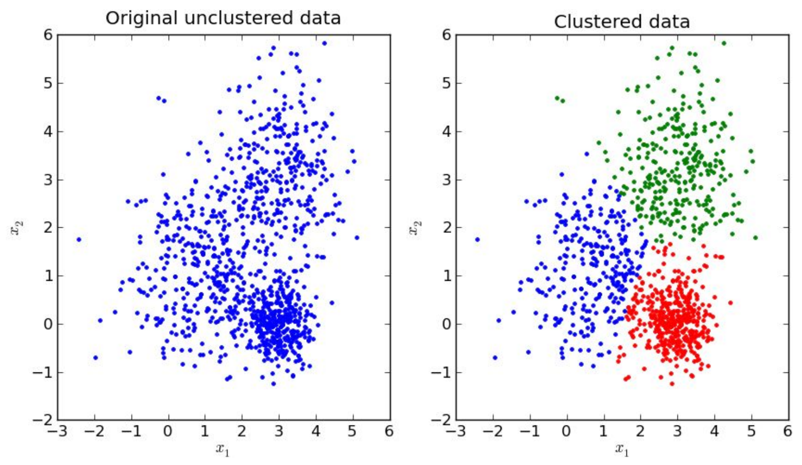
Final Project 3: *k*-means for Geolocation Clustering in Spark

Jacob Lee (jacob.lee / 435383) [Project Manager]
Frank Moon (f.moon / 431147) [Developer Local]
David Yang (yang.david / 451666) [Developer Cloud]
Nigel Kim (seunghwan.kim / 431659) [Key User]

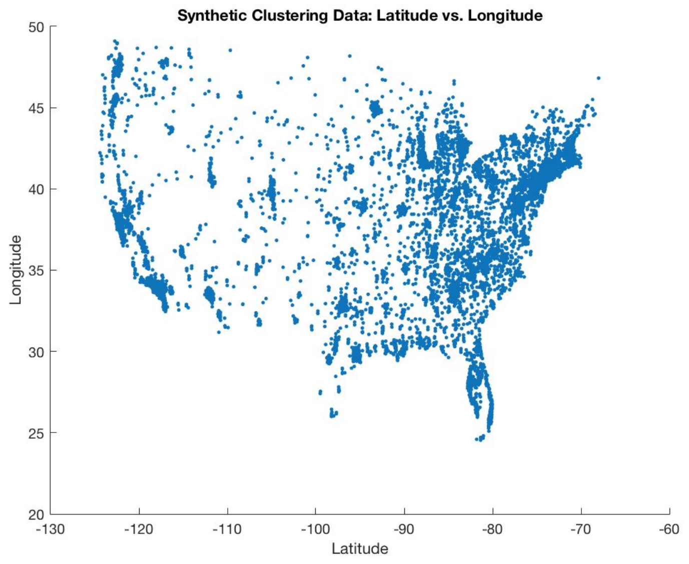
CSE427S

Motivation

For this project, we implemented the *k*-means clustering algorithm using Apache Spark. The *k*-means clustering algorithm is an unsupervised learning algorithm that groups *n* points of data into *k* different clusters, based on a similarity measure between each datapoint and the center of the nearest cluster. It then returns the location of each cluster center and classifications for each point in the dataset. A visual example of the input and output of the algorithm is shown below[1]. In this example, $k = 3$ and each color represents membership of a cluster.



We implemented the algorithm using Spark because it is optimized for distributed computing on volumes of data too large for a single computer to store and process. This proved to be crucial, as we eventually ran the algorithm on a very large dataset from the video game PlayerUnknown's Battlegrounds (PUBG), using a cluster on Amazon Web Services (AWS).



As we used geospatial data, the visualizations are a bit more intuitive than the fairly abstract example of *k*-means clustering above. This can be seen in our basic visualization of the provided Synthetic Clustering Data to the right, which is quickly recognizable as spatial data from the United States. Clustering would simply label each point with color while still preserving the general shape, allowing for easy understanding.

From a technical standpoint, this project gave us experience with working with Spark, distributed computing for big data, spatial data, the k-means algorithm, and data visualization. From an organizational standpoint, the project also gave us practice with the big

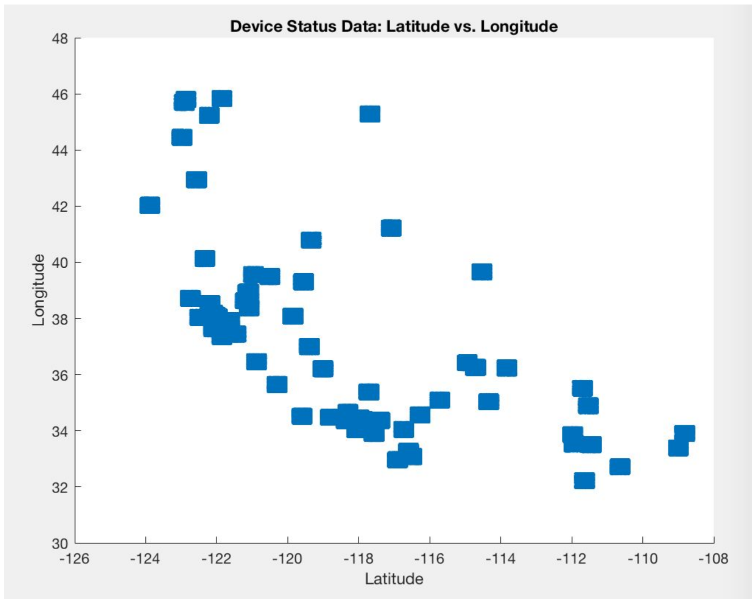
[1] https://mubaris.com/posts/kmeans-clustering/

data development pipeline. This project highlighted the necessity of the pipeline, especially as computation time on AWS is costly. All of this experience is highly valuable to working with big data in the real world.

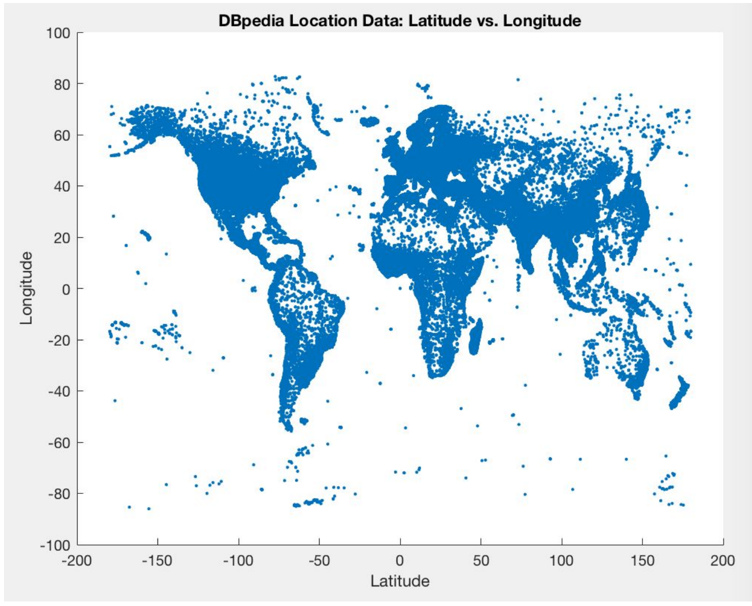
Approach

The project instructions essentially guided us through the development pipeline, with various aspects of the pipeline being managed by each team member.

To begin, the key user (Nigel) gathered and preprocessed the provided datasets, and then took small samples for the developers to develop the algorithm on. The local and cloud developers (Frank and David) then implemented and tested the algorithm locally based on these samples. After this, we ran the algorithm on the pseudo distributed cluster, using all three of the provided datasets in their entirety.



The first provided dataset is the *Loudacre Mobile* dataset, which is device location data from a fictional cell-phone carrier in the United States. A visualization of a subset of that dataset is shown on the right. The second is a synthetic location dataset from statistical-research.com. The third is *DBpedia* location data on 450,151 places that have Wikipedia articles about them (also shown on the right).



Because all of these datasets use the same type of simple location data (longitude + latitude), all we had to do was specify which parameters in the *k*-means algorithm were being used per run. These parameters can be adjusted to better capture the unique aspects of each dataset. The specifics of the implementation and parameters will be discussed in the Implementation section of the paper.

Concurrently, the key user worked with the cloud developer to find, process, and upload the real world dataset onto the AWS cluster. We chose the real world data (PUBG) because it used also used spatial data and because it had sufficient volume to be described as big data (9,845,492 x 2, before subsetting).

Similarly to above, the key user provided small samples of the real world data to the local and cloud developers, so that they could test both locally and on the pseudo distributed cluster. After that, the key user provided a larger volume of representative samples, which we ran on both the pseudo distributed and actual cluster.

As sending and processing data on AWS costs money, we were careful about checking our code and data at each stage. Finally, we then ran the algorithm with the full dataset on the actual cluster. After receiving our results, we worked on visualization and writing the report. Communication and organization of the project and report was managed by the Project Manager (Jacob).

Implementation

Parallel Processing and RDD Persistence

Parallelism is crucial in reducing computation time for big data jobs. To better understand the parallelism of our implementation, we focus on the number of "stages" in the overall Spark job. A stage is considered the basic unit of execution for Spark, because multiple stages cannot run in parallel with each other. In our implementation, the number of stages corresponds with the number of iterations in the algorithm, due to these action calls in each iteration: reduceByKey() and countByKey().

However, within a stage there can be multiple "tasks" executed in parallel. Although the high number of stages limits parallelism to an extent, we can still cache the resilient distributed dataset (RDD) that contains all input points.

We do this because at each iteration we must return to the input points and recalculate the nearest centroid to each point, as the centroid locations have been updated between iterations. Returning to the input points means partitioning the entire RDD that contains the points across the corresponding work nodes. This pipelining is very likely to remain consistent throughout the runtime of the algorithm since the points remain consistent and at each iteration the same actions are invoked. Thus, repartitioning for every iteration is redundant.

Caching allows each node not to have to return to the entire RDD but work from its own memory because caching saves the node's working partition of the RDD in its memory. This eliminates the need for any redundant repartitioning for a cached RDD. Therefore, this strategy greatly reduces the runtime of the application.

Input Handling

We had two processing tasks for input data within the program. Firstly, for each point, all of the coordinate values must be contained within a single container that allows public access to its members, to facilitate passing the values in as parameters to Python and SPARK functions. Secondly, to be applied various distance measures, a point container must support flexible conversion. This conversion is to and from latitude-longitude coordinates and Cartesian coordinates in the form (X, Y, ...).

Our solution was to use 2 classes:

- `latlon` takes in a (latitude, longitude) pair. This class was necessary since latitude and longitude is directly used in great circle distance, one the distance measures that we had considered. Also the format of input for many spatial are in latitude and longitude pairs, which calls for the class that can handle this type of data and represent a Point simultaneously.
- `XY` or `XYZ` handles the Cartesian representation of a Point. This class was necessary since the input PUBG data was in this format, and computing the mean of Points requires this representation.

Termination Criterion: Optimization by Centroid Convergence

Ideally, we would want the algorithm to immediately terminate when the centroids are at their optimal positions and no positional adjustments are needed between iterations. A perfect solution would have the change between iterations be 0, but this is difficult (if not impossible) to achieve in practice.

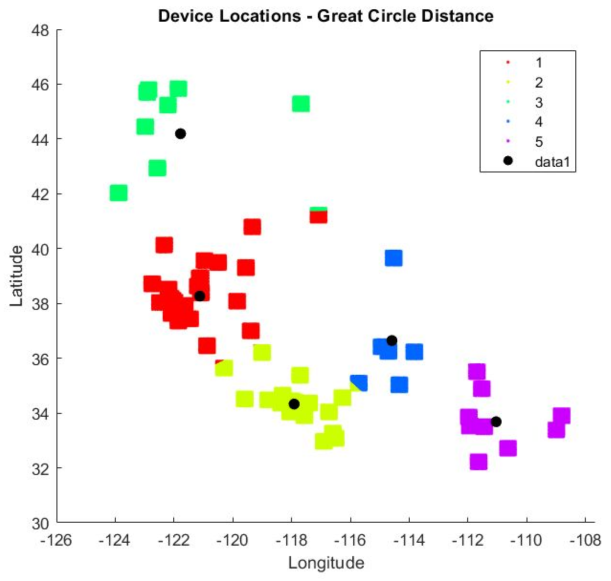
The essence of a *k*-means algorithm is that as the centroids continue to be updated as the mean of each new cluster per iteration, the changes in their locations before and after will decrease. In other words, the centroids converge. Therefore in our implementation, we set a small, positive, real number tolerance (`convergeDist`) to the total change in the positions of the centroids between before and after each iteration as our termination condition. We initially set `convergeDist` to be 1, as this is sufficiently small compared to the coordinate systems. As seen below, the termination condition gives the algorithm enough time to return near-optimal clusters while still be able to terminate in finite time.

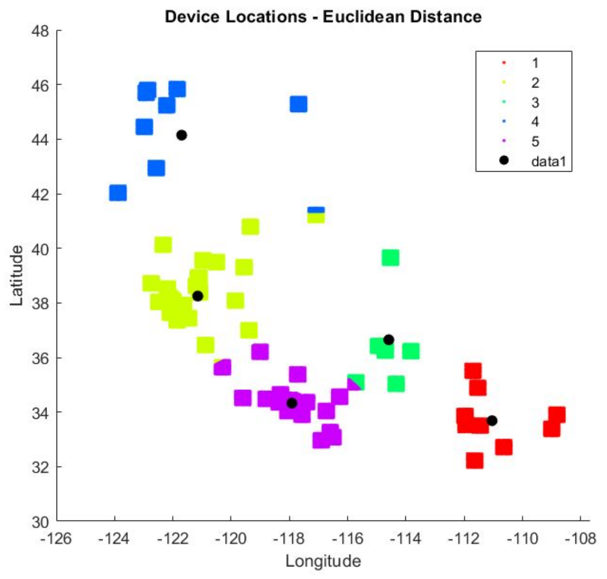
Performance on Provided Datasets

With the finished implementation of the algorithm, we started gathering data on its performance. We first measured its performance on the provided small datasets before moving to the big data application. This is because gathering performance metrics and testing on a huge dataset is impractical, so these smaller datasets gave us an opportunity to estimate performance. We used this information to better optimize the algorithm for the big data application later in the project.

1. Distance Measure Selection
   One important parameter of the algorithm was how distance is calculated. We considered two different distance measures: Great Circle distance and Euclidean distance. Great Circle distance measures the radial distance between two points, while Euclidean measures vector distance. We illustrate their impact on the clustering algorithm on the various datasets below:

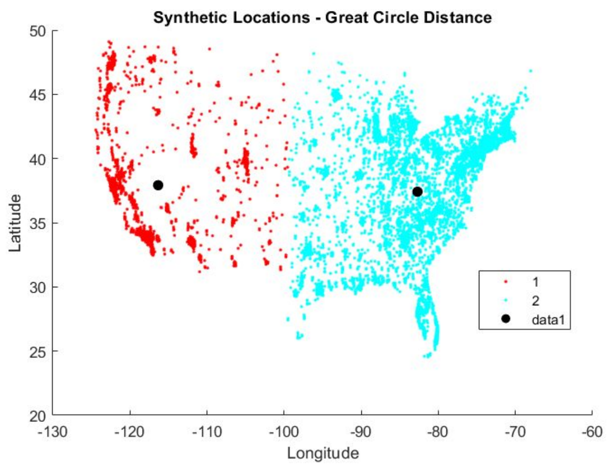
a. Device Location Data

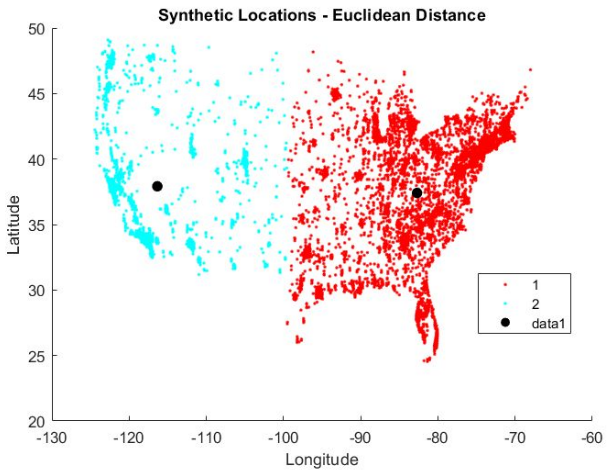




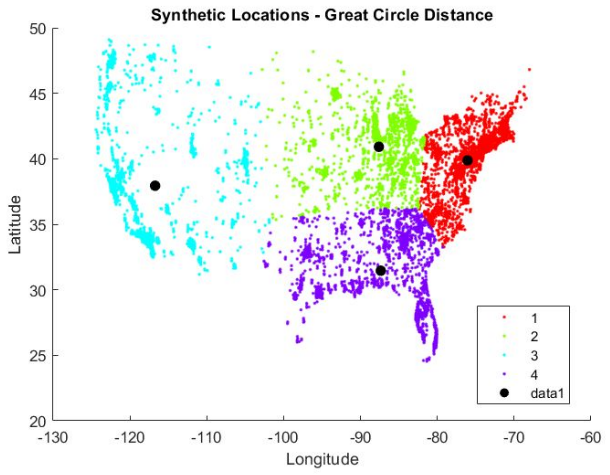
As seen above, there's not much of a difference between the Great Circle Distance and Euclidean Distance for this set of data points. This is likely because this dataset is a subset on a small region of earth, where curvature is not as significant. Therefore, measuring either radial distances or vector distances is close to equivalent.

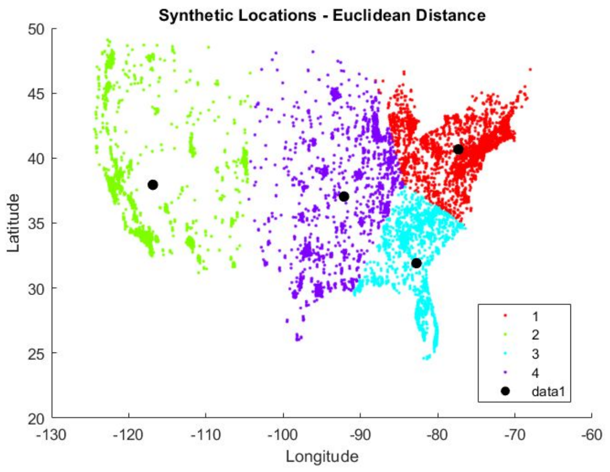
b. Synthetic Location Data





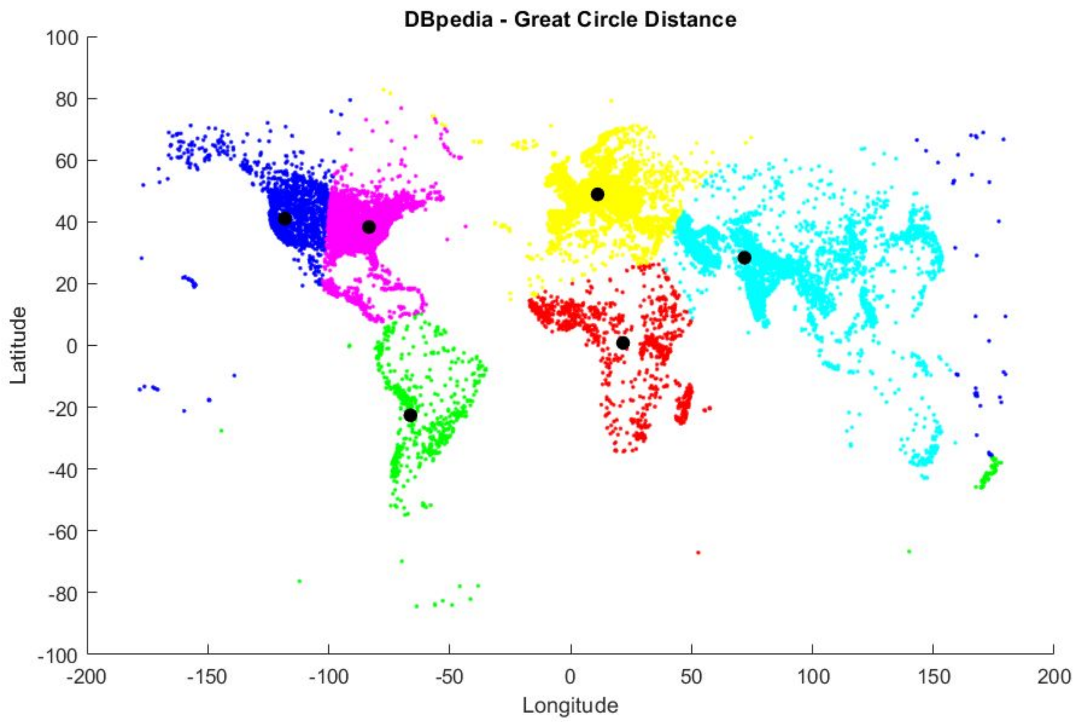
For the synthetic dataset, we initially set $k = 2$. However, this clustering did not show much useful information in terms of performance for reasons to be shown in the following discussion of finding the optimal value for k. The immediately following results, where we set $k = 4$, allows for better insight into what distance is to be used concerning latitude-longitude data.





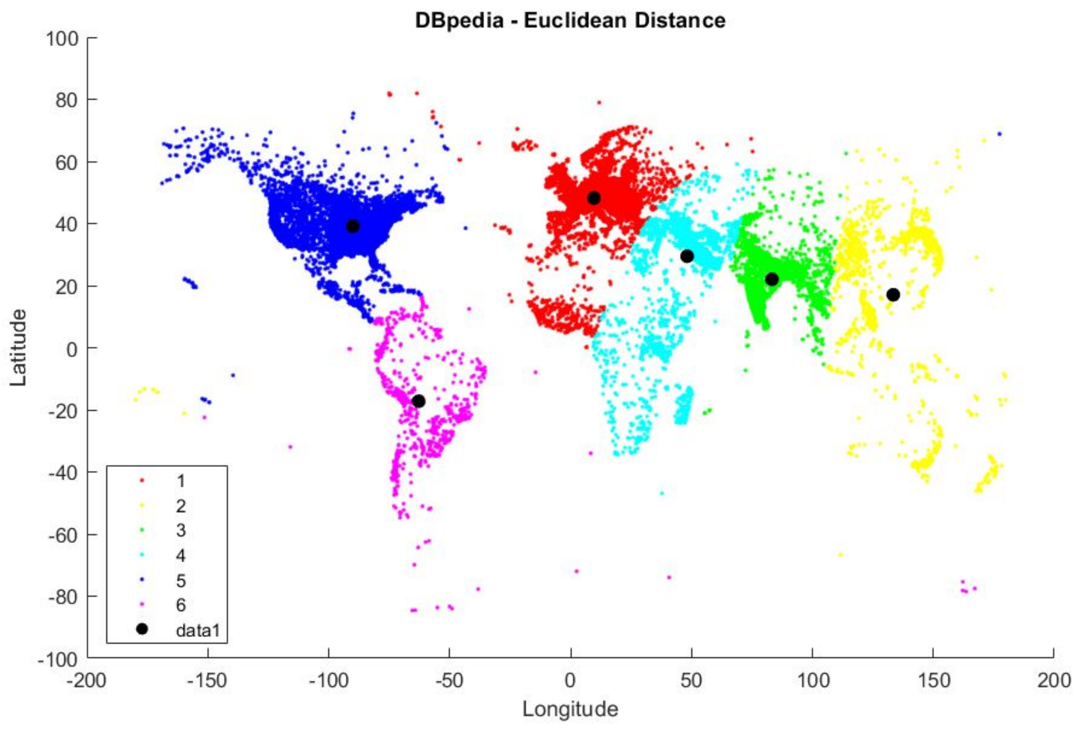
Here, the difference in clustering is evident. We can conclude that, assuming adequate *k*, the difference between the two distance measures becomes more significant as the domain of the spatial data increases.. Since this dataset is synthetic and no other information is available to us as to the specific meaning of the data, we could not make much interpretation as to what these clusters mean.

c. DBpedia



For Great Circle distance, the centroids along with their clusters each seem to represent a continent, with the exception of North America and Australia. These two exceptions may be attributable to the density of data points. North America seems to have an exceptionally abundant amount of data points, whereas Australia appears to be the opposite extreme.

It is expected that the clustering captures the distribution of continent as the dataset consist of locations on land as opposed to locations in the ocean, both of which are included in the data space: the entire Earth.



Interestingly, *k*-means using Euclidean distance does not capture the continents. Although the centroids do largely reside on land, their positioning ultimately results in a more or less uniform distribution longitudinally.

This is likely attributable to the fact that Euclidean distance does not accurately capture the geometry of Earth's spherical surface. Euclidean distance would be measuring the vector distance between two points in a given space. In this case, as the space is the entire Earth, Euclidean distance would measure the distance between locations "through" the sphere of the Earth. Thus, it appears impractical to consider Euclidean distance a useful measure on a spherical surface.

However, Great Circle Distance is costlier to compute compared to Euclidean distance. Since the algorithm involves iterative computation of means, it is inevitable that at one point during the application, the latitude-longitude coordinates need to be converted to Cartesian representation in a 3 dimensional space. Performing naive algebra on latitudinal and longitudinal values does not hold much significance. Once this conversion takes place, computing Euclidean distance is trivial, and at no point in the application are latitude-longitude coordinates needed. In contrast, Great Circle Distance requires that the computation is done using latitude-longitude form. The problem arises from that every iteration requires the computation of distances. The conversion therefore takes place twice for every point at every iteration. This increases the runtime of our application, especially as the number of data points increases.

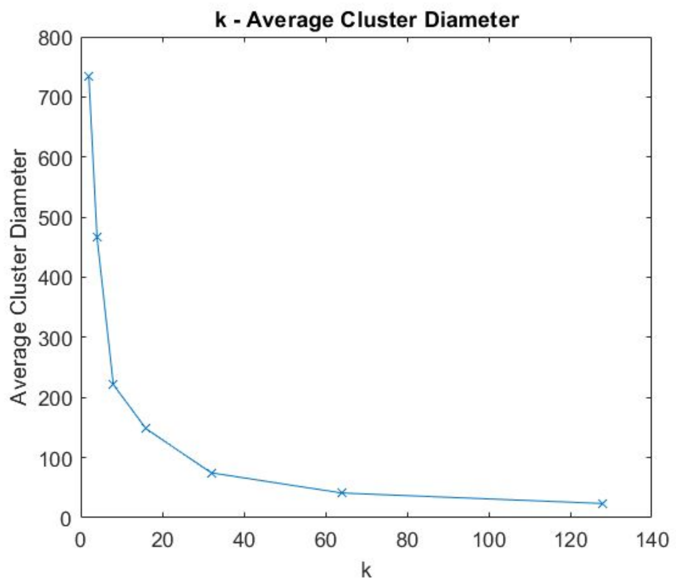
2. Parameter (*k*) Selection:

In order to analyze the behavior of our application in terms of varying *k*, our only true parameter, we chose one of the above small dataset, Device Locations, and along with it held all other variables constant, while only increasing *k*.

Results of increasing *k* at a 2-logarithmic rate, in elapsed time and the mean diameter of all clusters.

k	Elapsed Time (in seconds)	Average Diameter
2	260.76375699	733.585666803
4	328.523422003	466.635097488
8	1290.05509496	221.462470351
16	1002.21666598	148.160304039
32	2032.92604709	74.7129526224
64	7379.88112807	41.1508951686
128	13349.587765	23.6974115897

The table can be plotted as below:



This lines up with the conventional advice for the optimal value of *k*: as *k* increases, the average diameter of the clusters decrease to a certain point[2]. After said point, increasing *k* does not reduce the average diameter of clusters as much as it would increase the cost of each iteration. Adding in infinitely more *k* would eventually drive the average diameter to 0, but that would have no significant insights, as the output would essentially be the original dataset. For the device location dataset, the critical value for *k* is around 30.

[2] https://www.datascience.com/blog/k-means-clustering

3. Persistent RDDs

The following is a comparison table showing the performance of cached (persistent RDD) and uncached implementations, where all other variables are held constant. Each elapsed time is a mean taken from 3 trials and rounded to 1 decimal point.

		Cached	Uncached
DBpedia (k = 6)		**1.3 minutes**	3.6 minutes
Synthetic Location	k = 2	**12.5 seconds**	14.3 seconds
	k = 4	**18.2 seconds**	20 seconds
Device Location		**3.2 minutes**	4.9 minutes

As discussed in the Implementation section, caching significantly reduces the runtime of the application. This contrast is stark especially as the number of data points increases. The more data points, the more partitions the RDD with all the points must be divided into. By caching each partition into the memory of each node, we eliminate the redundant partitioning.

Big Data and Cloud Execution

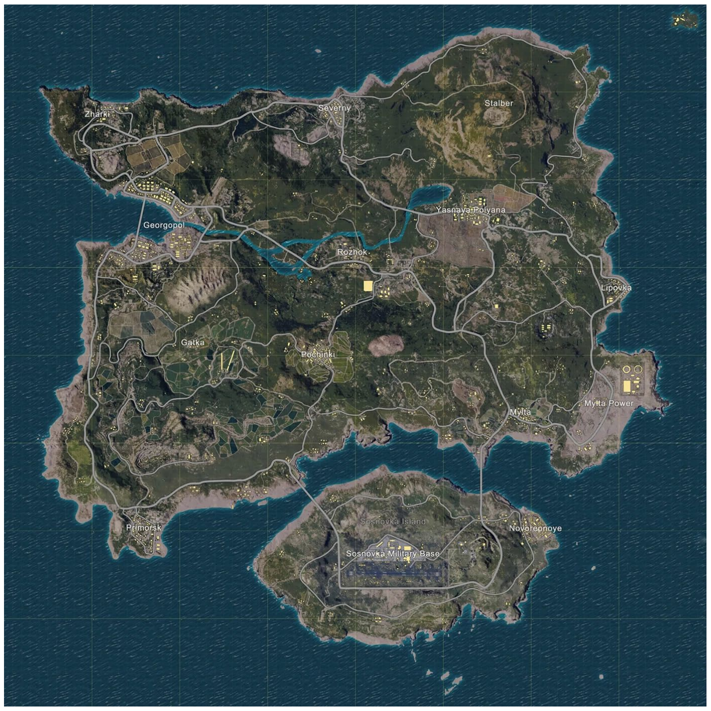
Dataset Description

We used a dataset based on the computer game PlayerUnknown's Battlegrounds (PUBG). PUBG is a first-person 'battle-royale' shooter, where over 90 players are placed on a large island. Teams of players and individuals fight to the death until one team or player remains.

We found the dataset on the website Kaggle. It contains (x,y) coordinates for player deaths in over 720,000 competitive game matches. The curator of the dataset said that he scraped the dataset from a PUBG metrics website[3]. To initialize the gathering, he used a seed player (his own low level character) that collected the information of all players he encountered[4]. He then took a random subset of 5000 players from the collected data records, and scraped their match history to create the final dataset. Note that there is a potential sampling bias issue here, which we will discuss in the conclusion.

Each row specifies a single player death, with these columns:

[killed_by, killer_name, killer_placement, killer_position_x, killer_position_y, map, match_id, time, victim_name, victim_placement, victim_position_x, victim_position_y]



We then subsetted the data to only be on one map (named Erangel[5]). We chose this map because it's the first and most popular map in the game[6]. Erangel itself is supposedly 8,000m x 8,000m (divided into 800,000 x 800,000 coordinate units).

After this, we preprocessed the dataset with Python Pandas library so it solely contains victims' death locations with just the x and y coordinates of the event. For visualization purposes, we also linearly scale the (x,y) coordinates so that they can be plotted onto the Erangel map in the range [0:800,000].

[3] http://pubg.op.gg
[4] https://www.kaggle.com/skihikingkevin/pubg-match-deaths
[5] https://vignette.wikia.nocookie.net/playerunknowns-battlegrounds/images/0/0d/Russia_map.jpg/revision/latest?cb=20170913160231
[6] https://www.eurogamer.net/articles/2018-12-07-pubg-map-best-spawn-locations-vehicle-boat-locations-where-to-start-4415

Implementation and Execution

Data Preprocessing and Choice of Dataset Size

The raw data from the Kaggle dataset is 2GB, 13.4 million rows by 17 columns. While we can store the raw data to the cluster and preprocess with another SparkStep before running the k-means clustering Spark step, uploading and downloading data with size of 2GB would risk overshooting the free data transfer limit of AWS S3 storage unit. Therefore, we decided to preprocess the data locally before uploading to the AWS. Using *pandas*[7], a popular open source Python data analysis library, we are able to drop the unnecessary columns as well as filter out rows from other map(s) locally within 3-4 minutes. The resulting file is 177.2MB with 9.8 million rows by 2 columns.

For our application, too many data points would increase the runtime without giving us better quality insights about the data. Therefore, we took a random sample of 500,000 rows of data from the preprocessed datafile to use for the algorithm. From our graphs (see below), we see that 500,000 data points not only takes a reasonable amount of time to run but also covers enough of the map to give us sufficient insights.

Distance Measurement

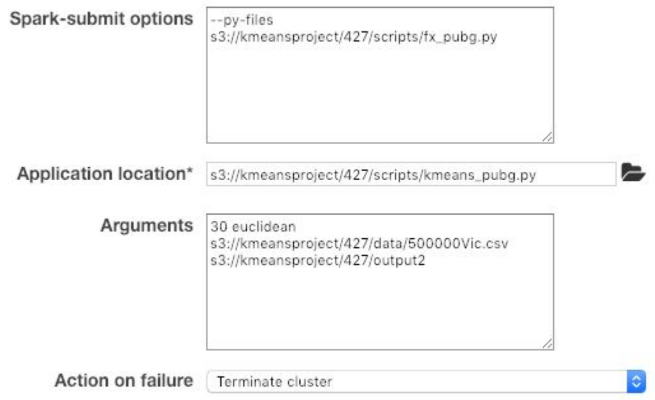
The main difference between these two datasets and the project-provided datasets is that these datasets use virtual (x,y) coordinates instead of (longitude, latitude) pairs. This means that great circle distance no longer applies, as these are just euclidean points on a "flat" surface.

Termination Criteria

We changed the `convergeDist` tolerance from 1 to 1000 (~10m), as 1 in the in-game coordinate system would correspond to 1 millimeter. That would be too short for the algorithm to converge in a reasonable amount of time.

Cluster Setup

We stored the python scripts as well as the preprocessed data files on an Amazon S3 Storage bucket, and then initialized a cluster with a Spark Step configured as below running on half of a million data points. The cluster contains one Master node and two Core node.



Spark-submit options	--py-files s3://kmeansproject/427/scripts/fx_pubg.py
Application location*	s3://kmeansproject/427/scripts/kmeans_pubg.py
Arguments	30 euclidean s3://kmeansproject/427/data/500000Vic.csv s3://kmeansproject/427/output2
Action on failure	Terminate cluster

7 https://pandas.pydata.org/

Results

Runtime

Data Size (in data points)	k	Runtime
0.5 mil	100	175 minutes
0.5 mil	30	27 minutes
0.5 mil	10	5 minutes

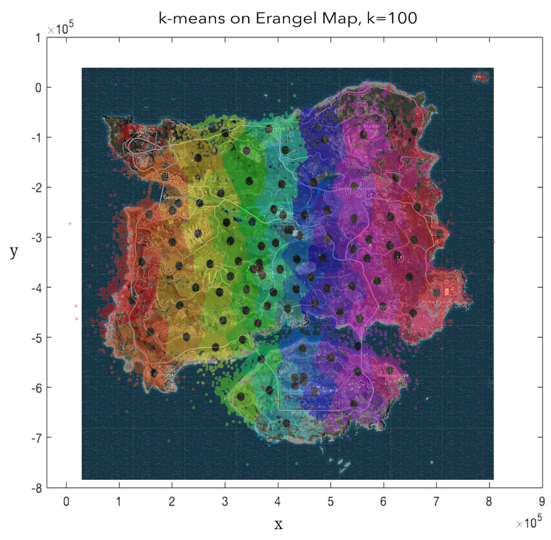
According to the *k*-means clustering Wikipedia page[8], the upper bound of Lloyd's algorithm for k-means is *O(nkdi)* where n is the number of data points, d is the dimension (2 in our case), k is the number of clusters, and i is the number of iterations. From the table above, we can see that reducing the number of clusters decreases the runtime as expected. This behavior resembles the logarithmic decay of the results using the small datasets in *Parameter Selection* of *Performance on Provided Datasets*, which is to be expected.

For our purpose, we fixed the dataset size at 0.5 million because it contains enough data to span the entire map while providing us valuable insights within an efficient runtime.

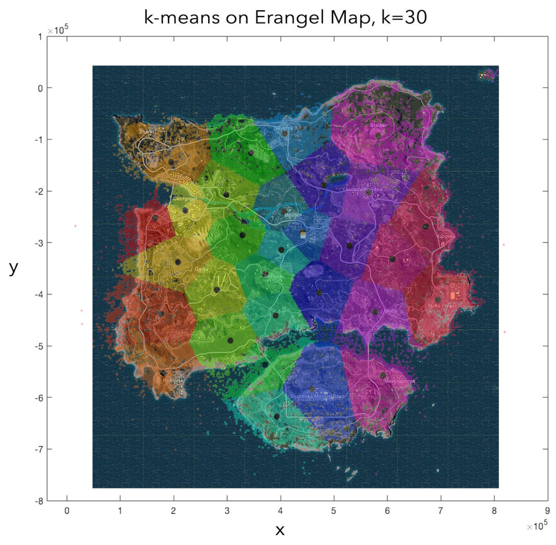
[8] https://en.wikipedia.org/wiki/K-means_clustering#Algorithms

Graphs

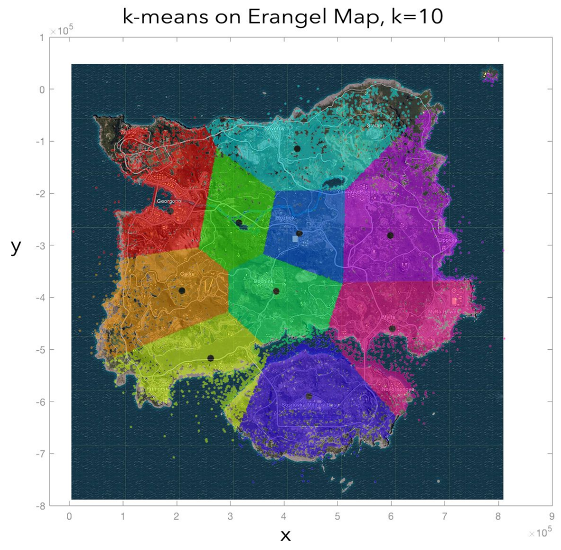
Below are the graphs from the resulting data from the experiment with parameters specified above (k=100,30,10), and we can see that from each graph we can draw insights of the data on a different scale. We also included a graph from a PUBG online gaming blog that marks the locations on the maps with the most loot. Loot include weapons and medical kits, which makes it essential for long term survival. Therefore, more loot attracts more people, which results in more conflicts. Since the PUBG location data is simulated on a flat 2-D surface, it would only make sense to use the Euclidean distance for k-means clustering.



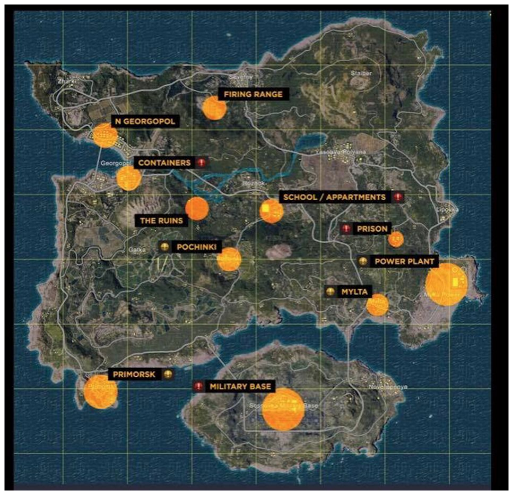
(1) k=100



(2) k=30



(3) k=10



(4) places with most loot[9]

[9] https://www.greenmangaming.com/intel-feature/playerunknowns-battlegrounds/starting-out-in-pubg/

Insights and Implications
From our various experiments, we were able to draw valuable insights regarding which areas on the graph have the most associated deaths.

Comparing graphs (3) and (4), we can see that our clusters correspond with areas that are reported to have the most loot. This serves as a validation of the correlation between loot locations and areas with high risk of death.

However, our higher choice of k also reveals more insights than the online gaming blog does, since the locations of loot is not the only factor that would increase the amount of conflict. In graph (2), for example, we find centroids corresponding to named areas of the map, such as “God Tower”, “Swamp Town” etc. These kinds of areas generally contain better cover and vantage points for sniping.

The insights generated from these graphs can be useful for modifying gameplay behavior. For example, if a player’s ultimate goal is to survive until the end (the win condition) regardless of how many kills they score, they can use the map to avoid the high-conflict areas as much as possible. Or, if a player has already gathered enough loot and is eager to find areas where conflicts often occur, they can also refer to the centroids in the graph to locate conflict.

Conclusion

Lessons Learned

As stated in the Motivation section, we gained a lot of valuable exposure to and experience with many technical concepts. This is especially true because of the big data nature of the project. The sheer volume of the data required careful planning and implementation. If we had used the full dataset or perhaps an even bigger one, computation time alone may span from hours to days to weeks. This is not only true for pseudo-clusters, but also on AWS cluster nodes, where computation time can be extremely expensive.

Therefore, it is important not only to understand the models being employed, but also to understand how to implement it efficiently and cheaply. We did this through understanding and manipulating Spark, both through its parallelization and also through using resilient RDDs. Although at first it appeared as though the action calls within the iterations would prevent parallelizing the iterations, caching the input data allowed us to increase parallelism and decrease the runtime significantly.

In addition to this, Spark’s compatibility with Python greatly added to the ease of writing the code, because of Python’s simple syntax and libraries. This helped us while we experimented with different visualization methods. In the end, we chose to use MATLAB because of its speed with large volumes fo data and ease of use.

Much like all other data, big data often requires preprocessing. Here, again we largely relied on Spark, while sometimes using *pandas* for Python. To sum up, we learned that we can complete any job with the right tools, of all of which we learned the fundamentals from the course.

Future work

An obvious next step after this project is predictive analytics. *k*-means clustering is an unsupervised learning algorithm, and outside of specific use cases (like feature selection or image compression[10]), its conclusions can be fairly esoteric. Developing a different model on this same data can likely generate more directly applicable conclusions.

For example, we could develop a predictive model that displays probability of death at each location. This number, along with a minimap displaying a death probability field, could be a useful tool for players learning to position themselves to maximize their chances of survival.

There is also an opportunity to include the columns that we excluded (like time of death) as additional predictors. In addition, the curator of this dataset also provided an additional dataset that can be joined with the death dataset on match_id. This other dataset contains variables like game_size (# of teams playing), player_kills, player_dmg, and player_dist_walk (total distance travelled on foot). All of these variables could also potentially be used to predict probability of death at any given moment.

At an even higher level, we could train a neural network to play this game, where the outputted probability of death from the above model is one input into its behavior. All of this work would likely require feature selection and additional research on model selection and parameter tuning to maximize effectiveness.

[10] https://medium.com/machine-learning-for-humans/unsupervised-learning-f45587588294