Geo-location Clustering Using the k-means Algorithm

Tyler Thomas

December 15, 2021

Contents

1 Introduction

1.1 What is k-means?

The goal of k-means clustering is to divide the input data set into k clusters of similarity. An element is assigned to the cluster which has the closest mean, or centroid, to the element. When an element is added to a cluster, the centroid is shifted to account for the bias of that element.

After every element has been assigned a cluster, the assign-shift process can occur again using the newly generated centroids. This process is iterative and can be ran until a satisfactory level of convergence has been reached.

Clustering disparate data is useful because the means and associations of elements with other elements in their clusters may provide insights that were not immediately clear from the raw data. It also provides a way to associate elements as "similar" to other elements.

1.2 Geo-location Data

The datasets provided for this project provide latitude/longitude pairs representing a point on the earth. The initial datasets are non-standardized and need sanitization in order to be used.

1.3 Considerations

Before implementing a k-means clustering algorithm the following problems must be considered:

Distance Measure

Problem: In order to assign an element to a cluster, it must be possible to identify which centroid it is closest to. This requires the definition of a function capable of numerically quantifying the distance between two elements.

Solution: The required distance measure algorithms were provided, Euclidean and GreatCircle. The GreatCircle distance formula takes the latitude and longitudes directly as input. However, Euclidean distance is calculated using Cartesian coordinates. Therefore, the lat/lon pairs must be converted to their Cartesian x/y/z representations before being passed to the formula.

Convergence Distance

Problem: How will the algorithm know when to stop iterating? One approach is to use the magnitude of the centroid deltas after each iteration. How is this quantity selected?

Solution: To determine when convergence has been accomplished, my approach takes the maximum change in distance of every centroid after each iteration. If every centroid's delta is less than 10 km, the iteration stops. 10 km was chosen somewhat arbitrarily as it is an insignificant distance at the scale of the world, but sufficiently large that the program does not run too long. This convergence threshold led to an average number of iterations of 8-12 for k=6 on the large data-set (DBPedia).

Shifting Centroids

Problem: After adding an element to a cluster, the centroid must be shifted to account for the new element. What operation will be used to generate this shift?

Solution: After adding a point to a cluster, the new centroid is the "sum" of every point in the cluster. In this case, "adding" two points together is taking the midpoint of the two. This is implemented as a map-reduce construct. Again, in order to do an operation on a Cartesian domain, we must convert the lat/lon pairs to x/y/z before taking the midpoint.

Centroid Initialization

Problem: The running state of the algorithm assumes that the centroids have already been initialized. How should points be selected as initial centroids?

Solution: I use the approach from the text to initialize the centroids. The first centroid is selected from the data-set at random. Then, subsequent centroids are selected by picking the point that is as far as possible from the already selected centroids. This is done by calculating the distance from each point to its closest centroid, and selecting the point with the largest distance.

2 Data sanitization

Three data-sets were provided with latitude longitude values. The three datasets will be referred to as small, medium, and large. Small consists of the data from the devicestatus.txt (device data) file containing around 10k locations from the west coast of the United States. Medium is from sample_geo.txt (synthetic data) and contains around 100k locations across the United States. Finally, large is from lat_longs.txt (DBPedia data) and contains around 500k locations from across the globe.

Each dataset was converted to the standardized format of a CSV with 2 columns, latitude and longitude. Before writing to the CSV, null values were filtered to prevent the need for sanity checks later in the clustering implementation.

Additionally, the large dataset required the extra step of determining the delimiter from a non-standardized format. To do this, the character at the first known delimiter index was extract and the python split function was used to separated the columns by the parsed delimiter.

The additional fields that were required to be parsed are shown in devicedata.png located in the GitHub repository.

3 Implementation

My implementation persists 1 pyspark RDD across the entirety of execution. This one RDD is the primary data structure operations occur on. From the highest level, my implementation is extremely simple. Algorithm 1 shows the high level logic of my approach.

The centroids are initialized using the algorithm shown in Algorithm 2. The clustering loop iterates while the delta is greater than the convergence distance threshold of 10km. Each iteration, there is a map-reduce operation where the points are mapped to the centroids they are contained within (assignment), then they are "added" (shift) to get the new centroid for each cluster. This requires only one collect call to occur per iteration. Collect is called to realize the new set of centroids (stored as a Python list). The new maximum delta is then calculated by computing the distance from each new centroid to its old location. The highest change in distance from one iteration is used as the delta.

Algorithm 2 shows the psuedo code implementation of the centroid initialization function. The first centroid is selected at random. The remaining points are selected by picking points as far as possible from the already selected centroids. The points RDD is used to map every point to its closest centroid. Then every point is mapped to the distance to the closest centroid. The point with the maximum distance from its closest centroid is then added to the list of initial centroids.

Algorithm 1 High Level Logic

```
points ← lat_long.asRDD()
if cache then
    points.cache()
end if
centoids ← initCentroids(k, points)                          ▷ [point1, ... pointk]
convDist ← 10km
delta = convDist + 1
assignedPoints ← null
while delta > convDist do
    assignedPoints ← points.map(p → assignPoint(p, centroids))   ▷ (centroid_index, point)
    temp ← assignedPoints.reduceByKey(addPoints)                 ▷ (centroid_index, point)
    new_centroids ← temp.sortByKey().map(p → p.point).collect()  ▷ [point1, ... pointk ]
    delta ← getHighestDelta(centroids, new_centroids)
end while
```

Algorithm 2 Centroid Initialization(k, points)

```
c ← emptyList()
c.append(points.takeSample(1))
for i ∈ range(k − 1) do
    close ← points.map(p → closestPoint(p, c), point)            ▷ (closest, point)
    distances ← closest.map(p → dist(p.closest, p.point), p.point)   ▷ (distance, point)
    new_centroid ← distances.max().point
    c.append(new_centroid)
end for
return c
```

4 Runtime Analysis

4.1 Speed

Figure 1 shows the differences in average runtimes (k=5) for each dataset when the points RDD is cached or not cached. There is a notable decrease in runtime when the data is persisted. Note that these times include the time spent to plot the figures as well.

It is interesting to note that the medium dataset consistently finished faster than the small dataset. I originally hypothesized this was because maybe there were less iterations required to reach convergence in this dataset. This was not the case. The average number of iterations to process each dataset increased linearly with both the size of the dataset as well as the size of the geographic area the locations were spread over. I cannot come up with any other theories explaining why the medium dataset was so much faster on average than the small dataset. However, the medium dataset had the largest percentage speed improvement when cached. The percentage runtime savings for caching across the datasets as well as the number of iterations for each dataset can be seen in Table 1

Dataset	Average Iterations	Percent Improvement w/ Cache
Small	3	2.74%
Medium	5	20.68%
Large	9	15.45%

Table 1: Performance for Each Dataset

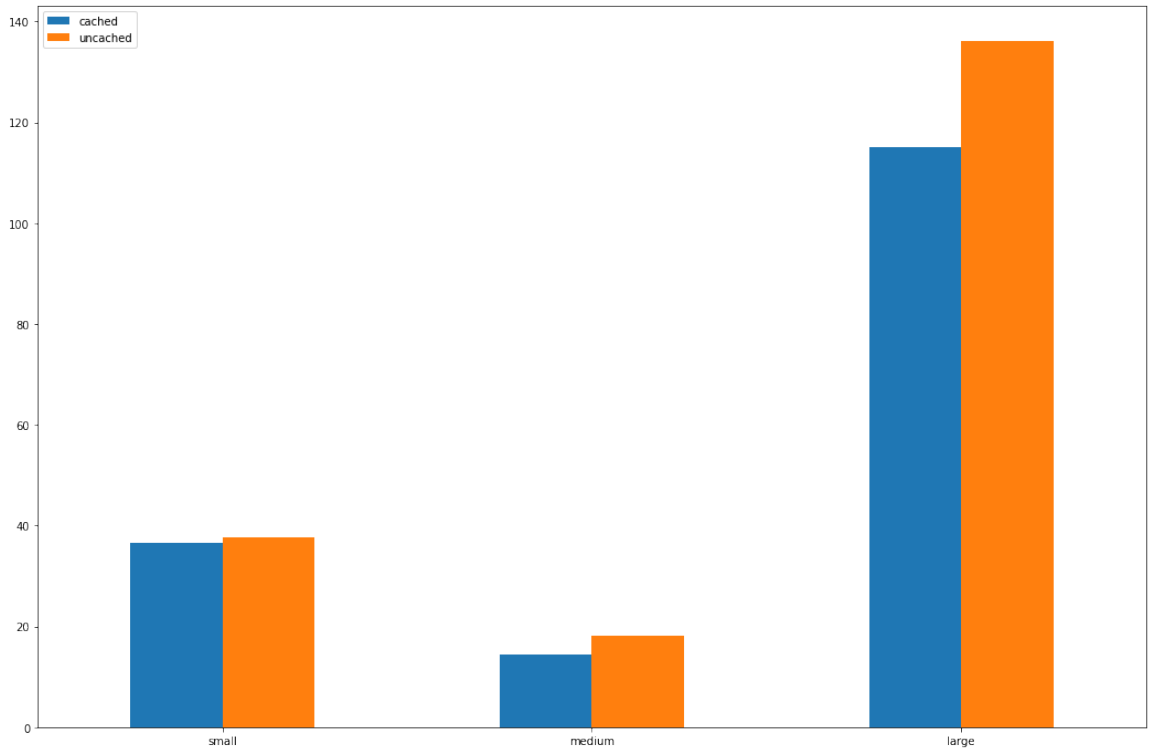

Figure 1: Cached vs Uncached runtimes (seconds) for each dataset, k=5

4.2 Quality

4.2.1 Device Data

Figures 2 and 3 show the output from the small dataset with k=5 using Euclidean and GiantCircle, respectively. They appear to be very similar. This is likely because locations are so close that the inaccuracy of the Euclidean calculations are insignificant. The GreatCircle one looks slightly better to me. Both are able to identify the population centers of Los Angeles and San Francisco.

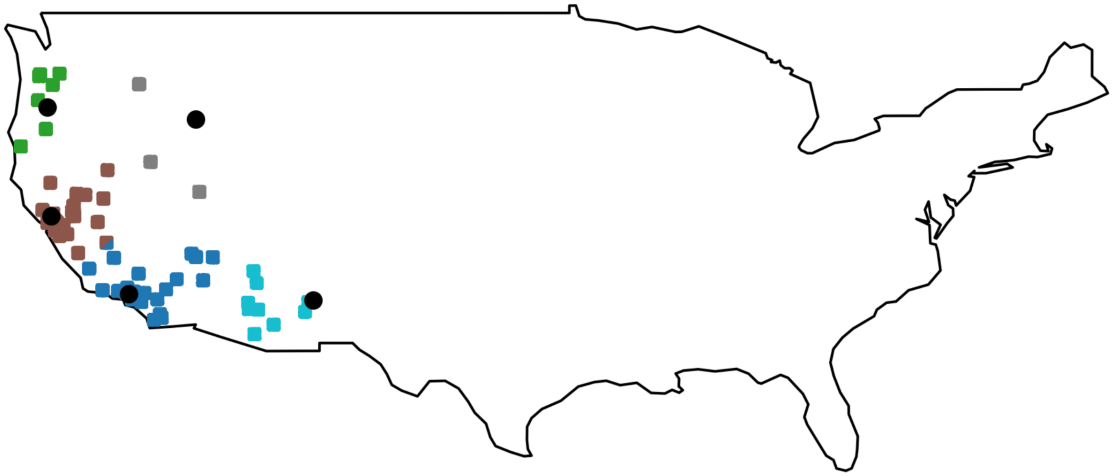

Figure 2: Small dataset, Euclidean, k=5

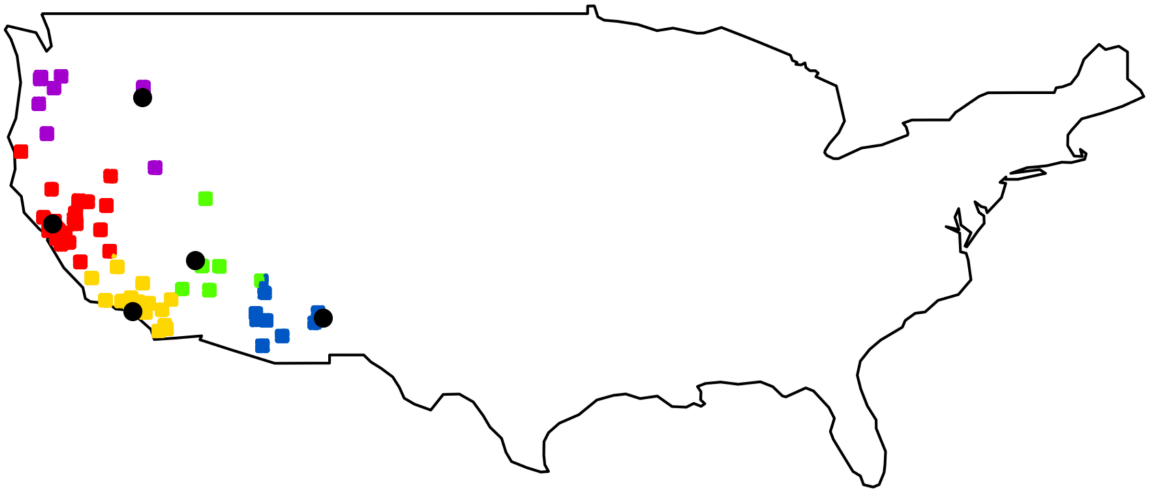

Figure 3: Small dataset, GiantCircle, k=5

4.2.2 Synthetic Data

Figures 4 and 5 show the output from the medium dataset with k=2 using Euclidean and GiantCircle, respectively. Again, they appear to be the same. Although this time they are even closer. This is likely because the east and west coast of the US are two extremely obvious clusters. I had to rerun the analysis several times to ensure these were in fact using the correct parameters.

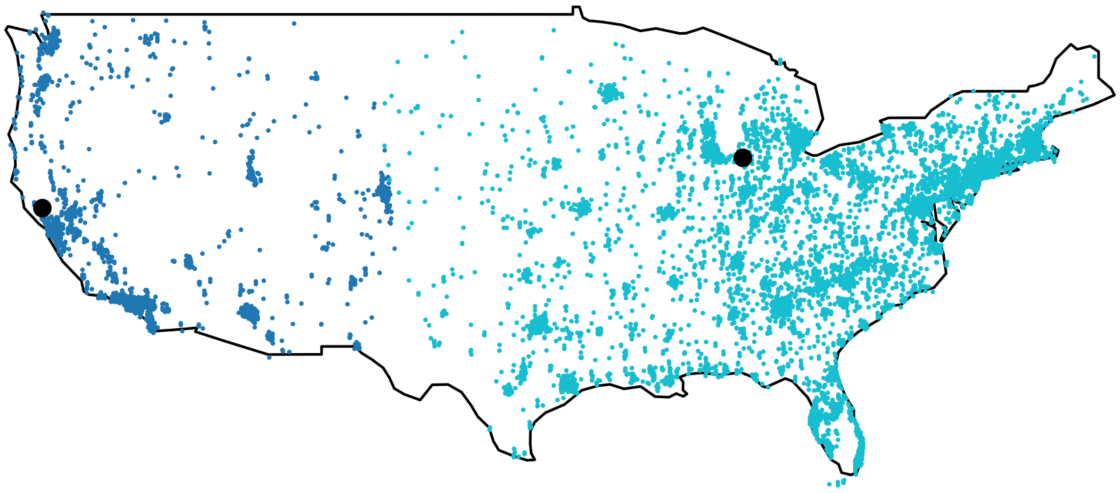

Figure 4: Medium dataset, Euclidean, k=2

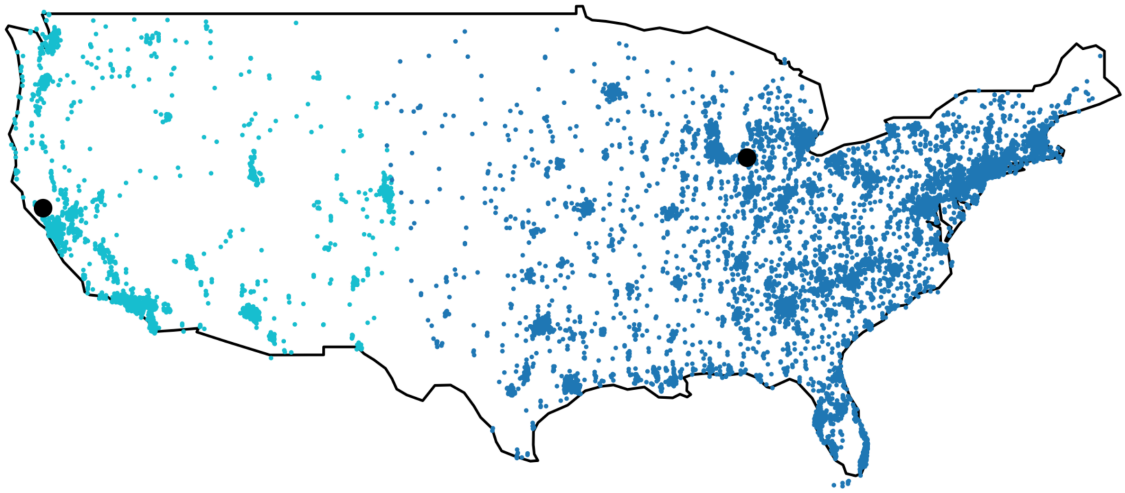

Figure 5: Medium dataset, GiantCircle, k=2

Figures 6 and 7 show the output from the medium dataset with k=4 using Euclidean and GiantCircle, respectively. GreatCircle looks significantly better here. GreatCircle accurately distinguishes the Deep South from the north east and Appalachia. While Euclidean splits the northeast in half and includes parts of Texas with Cape Cod.

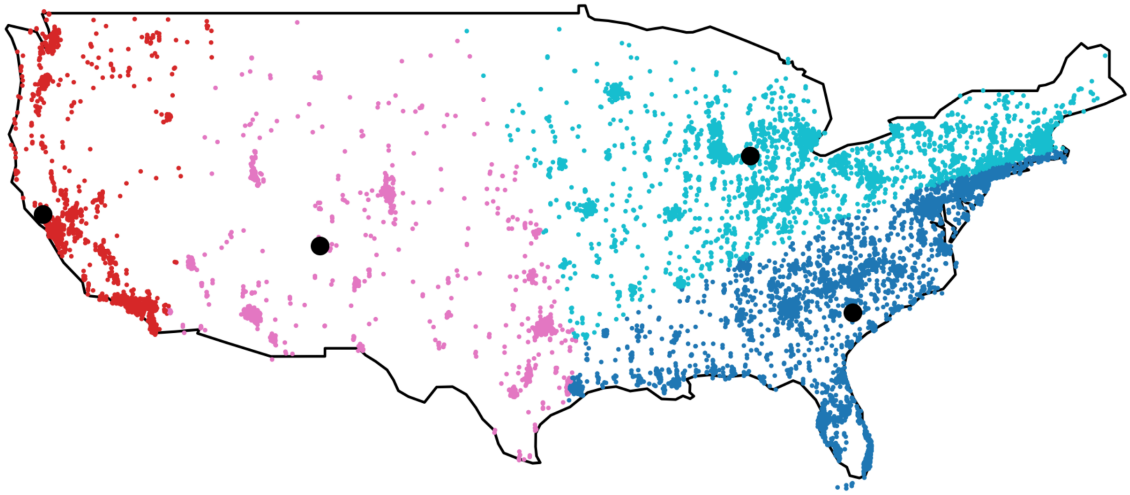

Figure 6: Medium dataset, Euclidean, k=4

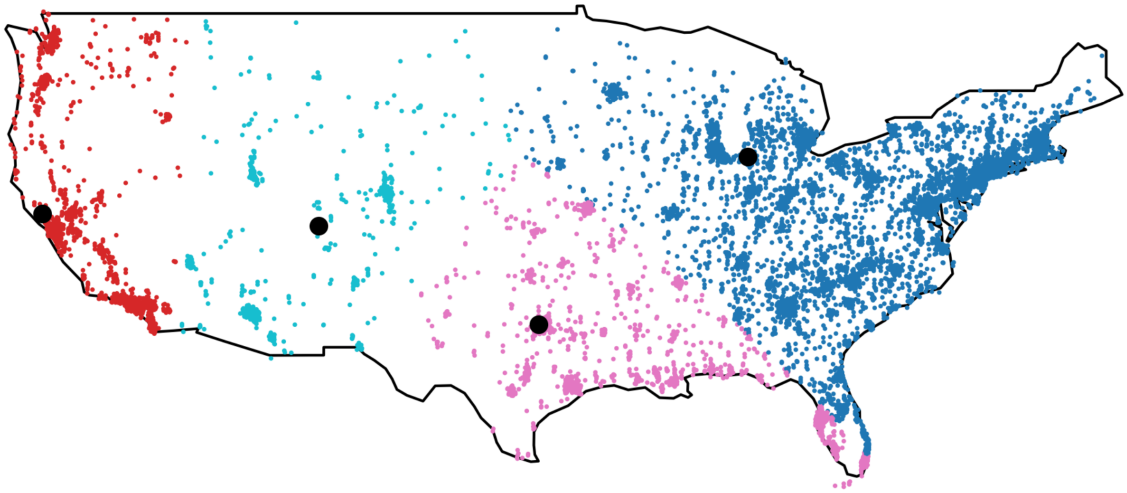

Figure 7: Medium dataset, GiantCircle, k=4

4.2.3 DBPedia Data

Figures 8 and 9 show the output from the large dataset with k=6 using Euclidean and GiantCircle, respectively. The edges between clusters in the Euclidean output are very sharp and the Americas are broken into three clusters. Africa is also awkwardly split in half. The GiantCiricle output nearly perfectly illustrates North America, South America, Europe and North Africa, Sub-Saharan Africa, urban China and Oceania, and the Indian subcontinent. The clusters very clearly map onto population centers.

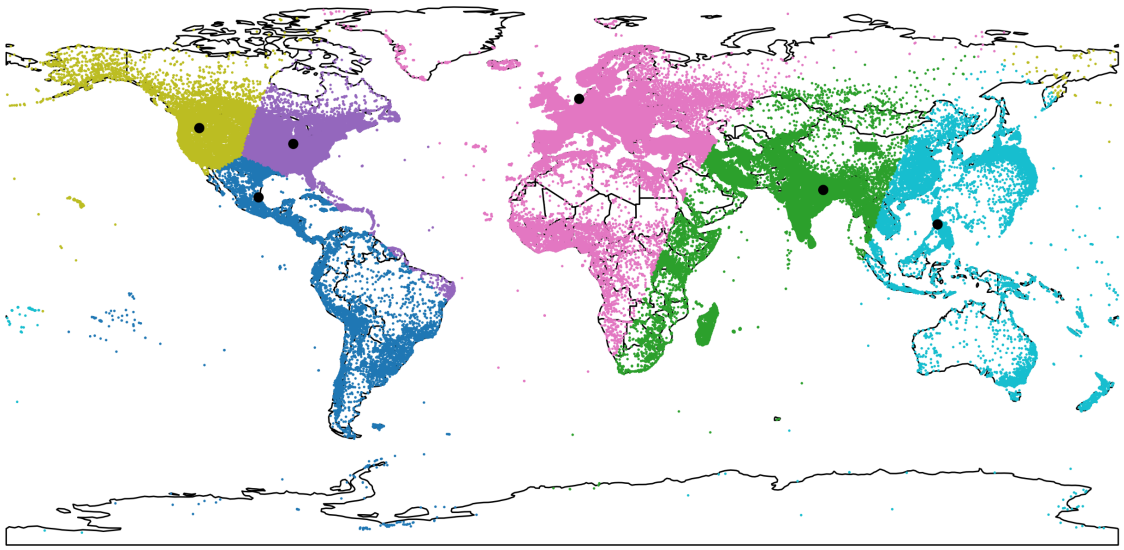

Figure 8: Large dataset, Euclidean, k=6

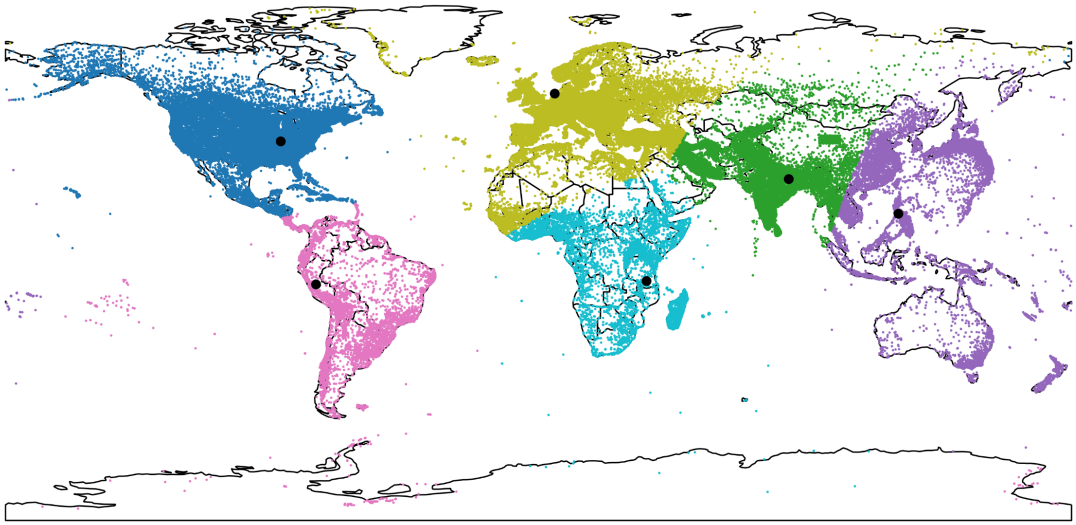

Figure 9: Large dataset, GiantCircle, k=6

Figures 10 and 11 show the output from the large dataset with k=8 using Euclidean and GiantCircle, respectively. Again, the Euclidean output is significantly worse than the GiantCircle output. GiantCircle identifies all the inhabited continents. GiantCircle additionally divides Africa between Sub-Saharan Africa and West Africa and Asia between the Indian subcontinent and urban China/Malaysia/Indonesia.

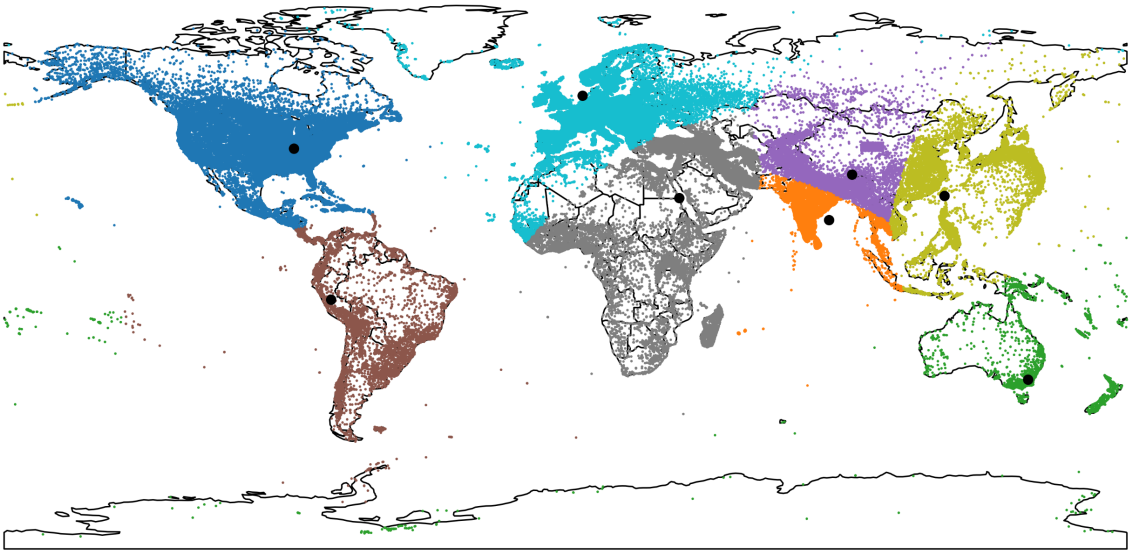

Figure 10: Large dataset, Euclidean, k=8

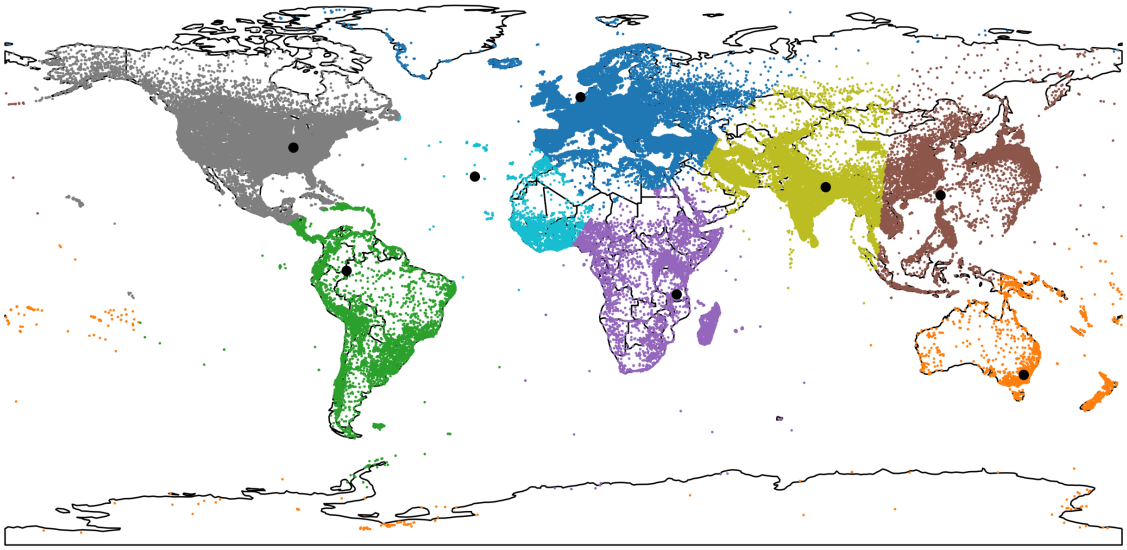

Figure 11: Large dataset, GiantCircle, k=8

5 Conclusion

The GiantCircle distance calculation appeared to give the best results that most accurately reflect my understanding of geography and world cultures. On data that spanned a smaller geographic distance, the error from the Euclidean calculation appeared to matter less, but still manifested itself through my testing.

The speed gained by caching the points RDD was significant and noticeable during testing.