GEO-LOCATION CLUSTERING USING THE k-MEANS ALGORITHM

Introduction

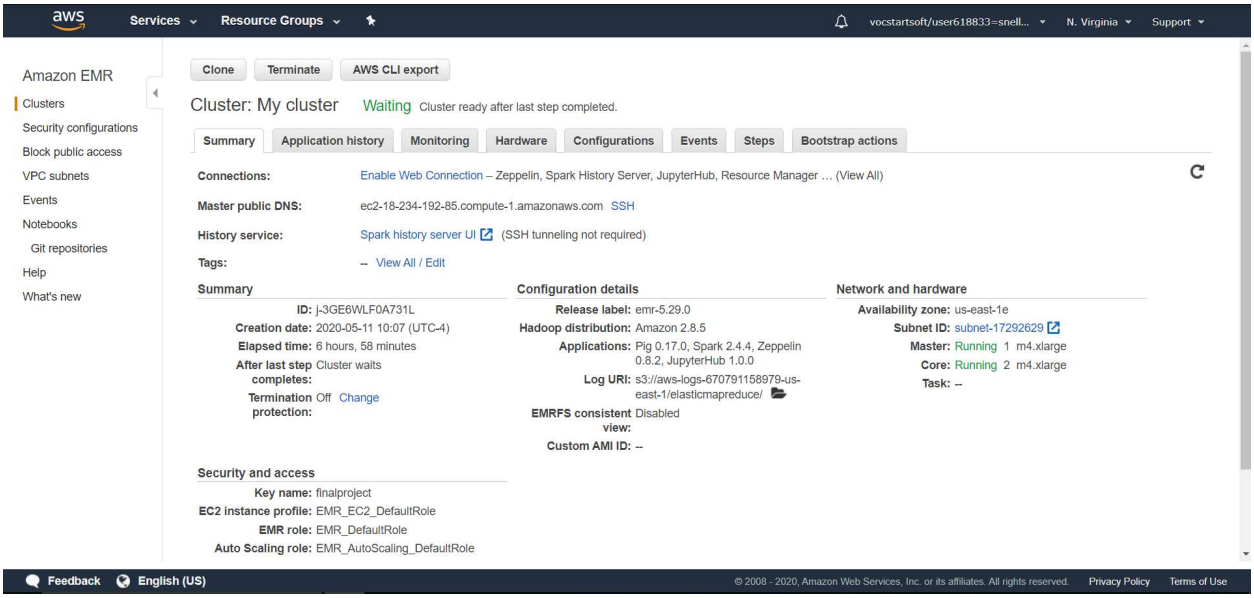
K-means clustering algorithm , a type of unsupervised learning which is used when you have non-categorical data (i.e., data without defined groups). The centroids of the K clusters, which can be used to label new data. Labels for each data point is assigned to a single cluster.

Here we will be using SPARK to implement an iterative algorithm that solves the clustering problem.

k-means is a distance-based method that iteratively updates the location of k cluster centroids until convergence. The main user-defined ingredients of this algorithm are the distance function (d_type) and the number of clusters (k).

Connection to AWS EMR cluster

In our AWS Educate account, its easy to create an Emr cluster.



This is what a successful cluster creation looks like. For steps to create one are mentioned in this link: https://docs.aws.amazon.com/emr/latest/ManagementGuide/emr-gs-launch-sample-cluster.html

Next, we install all necessary packages in our cluster using : sudo pip install jupyter pandas geopandas matplotlib Descartes

Next, we will integrate pyspark with Jupyter on port '8888' using this in our vi.bashrc file:

- export PYSPARK_DRIVER_PYTHON=/usr/local/bin/jupyter
- export PYSPARK_DRIVER_PYTHON_OPTS="notebook --no-browser --ip=0.0.0.0 --port=8888"

```
[hadoop@ip-172-31-54-58 ~]$ pyspark
[I 21:16:24.806 NotebookApp] Serving notebooks from local directory: /home/hadoop
[I 21:16:24.806 NotebookApp] The Jupyter Notebook is running at:
[I 21:16:24.806 NotebookApp] http://(ip-172-31-54-58 or 127.0.0.1):8888/?token=e65f65e2f8fbb30525401a0ab84b0132e2addd716f07e03b
[I 21:16:24.806 NotebookApp] Use Control-C to stop this server and shut down all kernels (twice to skip confirmation).
[C 21:16:24.812 NotebookApp]

    To access the notebook, open this file in a browser:
        file:///home/hadoop/.local/share/jupyter/runtime/nbserver-22977-open.html
    Or copy and paste one of these URLs:
        http://(ip-172-31-54-58 or 127.0.0.1):8888/?token=e65f65e2f8fbb30525401a0ab84b0132e2addd716f07e03b
```

Finally, copy the token acquired and use it to open a Jupyter notebook in this format:

http://<masternodeDNS>:8888/?token=<TOKEN>

Pre-Processing

As usual for every Machine Learning model, we will be doing pre-processing steps. We have 3 datasets (Device Location data, Synthetic Location data, DBpedia location data). The following are steps by each dataset:

- Load the dataset using inbuilt SparkContext (sc) as RDD (Resilient Distributed Datasets).
- Since some datasets are separated by comma (',') or by slash (/) or by vertical bar (|), so we need to careful in choosing the delimiter.
- Check to see if the dataframe are parsed correctly and data is correctly inserted.
- Change the column name to lower case, especially columns such as latitude and longitude, as they are more universally used.
- Filter any nan or null (or 0) values from all the columns in the dataframe
- For easier computation, splitting the columns with two values (separated by space)
- Visualize the dataset based on (latitude, longitude) pairs using matplotlib, geopandas and Descartes on World map.
- Store the final dataset to the s3 bucket using spark's 'write' function.

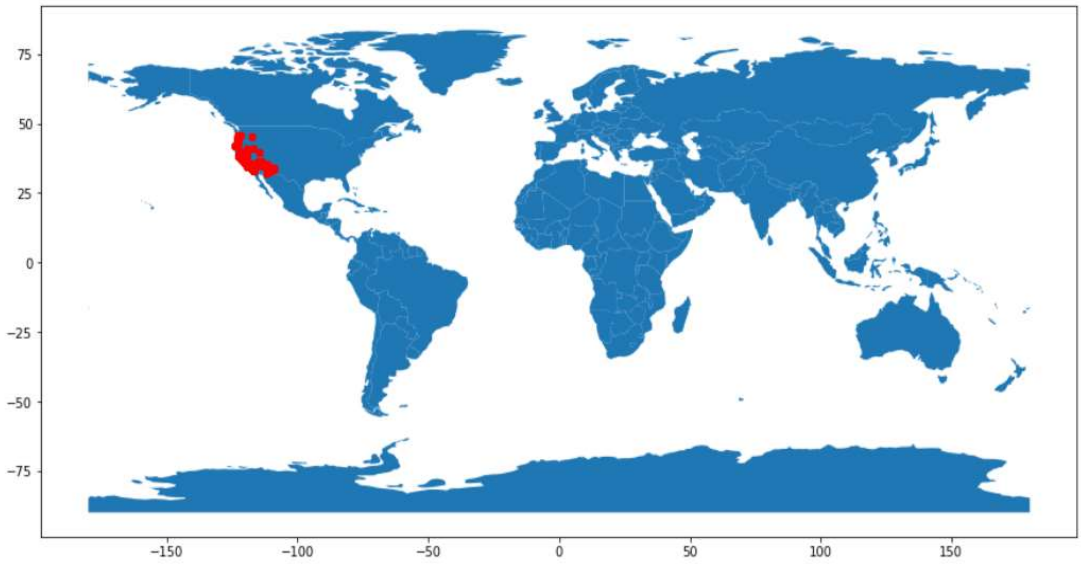
Also, store the following Jupyter codes for 3 datasets (.ipynb) file in GitHub repository under Clustering/step1 as advised.

Visualization of 3 Datasets

The following are:

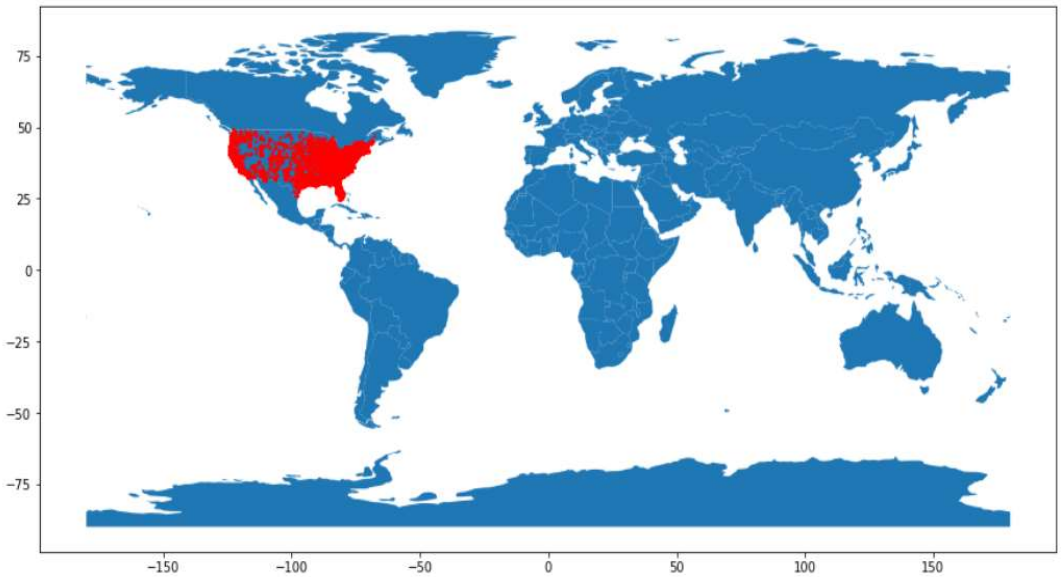
Device Location Data

```
Out[93]: <matplotlib.axes._subplots.AxesSubplot at 0x7f27ea4c4650>
```



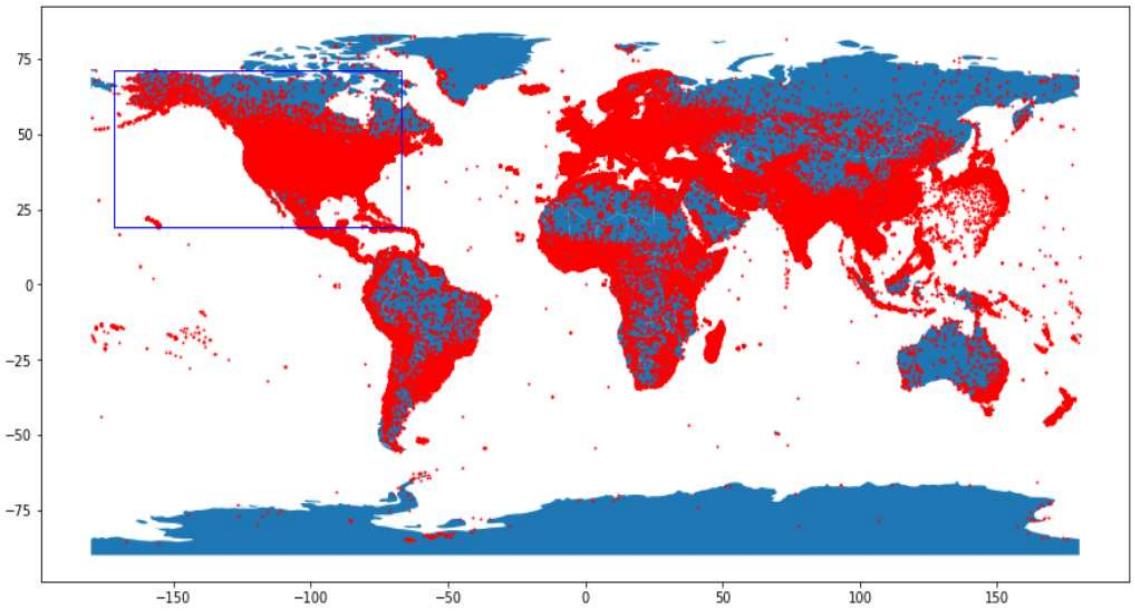
Synthetic Location Data

```
Out[55]: <matplotlib.axes._subplots.AxesSubplot at 0x7f41e708b950>
```

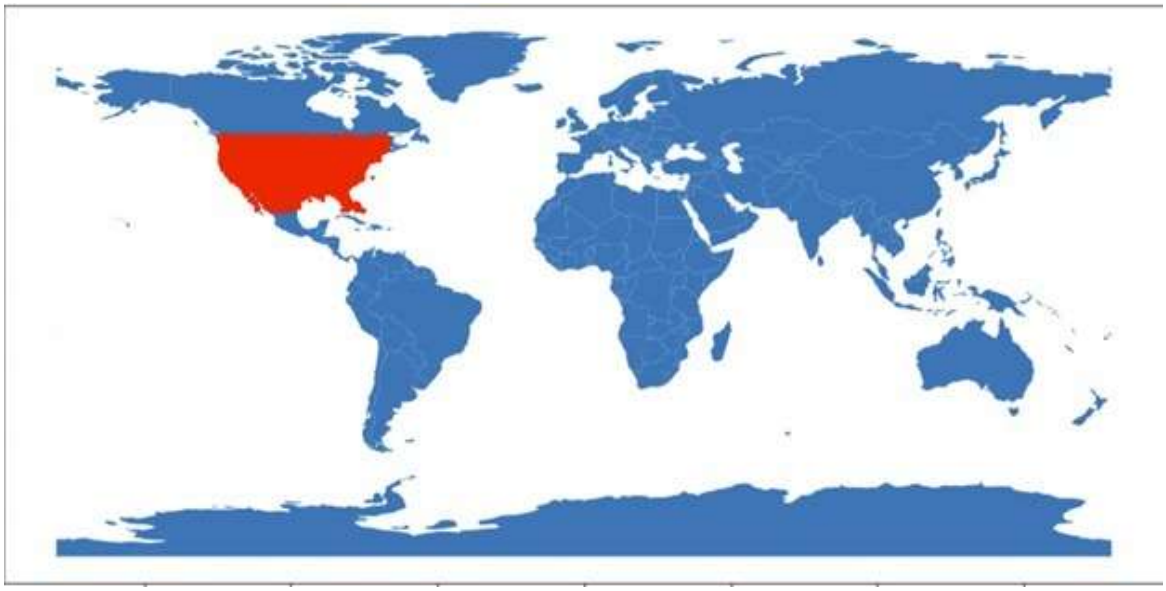


DPpedia Location Data

```
Out[11]: <matplotlib.axes._subplots.AxesSubplot at 0x7fa6792c7090>
```



Since the DBpedia data is scattered worldwide, we have filtered the data within the United States and a bounding box visualizes the filtered boundary.



The above figure is the filtered version of DBpedia data i.e. filtered within U.S country.

The filter is done using the U.S. coordinates: *(-124.848974, 24.396308) - (-66.885444, 49.384358)*

Clustering Approach

User Input:

- K = Number of clusters
- d_type = whether it is Euclidean or Great Circle Distance
- ip = data (device location, synthetic location, DBpedia location) data

1. Initially, we will be randomizing centroids , which are same value 'k' as the number of clusters which is given by the user.
2. Next is choosing between two types of distances:
   - Euclidean: Calculation of the line distance between two points
   - Great Circle: Calculation of spherical distance between two points along Earth radius.
3. Generally, device location only has around 100k along the coast of U.S as observed in the visualizations above. Synthetic Location covers the whole of U.S. whereas, DBpedia has co-ordinates that covers the whole world.
4. After getting our initial centroids, we find the closest distance to one of 'k' clusters with distance measure given above.
5. Getting our result in the form of a matrix with closest cluster centroids as its label.
6. Generating new centroids based on this data and old centroids by calculating the mean between them.
7. Mean is calculated by the data points within its each specific cluster.
8. Repeating the above steps until we satisfy a convergence distance, where the difference in the means calculated are greater than the given condition (in this case 0.5) i.e. convergeDist=0.5.
9. This is because, with the given data points, no algorithm can predict the clusters to its nearest zero.
10. This algorithm will keep on iterating and will give its final centroids when the difference in mean is less than 0.5
11. This algorithm showed promising clusters with faster runtime
12. For difference in runtimes, we have visualized output with the caching and without.

Implementation

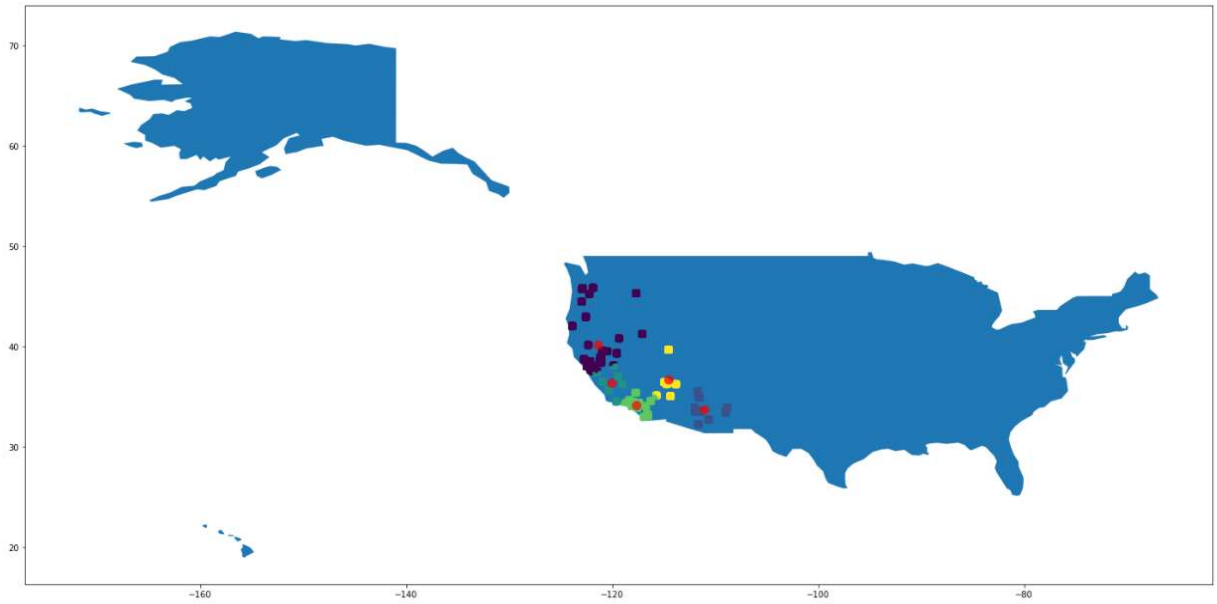
1. This algorithm initially has its *Euclidean* and *Great Circle* distance. Based upon user's choice either function is called.
2. A function called *'closestpoint'* has its use to calculate the closest point using either distance measures.
3. A new centroids function called '*getnew'* accepts old centroids and the result from previous iteration as its parameters. Output will be the new coordinates for centroids by calculating the means using the 'means' function.
4. A *'k-means'* function is used to iterate the algorithm recursively, until a final converge is met and the final centroids are thrown as an output.
5. Next comes the visualization function called *'visual'* , the purpose of this is to visualize the co-ordinates onto the world map whenever it is called.
6. Finally, *the main function*, where will be entering his choice of k values , distance measure and soon.
7. Later, several test cases are displayed using all the above function that satisfies a criterion

Results

Device Location Data

This dataset has around 100k points which are plotted along the coast of U.S. So, by using either Euclidean distance or Great Circle distance difference cannot be observed.

Euclidean (k=5)

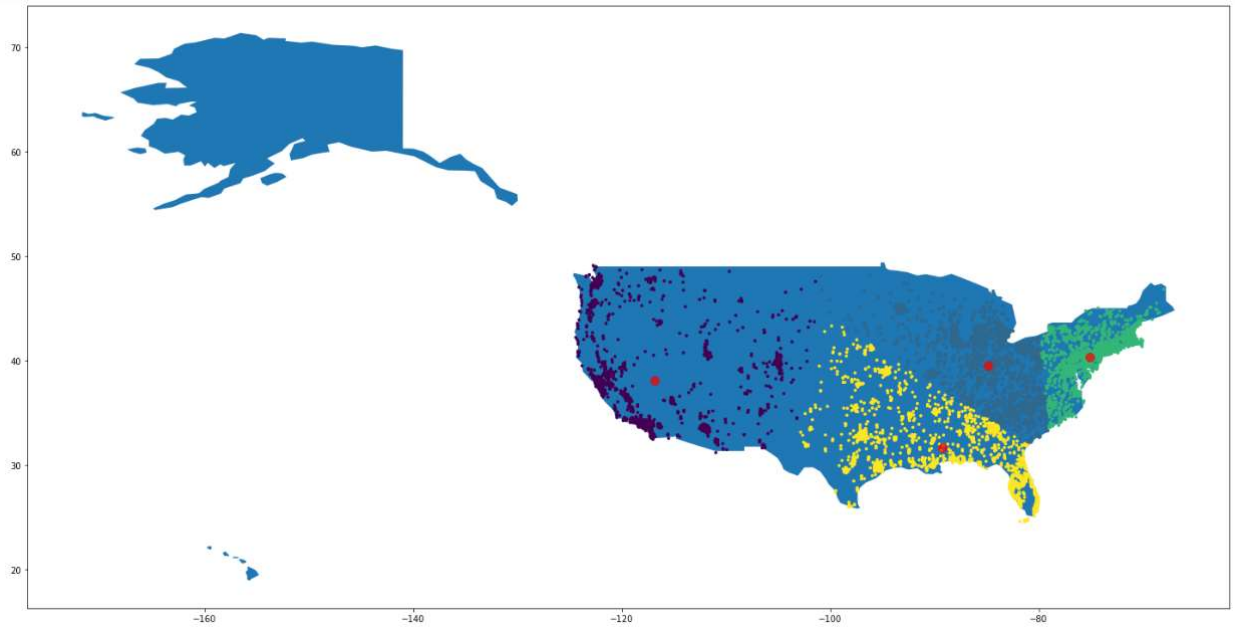


Great Circle (k=5)

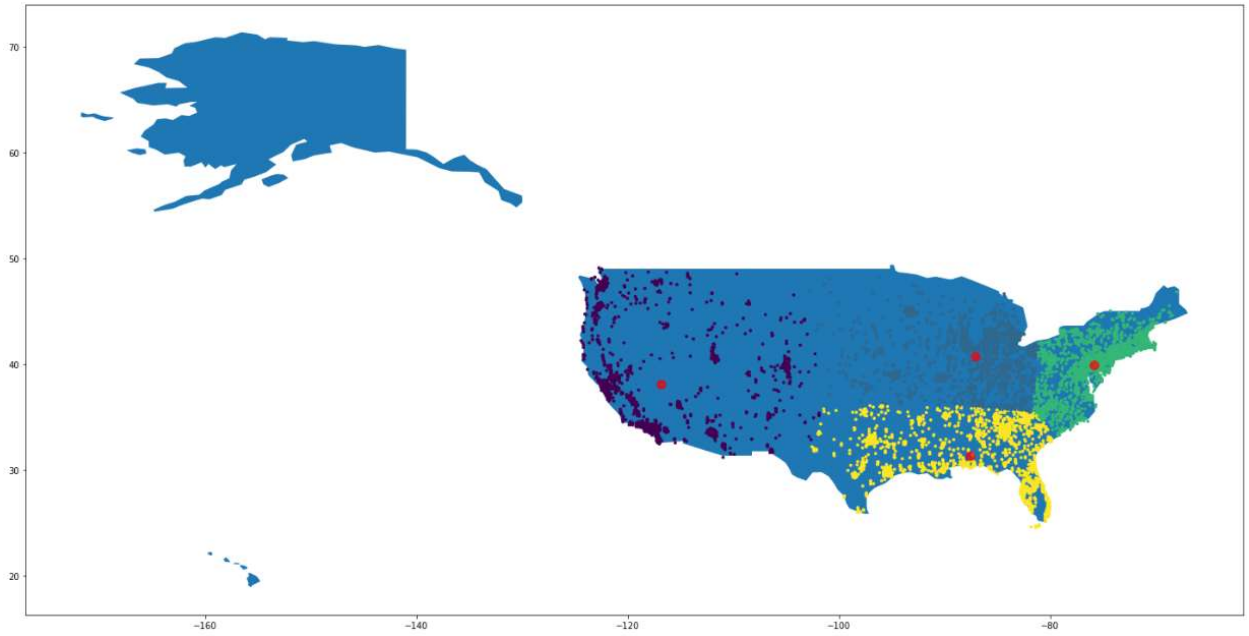
Synthetic Location data

For this, there are two conditions where k = 4 and k=6 values

Euclidean (k=4)



Great Circle (k=4)

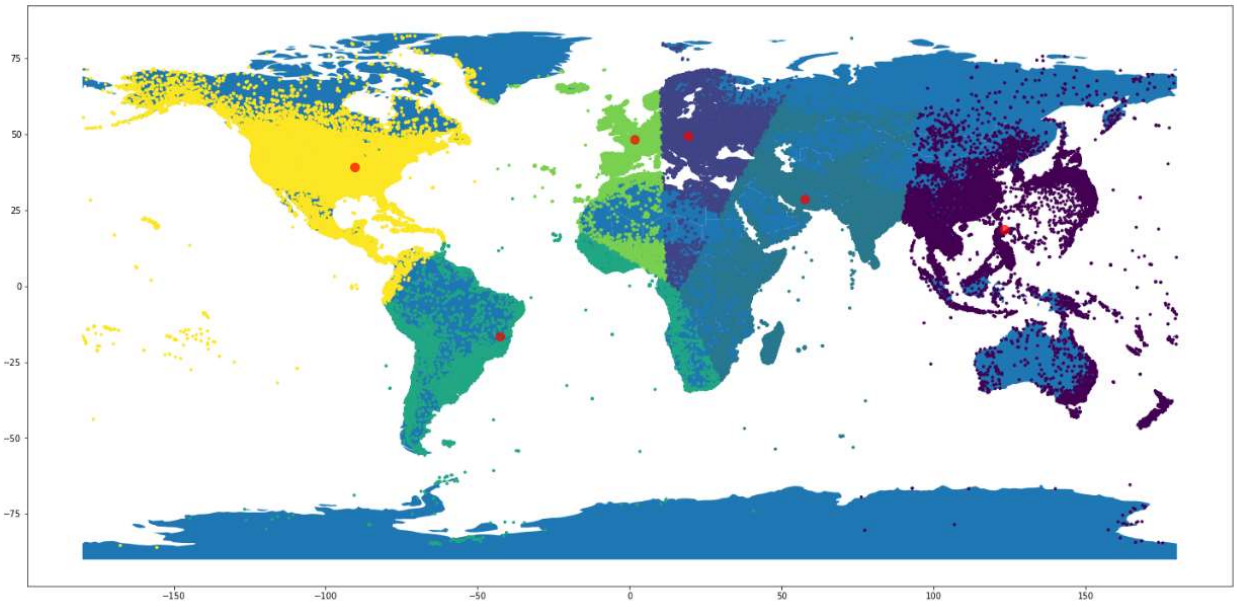


For k=6:

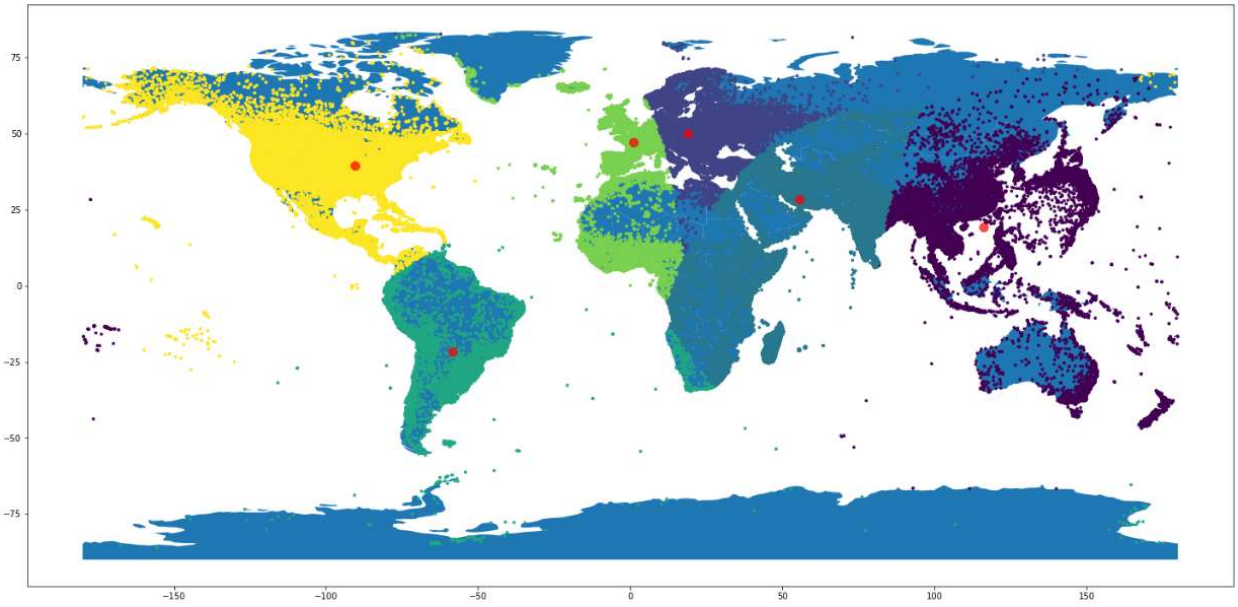
Euclidean (k=6)

DBpedia Location data

Euclidean (k=6)



Great Circle (k=6)



Argue, what choice of k makes sense by considering the problem context, i.e., what could the clusters mean/represent?

As of my observations, by calculating the final cluster centers for 3 datasets to both distance types, minimum k(number of clusters) value must be greater than or equal to 4.

The clusters can be used to interpret a kind of group each cluster represents based on the centroids.

In the above test cases, by placing k value as 2 or 3 , the points are unable to decide which cluster group it belongs to. Therefore, by increasing the minimum number of clusters to 4 for a huge dataset, it can find its centroids easily.

We can use the Elbow method for one in selecting the value of k,

Elbow method :

$$SSE = \sum_{i=1}^{K} \sum_{x \in c_i} dist(x, c_i)^2$$

Runtime Analysis

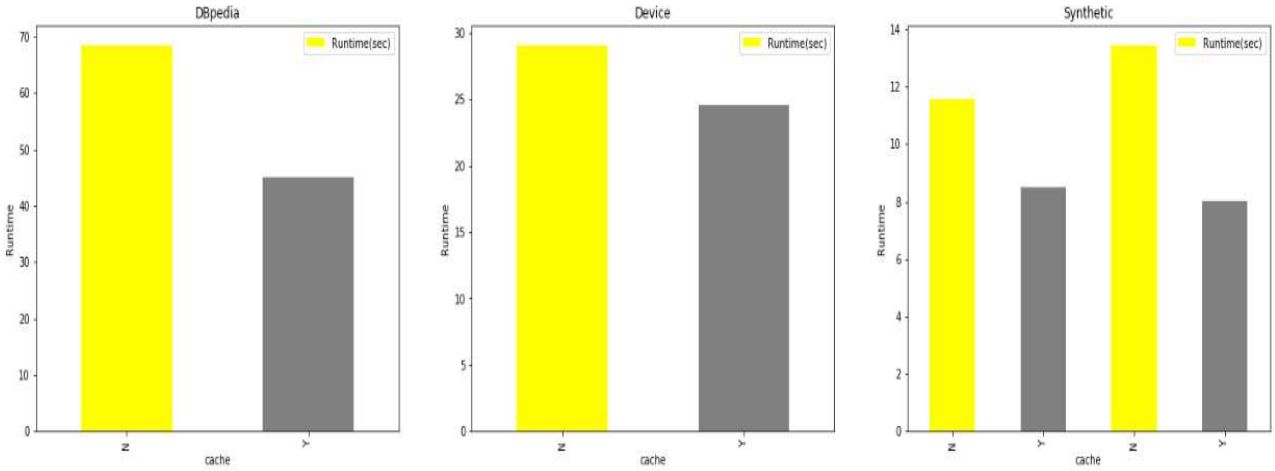
The below table shows the runtimes for each test cases and the bar plots are shown further below.

	k	data	d_type	cache	Runtime(sec)
1	6	DBpedialocation	euclidean	N	68.512020
4	6	DBpedialocation	euclidean	Y	45.063866
2	5	devicelocation	euclidean	N	29.084178
7	5	devicelocation	euclidean	Y	24.517869
3	4	syntheticlocation	euclidean	N	11.557487
6	4	syntheticlocation	euclidean	Y	8.489901
0	6	syntheticlocation	euclidean	N	13.454243
5	6	syntheticlocation	euclidean	Y	8.024613

Tabular chart

Findings

- From the above representation based on cached and non-cached data, we observe:
- For DBpedia data , k=6, cached algorithm executed approx. *23 seconds* faster than non-persist one.
- For Device Location and k=5, the algorithm ran faster when it is cached with a difference of almost *5 seconds* when compared to both the runtimes.
- For Synthetic Location, k=4 and k=6, not much change is observed i.e. a cached model ran approx. *3 seconds* for k=4 and approx. *5 seconds* for k=6 faster than normal one.



Bar plot

Conclusion

Final Conclusion is that not much difference is observed in case of a smaller dataset, but the change in the partitions on the map is much more in case of a larger one and caching the datasets has proven to increase the efficiency and decrease the runtime of the algorithm.

References

The introduction guide taken from Marion Neumann's CSE 427 (Fall 2019) course at Washington University in St. Louis, which was itself adopted from Pedro Domingo's' class on Data Mining/Machine Learning at University of Washington, 2012.