CSE 427

Final Project #3

Pu Chai & Shangqing Li

k-Means for Geo-location Clustering in Spark

Background

Clustering usually refers to the process of grouping a set of objects into multiple clusters that each consists similar objects. Through clustering, similar objects are grouped into the same cluster while objects that are not similar to each other are put in different clusters. Given this property, clustering has been widely used in real world applications. Companies can use clustering to group customers in order to deliver advertisements more efficiently. Also, we can analyze data such as web logs and browsing history through the use of clustering. Similar patterns and trends may be discovered through clustering these data. Due to the huge size and high dimensionality for data to be used in clustering, efficient implementation, both in terms of space and time, becomes an important problem for engineers to solve and improve. In this project, we will approach clustering by implementing the k-means algorithm with Spark in Python. We will then visualize and analyze the clustering results from three different datasets based on our implementation.

Approach

In this project, we approached the clustering problem of Geo-locations through implementing the k-means algorithm. The k-means algorithm is a distance-based method that updates the locations of cluster centroids iteratively until convergence. Specifically, the k-means algorithm aims at assigning all the target data points into k different clusters based on the distance between the target point and each cluster's centroid.

The k-means algorithm is initialized by picking up k centroids within the given data point set, one for each of the k clusters. In order to choose the initial k centroids, we randomly pick the

first centroid and then find the rest of the centroids by finding the point that is farthest from all previous centroid points. In order words, we choose the farthest point from the first two centroids as our third centroid point.

After the k centroids are initialized, we will assign each of our target data point to one of the clusters based on its distance to centroids. There are multiple possible approaches in order to calculate distance, and in this project we will look at Euclidean distance as well as Great Circle distance. Every single point is assigned to a cluster during one iteration of the algorithm. Then we update the centroid of the cluster by averaging all the data points within the cluster. The updated centroids would be compared to the centroids from previous iteration by calculating the change in distance for each centroid between the two iterations. The total distance change of all centroids would then be calculated. The algorithm terminates when the centroids do not change between iterations, which is equivalent to very trivial total distance change for all centroids. The ideal convergence criteria would be zero distance change, but usually in real world the algorithm terminates before this perfect criteria is met.

When the k-means algorithm terminates, it outputs both the final cluster centroids and the clusters consist of data points. The number k in the k-means algorithm can be chosen based on how users would like the data points to be clustered. The k-means algorithm groups the data by minimizing the sum of squared distances between the data points and their respective closest centroid.

Implementation

Our implementation consists of three stages: pre-processing data, implementing k-means algorithm, and visualizing result clusters. We pre-processed our input datasets and implemented the k-means iterative algorithm through Spark processing engine with Python. In this project, we implemented k-means clustering on three sets of data, including the device location data, the synthetic location data, and the DBpedia location data.

1. Preprocessing

   We pre-processed all three datasets with Spark in Python. First, we pre-processed our device location data with the following steps. After reading in lines of device location

data, we kept only the latitude and longitude pairs. Then we filtered out all pairs that contain any null or zero values. The pre-processed data is stored as comma-delimited latitude and longitude pairs. Similarly, we pre-processed the synthetic location data in the same manner and the result data is stored in the format "latitude, longitude" on each line. Given the massiveness of the DBpedia location data set and the limited memory of our machine, we decided to only use the US locations. We decided to use filter location data by latitude and longitude ranges so that instead of losing data (many US data records do not specify country) through filtering by country. We included some data points adjacent to US as well. The pre-processed data file is also stored in the format of comma-delimited latitude and longitude pairs.

2. Implementing k-means

   We implemented the k-means algorithm in Python. Our Python program accepts four input arguments: kmeans.py, input file path, user-specified k, and user-specified mode for calculating distance. Users can choose k by typing in integer number, and they can also specify whether Euclidean distance or Great Circle distance is used through the mode parameter. Specifically, integer 0 represents Euclidean distance while integer 1 represents Great Circle distance.

   Spark RDD is used in our implementation of k-means clustering. We cached all the data points in the RDD so that we do not have to load them in every iteration. In order to make comparisons, we also tried to run the k-means algorithm without using Spark RDD (without caching data points). The results section would discuss the performance difference of our implementation with and without using RDD.

   We implemented five functions to help our implementation. The closestPoint function takes in the target point, the list of centroid points, and the mode parameter. It then returns the index of the closest centroid point to the target point with the user specified distance calculation approach. The EuclideanDistance function returns the Euclidean distance between two points while the GreatCircleDistance function returns the Great Circle distance between two points. The addPoints function returns a point resulted from adding two points together, and the parsePoint function reads in a line of input file and then returns an input point.

We initialized the centroids by using Spark's takeSample() action. Then we update the centroids as well as the convergeDist variable iteratively. The convergeDist variable stores the sum of distances between the centroids of two iterations. When convergeDist is smaller or equal to 0.1, our k-means algorithm terminates and the final centroids are printed out. Finally, we assign each point to a cluster based on the final centroids and then store each point together with its cluster index in the output file.

3. Visualizing clusters

   Each cluster is visualized on the map tool provided by MapsData. We also visualized the locations of cluster centroids. We will discussion the visualization of clusters in our results section.

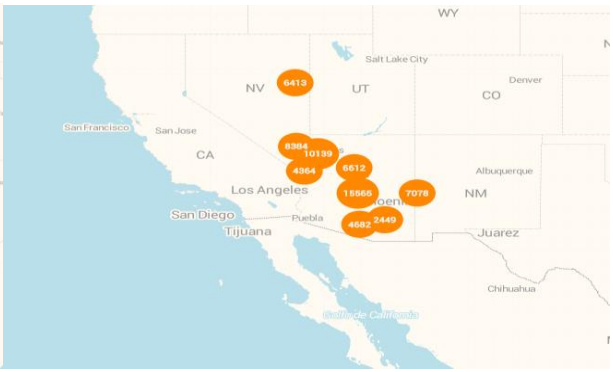
Results

1. Device Location Data

For each cluster, we created a separate plot to better juxtapose the results. Since we have five clusters (k = 5), we have five different clusters plots.

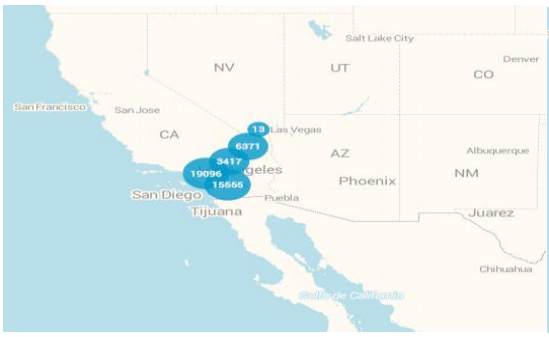
The first cluster is located around Arizona and Nevada.

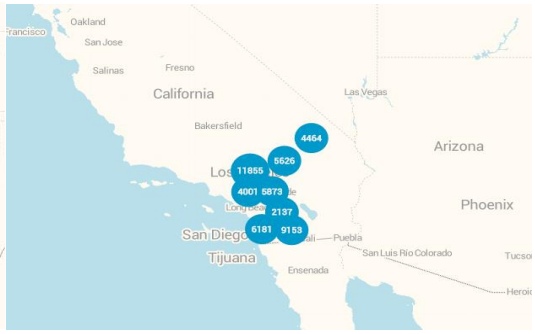
Euclidean Cluster 1



Great Circle Cluster 1

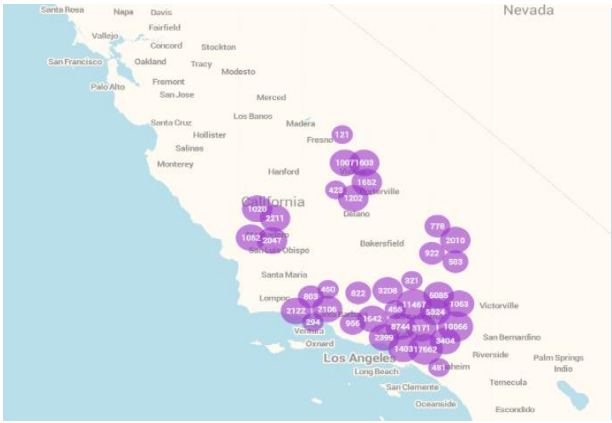
As we can see, the results from the two distance calculation approaches do not deviate from each other very much. Same result applies to the second cluster, which is located around Los Angeles and San Diego.





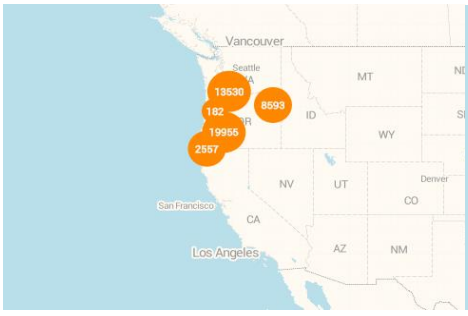
Euclidean Cluster 2 Great Circle Cluster 2

Similarly, the third cluster represents device locations around San Francisco and the Bay area. Although given the massive amount of data, MapsData cannot draw all single data points on the plot, we can still see that the device locations are located around the communities around the Bay area.



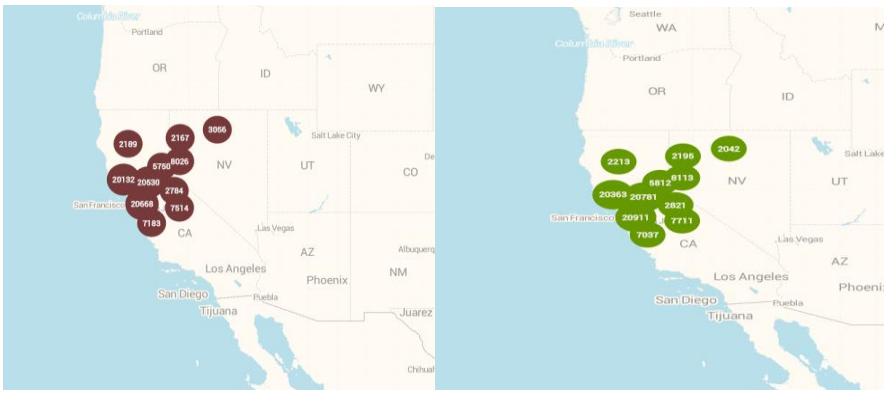
Euclidean Cluster 3 Great Circle Cluster 3

Similarly, the fourth cluster contains device location data in Washington and Oregon.



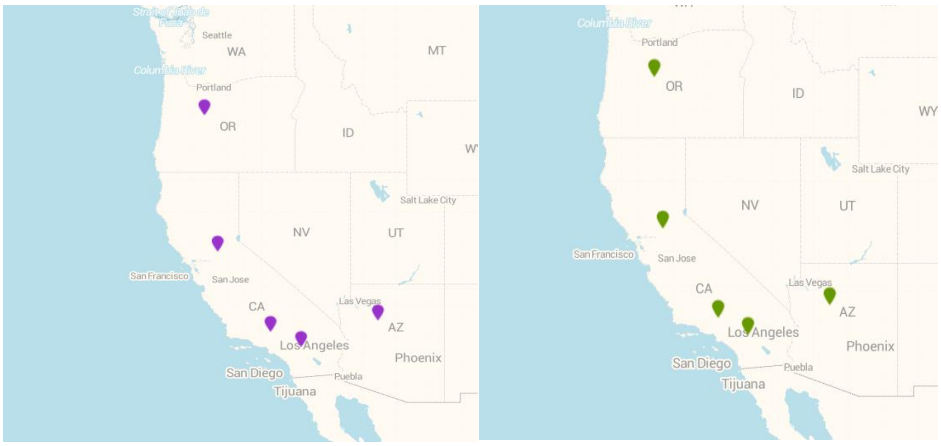
Euclidean Cluster 4 Great Circle Cluster 4

The last cluster contains device location data around north California, as shown in the plots below.



Euclidean Cluster 5 Great Circle Cluster 5

The plots below indicates that all the five cluster centroids using the two different distance calculation approaches are almost at the same locations.

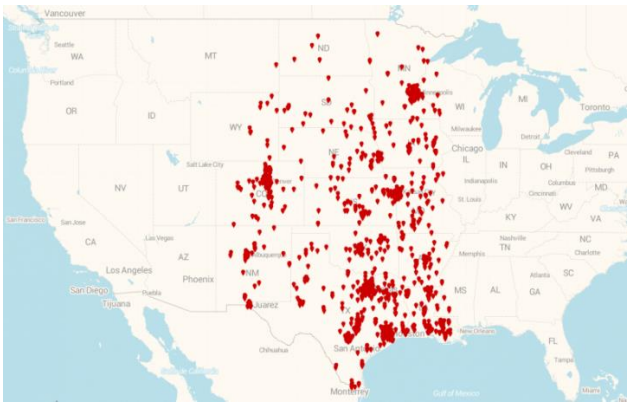


Euclidean Centroids Great Circle Centroids

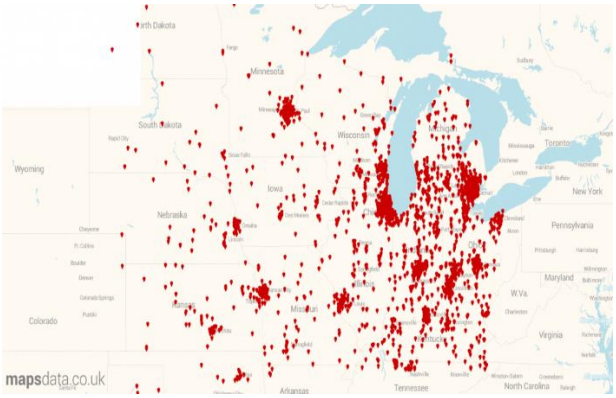
In conclusion, the first cluster is mainly allocated in Arizona and Phoenix, the second cluster is in the south side of California, the third one is located in middle California, the fourth cluster is in Seattle and Oregon and the last cluster located in north side of California and Nevada. And there exists slightly difference between using Euclidean distance method and Great Circle distance method. Since the data is pretty dense, both distance methods does not affect clustering result very much.

2. Synthetic Location Data

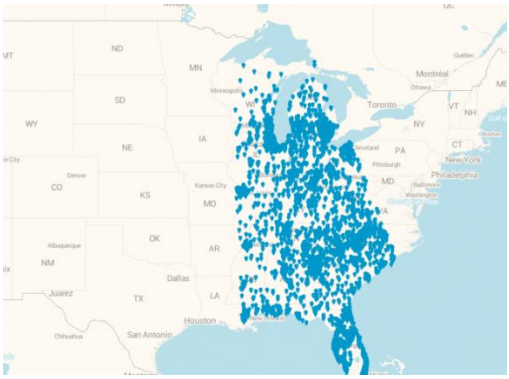
The left is the cluster computed with Euclidean Distance and the right clusters are computed with Great Circle distance.



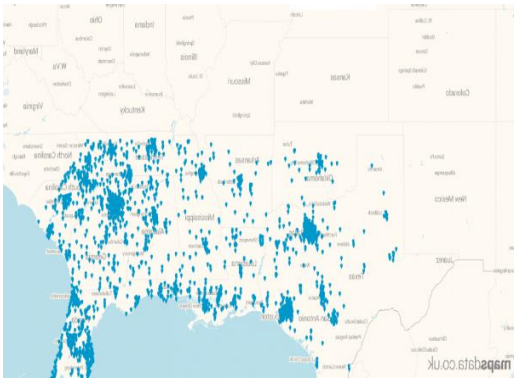
Euclidean Cluster 1



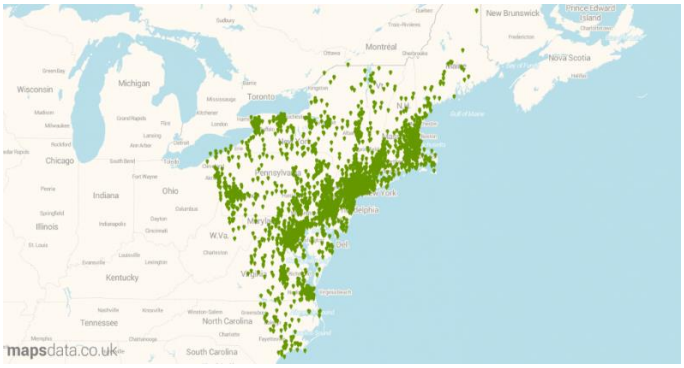
Great Circle Cluster 1



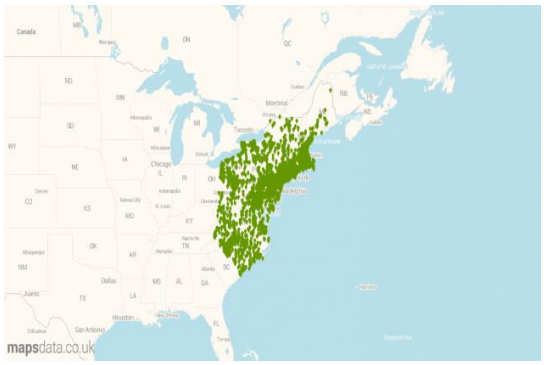
Euclidean Cluster 2



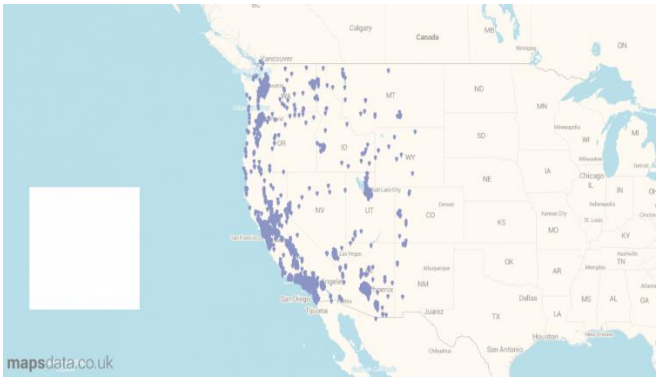
Great Circle Cluster 2



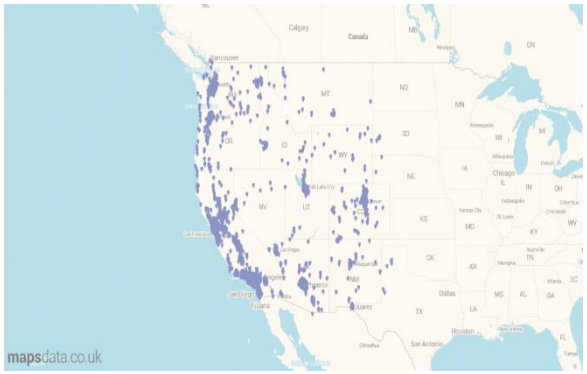
Euclidean Cluster 3



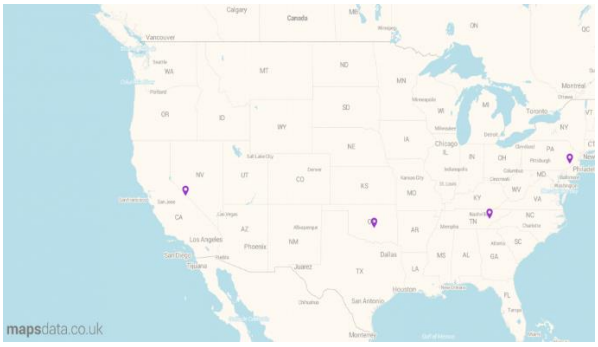
Great Circle Cluster 3



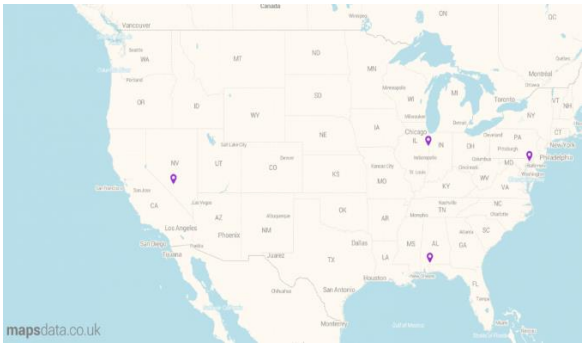
Euclidean Cluster 4



Great Circle Cluster 4



Euclidean Centroids

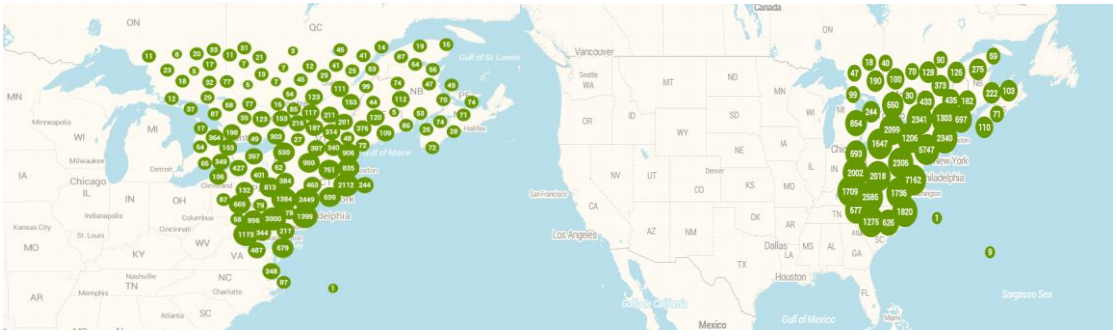


Great Circle Centroids

As we can see from the above plot comparisons, we can see that when the location data is not massive, the difference between the two distance calculation approaches can be quite obvious. The result for clustering is actually affected, and the clustering centroids deviate significantly.

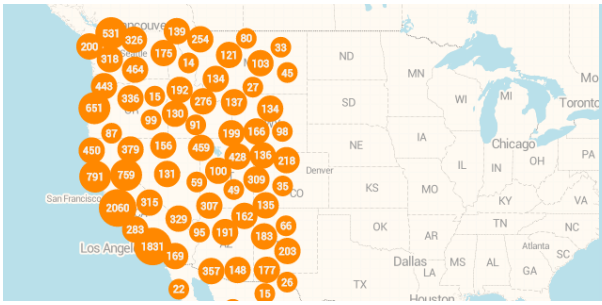
3. DBpedia Location Data

The left is the cluster computed with Euclidean Distance and the right clusters are computed with Great Circle distance.



Euclidean Cluster 1

Great Circle Cluster 1

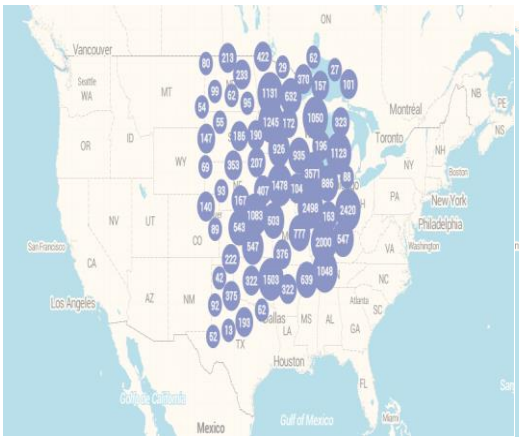


Euclidean Cluster 2

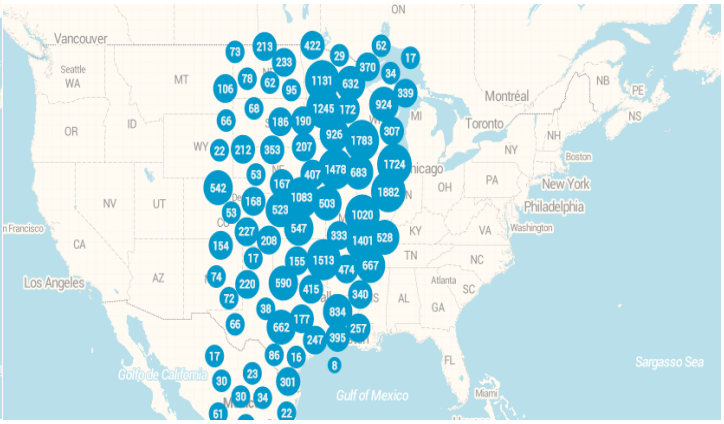
Great Circle Cluster 2

Euclidean Cluster 3

Great Circle Cluster 3



Euclidean Cluster 4



Great Circle Cluster 4

Euclidean Centroids

Great Circle Centroids

Same as in the case of Synthetic Location data, calculating distance with Euclidean approach and Great Circle approach results in different final centroids in our k-means clustering implementation. The size of this dataset is in-between the two previous datasets, since we only filtered out data belongs to US.

Analysis

RDD provides the system an easy and smart way of storing and caching data. With the use of RDD, the runtime of our data sets through k-means algorithms can be far shorter than without using RDD especially with large data.

With RDD cache:

DataSet1(devicesstatus)	k value (# of clusters)	Trial 1	Trial 2	Trial 3	trial4
EuclideanDistance	5	13mins	11mins	14min	11min
GreatCircleDistance	5	12mins	13mins	12mins	14mins
DataSet2 (geo_sample)	k value (# of clusters)	Trial 1	Trial 2	Trial 3	trial4
EuclideanDistance	4	56s	1.9min	2min	1.6min
GreatCircleDistance	4	2.6mins	1.4mins	2.4mins	2mins
DataSet3(US_Only)	k value (# of clusters)	Trial 1	Trial 2	Trial 3	trial4
EuclideanDistance	2	2.8mins	1.9min	2min	1.6min
GreatCircleDistance	2	2.6mins	1.4mins	2.4mins	2mins
EuclideanDistance	4	4.6min	4.4min	4.7min	4.9min
GreatCircleDistance	4	3.9min	4.2min	4.4mins	4.2mins

Without using RDD:

DataSet1(devicesstatus)	k value (# of clusters)	Trial 1	Trial 2	Trial 3	trial4	AVG
EuclideanDistance	5					
GreatCircleDistance	5		Over 30 minutes			
DataSet2 (geo_sample)	k value (# of clusters)	Trial 1	Trial 2	Trial 3	trial4	
EuclideanDistance	4	49s	1.4mins	2.6mins	3.1mins	
GreatCircleDistance	4	1.9mins	2.2mins	3.6mins	3.2mins	
DataSet3(US_Only)	k value (# of clusters)	Trial 1	Trial 2	Trial 3	trial4	
EuclideanDistance	4	6.7mins	5.8mins	8mins	7.2mins	
GreatCircleDistance	4	7.4mins	6.6mins	6.2mins	8mins	

When the data size is small, the runtime does not increase a lot. However, when we run through large data set (dataset1-devicesdata), the program runs really slow. And we have waited over 30 minutes but the program still does not successfully finished.

Conclusion

In this project, we successfully implemented k-means clustering iterative algorithm for geo-location with programming language python through spark processing engine. With the use of RDD provided in spark, we are able to run our implementation with large data sets in a reasonable amount of time. However, without using RDD, the runtime on large data sets can go overly long. Spark RDD provides a smart and easy way to cache data in memory and thus will greatly improve the efficiency.