Geo-Location Clustering with K Means

By:
Samantha Han
Hakkyung Lee
Skylar Nam
Zefang Tang

Introduction and Motivation

Clustering, according to the definition from Stanford University, is a process of grouping a set of data points into clusters so that points that are put within the same cluster are similar to each other whereas points from different clusters are dissimilar. Clustering has many useful applications for marketing, logistics, and document classification. We clustered geo-location data, which can be very intuitively visualized. The insight gained from location clustering can be used to determine information such as the best locations to add new cell towers, which the analysis of Loudacre's device data could be used for. In addition, we applied the algorithm to Pokemon Go data to find places where rare Pokemon are likely to spawn. This information can be valuable to avid players of Pokemon Go who are interested in catching rare Pokemon.

In order to solve the clustering problem in a parallel fashion, we implemented the *k-means algorithm* in Spark. As a distance-based method, the algorithm iteratively updates the location of *k* cluster centroids until the change in the mean of centroids converges to α=0.1 km, where α is convergeDist.

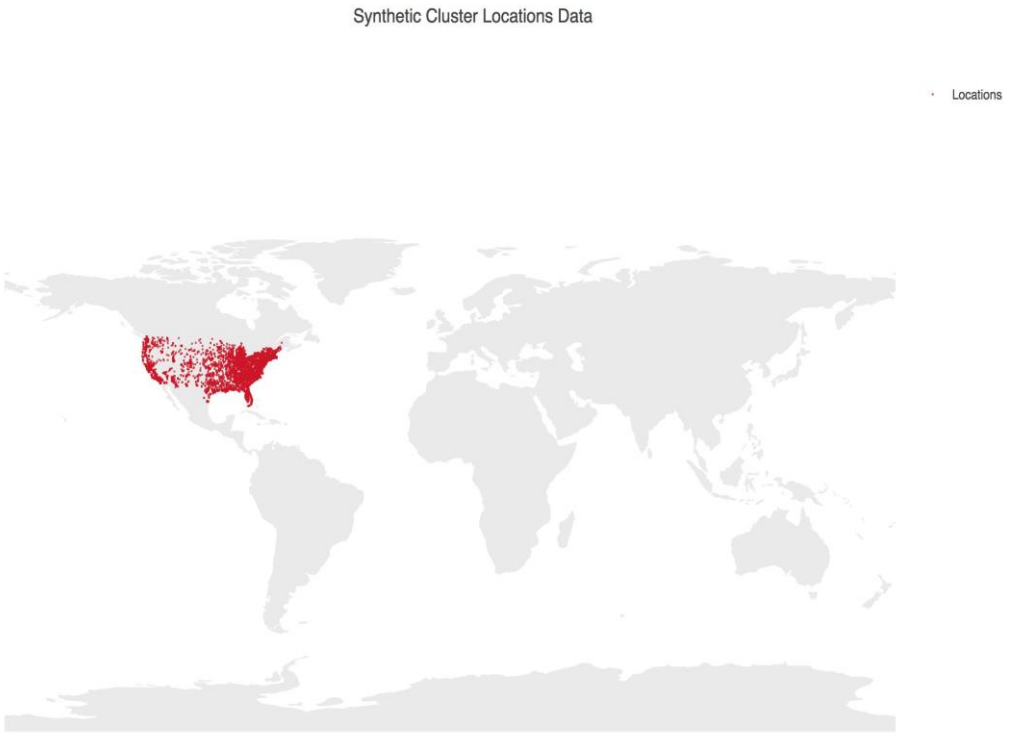
Data Preparation

Before implementing the actual algorithm, we went through pre-processing step in order to convert the data into a standardized format for later processing. The following describes the pre-processing process for device status data:

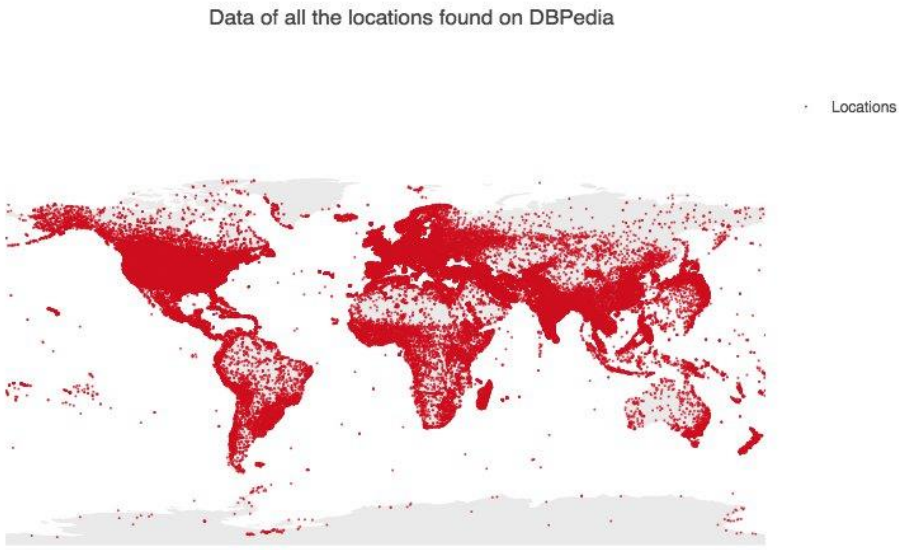
1. Load the dataset
2. Determine which delimiter to use
3. Filter out any records which do not parse correctly; each record should have exactly 14 values
4. Extract the date, model, device ID, and latitude and longitude
    a. date: 1st field
    b. model: 2nd field
    c. device ID: 3rd field
    d. latitude: 13th field
    e. longitude: 14th field
5. Store latitude and longitude as the first two fields
6. Filter out locations that have a latitude and longitude of 0
7. Split the model field that contains the device manufacturer and model name by spaces
8. Save the extracted data as comma separated values file in the /loudacre/devicestatus_etl directory on HDFS
9. Confirm the data in the file(s) was saved correctly

Visualization

Synthetic Location Data



DBPedia Location Data



Clustering Approach

The initial centroids are a *k*-sized random sample of all points in the dataset. On each iteration, the algorithm assigns each point to its nearest centroid, then calculates the new centroids by taking the mean of all points in that centroid's cluster. The distance between points and centroids is calculated using either Euclidean distance or Great Circle distance - this parameter is set by the user. The main difference between the two measures is that the former measures a straight-line distance between the points in 3D space, while the latter measures the distance across the spherical surface of the Earth.

Though a "perfect" algorithm would iterate until the change in centroid locations converges to 0, this algorithm continues iterating until the sum of all changes in centroid locations converges to α=0.1 km. That is, the algorithm calculates the distance between each centroid's new location and former location (using the distance measure specified by the user) on each iteration, and continues iterating until the sum of these distances for all centroids is less than 0.1 km. This requires larger values of *k* to converge more precisely than smaller values. Because data this algorithm runs on covers at least an entire continent, we determined that the alpha of 0.1 km or 100 m was a small enough value for this purpose. We found that this method gave us good results for the clusters without an extremely prohibitively long runtime.

Implementation

The implementation consists of two files, kmeans.py and point_utils.py. kmeans.py is the main Python script that should be submitted to Spark, whereas point_utils.py is a set of functions, classes, and constants that kmeans.py imports and uses. The Spark job can be submitted as follows:

```
spark-submit --master local kmeans.py \
<input_path> <output_path> <distance_measure> <k>
```

where input_path is the path to the input data (e.g. home/cloudera/cse427/final_project/dbpedia), output_path is the path where output data should be saved, distance_measure is either "Euclidean" or "GreatCircle" (case-insensitive), and k is an integer larger than 0.

In order to group latitude and longitude together cleanly, the implementation includes a `LatLonPoint` class, which contains a single point's latitude and longitude. Since some calculations (such as Euclidean distance and adding points) require latitude and longitude to be converted to Cartesian coordinates, we also created a `CartesianPoint` class as well as functions to convert between the two types of points.

The implementation represents the set of *k* centroids as a Python dict, mapping the centroid ID (0, 1, 2,...,*k*-1) to a `LatLonPoint` containing the centroid's latitude and longitude. On each iteration, each point is mapped to a key-value pair, where the key is the

nearest centroid ID and the value is the `LatLonPoint`. For each point, the nearest centroid is found by iterating through the dict of centroids and calculating the distance from the point to each centroid.

Next, the new centroids are found by calculating the means of the points in each cluster. To add the points correctly, all the points are first converted to `CartesianPoint`s. Since each point has been mapped to a key-value pair where the key is the ID of the nearest centroid, `reduceByKey` using `addPoints` adds all the points for each cluster. Next, `dividePoints` is called to divide those sums by the number of points in that cluster and convert the result back to a `LatLonPoint`.

Before updating the centroids, the distances between the old and new centroids are calculated. The sum of those distances is used as the value for `iterationDist`, which is the variable that is checked to decide whether the iterations should finish.

The implementation includes functions to calculate the distance between two points using the two distance measures: Euclidean and Great Circle. Calculating the Euclidean distance between points was done by first converting the latitude-longitude points to Cartesian coordinates, then calculating the distance, and finally converting the result back to a latitude-longitude point. The Great Circle distance formula works directly on latitude and longitude and does not involve conversion.

There are also multiple auxiliary functions in the implementation: `closestPoint` returns the index of the closest centroid given a point and the dict of centroids; `addPoints` takes in two `CartesianPoint`s and returns the sum as a `CartesianPoint`. As described above, the points are first converted to `CartesianPoint`s before `addPoints` is called, and after the final sums have been calculated, they are converted back to `LatLonPoint`s.

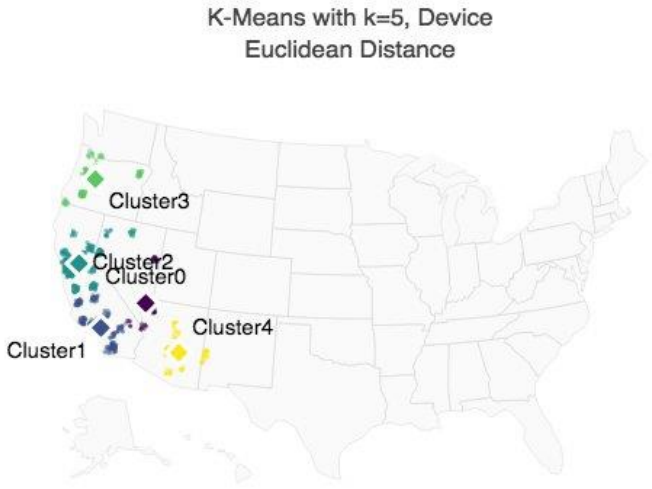
In addition to calculating the centroids and which points belong to which cluster/centroid, the implementation also calculates the mean distance between each point and its nearest centroid. This value is used as a metric for the quality of the model in order to determine appropriate values of *k* for the DBpedia and real-world datasets.

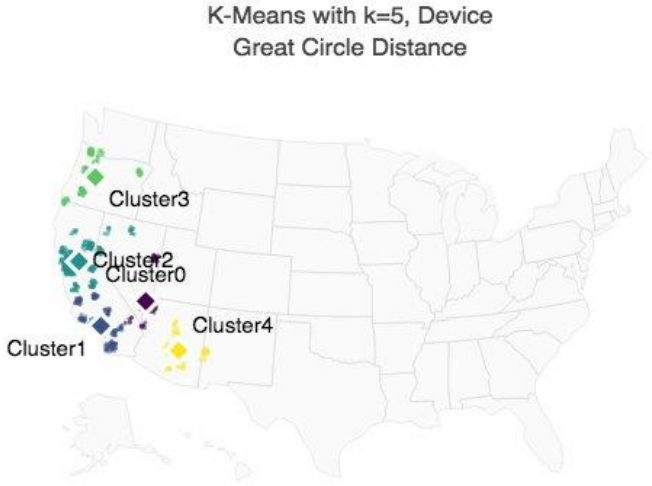
Results

Device Location Data

When generating a sample of the data points to determine the initial centroids, we used a fixed seed. This meant that the initial centroids were always the same for the same dataset and the same value of *k*. This allowed us to more directly compare the Euclidean and Great Circle distance measures without the confounding variable of having different initial centroids. Through this method, we found that the final centroids were consistently the same across the two distance measures. The difference between the two distance measures

was not dramatic enough to result in points being assigned to different final clusters, thus the final centroids were the same whether we were using Euclidean or Great Circle.

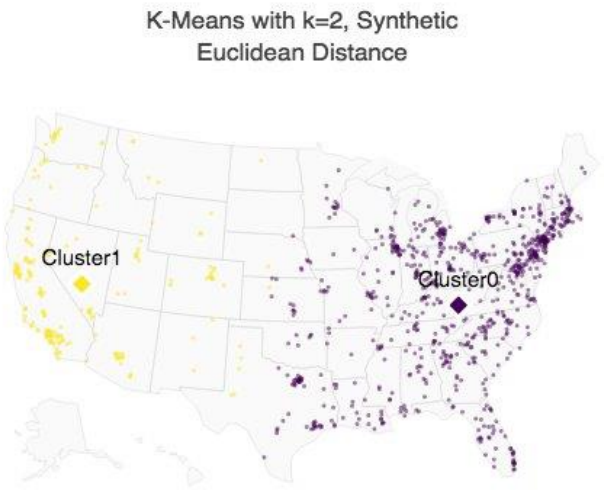


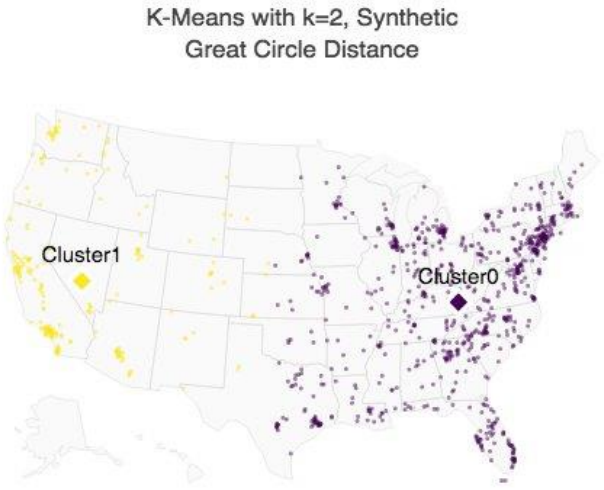


For the device location data, we found that our algorithm effectively grouped the locations into regions, with centroids in Oregon, Northern California, Southern California, Nevada, and Arizona. This corresponds well to the distribution of the synthetic data across the West Coast.

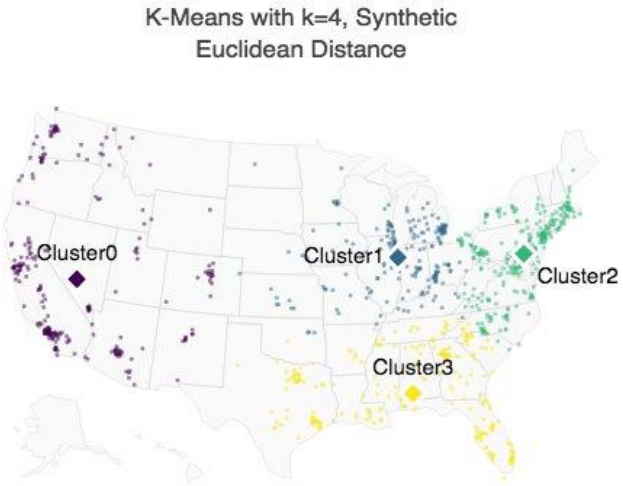
Synthetic Location Data

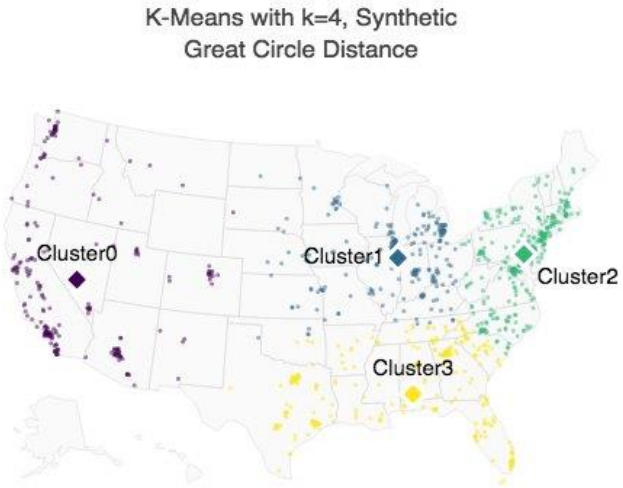
With *k*=2, the synthetic location data was grouped roughly into east and west:





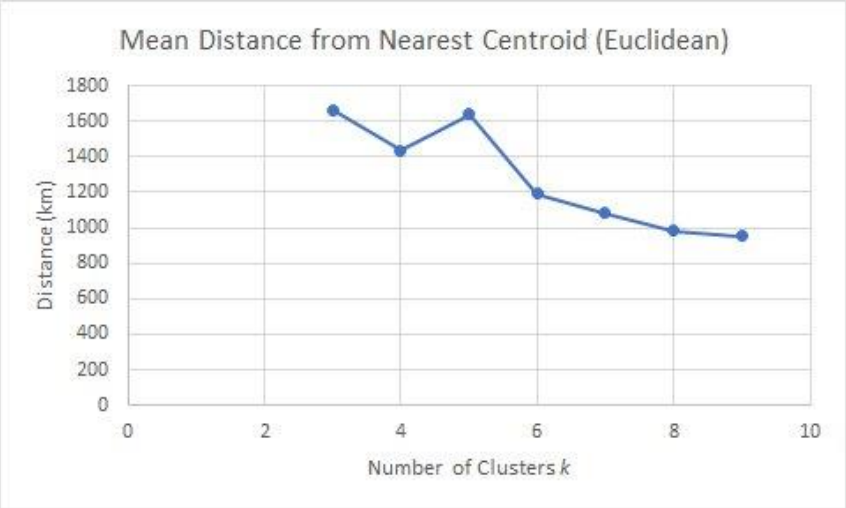
With *k*=4, the West Coast cluster remained, but the significantly larger number of locations on the eastern half of the map was able to be split into additional regions, roughly corresponding to the Midwest, Northeast, and South/Southeast.

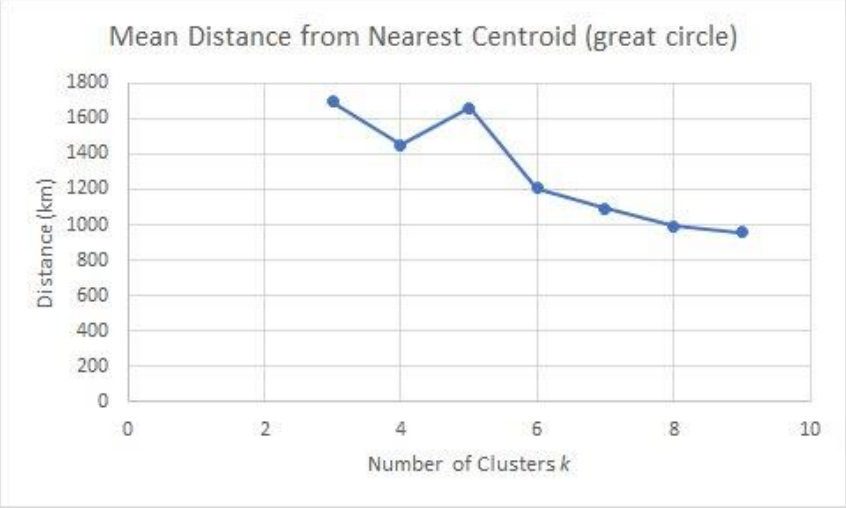




DBpedia Data

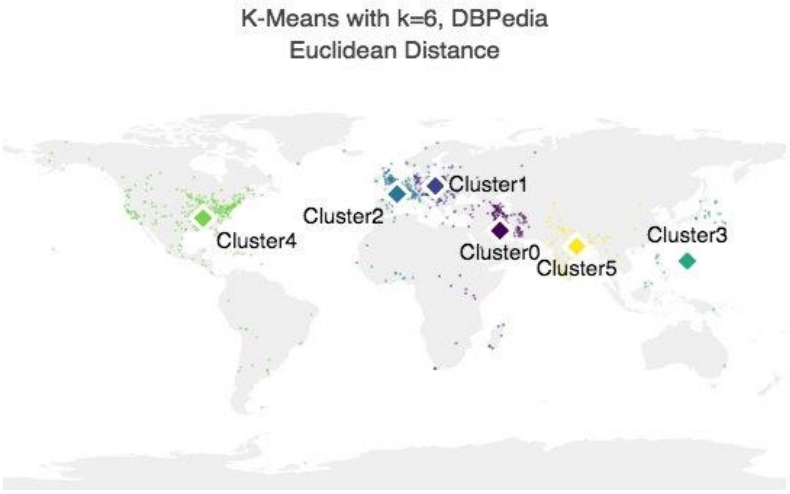
In order to determine a proper value of *k* for the DBpedia data, we tested with various values of *k*. We used the mean distance between points and their nearest centroids as a measure for the quality of the model. The results of the various experiments are graphed below:

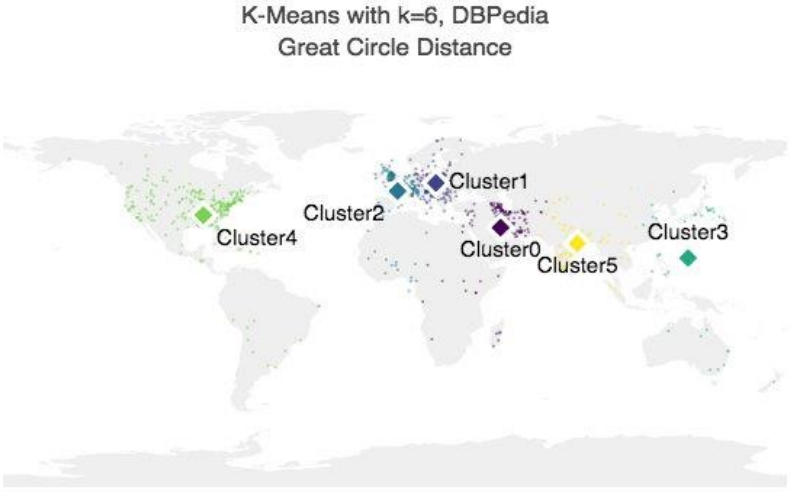




As seen in the graphs, the sharpest improvement in quality is from *k*=5 to *k*=6. After *k*=6, there is not much improvement with increasing values of *k*. In addition, visual inspection of the plotted clusters shows that *k*=6 is a sensible value. This is true for both Euclidean and Great Circle measures, which had very similar results. Using *k*=6 also makes sense because

it corresponds to the number of continents minus Antarctica, which has significantly fewer DBpedia locations than the other continents.



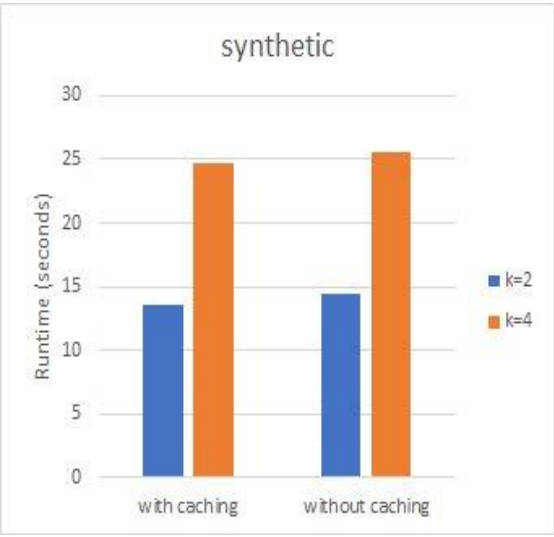


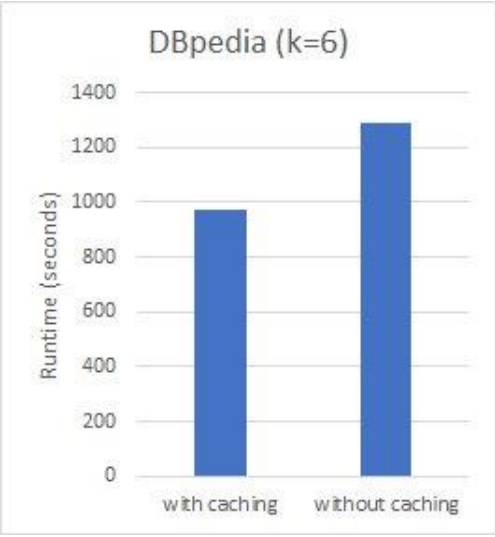
Runtime Analysis

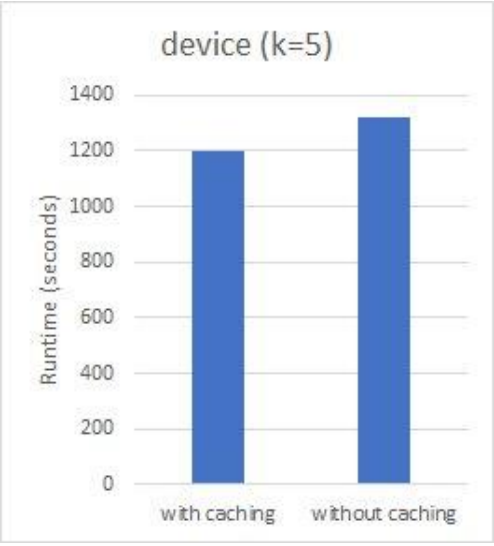
Dataset	K	Persist/Cache	Measure	Runtime (Sec)
DBpedia	6	Y	Euclidean	973.58963
DBpedia	6	N	Euclidean	1292.72722
synthetic	2	Y	Euclidean	13.55297
synthetic	2	N	Euclidean	14.45616
synthetic	4	Y	Euclidean	24.6351
synthetic	4	N	Euclidean	25.5668
device	5	Y	Euclidean	1198.696
device	5	N	Euclidean	1321.82274

<Runtime Analysis for DBPedia, Syntheic and Device Status>

After reading and parsing the input data into an RDD, the RDD is cached since it is reused on each iteration. Without caching, the runtime is consistently longer for each dataset. For smaller datasets, the difference is not large - as seen with the results for the synthetic data, the difference was only a second or two. However, for the larger device data and DBpedia datasets, caching made a very significant difference in the runtime. For these datasets, not caching increased the runtime by 3-4 minutes. Thus, for larger datasets, caching RDDs that are frequently re-used can be very beneficial for performance.







Cloud Execution on Real World Data

In order to execute our implementation on Amazon EMR, we were to:

1. Upload implemented code and pre-processed data on a S3 bucket, and determine an appropriate output location
2. Specify arguments for deployment file(s), input files, output location, distance measure, and *k*

Argument specifications are as below:

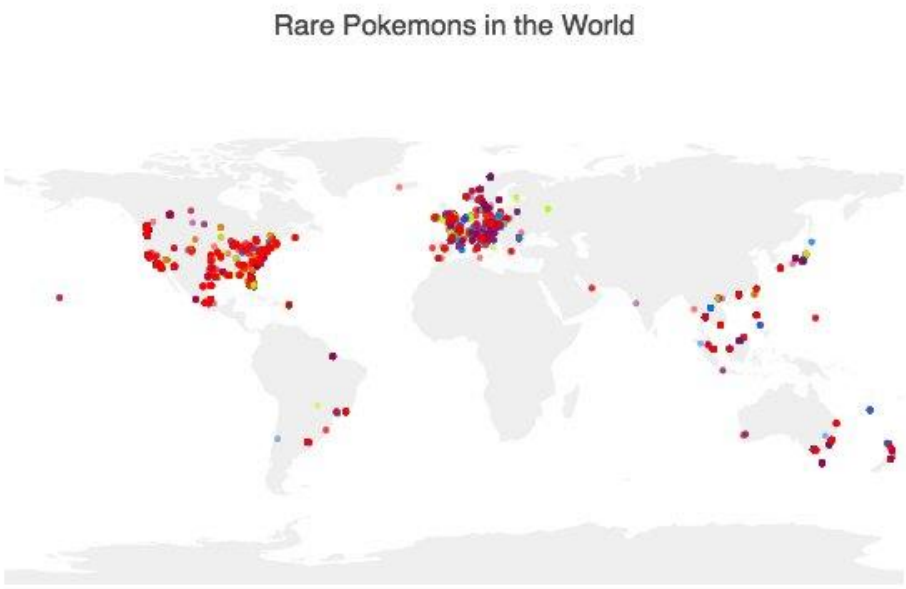
```
spark-submit --deploy-mode cluster \
# --py-files <custom_modules> \ # if necessary
<code_path> \ # Location of the Spark script to be run
<input_path> \
<output_path> \ # Should not be an existing directory
<distance_measure> \ # Euclidean or Great Circle
<k_clusters>         # An integer larger than 0
```

Below is an example run (using the Rare Pokemon data, Euclidean distance, and *k*=3):

```
# Jar location
command-runner.jar

# Arguments
spark-submit --deploy-mode cluster \
--py-files s3://teamipynb/input/point_utils.py \
s3://teamipynb/input/kmeans.py \
s3://teamipynb/input/pokemon_rare \
s3://teamipynb/rare_k3 \
euclidean \
3
```

Pokemon Go sightings around the world (Midsize data)
Data source: https://www.kaggle.com/semioniy/predictemall



<Map of rare Pokemon by their sighting locations and times*>

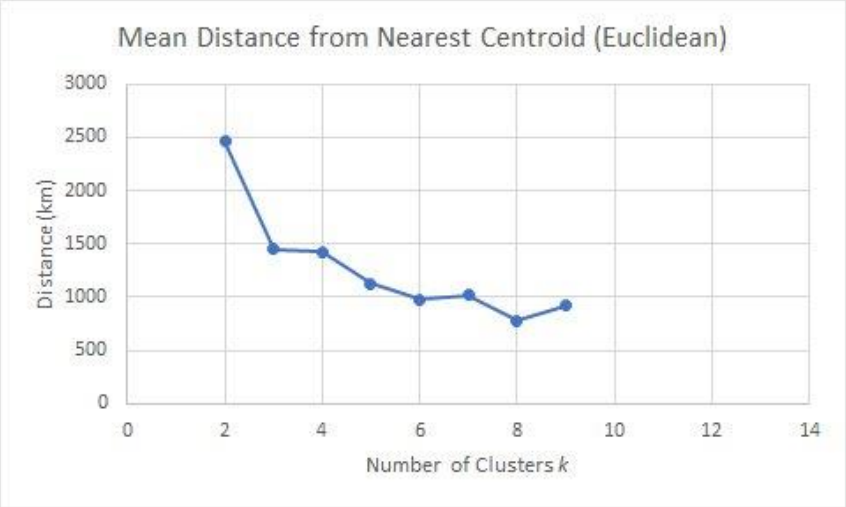
Pokemon Go is augmented reality game for mobiles developed by Niantic, Inc. and released in July 2016. As players move around in the real world, Pokemon appears on the player's device as though in the same real-world location as the player. Players can then catch these Pokemon in-game using their device. Players often seek to catch strong and rare Pokemon.

The data set consists of roughly 293,000 Pokemon sightings. The features in the original data set include Pokemon ID, coordinates of sighting, time, weather, population density, and distance to gyms/Pokestops. For the purposes of this project, the data set is filtered to have coordinates, time of day observed and Pokemon IDs only. The pre-processed data contains 12 part reducer files and was about 6.6 MB in size.

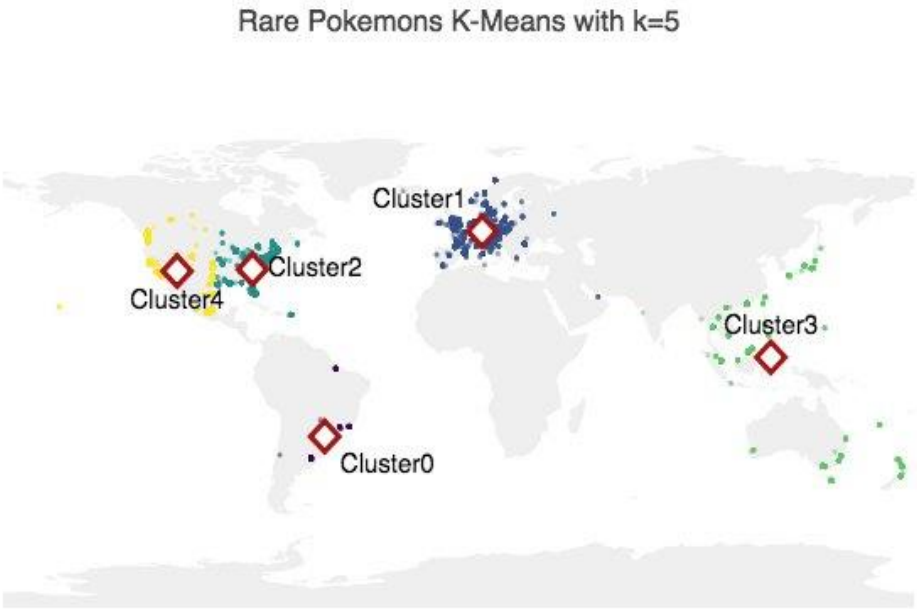
Since Pokemon players aim to catch rare Pokemon (Pokemon that appear relatively infrequently), we further filtered the data to consists only of rare Pokemon. The data was processed according to the Pokemon id in the "Very Rare" and "Super Rare" section in http://www.pokego.org/rare-pokemon-list/. The processed data was 400 KB in size with roughly 8000 data points.

To determine an appropriate value of *k* for this dataset, we followed the same procedure as with the DBpedia data.

**morning: purple, afternoon: blue, evening: yellow, night: red*



We chose *k*=5 as we determined that the final clusters were able to divide the data points into general regions as shown in the map below.



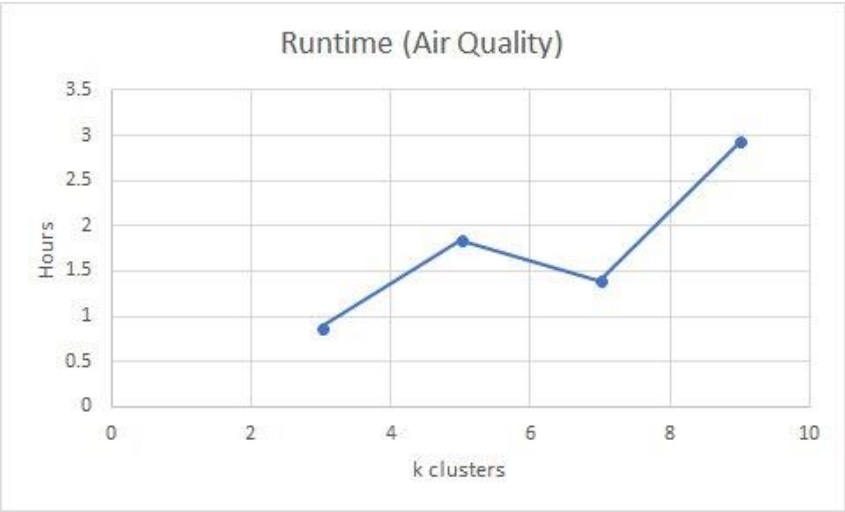
Hazardous Air Pollutant in US (Big Data)
Data source: https://www.kaggle.com/epa/hazardous-air-pollutants

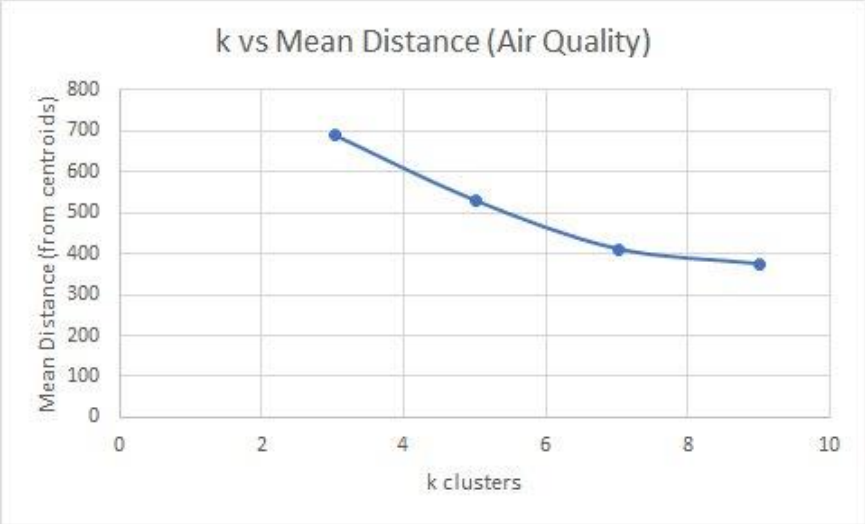
Since the Pokemon Go dataset was relatively small, we also tested the cluster with a much larger dataset to analyze the performance of the cluster.

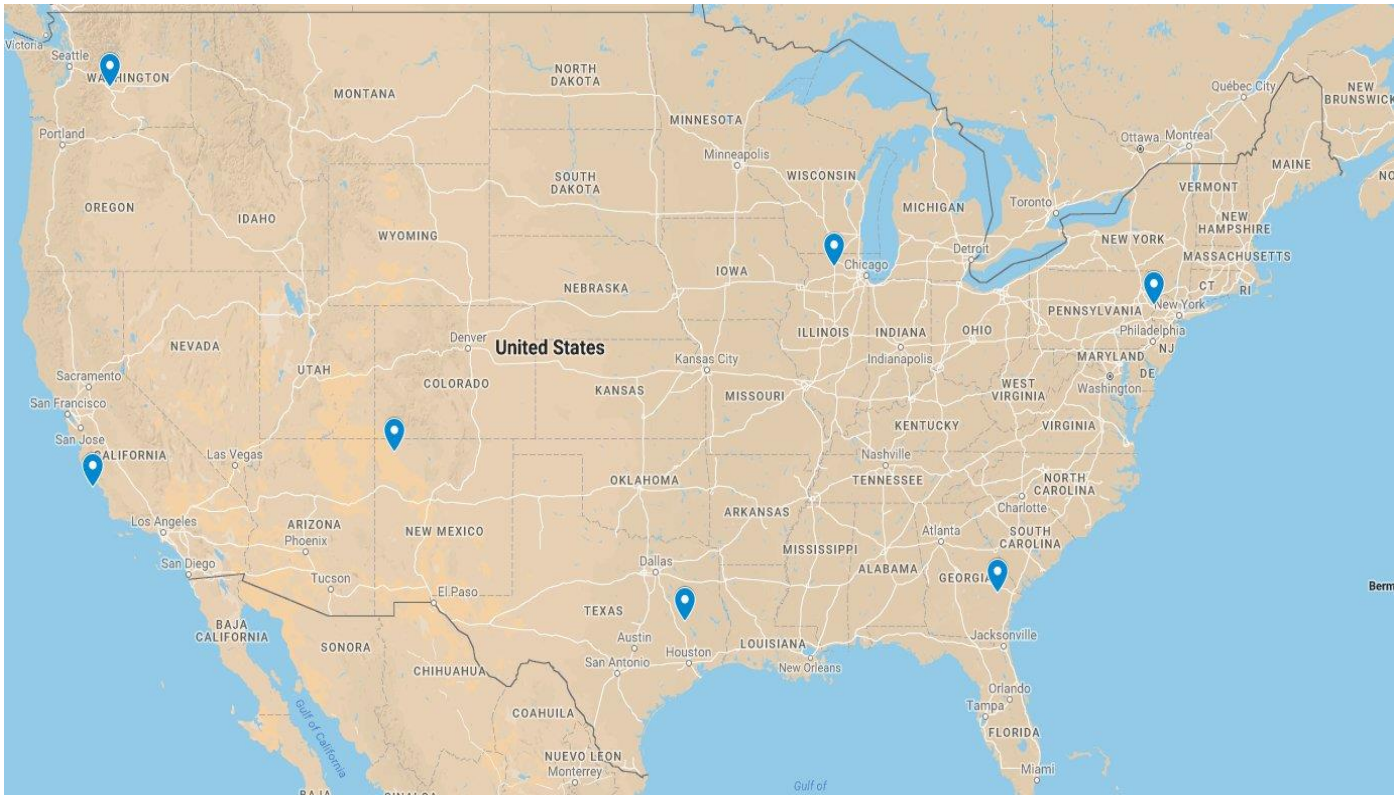
This data consists of 8 million measurements of the levels of 187 types of toxic air pollutants in the U.S. from 1990 to 2017. The air pollutants are suspected to cause cancer or other serious health effects. According to Environmental Protection Agency (EPA), people exposed to toxic air at certain concentrations may have increased chance of immune system damage, neurological, reproductive, developmental, respiratory and other health issues. By clustering this dataset, we can see where the EPA has the most monitoring stations, which could potentially be applied to identify places that already have a lot of monitors and places that may need more.

The original data set was about 628 MB. After pre-processing, the data used for the algorithm was about 300 MB that was divided into 74 part reducer files. The features in the processed data include latitude, longitude, and pollutant name.

The run time for *k*=3 was about 53 minutes, and larger k values had longer run time as shown in the graph below. After experimenting with various k values, *k*=7 seemed to be the drop point. The map below shows the final seven centroids. Because the seven points are around major cities in the US such as Austin, Chicago, and New York, it seems plausible that the EPA has the largest amount of monitoring stations in highly populated areas. It may be useful for the EPA to add more monitors to the northern United States, as for many of the northern states, the nearest centroid is quite far. However, this may simply be a result of that region of the U.S. having a lot of open land and not a very dense population, making a heavy amount of air pollution monitoring less useful.







<Map of final seven centroids>

Conclusion

The k-means algorithm has many applications as demonstrated by testing the algorithm on four data sets with different characteristics. This project clusters based on geo-locations, and can be applied to many different kinds of datasets with latitude and longitude as features.

In order to find meaningful information for each application, it was necessary to experiment with different values for the number of clusters, *k.* After several attempts, there indeed was a "drop" point where the mean of distances between clusters and data point significantly decreased. When the *k* was found for each application, it seemed plausible for the corresponding application. *k* was a metric that not only divided the data into clusters but also gave new insight for the data set.

Visualization was also a significant portion of the project. Though the computer is able to process large number of data, in the end, it was necessary for humans to interpret the meaning of the data and final clusters yielded by the algorithm.

This project also experiments with different methods to alter run time: RDD persistence and cloud execution. As *k* increased, the runtime of the algorithm increased. However, RDD persistence somewhat decreased the runtime of the algorithm on the same arguments, and cloud execution drastically improved the runtime. It was interesting to observe that running the algorithm on big data with the Amazon Elastic MapReduce service was faster by a few degrees of magnitude. This allowed us to observe some of the power of large-scale cluster computing.