CSE427 Final Project

k-means for Geo-location Clustering in SPARK

Ren Wan, Alicia Sun, Lei Wu

May 5, 2016

1 Introduction

In this project we implemented iterative K-means algorithm in Spark to solve clustering problems for three different datasets based on the geo-locations of datas. The problem is important since clustering can help to analyze users' behaviors based on their geo-locations. K-means is a distance-based method that iteratively updates the location of k clusters centroids until convergence. However, each dataset in this project contains from 9k records up to 450k records of spatial locations with different formats, which would take a lot of time and space to cluster.
Spark is particularly useful in the sense that datasets are stored in cached RDD and kept in memory, which saves time and memory when accessing the data in each iteration. Before deploying into the real clustering job, we need to first pre-process the data by filtering out incomplete/incorrect records and putting the datasets into a standardized format. In the K-means algorithm, a new list of centroids and clusters are produced in each iteration and used in the next iterations until the centroids remain stable and the algorithm converges. The implementation was successful and we visualized the clusters with Google Maps.

2 Approach

2.1 Pre-processing Data

1. Finding the delimiter: Since we have data from a variety of sources with different format, we need to separate fields according to the delimiter in different formats
2. Filter out bad records: As the valid record should have 14 fields in total, those records that are not of length 14 are considered bad data and will be filtered out
3. Filter out records without any location(latitude and longitude of 0)
4. For the model field, split it by space and only keep the model name
5. Extract output with format "latitude,longitude,date,model,deviceID"
6. Save the output to HDFS

2.2 Implementing k-means Algorithm in Spark

Initialization: randomly choose k points as initial centroids
In each iteration:

Task1 For every data point, find the closest center index in terms of great circle distance. Aggregate by center index and calculate sum of latitude,longitude and number of datas in each cluster.

Task2 For each cluster, calculate average latitude and longitude, and uses it as the new cluster center coordinates for the next iteration. Test if the amount of center location changes between iterations is less than the prespecified convergeDist (0.1 in our case), and if so, terminate the algorithm.

2.3 Visualizing Data

For visualizing geolocation data, we choose to convert the outputs from spark into "csv" files and feed them to Google Maps application. Since the datasets are quite large, and also as a consideration of better displaying data points, we choose a random subset of data points to visualize in Google Maps.

3 Implementation

3.1 MapReduce Job1

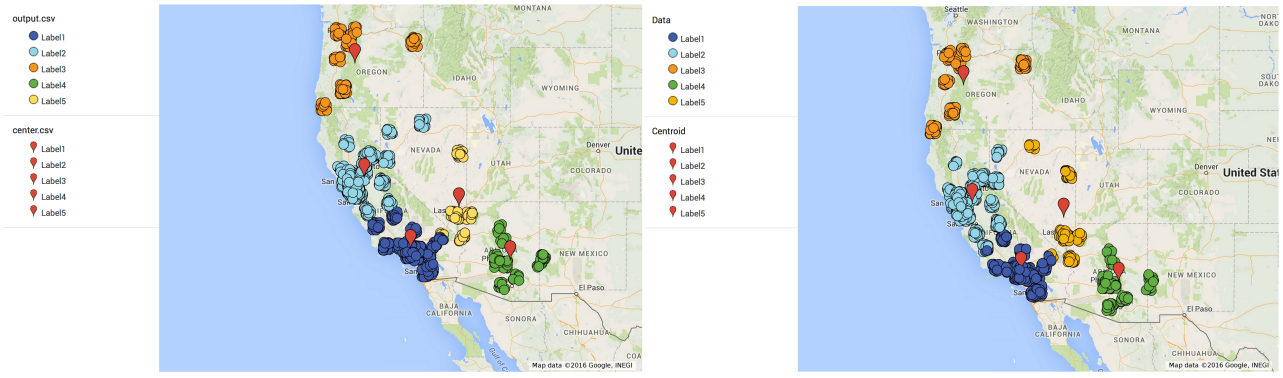
- Map Phase: Given K centroids, find closest center for each data points.
  input format:(latitude,longitude)
  output format: (closest center index,((latitude,longitude),1))
- Reduce Phase: For each cluster center, sum up latitude and longitude of points in that cluster, also count number of points in that cluster
  Input Format: (cluster center index,[((latitude,longitude),1), (latitude,longitude),1),.....])
  Output Format: (cluster center index,[(sum of latitudes,sum of longitudes),number of points in this cluster])

3.2 MapReduce Job2

- Map Phase: For each index, calculate the average latitude and longitude and use it as the new cluster center
  Input Format: (cluster center index, [(sum of latitudes, sum of longitudes), number of points in this cluster]) Output Format: (cluster center index, (average of latitudes, average of longitudes))
- Reduce Phase: calculate sum of location changes between current cluster centers and previous cluster centers. If less than the convergeDist, break out of the iteration and save the output. Otherwise, use the average as the new cluster center.

3.3 Converting to CSV

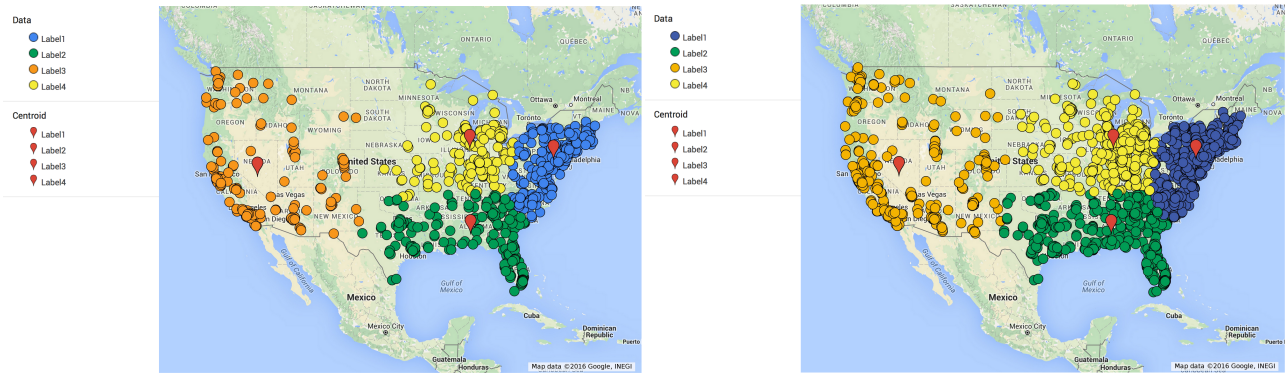
After getting the output from Spark, we then use a script to convert the result to the "csv" format for displaying in Google Maps. Each line in the "csv" file has 3 fields separated by ",", representing label, latitude and longitude respectively.



(a) Great Circle Distance

(b) Euclidean Distance

Figure 1: Device Data with K=5



(a) Great Circle Distance

(b) Euclidean Distance

Figure 2: Synthetic Data with K=4

4 Results and interpretation

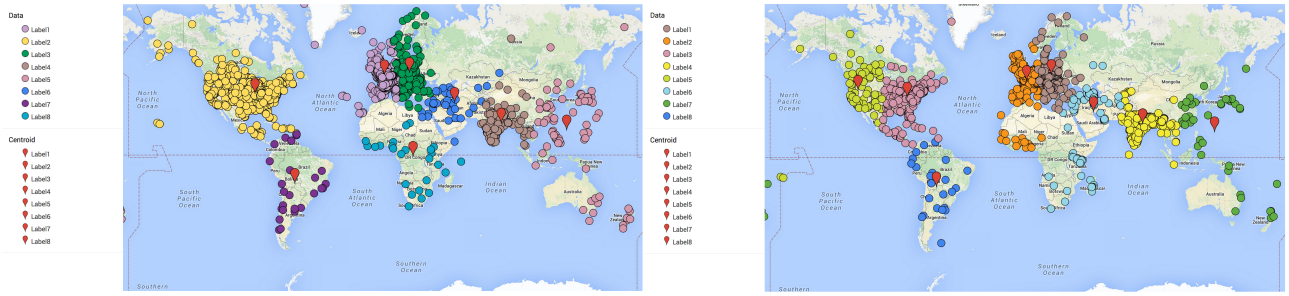
4.1 Visualization

See Figure 1, Figure 2 and Figure 3.
For DBpedia data, we experimented with K=6, K=7 and K=8, and found that K=8 gives the most reasonable clustering result.

4.2 Discussion

- For the device dataset, since the datas only spread out in western United States and are close to each other, euclidean distance very similar to the result from the great circle distance measure.

- In the device dataset, data points are all over United States, and great circle distance generates a better clustering solutions than euclidean distance.



(a) Great Circle Distance

(b) Euclidean Distance

Figure 3: DBpedia Data with K=8

- In the DBpedia dataset, the data points spread globally. That being said, data points are pretty far apart , and we can see that euclidean gives a much less desirable approximation compared to great circle distance.In the great circle measurements, datas are separated into eight clusters in their corresponding geolocations :North America, South America, Eastern Europe, Western Europe, Africa, Middle East, East Asian and Australia, and South Asia. While in Euclidean measure, US is divided into two and the real distance between datas are not persisted.

5 Runtime Analysis

We used local mode with 4 threads with great circle distance measure and $k = 4$. For the DBpedia dataset, we put a bounding box around US since the global dataset is unlikely to converge under K=4 and would be extremely slow running without persistant RDDs.

Runtime	Device Data	Synthetic Data	DBpedia Data
Size	240k	9970	115057
With RDDs	22min	36s	7.6min
Without RDDs	31min	43s	9.2min

Table 1: Runtime Analysis with/without the use of persisting RDDs.

5.1 Discussion

- Runtime increases along with the size of the dataset. The device data is the largest and it takes longest to run.
- Runtime increases dramatically if not using persistant RDDs. Persistant RDDs stores partition of dataset in memory and reuses them later, which allows future actions to be much faster in iterative algorithms.

6 Future Improvements

6.1 Finding Centroids

For finding centroids during each iteration, we simply calculate the average latitude and longitude of all points belong to the cluster. However, when we are using great circle distance as the measurement, this approach may cause errors, especially on high-latitude areas. For example, if we have 3 points which has the geolocation as 89N 0E, 89N 180E, and 0N, 0E respectively. If we use simple averaging to find the centroid, it will at around 60N 60E. But as we know 89N 0E and 89N 180E are all near the North Pole, which can be considered as the same point, and the correct centroid of great circle measurement should lay around 60N 0E.

6.2 Evaluating Clustering Results

Another part of this project that can be improved in the future is the way of finding the best k for clustering. Currently, we experimentally choose the values of k, and evaluate the results of clustering visually. In the future, we can use the internal or external methods for better evaluation. For internal evaluation, we can calculate the Davies-Bouldin index for each k in the range, and choose the k with the smallest value. For external evaluation, we can use some labeled data points as the benchmarks and calculate measures like Rand measure, F-measure or Jaccard index of each possible k.