For the first warm-up job, we performed word count Python implementation on “purplecow.txt” and “frostroad.txt”, and the results are as follows:

```
spark-submit wordcount.py file:///home/training/training_materials/data/purplecow.txt
```

spark-submit --master local wordcount.py frostroad.txt

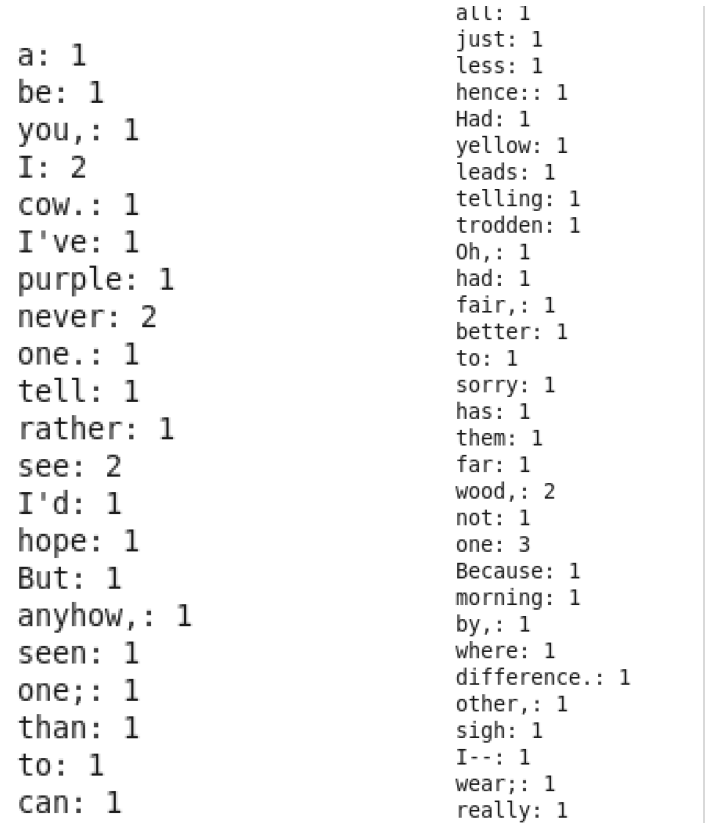

Image 1: Screenshots of results generated from the Python word count on "purplecow.txt"(left) and "frostroad.txt"(right).

The word count jobs are executed by the executor on the data node in HDFS. The execution mode can be changed by inputting different commands as below:

Cluster mode: spark-submit --master yarn-cluster abc.py ghj.txt

Client mode: spark-submit --master yarn-client abc.py ghj.txt

Local mode: park-submit --master local abc.py ghj.txt

Three parallel threads: park-submit --master local[3] abc.py ghj.txt

As many threads as cores: park-submit --master local[*] abc.py ghj.txt

Milestone II

The first step in milestone II is to pre-process the device status data into a cleaner, standardized format for later use. For documentation purposes, we will explain the Python code

in plain English and attach a screenshot of the code afterwards: after loading the data, we decided to unify all the field delimiters by replacing “pipes” and “slashes” with “commas”. “line.replace()” function will do the work. Next, as instructed, we filtered out data records that do not have 14 field values, which was achieved by “.filter()” function and equality operators “fields:len(fields)==14”. With this “cleaner” data, we used “.map()” function to extract latitude, longitude, date, model, and device ID for each record, plus filtering out those records with zero latitude and longitude values. Finally, we split the “model” field by adding the string “manufacturer” and “model” using “.map()” function. Screenshots of the Python code, data, and visualization of latitude/longitude pairs are attached below:

```
import sys
from pyspark import SparkContext
from pyspark import SparkConf


def newfields(field):
  field.split(" ")
  length= len(line)
  line.split(" ")

if __name__ == "__main__":

    if len(sys.argv)<2:
       print >> sys.stderr, "Usage: p2step1.py <logfile>"
       exit(-1)

    sc=SparkContext()
    #logfile=sys.argv[1]


    mydata=sc.textFile("file:///home/training/training_materials/data/devicestatus.txt").map(lambda line:line.replace("|",",").replace("/",",").split
(",")).filter(lambda fields:len(fields)==14).map(lambda fields:(fields[12]+","+fields[13]+","+fields[0]+","+fields[1]+","+fields[2])).filter(lambda
fields:fields[0]!='0' and fields[1]!='0').map(lambda line:line.replace(" ",",").split(",")).map(lambda fields:fields[0]+","+fields[1]+","+fields[2]
+",manufacturer "+fields[3]+",model "+fields[4]+","+fields[5])

    mydata.saveAsTextFile("loudacre/devicestatus_etl")
    sc.stop()
```

Image 2: Screenshot of main Python code responsible for device status data scrubbing.

```
33.6894754264,-117.543308253,2014-03-15:10:10:20,manufacturer Sorrento,model F41L,8cc3b47e-bd01-4482-b500-28f2342679af
37.4321088904,-121.485029632,2014-03-15:10:10:20,manufacturer MeeToo,model 1.0,ef8c7564-0a1a-4650-a655-c8bbd5f8f943
39.4378908349,-120.938978486,2014-03-15:10:10:20,manufacturer MeeToo,model 1.0,23eba027-b95a-4729-9a4b-a3cca51c5548
39.3635186767,-119.400334708,2014-03-15:10:10:20,manufacturer Sorrento,model F41L,707daba1-5640-4d60-a6d9-1d6fa0645be0
33.1913581092,-116.448242643,2014-03-15:10:10:20,manufacturer Ronin,model Novelty,Note
33.8343543748,-117.330000857,2014-03-15:10:10:20,manufacturer Sorrento,model F41L,ffa18088-69a0-433e-84b8-006b2b9cc1d0
37.3803954321,-121.840756755,2014-03-15:10:10:20,manufacturer Sorrento,model F33L,66d678e6-9c87-48d2-a415-8d5035e54a23
34.1841062345,-117.9435329,2014-03-15:10:10:20,manufacturer MeeToo,model 4.1,673f7e4b-d52b-44fc-8826-aea460c3481a
32.2850556785,-111.819583734,2014-03-15:10:10:20,manufacturer Ronin,model Novelty,Note
45.2400522984,-122.377467861,2014-03-15:10:10:20,manufacturer Sorrento,model F41L,86bef6ae-2f1c-42ec-aa67-6acecd7b0675
37.9248961741,-122.206868167,2014-03-15:10:10:20,manufacturer iFruit,model 3,27178d24-3a61-42f7-a784-e3263f25cc6f
38.1653163975,-122.151608378,2014-03-15:10:10:20,manufacturer Titanic,model 2400,b4a15931-9a69-469f-9823-a45974472c51
33.323126641,-116.472234745,2014-03-15:10:10:20,manufacturer Ronin,model S1,e75dc777-b531-4dbd-80d5-39c772666e6a
```

Image 3: Screenshot of partial records in the pre-processed dataset.

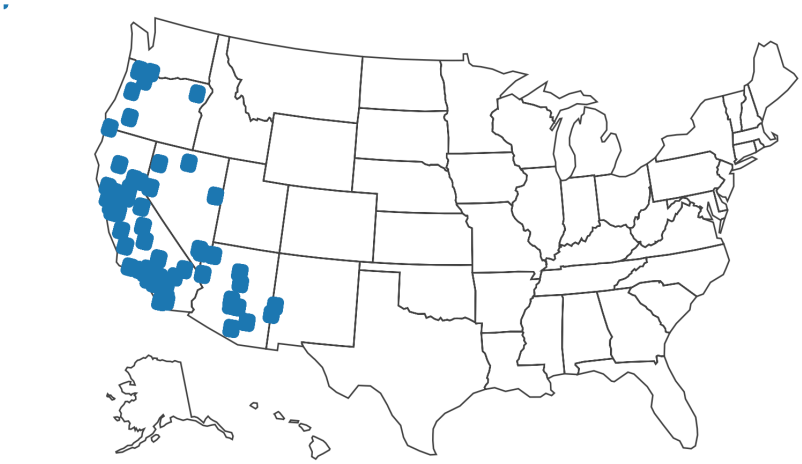

Image 4: Visualization of latitude/longitude pairs of device location data.

At first, we considered writing Python code to visualize the device location data. While saerching online for the proper use of Python graphing library, plotly, we found a more convenient way of visualizing data, which uses an online data visualization platform, http://plot.ly/. Instead of writing the entire Python code, we only needed to convert the "txt" file that contained the data into "csv" file. Likewise for the synthetic clustering data shown below.

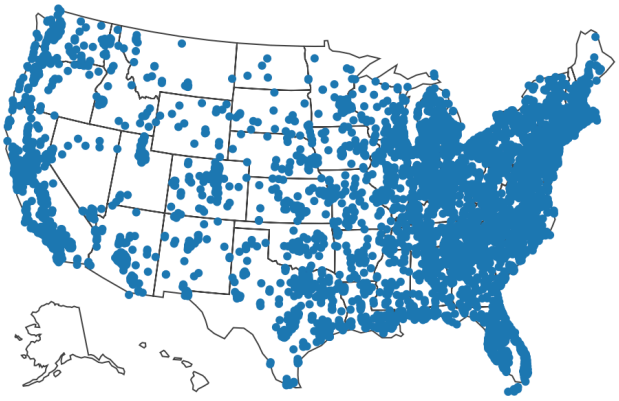

Image 5: Visualization of synthetic location data.

Problem 2 step 3 requires us to visualize a small sample of the DBpedia location data. To create the bounding box, we Google'ed the estimated latitude/longitude values for the four vertices that we selected. The visualized data is shown below:

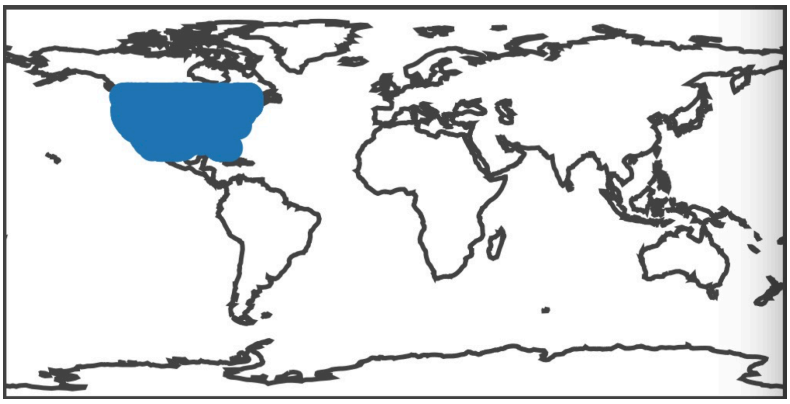

Image 6: Visualization of DBpedia location data.

Milestone III

In this section, we will be discussing the code that performs k-means algorithm and the implementation of k-means algorithm on the previous datasets with different parameters.

As instructed, we first defined the functions "closestPoint(closest)", "addPoints", "EuclideanDistance", "GreatCircleDistance", and "convergeDist(Cov)" for later use. Here we will only show the main defining part of the code. Complete code has been submitted to SVN.

```
def EuclideanDistance(point1,point2):
    dis = np.sqrt(np.sum((point1-point2)**2))
    return dis

def GreatCircleDistance(point1,point2):
    earthr = 6378137
    a=point1[0] * math.pi / 180.0
    b=point1[1] * math.pi / 180.0
    c=point2[0] * math.pi / 180.0
    d=point2[1] * math.pi / 180.0
    e=math.cos(b)*math.cos(d)*math.cos(c-a)+math.sin(b)*math.sin(d)
    if e >-1 and e <1:
        gcd=earthr*(math.acos(math.cos(b)*math.cos(d)*math.cos(c-a)+math.sin(b)*math.sin(d)))
        return gcd
```

Image 7: Screenshot of EuclideanDistance function and GreatCircleDistance function.

```
def closest(point, centers):
    finalindex = 0
    if Distancetype==0:
        mindis = EuclideanDistance(point, centers[0])
    else:
        mindis = GreatCircleDistance(point, centers[0])
    for i in range(len(centers)):
        if Distancetype==0:
            if(EuclideanDistance(point, centers[i]) < mindis):
                mindis = EuclideanDistance(point, centers[i])
                finalindex = i
        else:
            if(GreatCircleDistance(point, centers[i]) < mindis):
                mindis = GreatCircleDistance(point, centers[i])
                finalindex = i
    return finalindex

def addPoint(point1, point2):
    point = point1+point2
    return point

def Cov(centers1, centers2):
    dif = 0
    for i in range(len(centers1)):
        dif += EuclideanDistance(centers1[i], centers2[i])
    cov = float(dif/len(centers1))
    return cov
```

Image 8: Screenshot of closestPoint function, addPoints function, and convergeDist function.

Once we defined the major functions, the essential part of the code that is responsible for k-means algorithm came into play. Here, we used Spark SC to read the data and took some random samples from the points as original centers. Then, we used map to group the points according to their closest centers and count each point. Then, we used reduce by key to sum up the points with the same index of centers as well as their counts, and divide the sum by the total count to find out the new centers for each cluster. We kept repeating this operation until the convergence reach a reasonable value, in our case “0.1”. Eventually when “convergeDist” reached the value of 0.1, we wrote out our output in “csv” files.

```
    mydata=sc.textFile(logfile)
    points=mydata.map(parseVector).cache()
    centers=points.takeSample(False,K,1)
    cover=float(1.0)

    while (cover>0.1):
        p_cpairs = points.map(lambda point: (closest(point, centers), (point, 1)))
        sum_clusters = p_cpairs.reduceByKey(lambda a,b: (addPoint(a[0], b[0]), a[1]+b[1]))
        newcenters = sum_clusters.map(lambda a: (a[1][0]/a[1][1])).collect()
        cover = Cov(centers, newcenters)
        centers = newcenters
    p_c = p_cpairs.map(lambda a: (a[0], a[1][0])).collect()
#     clusters = p_c.reduceByKey(lambda a,b:(a,b)).collect()
    print(p_c)
    p_clists = [[] for i in range(K)]
    for i in range (len(p_c)):
        n = p_c[i][0]
        p_clists[n].append(p_c[i][1])
    for i in range (len(p_clists)):
        filename = 'cluster'+str(i)+'.csv'
        csvfile = open(filename, 'w')
        writer = csv.writer(csvfile)
        for j in range (len(p_clists[i])):
            writer.writerow([p_clists[i][j]])
        csvfile.close()
#     sc.close()
```

Image 9: Screenshot of the core of the code that's responsible for the algorithm.

After all the setup, we finally get to implement our k-means algorithm on the dataset as instructed in problem 3, step 3. Without loss of generocity, in this paper, we will only discuss one piece of comparison between Great-circle-distance k-means algorithm and Euclidean distance k-means algorithm, using the results from **synthetic location data**. Note that complete visualized results will be attached below for further review (not in the same order as in the project description). Visualized outputs are shown below:

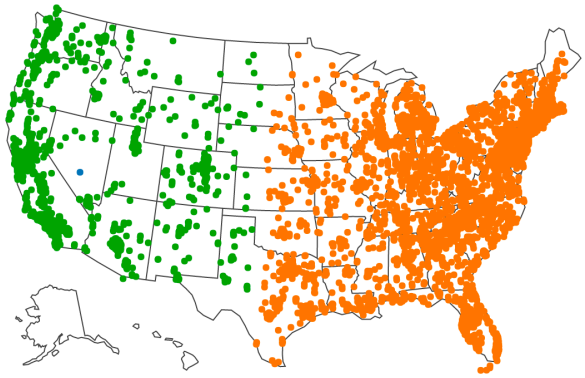

Image 10: Visualization of **Euclidean** k-means clustering on synthetic location data using k = 2. Each color represents a different cluster. If you look closely, you will see the blue dots, which represent the centroid of each cluster.

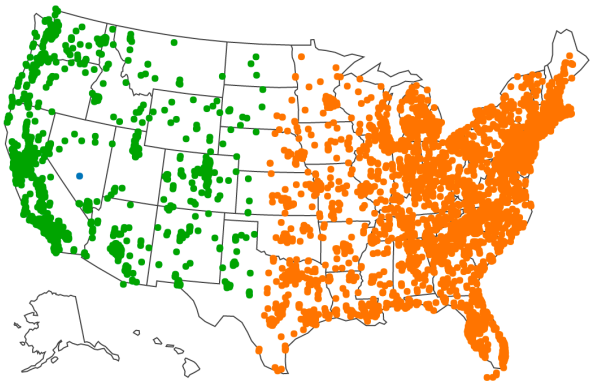

Image 11: Visualization of **Great-circle-distance** k-means clustering on synthetic location data using k = 2. Again, each color represents a different cluster. Blue points represent the centroids for each cluster.

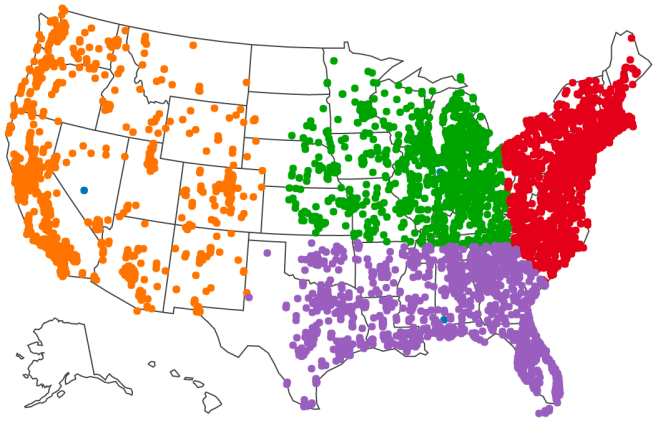

Image 12: Visualization of Euclidean k-means clustering on synthetic location data using k = 4.

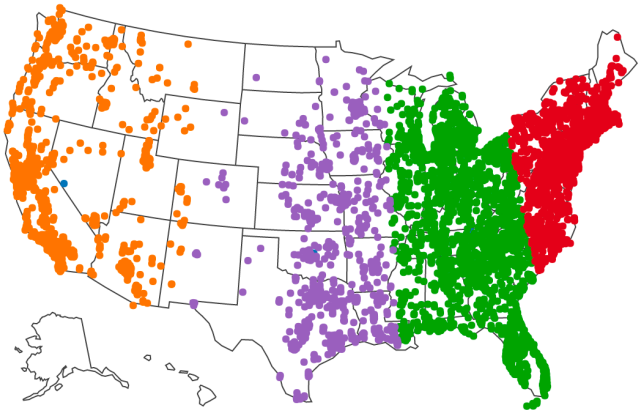

Image 13: Visualization of Great-circle-distance k-means clustering on synthetic location data using k = 4.

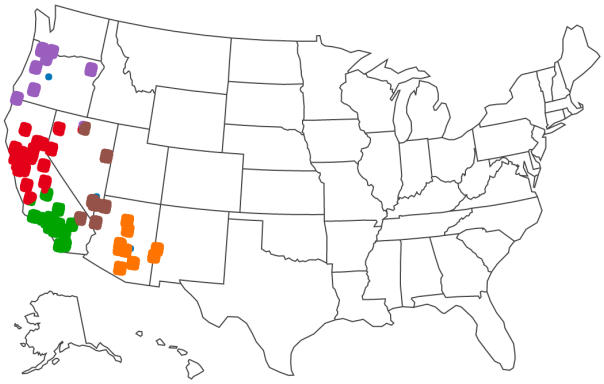

Image 14: Visualization of Euclidean k-means clustering on device location data using k = 5.

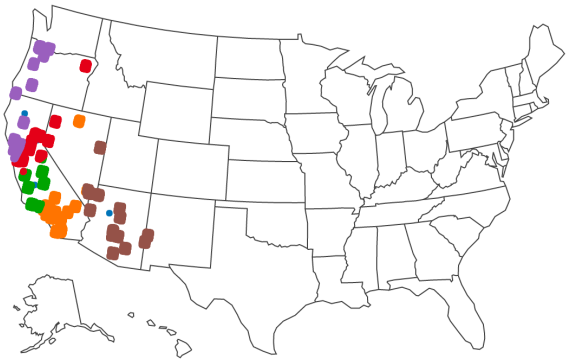

Image 15: Visualization of Great-circle-distance k-means clustering on device location data using k = 5.

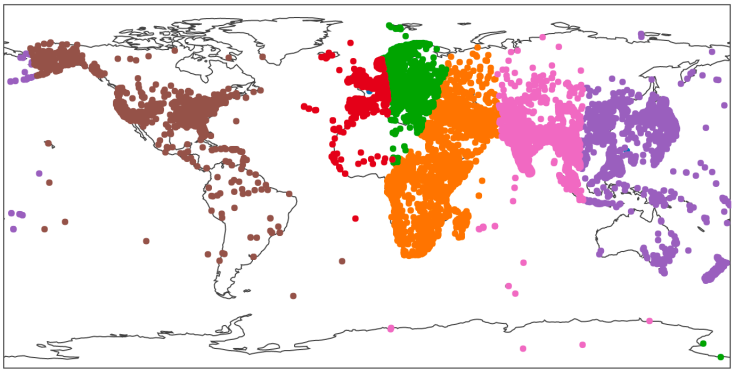

Image 16: Visualization of Euclidean k-means clustering on the DBpedia location data using k = 6.

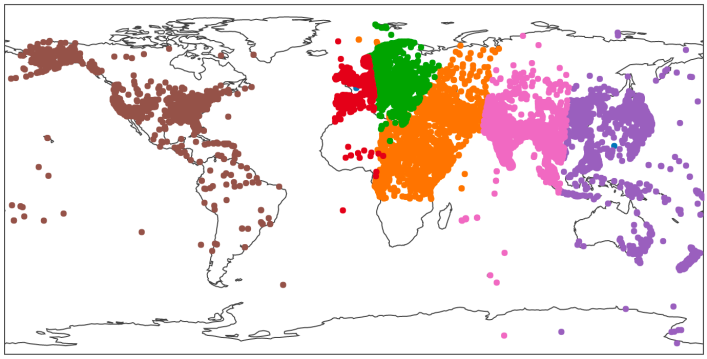

Image 17: Visualization of Great-circle-distance k-means clustering on the DBpedia location data using k=6.

As we can see, for the same synthetic location dataset, if we have “k=2”, the two distance calculation methods, namely Great-circle-distance and Euclidean distance, generated practically the same results. However, as the value of parameter k increases to 4, the resulting clusters are a lot different. The question comes down to “which method is better?”. Our conclusion is that for small values of k, such as 2 or 3, Great-circle distance and Euclidean distance will generate very similar results; however, when we have large values of k, Great-circle distance k-means algorithm will be more realistically reasonable because our earth is spherical. Great-circle distance will be more accurate than Euclidean distance.

Runtime Analysis

	Device Location Data	Synthetic Location Data	DBpedia Location Data
Persistent RDD	189568ms	175348ms	4822389ms
W/O Persistent RDD	208805ms	195887ms	5199625ms

From our findings, we conclude that Persistent RDD significantly decreased the processing time and improved running efficiency. Although Persistent RDD stores RDD into

buffer and takes up more memory, it's a lot more time saving. Such finding validates the space/time tradeoff hypothesis.

Big Data and Cloud Execution

For this final execution, we found the Taxi Service Trajectory dataset on UCI Machine Learning Repository. The dataset contains more than 1.7 million records of the one-year trajectory data collected by 442 taxis running in the city of Porto, Portugal. We intended to study the pick-up location latitude/longitude pairs in order to find the optimal locations for taxi stations in Porto, Portugal.

TRIP_ID	CALL_TYP	ORIGIN_C	ORIGIN_S	TAXI_ID	TIMESTAN	DAY_TYPE	MISSING_I	POLYLINE
T1	B	NA	15	20000542	1.41E+09	A	FALSE	[[-8.585676, 41.148522], [-8.585712000000001, 41.148638999999996], [-8.585685000000002, 41.148855000000005],
T2	B	NA	57	20000108	1.41E+09	A	FALSE	[[-8.610876000000001, 41.14557], [-8.610858, 41.145579000000005], [-8.610903, 41.145768], [-8.610444, 41.1461905
T3	B	NA	15	20000370	1.41E+09	A	FALSE	[[-8.585739, 41.148558], [-8.585730000000002, 41.148828], [-8.585721, 41.14897200000001], [-8.586288, 41.149017(
T4	B	NA	53	20000492	1.41E+09	A	FALSE	[[-8.613963, 41.141169], [-8.614125000000001, 41.141124000000005], [-8.615088, 41.140926], [-8.615276999999999
T5	B	NA	18	20000621	1.41E+09	A	FALSE	[[-8.619902999999999, 41.148036000000005], [-8.619894, 41.148036000000005]]
T6	A	42612	NA	20000607	1.41E+09	A	FALSE	[[-8.630613000000002, 41.178248999999994], [-8.630613000000002, 41.178248999999994], [-8.630739, 41.1782310(
T7	B	NA	15	20000310	1.41E+09	A	FALSE	[[-8.585622, 41.148918], [-8.585640000000001, 41.1489], [-8.585919, 41.148945], [-8.586369000000001, 41.1489269
T8	A	31780	NA	20000619	1.41E+09	A	FALSE	[[-8.582922, 41.181057], [-8.582004, 41.181813000000005], [-8.581077, 41.182965], [-8.580114, 41.184018], [-8.5792
T9	B	NA	9	20000503	1.41E+09	A	FALSE	[[-8.606529, 41.14467], [-8.606673, 41.144724], [-8.606799, 41.144706], [-8.60679, 41.144679000000004], [-8.606708
T10	B	NA	15	20000327	1.41E+09	A	FALSE	[[-8.585658, 41.148576000000006], [-8.585703, 41.148603], [-8.585730000000002, 41.148603], [-8.585739, 41.14860
T11	B	NA	56	20000664	1.41E+09	A	FALSE	[[-8.591228999999998, 41.16270600000001], [-8.591228999999998, 41.162715], [-8.59122, 41.162715], [-8.59100400
T12	C	NA	NA	20000160	1.41E+09	A	FALSE	[[-8.585694, 41.148603], [-8.585757000000001, 41.148684], [-8.585712000000001, 41.148882], [-8.585757000000001
T13	C	NA	NA	20000017	1.41E+09	A	FALSE	[[-8.580105000000001, 41.159394], [-8.580231, 41.159358000000005], [-8.581517999999999, 41.15934899999999], [
T14	C	NA	NA	20000312	1.41E+09	A	FALSE	[[-8.665353, 41.186161], [-8.666892, 41.188311], [-8.667045, 41.190705], [-8.665812, 41.194017], [-8.662734, 41.19715

Image 13: Screenshot of a little part of the original Taxi Service Trajectory dataset.

At first, we created our own AWS EMR and established a cluster with Spark and Zeppelin Notebook. Then, we pre-processed our Taxi Service Trajectory dataset. The two screenshots below represent these two steps:

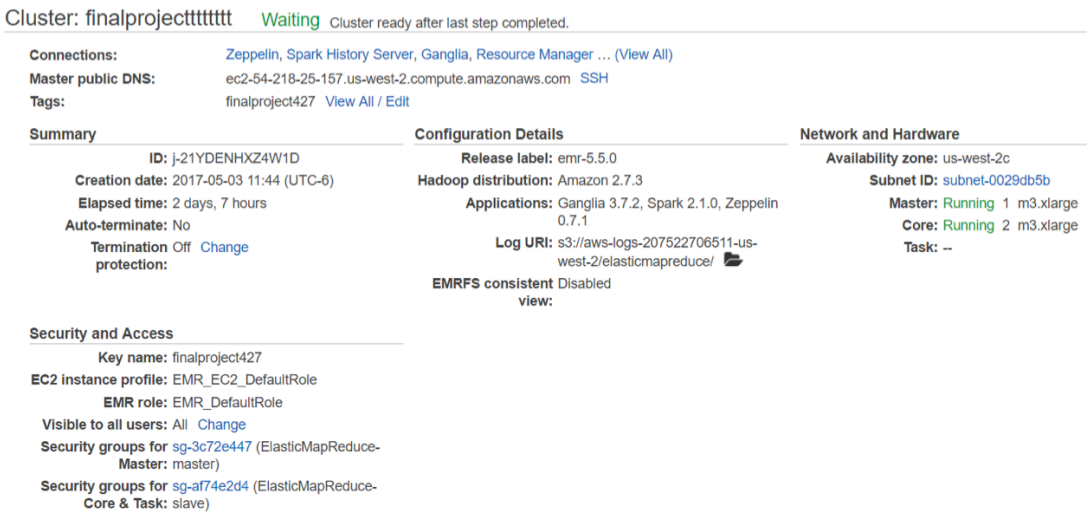

Image 14: Screenshot of EMR cluster.

956154	-8.64913	41.15442
956155	-8.5863	41.15006
956156	-8.61213	41.17274
956157	-8.59588	41.15395
956158	-8.6156	41.14096
956159	-8.60428	41.16098

Image 15: Screenshot of the last few records of the pre-processed data. We ended up with 956159 records.

Now, we implement our k-means algorithm on the cluster, using Zeppelin Notebook. Throughout the process, we tried different values of k, namely "k=2, 4, and 6", to find the optimal results that are align with both our intuition and reality. To achieve this, we used Google Map to check if the resulting latitude/longitude pairs are viable (for example, we check if there are human/economic activities around the resulting spot.). We attached the screenshot of the result that shows the locations of six taxi stations on Google Map below:

Image 16: Screenshot of the resulting six taxi stations on Google Map. Yellow stars represent six centroids of the clusters, which are optimal locations for six taxi stations.

Their latitude/longitude values are as follows:

41.14553267	-8.61895538
41.16144281	-8.62986833
41.16282774	-8.65925795
41.16635115	-8.60612597
41.14578482	-8.60973315
41.15932692	-8.5823413

Runtime Table

	Data Size	Runtime
K = 4	956159 records	2789375ms
K = 6	956159 records	3839582ms

As we can see, by comparing with the previous runtime table, implementing our k-means algorithm on EMR significantly improved processing efficiency.

Reference

Plotly. N.p., n.d. Web. 4 May 2017. <https://plot.ly/>.

"Taxi Service Trajectory - Prediction Challenge, ECML PKDD 2015 Data Set." *UCI Machine Learning Repository*. N.p., 11 July 2015. Web. 3 May 2017.