Geo-location Clustering Project

Distributed and Scalable Data Engineering

By Sam Watson

5/10/2020

I. Motivation

Data is growing exponentially, and there is not any sign of it slowing down. Computing cores are our tool for processing and delivering data, but one computer can only do so much. It is simply not feasible for one computing core to do an aggregation, filter, or sort on hundreds of terabytes of data. The results of this will take hours, maybe even days!

This is where parallel processing comes into play. Have you ever tried to make a one, giant cookie from Betty Crocker's cookie dough pouches? Speaking from experience, the amount of time it takes to bake one giant cookie takes almost two times longer than the recommended amount of time. If you bake them as 12, separate cookies, however, it then takes half the amount of time to bake!

The same line of thought can be applied to big data. Using multiple computers to work in parallel will run much, much quicker than one computer working alone. Spark is a software that leverages computer memory along-side the map-reduce framework. It provides an interface that allows users to run jobs in parallel on a cluster of computers. It makes running aggregations on large datasets easy and fast.

Geo-location data is a great example of big data that can benefit from parallel processing. The goal of this project is to implement a K-means clustering algorithm on geo-location data using Spark.

People stream videos, do google searches, scroll through social media, etc. The implications of being able to run complex machine learning algorithms on large datasets (like a geo-cluster dataset) are huge. For example, this same application is used every day in the real world all around us. Society consumes data and wants the consumption to be quick.

II. Documentation and Approach

The purpose of this assignment is to implement a K-means clustering algorithm on geo-location data using python and Spark. Before I could do anything, I first needed to find a place to store my data. Then, I had to create an environment that would allow me to run Spark. From there, I had to clean and examine the data. Once the data was cleaned, I then could create a K-means algorithm in Spark, apply it to the three datasets, and plot. Finally, just to further prove why parallel processing is the bomb, I then had a to find a way to record the time it took to run the processes with and without parallelization.

III. System Configuration

As mentioned above, I needed to be able to store the data and process it with Spark. I used Amazon's web services (AWS) to accomplish these tasks. I used the S3 service to store the data, and the EMR service to process it. On my EMR cluster, I wrote python code. I used pyspark, which is an import that allows python to interact with the Spark API.

S3 is a way to store data, such as photos, videos, documents, etc. I first uploaded the three datasets into their own folders.

Then, I created a cluster of computers using EMR. EMR is a service that creates a cluster of computers that can run in parallel. I chose Spark as my application, chose m4.xLarge as my instance type, and chose 3 as my number of instances. Once the cluster was created, I edited my security groups to allow "All Traffic" into the master node.

Next, I ssh'd into the master node of the cluster through my Linux terminal. I ran the following commands:

```
-   sudo pip install pyyaml ipython jupyter ipyparallel pandas boto -U
-   export PYSPARK_DRIVER_PYTHON=/usr/local/bin/jupyter
-   export PYSPARK_DRIVER_PYTHON_OPTS="notebook --no-browser --ip=0.0.0.0 --
    port=8888"
-   source ~/.bashrc
-   pyspark
```

The first command installs the appropriate software on my master node. The next two configure pyspark to open and run on Juypter notebook. The fourth refreshes the bashrc, and the last one creates a Jupyter notebook instance with pysark. The terminal then displays a token id. That token id can be pasted at the end of the following URL to access the Jupyter notebook.

http://ec2-XXX-XXX-XXX-XXX.compute-1.amazonaws.com:8888/tree?token=

All of the code to load the data, implement the K-means algorithm, and graph the results was done on this EMR cluster.

IV. Big Data Application/Dataset

a. Description

The goal is to create a K-means cluster algorithm, run it on geo-cluster data, and analyze it all with Spark. There were three geo-cluster datasets that were given: device data from fictional mobile company (94K rows), Synthetic data geo-location data (10K rows), and latitude/longitude coordinates from DBpedia users (450K rows). I Clustered the three datasets, visualized them, and analyzed their run-time.

b. Implementation

Cleaning

First, I loaded the data from my s3 bucket into Jupyter notebook as a Resilient Distributed Dataset (RDD). Now it is time to clean the datasets.

The first dataset was not delimited uniformly; some records were parsed with a ",", some with a "|", and others with "/". To parse the data uniformly, I replaced all "|" and "/" with a ",". I did this by mapping a lambda function to each line in the RDD. Now that the data is parsed uniformly, I was then able to split the data by the "," value. Next, I only kept the latitude/longitude (13th and 14th element), the date (first element), the model (second field), and the device id (third field) fields. From there, I filtered out locations where the latitude and longitude (now the first and second field) equal 0. Finally, I separated out the device manufacturer and model name from the model field (now the 4th list element). This is done by splitting the model field by spaces. I then take the first two elements of that split string and tack them on at the end of my row as two new columns.

The result of this cleaning of the first dataset is an RDD containing the fields: latitude, longitude, timestamp, the full model name, the device id, the model manufacturer, and the model name. (*see Figure 1*) This cleaned dataset was then saved to S3 as a csv file for later use.

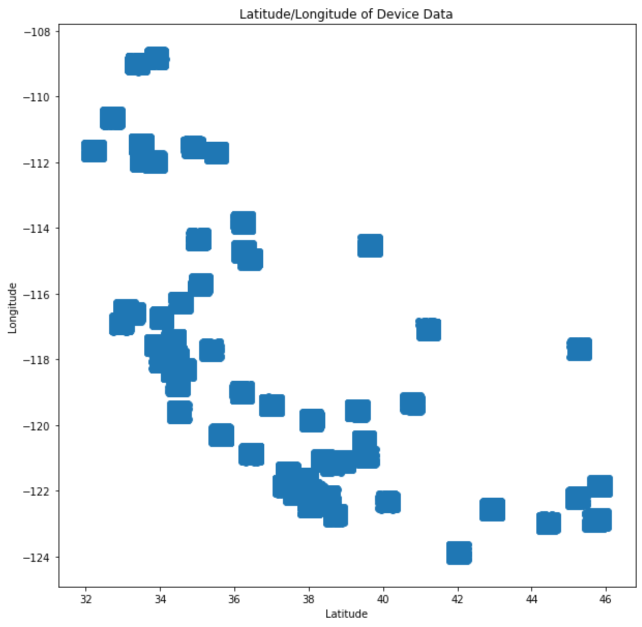


Figure 1

Cleaning the Synthetic and DBPedia data was much easier. For both datasets, I loaded the data from S3. For the Synthetic data, I split the times by “/t” (tab). For the DBPedia data, I parsed the data by “ “. For both, I removed blank values, and then loaded the data to S3 as a csv files. The struggle with the DBPedia dataset was that it was much larger than the others. I only plotted the geo-coordinates where the latitude was between 25 and 47, and the longitude was between -124 and -68. (*see Figure 2*).

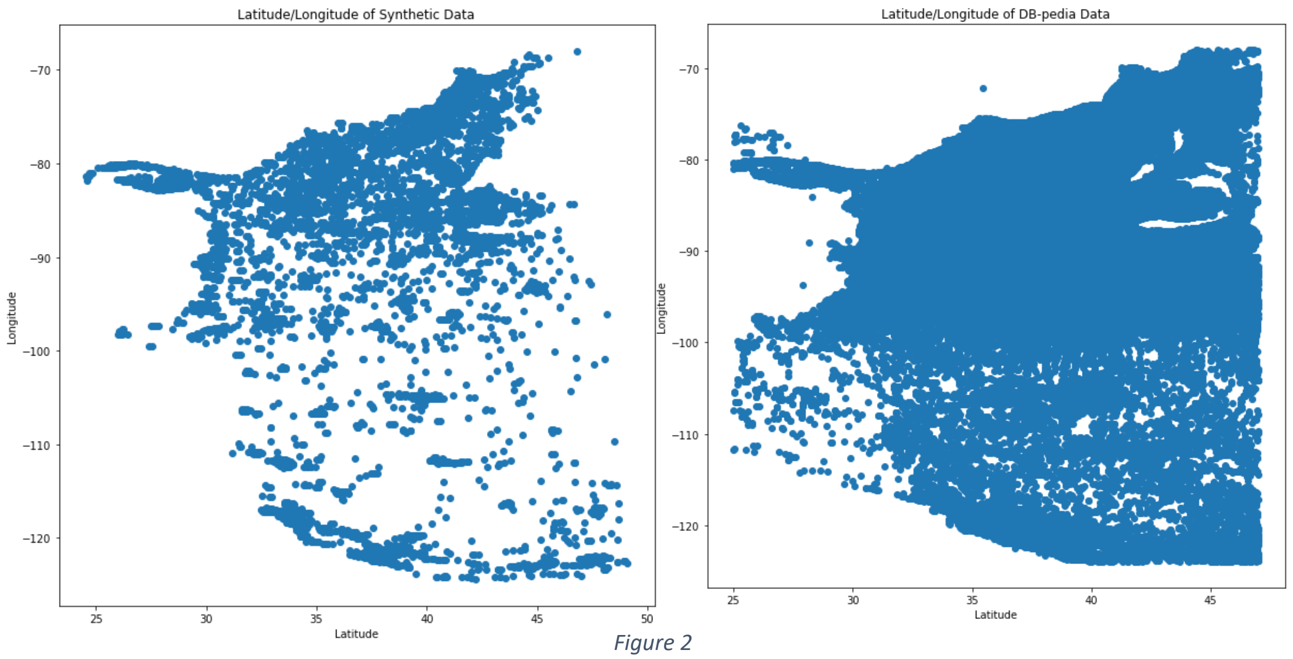
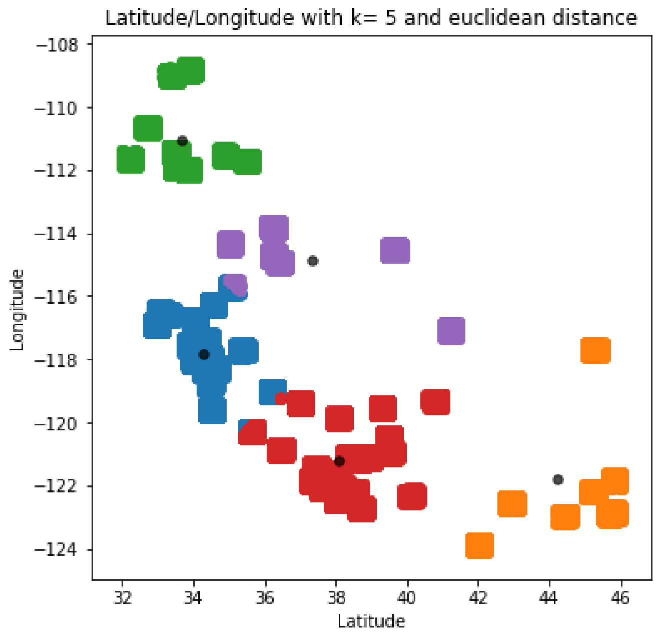


Figure 2

K-Means

Now that the datasets are cleaned, we can apply a K-means clustering algorithm, and plot them. K-means is an unsupervised machine learning model that clusters data around a specified number of centers. I created a python function that uses pyspark’s machine learning library. It takes a pyspark dataframe, the number of desired clusters, and the distance measuring (Euclidean or cosine/great circle) as parameters. Then I created a python function that graphs the data’s clusters with their cluster centers. I used the libraries default convergence distance. *Figure 3* is a graph of the first dataset using Euclidean distance and 5 clusters.



I also plotted the Synthetic dataset with both cosine and Euclidean distances with values 2 and 4 for the number of clusters. See *Figure 4*.

Figure 3

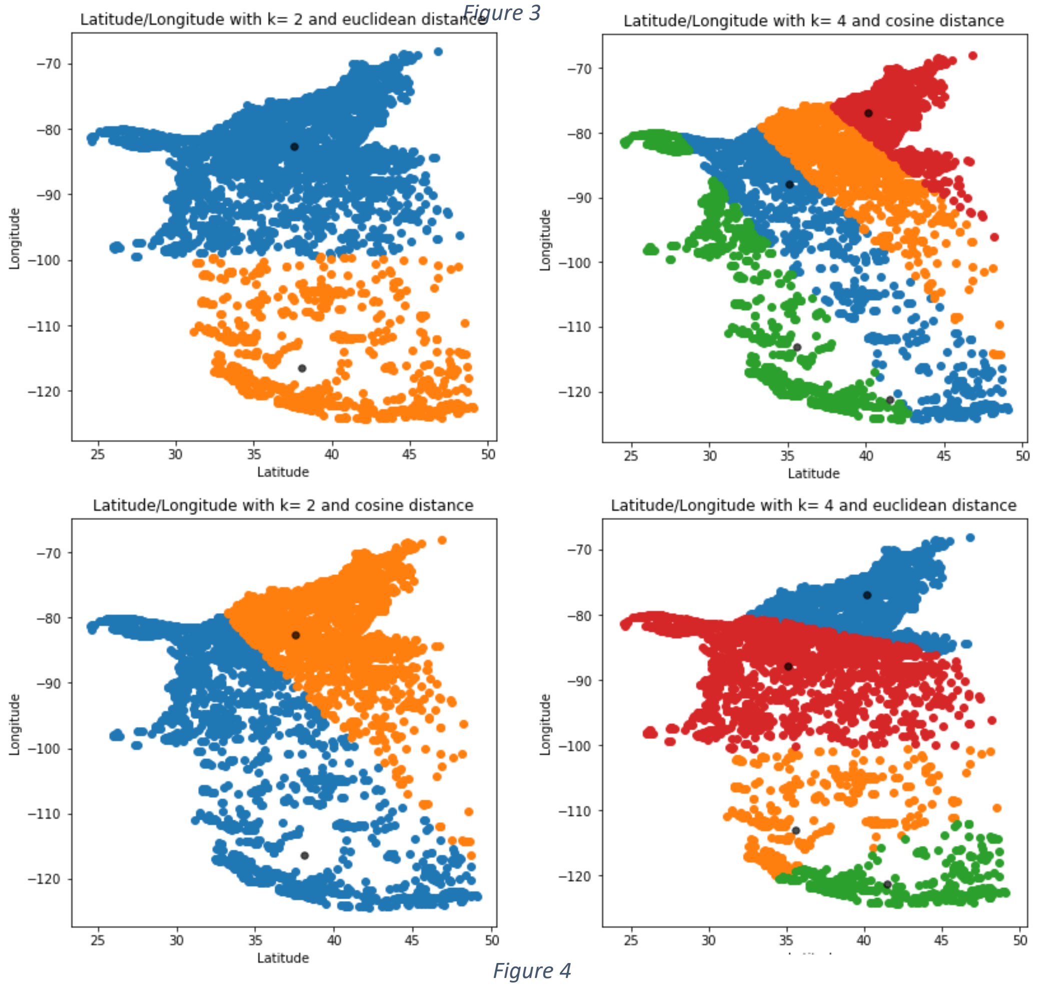


Figure 4

Lastly, I plotted *all of* the data points for the DBpedia data using both cosine and Euclidean distance metrics. I chose 2 and 6 as the number of clusters. *Figure 5* is a graph of the third dataset using 6 clusters and Euclidean distance.

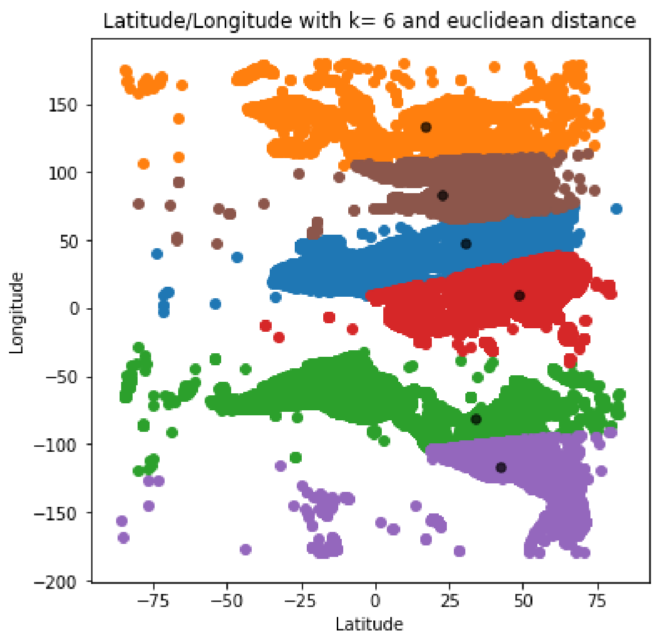


Figure 5

Runtime Analysis

Thank you, you have been a great audience. For my last trick, I will compare the runtimes of my k-means implementation for all three datasets. Then, I will compare this overall runtime to that done with three threads going on. Fear the might of parallel processing! Muhuhahaha.

Figure 6 shows a bar chart of the time it took to run the K-means cluster with 4 clusters and Euclidean distance metric. These results were run three times with python's %timeit function, and the graph is displaying the best of the three loops.

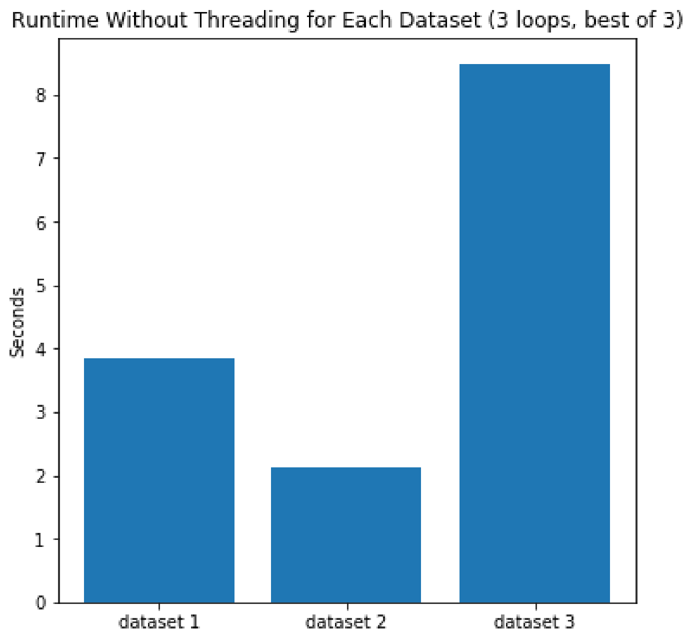


Figure 6

Figure 7 shows the time results of running all three clustering algorithms on a single thread (without parallelization) and on multiple threads (with parallelization).

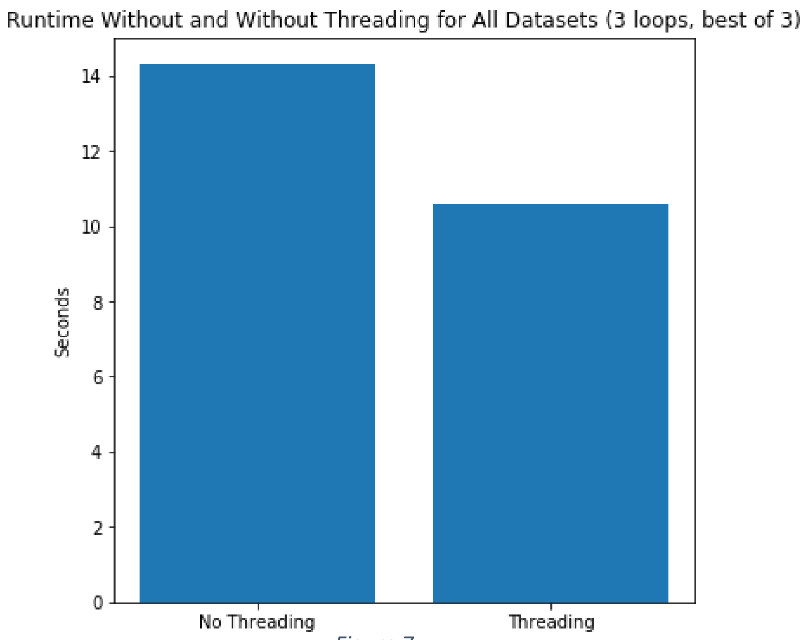


Figure 7

c. Results/discussion

For the Synthetic dataset, if you choose Euclidean distance, then the decision boundary is basically a horizontal line. for cosine distance, the decision line is more at a 45 degree angle.

The value for k that makes the most sense for the third dataset is 6. This is because the dataset covers the entire world, and there are six populated continents in the world. The third dataset took the longest to run the K-means algorithm (9 seconds). This makes sense, as it was by far the largest one. 9 seconds is still quick, though! Thank you, pyspark!

Probably the coolest part of this entire assignment was comparing the runtime of doing K-means on the three datasets on a single thread vs on multiple. Having the job split up on multiple threads reduced the overall runtime by 25% (14.3 seconds to 10.6). The implications of this are huge! Pooling tasks and threading them means we can get our data much, much faster.

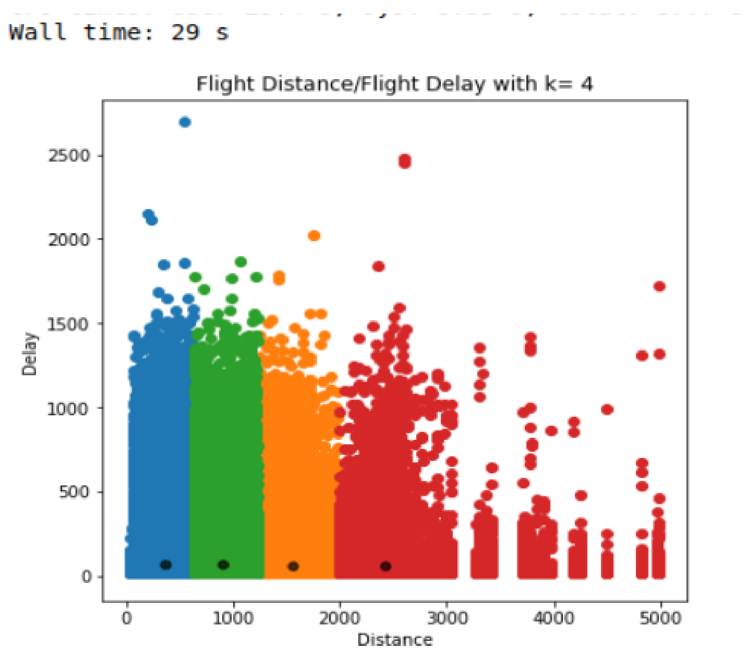
V. Bonus

As a bonus, I found an even bigger dataset to do my clustering algorithm on. The data is retrieved from the United States Department of Transportation.

The website holds all airline data reported to the U.S Department of Transportation (DOT) from the top 12 U.S air carriers. Each of these air carriers has at least 1% of the total domestic scheduled-service passenger revenue. The dataset will include attributes like the date of the flight, the tail number for the plane, the departure and arrival time and location, the total distance traveled, the amount of time delayed, and more.

The dataset is large; There are about 450,000 flights each month. I am going to try to do a years-worth of data (all of 2018), which would mean roughly 5,400,000 samples The link to dataset is: https://transtats.bts.gov/ONTIME/.

I trimmed the dataset down to only look at 2018 flights that experienced a delay, bringing the total number of records here is about 1.4 million (still big!). I Then clustered the data around the total distance the plane traveled and total delay time. It took about 30 seconds to run the data on my EMR cluster (*see Figure 8*)



```
Wall time: 29 s
```

Figure 8

VI. Conclusion

To recapitulate, big data is constantly getting bigger, and society as consumers want the data readily accessible. If our objective is to process the data quickly, then processing terabytes of data on a single computer is not the answer. Clustering computers together and running Spark is a cheap and easy solution to this problem.

The project was to run a K-means algorithm large geo-location data. I used AWS services to create a cluster on EMR and store the data on S3. I used python's pyspark to access Spark's API. The three geo-location datasets were cleaned, stored, clustered using K-means, and plotted. I then analyzed the runtimes of the algorithm.

For the second dataset, when I choose Euclidean distance, the decision boundary forms a horizontal line. For cosine distance, the decision line is more at a 45 degree angle. The third dataset took the longest to run the K-means algorithm (9 seconds). Using 6 clusters makes the most sense on this dataset, as it covers the entire world, and there are six continents.

As a bonus, I ran the same algorithm on delayed flights data in 2018, containing 1.4 million records. The job took 29 seconds to run and plot.

Running a k-means algorithm on all three geo-location datasets takes a while; it took me an average of 14.3 seconds. However, when you run the same job on 3 concurrent threads, the time drops down to 10.6 seconds. This proves just how powerful parallel processing can be.