COMP 4106

Final Project

Tyler McBride

100888344

Problem Domain

Given the set of n latitude and longitudes coordinates,

$$S = \{(lat_1, long_1), (lat_2, long_2), \dots, (lat_{n-1}, long_{n-1}), (lat_n, long_n)\}$$

The problem I proposed was to split S into k clusters—where $k \leq n$—of roughly equal size,

$$C = \{C_1, C_2, \dots, C_{k-1}, C_k\}$$

In other words, these clusters require a minimum cardinality of $\left\lfloor \frac{n}{k} \right\rfloor$ and a maximum cardinality of $\left\lceil \frac{n}{k} \right\rceil$. This is because unless n is a multiple of k, $\frac{n}{k}$ will produce a remainder; therefore, some subsets will have a cardinality of $\left\lceil \frac{n}{k} \right\rceil$. This gives us,

Each cluster has a respective center coordinate, and the problem is to minimize the sum of the squared Euclidean distances to these respective cluster centers. In other words, achieve the minimal distance between each cluster center and that of each element within the respective cluster. In mathematical terms, the problem is to find the minimum for:

$$\sum_{i=0}^{k} \sum_{p \in C} \|p - c_i\|^2$$

Where c_i is the respective cluster center for C_i, and each C_i has roughly equal cardinality.

Motivation

I have volunteered with the Peterborough Youth Soccer Club for many years, and I am currently developing an application for their convenors to create and manage teams. Growing up, I too played in the Peterborough Youth Soccer Club league and found that often times my family had to drive across town to my soccer game.

With the solution to this problem, I am trying to both reduce gas consumption and save time by having members of each team live in proximity to one another; therefore, teams can be scheduled to play games at fields within close proximity and even potentially carpool to said games.

AI Techniques

K-Means++

The solution begins by initializing cluster centers using the k-means++ algorithm. In the code this algorithm is contained in the KMeansPlusPlus.java class. This algorithm will begin by choosing a point at random and assigning the first centroid to said point location. After this first centroid is chosen, the algorithm will then for each individual point, calculate the distance to the nearest initialized centroid.

These distances will be used to determine the probability of being chosen; therefore, the further away a point is from that of an initialized centroid the more likely it will be chosen. In order to increase the likelihood of a further distanced point, each distance will be squared.

Afterwards, with the use of a weighted probability distribution, a point will be randomly chosen to be the next centroid. Overall, it will be much more likely for a centroid to be a point that is far away from the set of already initialized centroids, or the point in a dense area of points, and even more likely if both of these conditions are met.

With the use of k-means++, the optimal centroid locations will be estimated to minimize the average squared distance between points in the same cluster. This algorithm will effectively try to estimate the solution to the best degree as possible in order to avoid the potential poor set of clusters found by the using the standard k-means algorithm alone.

K-Means with Equal Size

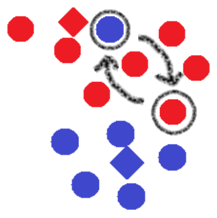
After the k-means++ initialization, the k-means algorithm is then executed. I have created my own variant of this algorithm that achieves roughly equal sized clusters. The method begins by pre-computing the distances between each point and each centroid.

After these distances are computed, the algorithm then iterates over the points and assigns them to their nearest centroid; however, if the cluster has reached the minimum cardinality of $\left\lfloor \frac{n}{k} \right\rfloor$, the point is instead assigned to the nearest centroid which has not yet reached the minimum cardinality.

After each cluster has reached the minimum cardinality, the remaining points will be assigned to the nearest centroid which has not yet reached the maximum cardinality of $\left\lceil \frac{n}{k} \right\rceil$. This will guarantee roughly equal cardinality; however, because k-means++ does not guarantee initial centroids that have equal cardinality, there will more than likely be points whom are assigned to the incorrect centroid.

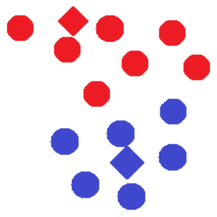
For example, if $k = 5$ and $n = 30$, but the k-means++ algorithm creates a centroid for which 7 points are nearest to, it is virtually impossible to know which 5 points of the 7 should be assigned to said centroid. If we choose the 5 points that are closest, the remaining two may be outlier and are no closer to the given centroid than any other centroid; therefore, there will almost always be incorrectly assigned points that will need to be addressed.

To combat these incorrectly assigned points, I implemented a swapping functionality. This functionality will begin by iterating over each point and fetching the nearest centroid to said point. If the nearest centroid to said point does not equal the centroid assigned to that point, then the algorithm will search each point within the cluster respective to the assigned centroid to find a point which is closer to the nearest centroid to swap with.



The example above illustrates two points swapping. The colours depict the different clusters, while the circles represents a point and the diamond represents a cluster.

As you can see from this example, the blue point within the red cluster is assigned to the blue centroid, but the red centroid is actually closer to the point; therefore, the blue point will attempt to switch with a point within the red cluster that has a minimal distance to that of the blue centroid, if said point exists the two points will switch assignments.



As you can see, after swapping the two points, the clusters have effectively minimized the average squared distance between points in each cluster. There are many edge cases that exist, which this algorithm covers but the main logic of this k-means variation is captured in the above example.

Addressing these edge cases the algorithm is a little more complex, if the nearest centroid to the incorrectly assigned point does not contain a point that minimizes the distance—say the point is already as close as possible to the assigned centroid within the cluster—the point will then attempt to swap with a point in an adjacent cluster.

After the point travels to the adjacent cluster, the point will then recursively try again to swap within the new cluster. This will allow points to jump from cluster to cluster until they reach their assigned cluster. Overall, these swaps are implemented in order to minimize the average squared distance between points in each cluster.

State Space

The goal of the design was to reduce the amount of memory required for the program by storing all of the information in either primitives or the lightweight coordinate class. This coordinate class contains two `double` primitives, one representing the latitude and the other representing the longitude. Both the centroids and the points are stored within a separate

`Coordinate[]` array; therefore, each centroid and point may be referred to by there index within these two arrays.

The distances between each point and each centroid are stored within a `double[][]` array, where the first dimension represents the centroid index and the second dimension represents the point index. These distances are calculated using the Haversine formula. This formula is "an equation important in navigation, giving great-circle distances between two points on a sphere from their longitudes and latitudes." (rosettacode, n.d.)

To keep track of which point is assigned to which centroid, there is a separate `int[]` array where the indices correspond to the point `double[][]` array, while the value represents the indices of the centroid `double[][]` array. These centroids are calculated by converting each coordinate within the cluster into a 3D vector, summing all these vectors, normalizing the resulting vector and finally converting back to a latitude longitude coordinate.

The sizes of each of each centroid is stored within an `int[]` array where the indices correspond to the centroid `double[][]` array and the values correspond to the cluster size of that given centroid. A single `int` is stored to represent the maximum size of each cluster to prevent each cluster from violating the size constraint.

Application Results

The results of the variant I designed to produce roughly equal sized clusters proved to be quiet satisfying. The resulting clusters are spatially cohesive and never violate the cardinality restrictions. The average squared distance between points does not seem to be minimized to the full extent, but with the size constraint it would be extremely difficult find the most optimal solution.

With that being said there sometimes exists strange behaviour, where a point seems to be in the incorrect location, there is more than likely an edge case that I have no thought of. These in-corrections are still relatively close in proximity to their assigned clusters and can easily be fixed a post-processing step. It seems to be a swap is occurring when it should not have, but at the end of the day, the clusters formed are nearly perfect.

The performance is fair, I have not written the formal math to prove the time complexity of my variant on the k-means algorithm, but it runs fairly well even with 5,000 points and 100 clusters.

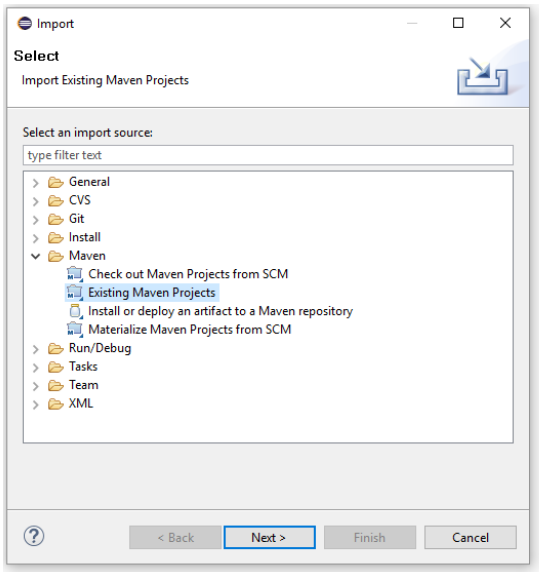
Possible Enhancements

I realized that because k-means is based on the computation of the mean and minimizing variance, it is designed to work with linear data; however, latitude longitude pairs are spherical in nature. Unfortunately, it did not dawn on me until I was almost completely finished implementing the k-means algorithm.

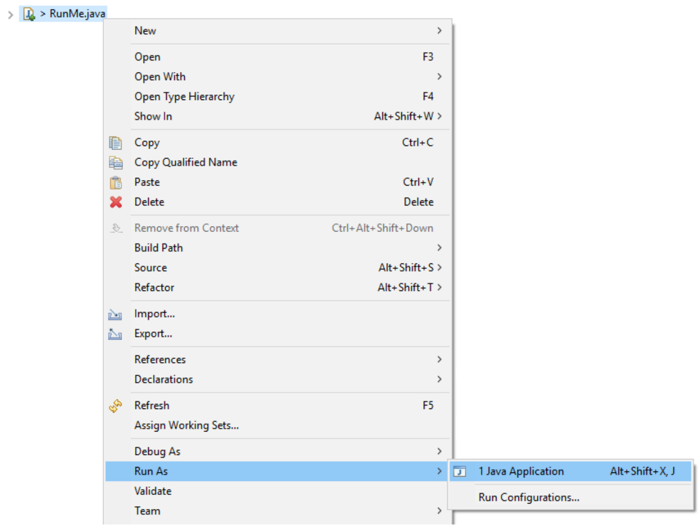
With that being said, a likely enhancement would be to choose a different algorithm that plays nicer with latitude longitude coordinates. These possible algorithms may include Hierarchical clustering, PAM, CLARA, or DBSCAN.

Running the Program

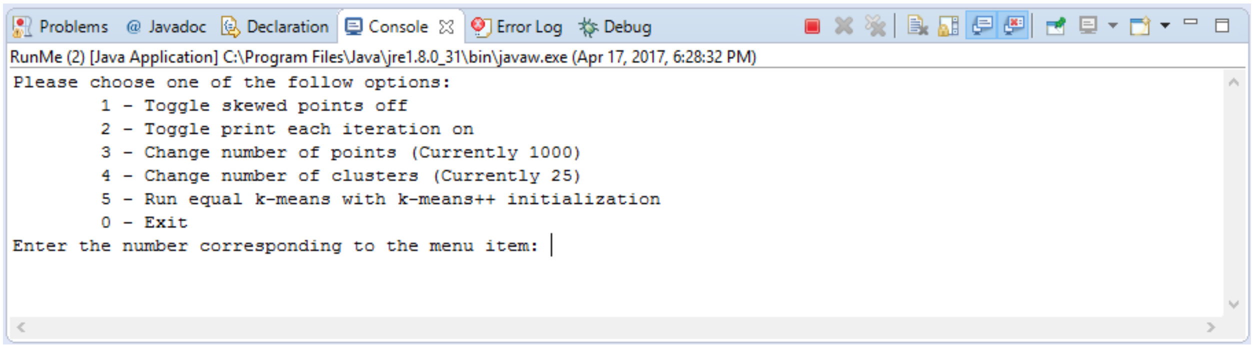
To begin, import the maven project into eclipse.


Import
Select
Import Existing Maven Projects
Select an import source:
type filter text
General
CVS
Git
Install
Maven
Check out Maven Projects from SCM
Existing Maven Projects
Install or deploy an artifact to a Maven repository
Materialize Maven Projects from SCM
Run/Debug
Tasks
Team
XML
< Back
Next >
Finish
Cancel

To run the program, right click on the **RunMe.java** class and select **Run As → Java Application**. This class can be found in the **Package Explorer view**.

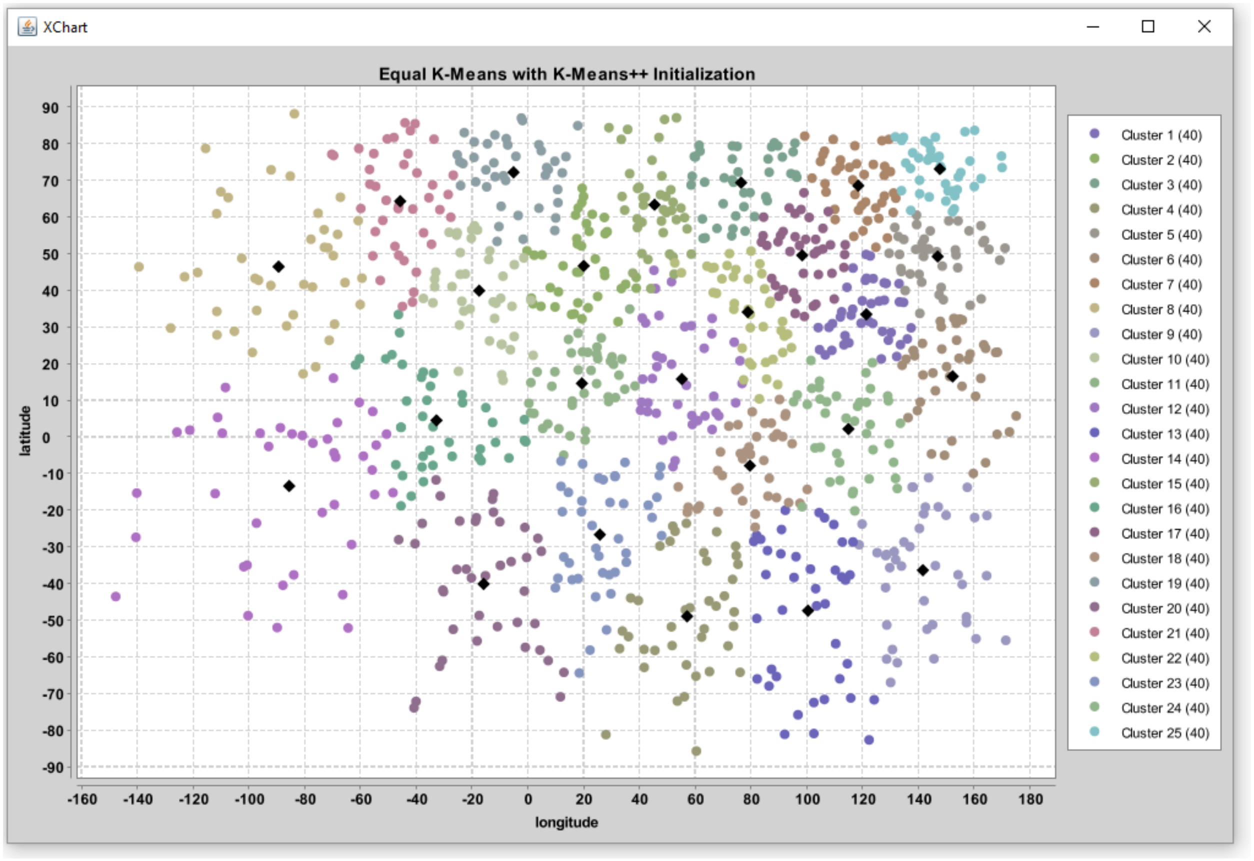


This will prompt the following options in the **Console view**.



```
RunMe (2) [Java Application] C:\Program Files\Java\jre1.8.0_31\bin\javaw.exe (Apr 17, 2017, 6:28:32 PM)
Please choose one of the follow options:
        1 - Toggle skewed points off
        2 - Toggle print each iteration on
        3 - Change number of points (Currently 1000)
        4 - Change number of clusters (Currently 25)
        5 - Run equal k-means with k-means++ initialization
        0 - Exit
Enter the number corresponding to the menu item: 
```

Choose option five to run the algorithm. This algorithm will be run with the provided number of points and clusters provided. The data will be randomized, there is a flag to skew the data which may be toggled. Once the algorithm is finished running the result will be displayed in a scatterplot.



References

Arthur, D., & Vassilvitskii, S. (n.d.). *k-means++: The Advantages of Careful Seeding.* Retrieved from http://ilpubs.stanford.edu:8090/778/1/2006-13.pdf

ELKI. (n.d.). *ELKI Data Mining*. Retrieved from Same-size k-Means Variation: https://elki-project.github.io/tutorial/same-size_k_means

Piech, C. (n.d.). *K Means*. Retrieved from Stanford: http://stanford.edu/~cpiech/cs221/handouts/kmeans.html

pierredavidbelanger. (n.d.). *A Java K-means Clustering implementation*. Retrieved from GitHub: https://github.com/pierredavidbelanger/ekmeans

rosettacode. (n.d.). *Haversine Formula*. Retrieved from rosettacode: https://rosettacode.org/wiki/Haversine_formula