CS5800 –Final Report
Keegan Mosley
Ryan Moore
Andrea Croak
April 26, 2023

Introduction (5 marks)

Question:

As a tourist, what is the minimum spanning tree between a large number of haunted destinations?

Scope:

Using k – clustering, we will cluster GPS coordinates of haunted sites across New England. We will then create edges of driving distance between the cluster centroids. With this graph, we will determine an MST between all clusters. This will give us the minimum distance required to link concentrations of haunted and historical sites. We will then find the MST of each individual cluster.

Context:

Keegan Mosley:

I do not believe in the supernatural in any way, but I do enjoy visiting locations that are supposedly haunted. The reason why a location is deemed haunted is usually because there is some kind of story involving the location & people who were there. For my undergraduate degree, I majored in American History. I chose this major because I found exploring the perspectives, cultures, and lives of those in the past to be interesting. In many ways people living a few hundred years in the past were just like us, but they perceived the world around them in radically different ways. As someone who grew up just outside Boston, I had numerous historical sites within walking distance of my house when I was young. Seeing these very old sites, contrasted with the modern buildings surrounding them, had a very important part in sparking my interest in the past. Learning about these buildings and the stories of those who lived in them ignited my imagination. It scratched the same itch as traveling to a place I'd never been before.

Haunted locations create opportunities for the general public to hear real stories about those who lived in the past. Most Americans perceive history as the series of dates they had to memorize in high school. I think this makes many people miss out on the joys of studying the past. Large trends are interesting in their own right, but focusing on the stories of individuals can bring about an overwhelming sense of humanity. For instance, there is something special about reading a handwritten private letter from one person to another, both of whom have been dead for hundreds of years. So many feelings that are expressed in these primary sources are the same that we feel in our lives - joy, sadness, love, grief, passion, ect. We can see stories of the great kindness that humans show each other as social animals, and inversely the great pain that we can inflict upon one another. These complicated, messy, human stories are what makes studying history so fascinating, just as reading a book with heavily flawed characters can be. Historical haunted sites offer an opportunity to tell a short story of what happened here. Since these tales are told as stories, they can have an easier time capturing the imaginations of the general public. This is supported by the fact that an entire tourism industry has sprung up around touring haunted sites. These sites provide an opportunity to trojan-horse historical interest to those who may not have had a previous experience where learning about the past is fun.

Ryan Moore:

My parents owned a flower shop for 40 years and many of my afternoons as a child were spent riding around with the delivery drivers and "helping" them. When we visited cemeteries to install flowers on gravestones and headstones, I did not see sad or scary locations, I saw something like a park with trees, flowers, and birds. Somewhere you could be at rest. As a result, I grew up with a rosier relationship with these locations than most young children. Similarly, my mom and I would spend every Sunday morning delivering funeral work. Being a natural helper, I often held doors and carried small arrangements. My parents had deep relationships with the funeral directors in our town so I was naturally comfortable asking them the questions that would come up. I would respectfully ask to see the deceased and would ask the funeral director a series of questions "How did they die? Did they have a family?" These early experiences have given me a familiarity with death, dying, and the traditions that encompass those experiences. I am comfortable in cemeteries, funeral homes, and legitimately haunted locations. Oddly enough, I hate the fake haunted houses that pop up around Halloween. I care less about being scared and more about experiencing legitimate paranormal activities.

Andrea Croak:

Having spent the early half of my childhood growing up in Arlington MA my mother was always looking for things to do out of the house with us, particularly activities that got us out into nature. Being an incredibly suburban area made this a challenge. The answer became the family visiting historical sites and cemeteries. Not many folks know that cemeteries planned during the Victorian era and after looked more towards being park-scapes, working to be inviting towards the living who wished to visit their loved ones. Despite this fact most people today are disquieted visiting these greenspaces that while being reserved for the dead are full of nature and life, and instead feel a sense of haunting. Many historical sites like to add flavor to their tours with examples of paranormal experiences over the centuries. These early exposures have led to a lifelong enjoyment of skepticism about the paranormal, psychology, and history.

- **Analysis (25 marks)**

How we gathered our Data:

From the start of the project, we were focused on the New England area for destinations (data points) to add to our map. The group's first steps were to divide up the states of New England and embark on some quality research time searching for interesting haunted historical sites and stories. These points of interest range from haunted inns and taverns to disquieting stretches of woodlands haunted by ghosts of witches to shuttered and abandoned asylums left to rot into oblivion. Since New England has a wealth of history each state provided excellent results and interestingly aligned with the more populous areas of each state. We utilized a shared spreadsheet to list the names, addresses, and GPS coordinate (latitude and longitude) of each of these sites.

These coordinates are the basis for our mapping, minimum spanning tree, and clustering programs. Upon our next meeting, we realized that in a short span of time we had amassed over 100 points of interest. Since we were interested in a road trip theme, we cut this list back by removing destinations that were not easily accessible by road or foot, and destinations with vague locations. This left 102 coordinates to process, which has been included in the appendix of this paper alongside our programs.

What did we do and why?

We continued our research by looking into Kruskal's Minimum Spanning Tree algorithm, K-Means Clustering, using the elbow method to determine an optimal k, data visualization techniques, and accounting for the Earth's curvature. Modules 7.3: Prims and Kruskal's and 7.4: Clustering via MST provided a solid foundation. From there we divided and conquered what needed to be accomplished. We made a point to communicate via text and email about our individual progress and met either before class or virtually to plan and discuss the project. Being scattered between different locations, GitHub has been a huge boon to our process.

First, we ran Kruskal's algorithm to give us an MST between all data points. Next, we created a k-clustering algorithm in order to group sites into clusters. This required creating methods to determine the distance between GPS coordinates, and to find the centroid of a set of GPS coordinates. With this algorithm created, the next step was to determine what k value to use for our k-clustering. To accomplish this we used the "elbow" method, and settled on k = 6. Next, we clustered the data and used Kruskal's algorithm to find the minimum spanning tree of the cluster centroids. This produced a path that we could consider for a road trip.

We then considered what would be the MST of each individual cluster. With our clusters already formed, we were once again able to apply Kruskal's algorithm to each cluster. We could then compare the MST of each cluster, to see alternative potential road trip paths.

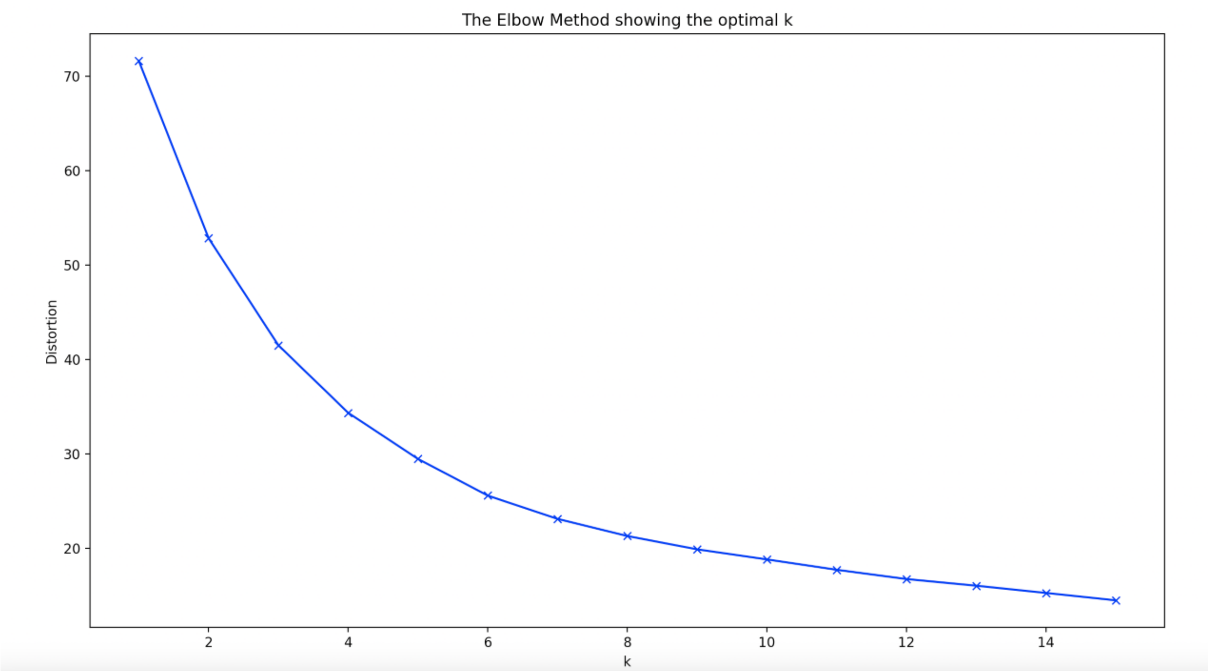
Process

We begin with the kruskal_mst.py program, it takes in the latitude and longitude data for all our points using the Pandas library to grab the necessary column data and iterate through it. From our 102 points a substantial MST list is produced.

8 - 9: 0.001624 35 - 36: 0.006275 70 - 71: 0.010002 9 - 10: 0.010086 83 - 87: 0.013149 40 - 41: 0.014735 84 - 85: 0.014809 51 - 54: 0.018430 26 - 66: 0.021076 84 - 86: 0.023216 36 - 38: 0.024958 69 - 71: 0.025287 37 - 38: 0.027216 3 - 11: 0.027259 31 - 32: 0.029761 34 - 37: 0.032717 87 - 89: 0.056426 30 - 39: 0.060222 39 - 40: 0.061531 45 - 57: 0.066273 92 - 94: 0.070863 89 - 90: 0.073532 0 - 73: 0.078599 81 - 82: 0.081204 77 - 82: 0.081390 17 - 23: 0.089695 2 - 42: 0.090102 29 - 31: 0.114436 46 - 49: 0.117542 88 - 89: 0.117755 63 - 72: 0.118728 48 - 53: 0.128617 83 - 84: 0.134863 78 - 85: 0.138162	50 - 57: 0.138326 24 - 25: 0.139861 53 - 54: 0.140211 20 - 61: 0.142232 80 - 93: 0.142631 2 - 7: 0.146819 6 - 14: 0.148951 91 - 98: 0.150699 65 - 66: 0.153325 46 - 48: 0.154154 5 - 6: 0.158536 13 - 14: 0.159586 51 - 55: 0.162692 23 - 60: 0.163051 75 - 79: 0.164375 38 - 42: 0.170085 0 - 75: 0.170527 29 - 41: 0.171670 24 - 60: 0.173108 70 - 78: 0.188323 17 - 18: 0.188373 21 - 60: 0.188513 28 - 33: 0.189274 32 - 34: 0.189759 71 - 74: 0.199773 29 - 33: 0.203676 67 - 77: 0.206065 11 - 98: 0.206137 8 - 13: 0.217449 91 - 101: 0.229179 7 - 12: 0.242450 20 - 65: 0.264667 33 - 44: 0.265285 4 - 76: 0.268194	47 - 50: 0.268689 94 - 100: 0.277533 91 - 99: 0.282334 67 - 74: 0.301440 1 - 96: 0.301800 6 - 90: 0.302377 4 - 16: 0.305643 16 - 28: 0.315823 43 - 46: 0.333629 52 - 53: 0.338350 12 - 13: 0.347179 93 - 97: 0.352289 27 - 34: 0.354384 49 - 56: 0.354534 10 - 79: 0.401535 44 - 50: 0.409689 47 - 56: 0.425477 22 - 64: 0.446416 97 - 99: 0.459634 68 - 72: 0.463599 59 - 64: 0.466030 23 - 62: 0.476076 72 - 77: 0.478306 59 - 62: 0.483552 1 - 95: 0.483789 25 - 63: 0.503874 15 - 16: 0.517962 92 - 99: 0.519378 92 - 96: 0.536722 61 - 62: 0.570384 26 - 58: 0.604058 76 - 97: 0.626891 19 - 58: 1.462123

Next, we implemented a version of K-means clustering. This algorithm initialized k cluster centroids as random coordinates from our dataset. From there, all datapoints were assigned to the cluster of the closest centroid. Once all datapoints had been assigned to clusters, new centroids of every cluster were found via the geographic midpoint of all cluster coordinates. The algorithm then repeats, assigning datapoints to the closest centroid. This continually repeats until there has been no change in the cluster centroids between iterations.

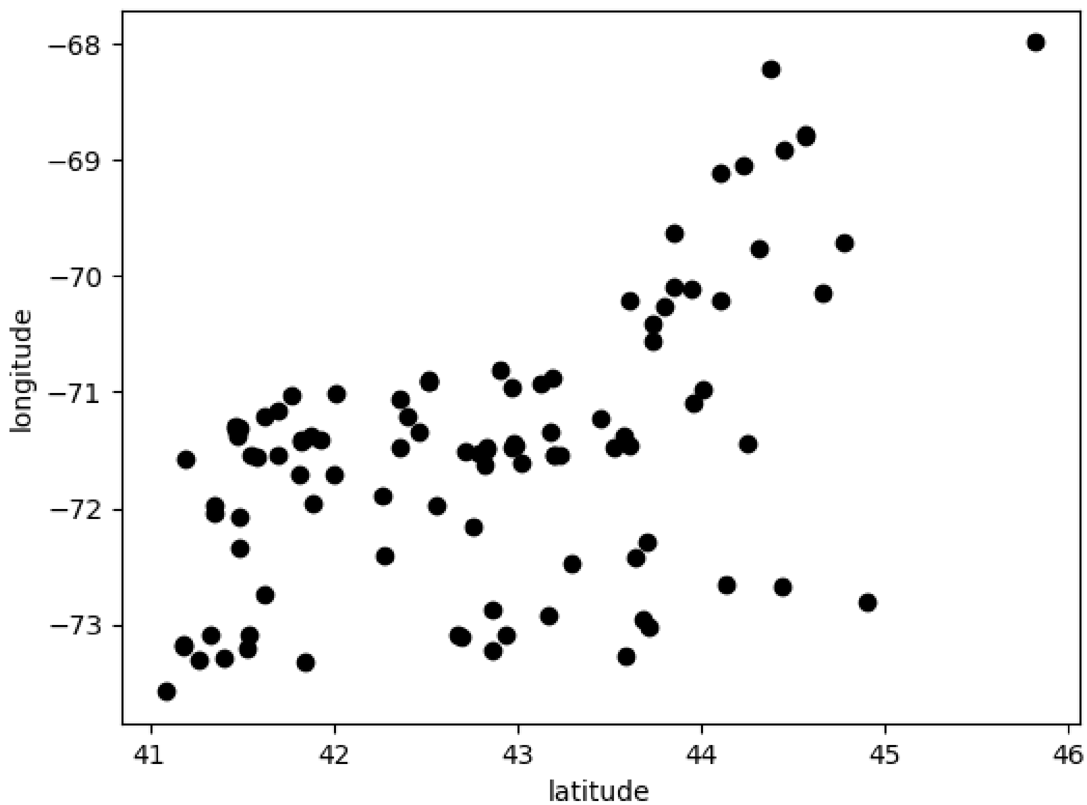
Next, we created a line graph in order to determine an optimal k. K-means clustering was ran 100 times, across a range of k values. The distortion (average difference between data points and their centroids) was found for each k iteration, and the average of the 100 distortions was graphed on the y axis for each k value. This allowed us to examine how the average intra-cluster variation changed with different k values. The k value from which the rest of the graph descends from in a linear-like fashion is the elbow point. This elbow point is an optimal k. We ran the algorithm 100 times since the clusters, and therefore the distortion, could vary depending on which datapoints were selected to be the initial centroids.



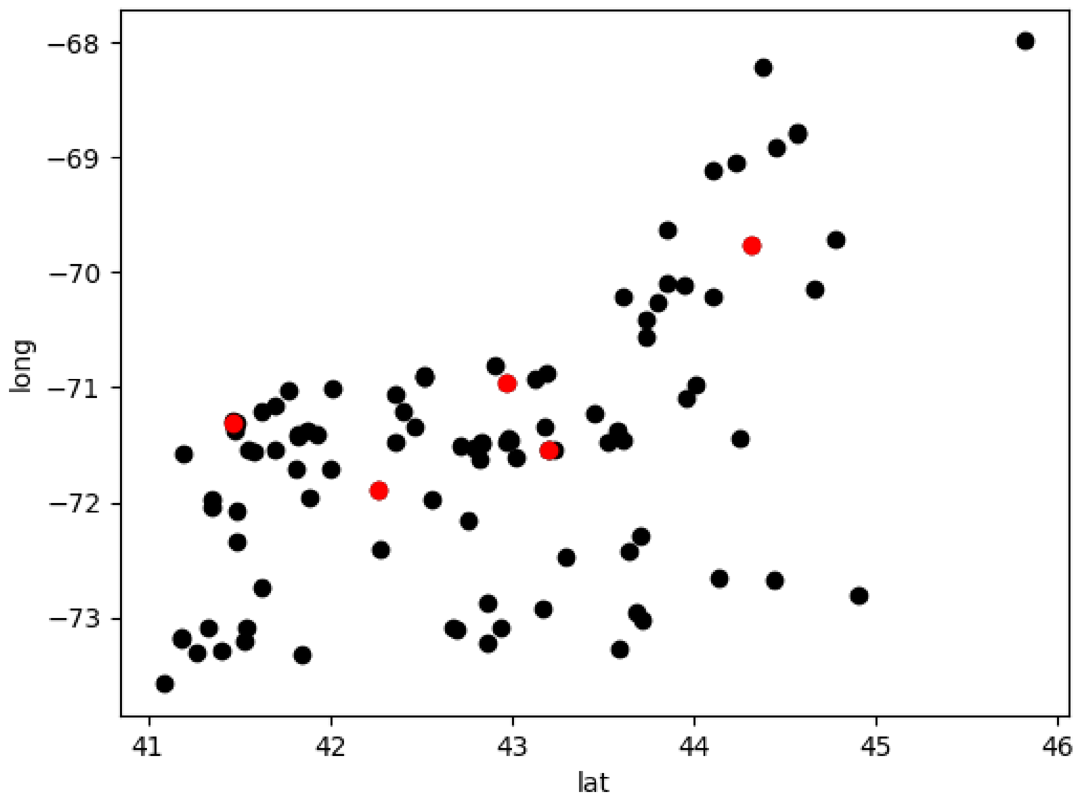
The group met briefly to confer on what the graph results depicted and decided that based on the results the optimal k value was 6. This was a partially expected result as we have 6 states to gather data from, clustered primarily around dense population centers.

After reviewing several articles and tutorials online about the K-Means Clustering algorithm we continued our development. For testing purposes, we continued to utilize the matplot library to get some quick confirmations that our algorithm was processing properly. If you look at the code, we have left in some commented out methods for printing specific evolutions of the graph such as the initial plotting of our points, the first iteration of our centroids being applied to the graph and finally the results of the machine learning algorithm (locally optimal centroids). These graphs are included below.

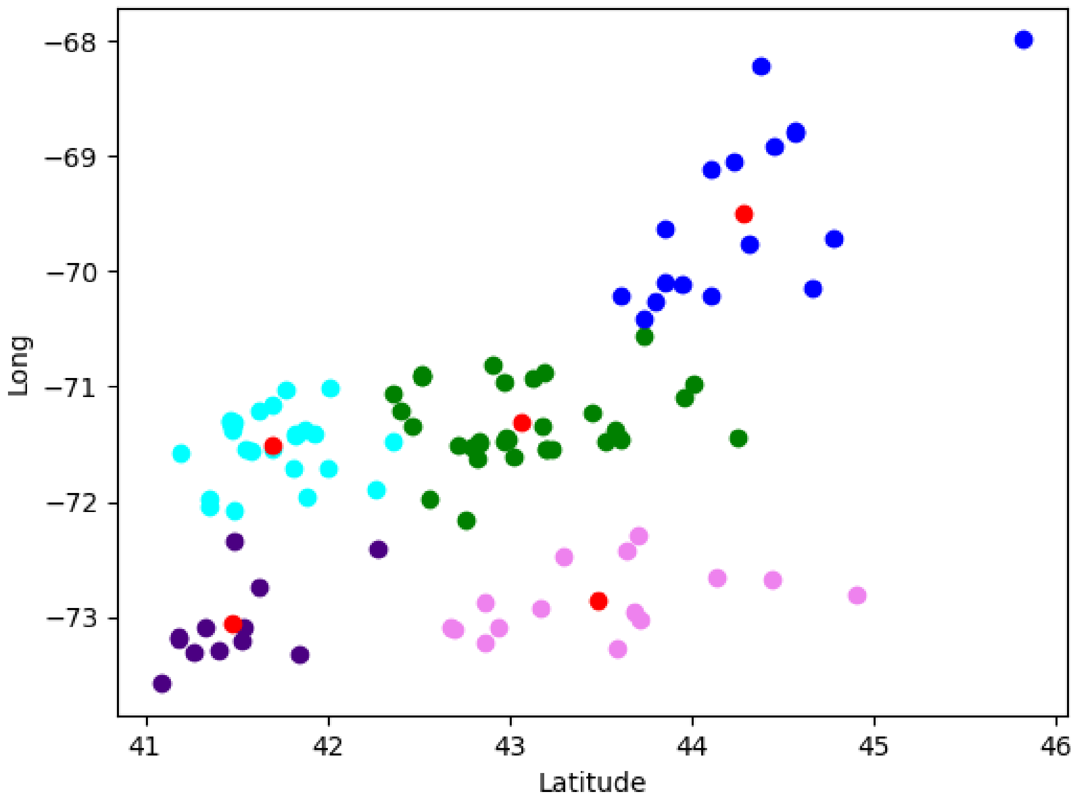
The Initial Scatter Plot of our Coordinates



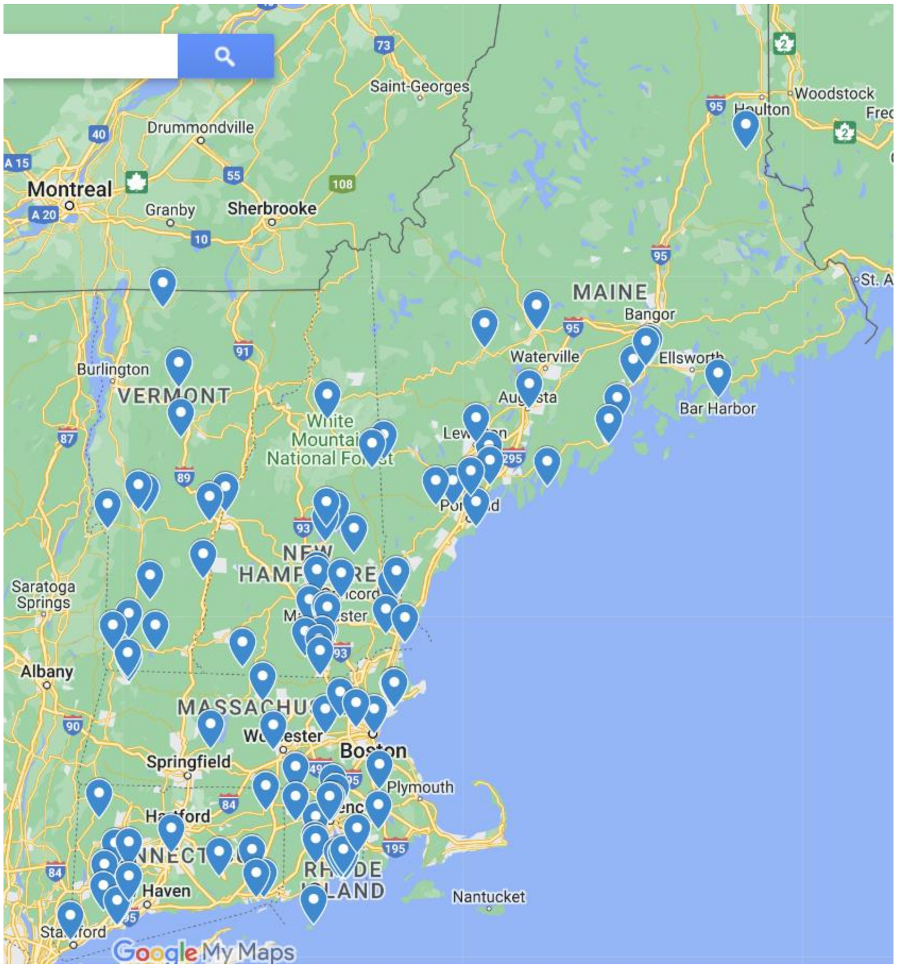
Adding 5 randomly generated centroids



Scatter plot with clusters



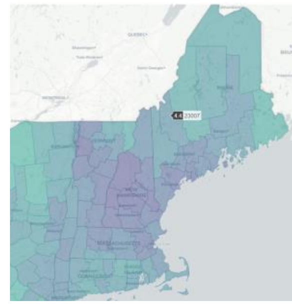
When comparing the scatter plot to a map with the coordinates laid out the outlier coordinates pop out. Such as the top right most blue point corresponding to Haynesville, ME. Or the bottom left most purple coordinate being Stamford, CT. It is also interesting looking at the scatterplot and physical map and seeing that the data only vaguely abstracted the shape of New England.



Once these methods were complete there was a shift in focus towards accounting for the curved surface of our planet, which affects the distance between points. While the Euclidian distance between coordinates plotted on an x – y plane (Latitude & Longitude) can be useful for seeing some relationships between coordinate points, it is a flawed system. Coordinates exist on a sphere. They are a distance along the curve of the Earth North or South of the Equator (latitude), and a distance along the curve of the Earth East or West of the Prime Meridian (longitude). At first, we were clustering our data simply based off of the Euclidean distance between points, where the latitude was the x coordinate and the longitude was the y coordinate. We decided to use a more accurate method of calculating the difference between points by creating the Coordinate and Point (x,y,z) classes. These work in conjunction to allow for the distance along the curve of the Earth between coordinates to be found. They also allow for the centroid of a set of coordinates to be found. By creating Coordinate and Point versions of our dataset, we were able to apply K-means clustering more accurately.

The distance between two Coordinates is found by taking a circle that is the circumference of the Earth, and shifting it by an angle until both coordinate points lie on the circle. Then, the distance between these two points on the circle can be found. The geographic midpoint of a set of Coordinates was found by first converting all Coordinates to (x, y, z) Points.

The origin of these points is the center of the Earth, and the x-y plane is in line with the equator. The positive x line passes through the Prime Meridian, the positive y line passes through 90 degrees East, and the positive z line passes through the north pole. With these (x, y, z) points, an average point could be found. A line is made from the origin, through this average point. Where that line intersects with the surface of the earth is the geographic midpoint of the set.



When thinking about the final vision of this project, it was important to us that our results be displayed in an appealing and meaningful way. To do so required research and learning how to work with various python libraries to read, interpret, and display the results of the graphing algorithms we created. One such way was to create a choropleth map, showing the density of our dataset. As striking as these maps are, however, they would not ultimately make much sense given the scope of the project. Instead, we turned to a scatterplot map. Leveraging the Plotly library in Python, we were able to design a simple program to read and interpret the data returned by our clustering algorithms, to show how our data would be clustered. This part of the project took a fair amount of time given the lack of experience with this library. Once we had a working prototype, however, it was quite fun to see our work visualized.

Once the data had been clustered and the centroids of these clusters were found, Kruskal's algorithm could be applied to find a minimum spanning tree between the centroids. We decided to use the minimum driving distance between centroids as the edge weights. The resulting MST edges & centroids are displayed below.

Final Centroids:

```
Index,Lat,Long
0,41.64082480214136,-71.52807792820188
1,44.52529156786832,-69.1046717756104
2,42.770479899246176,-71.42672616888274
3,43.76125133368217,-70.78613143742854
4,41.39977097681344,-73.19884195086995
5,43.49072018565226,-72.85867763421005
```

MST Edges:

```
0 - 2: 100.000000
2 - 3: 106.000000
0 - 4: 108.000000
2 - 5: 111.000000
1 - 3: 125.000000
```

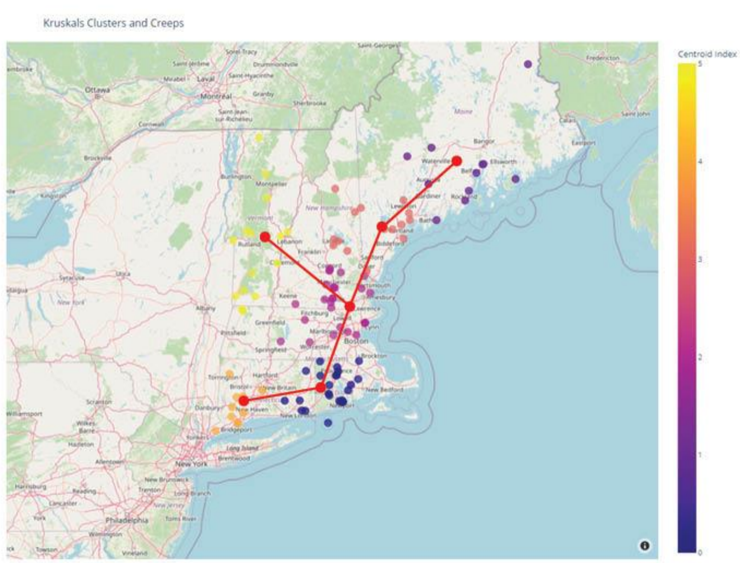
The individual cluster MSTs were then found with Kruskal's algorithm as well (shown in the conclusion). These produced alternative potential road trip paths, within each cluster.

• **Conclusion**

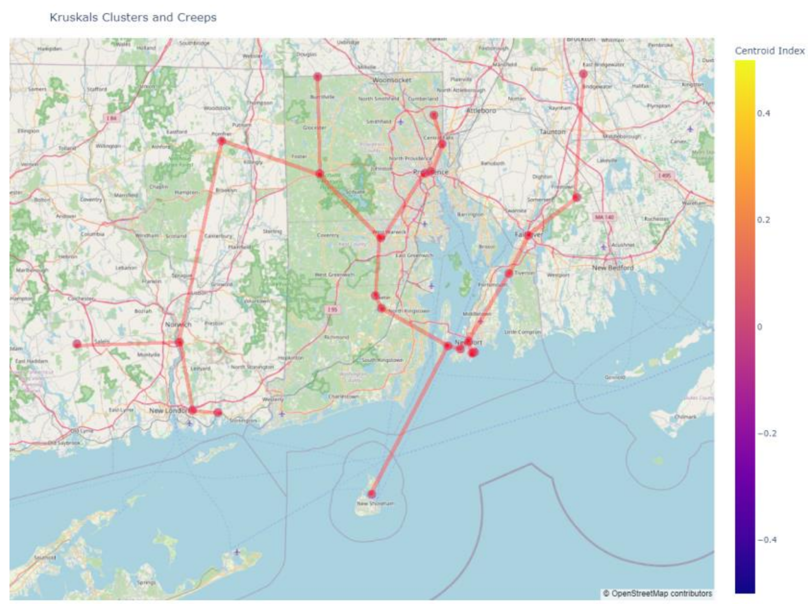
Overall Take Away –here we write our combined thoughts about the project

As a tourist, what are the minimum spanning trees between a large number of haunted destinations to visit?

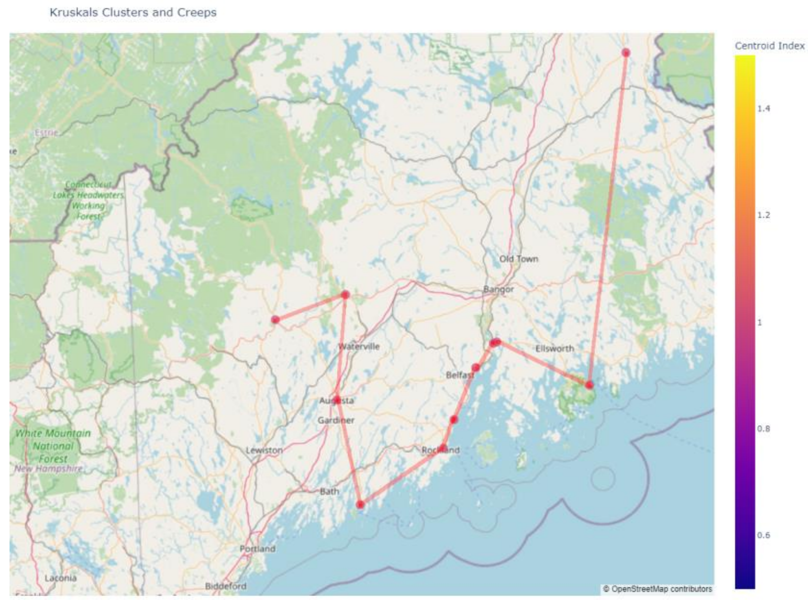
The following map displays the minimum spanning tree between all cluster centroids, which could be considered for a road trip (with some backtracking of course). The edges between cluster centroids were specifically weighted as driving distance. The centroid MST had an overall value of 550 miles.



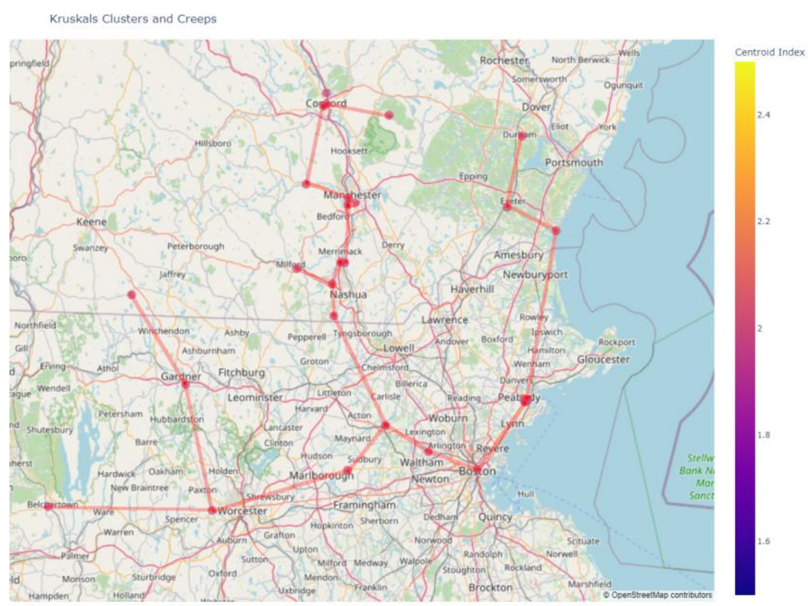
The below table depicts the minimum spanning tree of each cluster, where the difference between the latitudes and longitudes are the edge weights. By creating an MST of each cluster, we are finding the minimum weight to link all sites. Taking this data, you can create a road trip plan, or at the very least help depict what areas of New England are the most jam packed with destinations. Cluster 2 looks like an excellent candidate.

Cluster 0	
11 - 12: 0.006275 16 - 17: 0.014735 12 - 14: 0.024958 13 - 14: 0.027216 7 - 8: 0.029761 10 - 13: 0.032717 6 - 15: 0.060222 15 - 16: 0.061531 20 - 23: 0.066273 0 - 18: 0.090102 5 - 7: 0.114436 22 - 23: 0.138326 0 - 1: 0.146819 14 - 18: 0.170085 5 - 17: 0.171670 4 - 9: 0.189274 8 - 10: 0.189759 5 - 9: 0.203676 1 - 2: 0.242450 9 - 19: 0.265285 21 - 22: 0.268689 3 - 10: 0.354384 19 - 22: 0.409689	
Cluster 1	

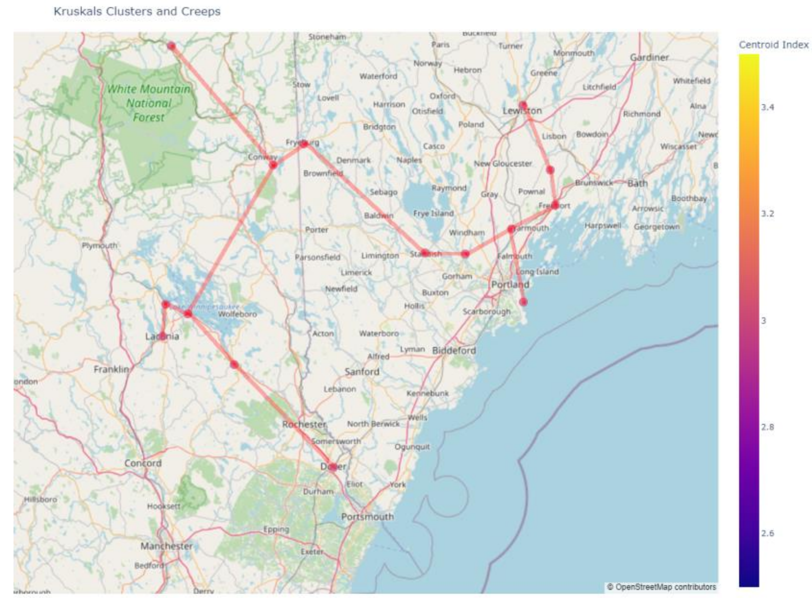
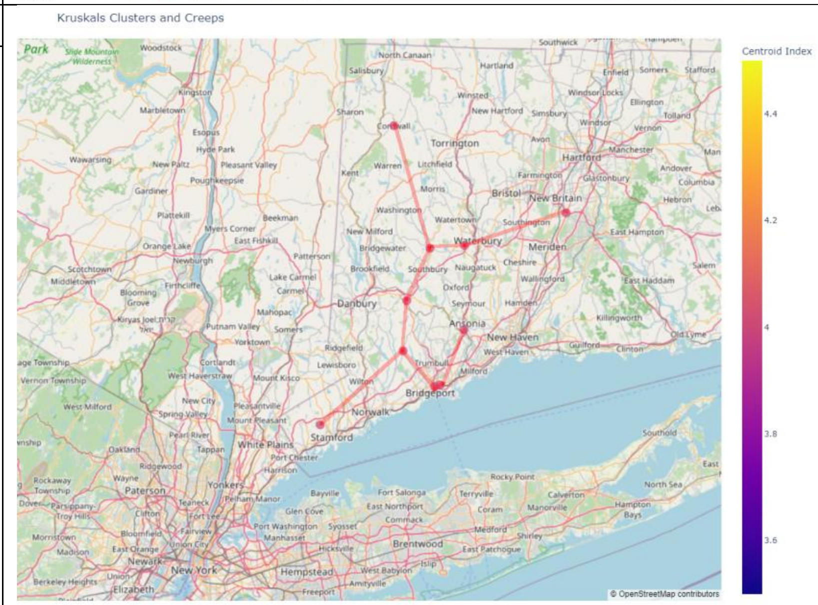
<table>
<tr><td>3 - 10: 0.021076
1 - 6: 0.142232
9 - 10: 0.153325
1 - 9: 0.264667
2 - 8: 0.446416
5 - 8: 0.466030
5 - 7: 0.483552
6 - 7: 0.570384
3 - 4: 0.604058
0 - 4: 1.462123</td><td>

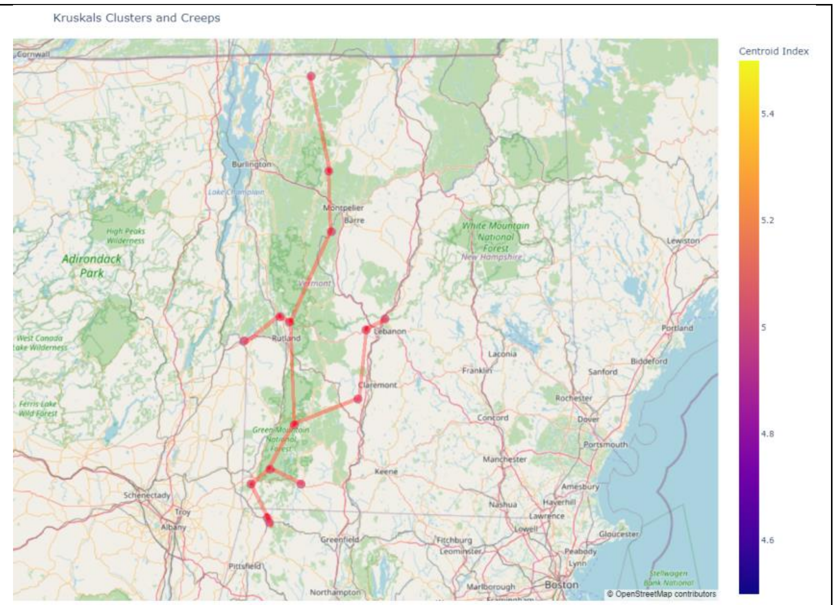


</td></tr>
<tr><td>Cluster 2</td><td rowspan="2">



</td></tr>
<tr><td>4 - 5: 0.001624
12 - 13: 0.010002
5 - 6: 0.010086
19 - 23: 0.013149
20 - 21: 0.014809
20 - 22: 0.023216
11 - 13: 0.025287
23 - 25: 0.056426
25 - 26: 0.073532
24 - 25: 0.117755
19 - 20: 0.134863
17 - 21: 0.138162
3 - 8: 0.148951
2 - 3: 0.158536
7 - 8: 0.159586
15 - 18: 0.164375
0 - 15: 0.170527
12 - 17: 0.188323
13 - 14: 0.199773
4 - 7: 0.217449
1 - 16: 0.268194
3 - 26: 0.302377
1 - 10: 0.305643
6 - 18: 0.401535
2 - 10: 0.437705
9 - 10: 0.517962</td></tr>
<tr><td>Cluster 3</td><td></td></tr>
</table>

13 - 14: 0.081204 12 - 14: 0.081390 0 - 3: 0.089695 7 - 10: 0.118728 4 - 5: 0.139861 3 - 6: 0.163051 4 - 6: 0.173108 0 - 1: 0.188373 2 - 6: 0.188513 8 - 12: 0.206065 8 - 11: 0.433409 9 - 10: 0.463599 10 - 12: 0.478306 5 - 7: 0.503874	
Cluster 4	
4 - 7: 0.018430 1 - 3: 0.117542 2 - 6: 0.128617 6 - 7: 0.140211 1 - 2: 0.154154 4 - 8: 0.162692 0 - 1: 0.333629 5 - 6: 0.338350 3 - 9: 0.354534	
Cluster 5	

1 - 2: 0.027259 5 - 7: 0.070863 3 - 6: 0.142631 4 - 11: 0.150699 2 - 11: 0.206137 4 - 14: 0.229179 7 - 13: 0.277533 4 - 12: 0.282334 0 - 9: 0.301800 6 - 10: 0.352289 10 - 12: 0.459634 0 - 8: 0.483789 5 - 12: 0.519378 5 - 9: 0.536722	

At this time one of the larger weaknesses of our project is that some data processing methods require one file being run at a time as opposed to simply running main.py. This limits the efficiency and cohesiveness of our code base.

Future routes of research could include applying a shortest path algorithm, such as Dijkstra's or Floyd-Warshall, to each cluster to plan a trip optimally based on milage. Since there are relatively few sites per cluster, the shortest path to visit all sites in each cluster could be found. This would be computationally expensive, but since there are relatively few data points per cluster it may be viable. Another opportunity could be diving further into the interesting world of data visualization, creating a map of road directions for a fully planned road trip. Or even integrating the google maps API to help us get real world information such as driving time and real life milage for the trip.

Keegan Moseley

Overall, I greatly enjoyed working on this project. An interesting experience I had when coming up with the project idea with my group mates was that we stumbled upon an early glimpse of NP-Complete problems. We initially wanted to find the shortest path to visit all points of interest. However, after some discussion, we decided that this problem had too many factors to apply a simple greedy algorithm too. We had not yet started our NP unit, but we were vaguely familiar with the Traveling Salesman Problem, and decided that finding the shortest path to visit all nodes was a similarly difficult problem. This project was also great practice for using Github and working collaboratively with other programmers. We all had our own assignments, which had to fit together cohesively. From what I've heard about working in the industry, the programing you do is almost never a solo endeavor, so it was nice to have further experience working with others. I was very pleased with how well my

group worked together. We never went a few days without emailing or texting about our progress, with very frequent video calls and meetings.

Ryan Moore

This project helped bring the concepts we covered in class to life in a concrete and creative way. I have little background in programming beyond the scope of my time at the Roux Institute and as a learner, I have found the best way to solidify a concept is to see it in action. Working with Keegan and Andrea gave me a fantastic opportunity to explore my understanding of minimal spanning trees, Kruskal's algorithm, and k-means clustering in a collaborative and unusual way. At the outset of this project, our group had lofty goals, especially my own goal to visualize the results of our graphing algorithms in a sleek, professional way. I have a background in communication and marketing and can't resist the opportunity to tell a story visually. This project gave me the chance to explore ways of visualizing data and the many tools available to do so. I settled on the Plotly library for Python because it was well-documented and supported. Plotly has a wide range of tools allowing you to represent data in unusual and creative ways. Though I was initially enamored by choropleth maps, I settled on a scatterplot map that reads information from the CSV generated by the graphing algorithms to display and color-code each cluster, showing which point belongs to which cluster. If there had been more time, I really would have enjoyed expanding the scope of this project to allow users to filter, sort, and search along with supplying real-world directions. In a future iteration of this project, it would be interesting to expand the scope of the shortest path algorithms to consider real-word travel time, tolls, CO2 emissions, accessibility (to include locations such as islands), and other constraints. I look forward to learning more about Data Science and Machine Learning next semester. Though we did narrow the scope of our project, I cannot say enough good things about working with Keegan and Andrea. I am enormously proud of our work and grateful to have such a cohesive and collaborative team.

Andrea Croak

This project was an enjoyable experience. Being a hands-on learner, getting to sit down, code, and apply concepts from class always brings joy. It also satisfied my curiosity for learning a little bit more about machine learning. The topic seemed like a magic black box where computers take in information, magically analyze it and spits back out something. But it's just loops and stop conditions, which was oddly comforting. The concept of Kruskal's algorithm and minimum spanning trees really became crystal clear too as we wound our way through writing the algorithms. In all honesty, the tree, pathing, and graphing sections have been some of my favorites in class. While we dreamed big with our initial proposal, we didn't accomplish everything we had hoped. I believe time was not on our side as the semester came to a close. Early on in the process I had more time to apply towards the group, our research, troubleshooting, and our research. The time flew by and having to balance my fulltime course load with end of the semester assignments, regular life, and spending time outdoors things left me feeling strained and drained. Despite that, I am exceedingly proud of the work our group accomplished in a short period of time, and look forward to potentially partnering up again in future courses. I have caught a little bit of the machine learning bug, so there may be a need for some Udemy experimentation over the summer.

- **Program Appendix**

Our Programs can be found on Khourys GitHub at the following link:
https://github.khoury.northeastern.edu/acroak/5800_group_project/tree/keegan

- **5800_dataset.csv** - This is the base CSV containing our researched datapoints across the New England area. We don't mutate our primary csv when creating centroid assignments, instead producing a secondary csv called "Final_dataset" that is automatically updated for each new run of centroids.

```
ID,lat,long,Place,State,City
0,43.1310080,-70.918525,Three Chimneys Inn,NH,Durham
1,44.4404056,-72.6798578,Emily's Bridge,VT,Stowe
2,41.6988070,-71.156407,Lizzie Borden House,MA,Fall River
3,42.6733866,-73.0915873,The Hoosac Tunnel,MA,North Adams
4,42.5574852,-71.9807807,S.K. Pierce Mansion,MA,Gardner
5,42.3575313,-71.4691568,Longfellow's Wayside Inn,MA,Sudbury
6,42.4617047,-71.3496517,Concord's Colonial Inn,MA,Concord
7,41.7728168,-71.0296064,Freetown-Fall River State Forest,MA,Assonet
8,42.5162160,-70.910225,Gallows Hill,MA,Salem
9,42.5174319,-70.9091478,Proctor's Ledge,MA,Salem
10,42.5212124,-70.8997968,Ropes Mansion,MA,Salem
11,42.6962960,-73.1063591,Houghton Mansion,MA,North Adams
12,42.0146330,-71.012082,Hockomock Swamp,MA,Raynham
13,42.3585002,-71.0599241,Kings Chapel Burying Ground,MA,Boston
14,42.4003397,-71.2139282,Metropolitan State Hospital,MA,Waltham
15,42.2726406,-72.4145564,Belchertown State School,MA,Belchertown
16,42.2636420,-71.896673,Spider Gates Cemetery,MA,Leicester
17,43.9463549,-70.1192825,Royalsborough Inn,ME,Durham
18,44.1086730,-70.2148714,Riverside Cemetary,ME,Lewiston
19,45.8263231,-67.9879768,"Haynesville Road, Route 2A",ME,Haynesville
20,44.2304611,-69.0476748,Maiden Cliff Trail,ME,Camden
21,43.6143735,-70.2131151,Beckett's Castle,ME,Cape Elizabeth
22,44.6691381,-70.1489941,USM Farmington,ME,Farmington
23,43.8582211,-70.1026224,Jameson Tavern,ME,Freeport
24,43.7355626,-70.4156038,Smith-Anderson Cemetary,ME,Windham
25,43.7381665,-70.555441,Old Red Church ,ME,Standish
26,44.5717667,-68.7843204,Cursed Tomb of Colonel Buck,ME,Bucksport
27,41.1895771,-71.5679001,The Palatine,RI,Block Island
```

28,42.0091173,-71.7096935,The Perron House,RI,Burrillville
29,41.6942734,-71.5440491,Nathanael Greene Homestead,RI,Coventry
30,41.9340864,-71.4042122,Cumberland Monastary/Library,RI,Cumberland
31,41.5807467,-71.5584478,Grave of Mercy Brown,RI,Exeter
32,41.5556950,-71.542381,The Ladd School,RI,Exeter
33,41.8199182,-71.7043525,The Ramtail Mill,RI,Foster
34,41.4818983,-71.3675597,Fort Wetherill,RI,Jamestown
35,41.4699145,-71.2981522,The Breakers,RI,Newport
36,41.4678179,-71.3040671,Seaview Terrace,RI,Newport
37,41.4758017,-71.3354155,Fort Adams,RI,Newport
38,41.4911463,-71.3129371,The White Horse Tavern,RI,Newport
39,41.8777376,-71.3829646,Slater Mill,RI,Pawtucket
40,41.8242752,-71.413425,The Biltmore Hotel,RI,Providence
41,41.8204641,-71.427659,Barnaby Castle,RI,Providence
42,41.6244848,-71.2073449,Fort Barton Woods,RI,Tiverton
43,41.8437058,-73.3292848,Dudleytown,CT,Cornwell
44,41.8835617,-71.9618903,Bara-Hack,CT,Pomfret
45,41.3509790,-71.9721425,Captain Daniel Packer Inn,CT,Mystic
46,41.5335257,-73.2064158,Curtis House,CT,Woodbury
47,41.4856693,-72.341798,Devils Hopyard State Park,CT,East Haddam
48,41.4010210,-73.2851937,Fairfield Hills State Hospital,CT,Newton
49,41.5407091,-73.0890935,Little Peoples Village,CT,Middlebury
50,41.4892648,-72.0731327,Norwich State Hospital,CT,Norwich
51,41.1863410,-73.172713,Remington Arms Munitions Factory,CT,Bridgeport
52,41.0843736,-73.5786036,Fort Stamford,CT,Stamford
53,41.2730160,-73.297721,Union Cemetery,CT,Easton
54,41.1824094,-73.1907184,Palace & Majestic Theaters,CT,Bridgeport
55,41.327390,-73.091633,Sterling Opera House,CT,Derby
56,41.6234491,-72.7443492,Meeting House Square,CT,Hartford
57,41.3554050,-72.038268,Pequot Hill,CT,Groton
58,44.3813200,-68.21107,Coach Stop Inn,ME,Bar Harbor
59,44.3163186,-69.7693596,Kennebec Arsenal,ME,Augusta
60,43.7983371,-70.2542786,Greely High School,ME,Cumberland
61,44.1046650,-69.114048,Lime Rock Inn,ME,Rockland
62,43.8543322,-69.6265625,Boothbay Opera House,ME,Boothbay
63,44.0119830,-70.978423,Admiral Peary Inn,ME,Fryeburg
64,44.7793290,-69.716391,The Strand Cinema,ME,Skowhegan
65,44.4589710,-68.914138,Carriage House Inn,ME,Searsport
66,44.5663440,-68.804687,Fort Knox,ME,Prospect
67,43.4564110,-71.220617,Alton Town Hall,NH,Alton
68,44.2568473,-71.4397775,The Mount Washington Hotel,NH,Bretton Woods
69,43.2303140,-71.536101,Margarita's,NH,Concord
70,43.2003310,-71.544889,New Hampshire Asylum for the Insane,NH,Concord
71,43.2050273,-71.5360586,Siam Orchid Restaurant,NH,Concord

```
72,43.9588815,-71.0846146,Stark Road Cemetary,NH,Conway
73,43.1959288,-70.8742176,The Old Dover Mills,NH,Dover
74,43.1787732,-71.3380185,The Epsom Red Schoolhouse,NH,Epsom
75,42.9672400,-70.96606,Exeter River Mobile Home Park,NH,Exeter
76,42.7648060,-72.150916,The Amos J Blake House,NH,Fitzwilliam
77,43.5843727,-71.382137,Kimball Castel,NH,Guillford
78,43.0203600,-71.60035,St. Anselm College,NH,Goffstown
79,42.9127980,-70.810963,Island Path Road,NH,Hampton
80,43.7033054,-72.2893869,Dartmouth College,NH,Hanover
81,43.5281930,-71.4700656,The Colonial Theater,NH,Laconia
82,43.6087448,-71.4597921,Winnipesaukee Marketplace,NH,Laconia
83,42.8387459,-71.4783688,Griffin School,NH,Litchfield
84,42.9732367,-71.4683568,Hesser College,NH,Manchester
85,42.9878672,-71.4660628,RG Sullivan Building,NH,Manchester
86,42.9769302,-71.4454367,Saint Joseph Middle School,NH,Manchester
87,42.8396802,-71.4914846,The Common Man Restaurant,NH,Merrimack
88,42.8257970,-71.629517,Lorden Plaza,NH,Millford
89,42.7895921,-71.5174664,The Country Tavern Restaurant,NH,Nashua
90,42.7161998,-71.5129427,Gilson Cemetary,NH,Nashua
91,42.9440210,-73.0878364,Glastenbury Wilderness,VT,Binnington
92,43.6878797,-72.9517848,The Eddy House,VT,Chittenden
93,43.6477450,-72.420751,The Quechee Inn at Marshland Farm,VT,Quechee
94,43.7145086,-73.0174541,Vermont Police Academy,VT,Pittsford
95,44.9084866,-72.8021359,Opera House at Enosburg Falls,VT,Enosburg
96,44.1391790,-72.6612695,Norwich University,VT,Northfield
97,43.3000260,-72.477309,Hartness House Inn,VT,Springfield
98,42.8691739,-73.2186346,Southern Vermont Colleve,VT,Bennington
99,43.1695980,-72.918049,The Bennington Triangle,VT,Winhall
100,43.5931020,-73.2670234,Marble Mansion,VT,Fair Haven
101,42.8684100,-72.87149,The White House Inn,VT,Wilmington
```

- Centroids.csv - Created csv to store the centroids for later use in other methods

```
Index,Lat,Long
0,41.64082480214136,-71.52807792820188
1,44.52529156786832,-69.1046717756104
2,42.770479899246176,-71.42672616888274
3,43.76125133368217,-70.78613143742854
4,41.39977097681344,-73.19884195086995
5,43.49072018565226,-72.85867763421005
```

- **Clusters_to_csv.py** - Simple Program to create a csv from the kcluster dataframe info

```
""
Simple Program to create a csv from the kcluster dataframe info
'''
import kmeans
import pandas

def main():
    k = 6
    #Tuple is:
        #[0] - the dataframe
        #[1] - the array of centroids
    tuple = kmeans.kmeans_starter(k)
    #print(tuple[0].head())
    tuple[0].to_csv("final_dataset")

main()
```

- **Coordinate.py -** Class to represent each location. Its methods allow for the distance between coordinates to be calculated, and the geographic midpoint of a set of coordinates to be found.

Coordinate.py

```
import math
import point

'''
Method to return a centroid (gravitational center) Coodinate from a set of Coordinate objects
'''
@staticmethod
def centroid(input_set):
    #create set of cartestian points of all cities
    point_set = set()
    for coordinate in input_set:
        point_set.add(coordinate.point)


    #get average cartesian point of all cities
    middle_point = point.average_of(point_set)
    #convert the cartestian point into a coordinate
    middle_coordinate = point_to_coordinate(middle_point)


    #print("IN CENTROID, CENTROID IS:")
    #print(middle_coordinate)
```

```
    return middle_coordinate


'''
Method to create a Coordinate point from a cartesian (x,y,z) point
'''
@staticmethod
def point_to_coordinate(point):
    radian_longitude = math.atan2(point.y, point.x)
    hypotenouse = math.sqrt(point.x * point.x + point.y * point.y)
    radian_latitude = math.atan2(point.z, hypotenouse)


    degree_latitude = radian_latitude * 180/math.pi
    degree_longitude = -1 * radian_longitude * 180/math.pi


    new_coord = Coordinate(degree_latitude, degree_longitude)


    return new_coord


'''
Class representing a DD Coordinate (Latitude, Longitude)
Only works for locations in both the North and Western hemispheres


The radian versions of the latidude and longitude are stored, as well
as a representation of the coordinate as a cartesian point (x,y,z)
'''
class Coordinate:
    def __init__(self, latitude_deg, longitude_deg):
        #degrees of lattitude
        self.latitude = latitude_deg
        #degrees of longitude
        self.longitude = longitude_deg


        #latitude in radians
        self.radian_latitude = latitude_deg * math.pi/180
        #longitude in radians   Fliped negative since all points are in W hemisphere
        self.radian_longitude = longitude_deg * -1 * math.pi/180
```

```
        #cartesian point of this coordinate.
        # x and y plane is at the equator, with origin at center of the earth.
        #   Positive x line passes through 0 degrees E
        #   Positive y line passes through 90 degrees E
        # Positive z line is from the center of the earch to the north pole
        x = math.cos(self.radian_latitude) * math.cos(self.radian_longitude)
        y = math.cos(self.radian_latitude) * math.sin(self.radian_longitude)
        z = math.sin(self.radian_latitude)
        self.point = point.Point(x,y,z)


    '''
    Method to get the distance between two coordinates (in Nautical Miles)
    '''
    def distance(self, otherCoordinate):
        try:
            #check if these are the same point
            if (self.latitude == otherCoordinate.latitude and self.longitude ==
otherCoordinate.longitude):
                return 0


            #cental angle between two coordinates. See this article for math explaination:
https://en.wikipedia.org/wiki/Great-circle_distance
            value = (math.sin(self.radian_latitude) * math.sin(otherCoordinate.radian_latitude) +
                    math.cos(self.radian_latitude) * math.cos(otherCoordinate.radian_latitude) *
                    math.cos(self.radian_longitude - otherCoordinate.radian_longitude))
            angle = 0
            #value can sometimes be 1.0000000000000002, due to so many floats being used.
            if (value > 1):
                #print(value)
                angle = math.acos(1)
            elif (value < 0):
                #print(value)
                angle = math.acos(0)
            else:
                angle = math.acos(value)

            #in nautical miles
            #NM are nice b/c 1 NM == 1 minute, but this can easily be changed to a different unit
            distance = angle * 3443.92
```

```
            return distance
        except ValueError:
            print("Issue with acos due to Floating Point numbers. Input:")
            print(math.sin(self.radian_latitude) * math.sin(otherCoordinate.radian_latitude) +
                        math.cos(self.radian_latitude) * math.cos(otherCoordinate.radian_latitude) *
                        math.cos(self.radian_longitude - otherCoordinate.radian_longitude) )

    def __str__(self):
        string = "Latitude : " + str(self.latitude) + " Longitude : " + str(self.longitude)
        return string


def main():
    boston = Coordinate(42.3601, -71.0589)
    portland = Coordinate(43.6591, -70.2568)
    burlington = Coordinate(44.4759, -73.2121) #Vermont, not mass


    #test over most of NE
    set_1 = set()
    set_1.add(boston)
    set_1.add(portland)
    set_1.add(burlington)
    #print(centroid(set_1))


    #test over a short distance, the boston area
    arlington = Coordinate(42.4154, -71.1565)
    brookline =  Coordinate(42.3318, -71.1212)


    set_2 = set()
    set_2.add(boston)
    set_2.add(arlington)
    set_2.add(brookline)


    #print(centroid(set_2))

main()
```

- **Csv_parser.py** - Built initially due to skill gap with Pandas, but we learned Pandas and allowed this method to help support our node class.

Csv_parser.py

```
import pandas


# array for storing Node Objects
GPS_Points = []


class Node:
  def __init__(self, id, xcoord, ycoord):
    self.id = id
    self.xcoord = xcoord
    self.ycoord = ycoord


# Methods to fetch data from CSV based on column name
def getID(dataset):
  return dataset.ID

def getPlace(dataset):
  return dataset.Place


def getState(dataset):
  return dataset.State

def getCity(dataset):
    return dataset.City


def getX(dataset):
    return dataset.lat #returns lat


def getY(dataset):
    return dataset.long #returns long


# Fetch Datapoints from CSV and create a Node object, add the object to a globablly
available array, GPS_Points. These objects indices and ID number correlate. ie object ID
24 will be located at index 24. The count starts from 0
def createNodes(filepath):
  dataset = pandas.read_csv(filepath)
  IDList = getID(dataset)
  #   PlaceList = getPlace(dataset)
```

```
   #    StateList = getState(dataset)
   #    CityList = getCity(dataset)
   XCoordList = getX(dataset)
   YCoordList = getY(dataset)
   # print(len(XCoordList))


   # For every row in the provided CSV, grab the information based on the column name
and create a node object to contain it.
   for i in range(len(IDList)):
      # print(i)
      GPS_Points.append(Node(IDList[i], XCoordList[i], YCoordList[i]))

   return GPS_Points

# createNodes("5800_dataset.csv")
# print(GPS_Points[4].xcoord)
# print(GPS_Points[0].id)
# print(GPS_Points[0].xcoord)
# print(GPS_Points[0].ycoord)


# For K-Means we will want to create another csv sheet to store the new cluster
information
```

- **Elbow Method** – In order to determine what k value to use, we employed the “elbow” method. The graph created by this file allows us to interpret how the average intra-cluster variation changes with different k values. Using this graph, we decided to have our k value be 6.

Elbow_method.py

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import warnings
import coordinate as coord
import kmeans


warnings.simplefilter(action='ignore', category=FutureWarning)


def create_elbow_graph():
   orig_df = pd.read_csv('5800_dataset.csv')
```

```
#distortians
y= []
#k's
x = [1,2,3,4,5,6,7,8,9,10,11,12,13,14,15]
#get a distorians for each k
for k in range(1, 16):
   #print(k)
   total_distorian = 0


   #run the algo 100 times
   for i in range(0, 100):

      #cluster the data into k clusters
      result = kmeans.kmeans_starter(k)


      #find total distance of each coordinate to its centroid
      total_distance = 0
      for i in range(len(orig_df)):
         #get coordinate Object
         coordinate = result[0]["Coord"][i]
         #centroid object
         centroid_index = result[0]["Centroid Index"][i]
         centroid = result[1][centroid_index]


         #find distance
         distance = coordinate.distance(centroid)
         total_distance += distance


      #get average distance (distorian)
      distorian = total_distance/(len(orig_df))


      #record distorian for this iteration
      total_distorian += distorian


   #average distorian for 100 k clusters
   y.append(total_distorian/100)
```

```
    #print(y)


    plt.figure(figsize=(16,8))
    plt.plot(x, y, 'bx-')
    plt.xlabel('k')
    plt.ylabel('Distortion')
    plt.title('The Elbow Method showing the optimal k')
    plt.show()



def main():
    create_elbow_graph()
main()
```

- **K-Means Program** – This is the implementation of our K-Means machine learning algorithm. Using the Pandas library, the program reads in the csv data for the latitude and longitudes of our points of interest. It then can plot these items on a graph. The application will then apply random centroid values and self-update until those centroids are an optimal distance for each of their clusters

Kmeans.py

```
#import libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import coordinate as coord

'''
1. specify number of clusters you want (k)
2. randomly init centroid for each cluster
3. discover which data points belong to which cluster by finding closest centroid to each data
point
4. update centroids based on geometric mean of all data points in cluster. update datapoints
to what cluster they belong to
5. iterate through/run 3 and 4 until centroids stop changing position.
'''
'''
Parameters:
```

```
    Dataset containins Latitude, Longitude, and Coordinate object columns. Columns are added
of Centroid Indices, the centroid latitude, and the centroid longitude
    k - num clusters
    Coordinates - array of the Coordinate objects, same order as Dataset Coordinate sequence
    Centroids - array of k random coordinates to be the initial centroids
Returns:
    Final array containing centroids
'''
def KMeans(dataset, k, Coordinates, Centroids):
    #arrays to preserve data gained in While loop cuz im trash at Pandas

    #2d array, where each inner array is a cluster. final_clusters[i] is cluster belonging to
final_centroids[i]
    final_clusters = []

    #the k final centroids of the clusters
    final_centroids = []

    #Array containing index that each Coordinate's centroid is in (of the final_centroids array)
        #Ex: Coordinate at index 4 in Coordinates will belong to the Centroid:
        # final_centroids[final_centroid_index[4]]
    final_centroid_index = []

    diff = 1
    counter = 0
    # Step 5, repeat commands until the diff is 0
    while(diff!=0):
        # Step 3 which data points belong to which cluster
        #print("Iteration " + str(counter))

        #Create 2d array k long. Array at index i will contain
        #list of distances of coordinates to the centroid at index i
        distances = []
        for i in range(0,k):
            distances.append([])

        #Find the distance of every coordinate to every centroid
        for i in range(0, k):
            #get the centroid
            centroid = Centroids[i]
            for j in range(0,len(Coordinates)):
                #find distance of site at index j to this centroid
                site = Coordinates[j]
                distance = centroid.distance(site)
```

```
        #append the distance of this coordinate to inner array of index i
        distances[i].append(distance)

    #2d cluster array. Each internal array at index i contains
    #the Coordinates belonging to the cluster of the Centroid at index i
    clusters = []
    for i in range(0, k):
      clusters.append([])

    #holds which index to use to look up Centroid of each coordinate
    cluster_indices = []

    # Step 4 find which centroid each point is closest to, and add them to that centroid's
cluster
    #find closest cluster for the Coordinate at index i
    for i in range(0, len(Coordinates)):
      min_dist = 10000000000
      #index of closest centroid found so far
      cluster_index = -1

      #j is index of centroid. Find which centroid is closest to this site
      for j in range(0, k):
        possible_min_distance = distances[j][i]
        if (possible_min_distance < min_dist):
          #closer centroid found
          min_dist = possible_min_distance
          cluster_index = j

      #print("Min dis:" + str(min_dist))
      #add the Coordinate to a cluster
      clusters[cluster_index].append(Coordinates[i])

      #save the cluster index, so can look up the centroid of this Coordinate
      cluster_indices.append(cluster_index)

    #Find the new average centroid of each cluster
    new_centroids_list = []
    for list in clusters:
      centroid = coord.centroid(list)
      new_centroids_list.append(centroid)

    diff = 0
    #check if the new centroids generated are the same as the previous centroids, meaning
```

```
the algo is done
    for i in range(0, k):
      new_centroid = new_centroids_list[i]
      old_centroid = Centroids[i]

      if new_centroid.latitude == old_centroid.latitude and new_centroid.longitude ==
old_centroid.longitude:
        #same centroids
        continue
      else:
        #centroids are not equal. Update the Centroids and do another iteration
        i = k
        diff = 1
        Centroids = new_centroids_list
        counter += 1

    #save the array of clusters ,array of final centroids, and centroid indices for Coordinates
    final_clusters = clusters
    final_centroids = new_centroids_list
    final_centroid_index = cluster_indices

  #add a column to the dataframe, with the index to find each Coordinate's Centroid
  dataset.insert(6, "Centroid Index", final_centroid_index)
  #print(dataset.head())

  #populate array of centroid latitude/longitude for each coordinate
  centroid_latitudes = []
  centroid_longitudes = []
  for i in range(len(dataset)):
    centroid = final_centroids[dataset["Centroid Index"][i]]
    centroid_latitudes.append(centroid.latitude)
    centroid_longitudes.append(centroid.longitude)

  #add centroid lat/long to the dataset
  dataset.insert(7, "Centroid Lat", centroid_latitudes)
  dataset.insert(8, "Centroid Long", centroid_longitudes)

  #print(dataset.head())

  '''This works, just commented out to prevent errors when k > 5
  # print out the final scatter plot. Not a totally accurate representation of 3d lat/long points,
but close enough to see many relationships
  color=['green','blue','cyan','indigo','violet']
  #plot each site
```

```
    for i in range(0, len(Coordinates)):

        #get centroid index of this val, to assign color to this point
        #allows for unique color per cluster
        centroid_of_coordinate = dataset["Centroid Index"][i]
        cluster_col = color[centroid_of_coordinate]

        latitude = dataset["Coord"][i].latitude
        longitude = dataset["Coord"][i].longitude
        plt.scatter(latitude,longitude,c = cluster_col)
    #plot the centroids
    for i in range(0, len(Centroids)):
        centroid = Centroids[i]
        latitude = centroid.latitude
        longitude = centroid.longitude

        plt.scatter(latitude,longitude,c='red')
    plt.xlabel('Latitude')
    plt.ylabel('Long')
    plt.show()
    '''


    return final_centroids

''' Would need to make some changes to these to make usable after refactoring
def show_init_plot(dataset):
    #this scatter plot will show us how all the datapoints look on a graph
    plt.scatter(dataset["lat"],dataset["long"],c='black')
    plt.xlabel('latitude')
    plt.ylabel('longitude')
    plt.show()

def show_init_plot_and_rnd_centroids(dataset, Centroids):
    # this scatter plot will show all our points and random centroid prior to clustering
    plt.scatter(dataset["lat"],dataset["long"],c='black')
    plt.scatter(Centroids["lat"],Centroids["long"],c='red')
    plt.xlabel('lat')
    plt.ylabel('long')
    plt.show()
'''

'''
Helper method, to make calling kmeans easier outside this file.
```

```
Parameters:
   k, the number of clusters
Returns:
   a tuple:
      [0] - dataframe containing Latitude, Longitude, Coordinate Objects, the Index of the
cluster centroid of each row,
         the centroid latitude, and the centroid longitude
      [1] - array of k centroids
'''
@staticmethod
def kmeans_starter(k):
   try:
      data = pd.read_csv('5800_dataset.csv')
      dataframe = data[["lat","long","Place","State","City"]]
      print(dataframe)
      #array of coordinate objects
      coordinates_array = []
      #create coordinate objects and add them to the data frame
      for index, row in dataframe.iterrows():
         new_coordinate = coord.Coordinate(row["lat"], row["long"])
         coordinates_array.append(new_coordinate)
      #add a column of coordinates to dataFrame
      dataframe.insert(2,"Coord", coordinates_array)

      #get random starter centroids
      Centroids = (dataframe["Coord"].sample(n=k))

      #create list to overwrite indices of Centroids. Then allows 0 indexing like an array.
      index = []
      #make 0 - (k-1) list
      for i in range(0,k):
         index.append(i)
      #set index of Centroids
      Centroids.index = index

      #simple array containing centroids
      centroids_array = []
      for i in range(0, k):
         centroids_array.append(Centroids[i])

      #call kmeans
      return dataframe, KMeans(dataframe, k, coordinates_array, centroids_array)
   except ValueError as e:
      #very rare exception, where one cluster has no members after a few iterations of
```

```
kmeans.
      #Simply retry the algo with new randomized starting coords
      return kmeans_starter(k)


def main():

   # read in data
   data = pd.read_csv('5800_dataset.csv')
   #print(data.head())

   # take only the lat and long columns for graphing
   X = data[["lat","long"]]
   # show_init_plot(X)

   #array of coordinate objects
   coordinates_array = []
   #create coordinate objects and add them to the data frame
   for index, row in X.iterrows():
      new_coordinate = coord.Coordinate(row["lat"], row["long"])
      coordinates_array.append(new_coordinate)
   #add a column of coordinates to dataFrame
   X.insert(2,"Coord", coordinates_array)

   # Step 1 number of clusters
   # we get this number from our elbow graph
   K=5

   # Step 2 Select random observation as centroids
   Centroids = (X["Coord"].sample(n=K))
   # show_init_plot_and_rnd_centroids(X, Centroids)

   #create list to overwrite indices of Centroids. Allows 0 indexing like an array.
   index = []
   for i in range(0,K):
      index.append(i)
   Centroids.index = index

   centroids_array = []
   for i in range(0, K):
      centroids_array.append(Centroids[i])

   KMeans(X, K, coordinates_array, centroids_array)

if __name__ == "__main__":
```

```
main()
```

- **Kruskal's MST** - An implementation of Kruskals Algorithm for finding the minimum spanning tree for a given data set. This program leveraged the math and pandas python libraries. Provided that the supplied .csv of data contains a column titled “lat” and “long” the program will print out a list of nodes and edges that form the minimum spanning tree.

Kruskals_mst.py

```
import math


class Graph:
    def __init__(self, vertices):
        self.vertices = vertices
        self.graph = []


    # p = node1, q = node2, weight is the weight of the edge between them
    def add_edge(self, p, q, weight):
        self.graph.append([p, q, weight])


    # find the i elements set
    def find(self, parent, i):
        if parent[i] == i:
            return i
        return self.find(parent, parent[i])


    # apply a union (edge) between 2 node points. This is completed by leveraging the rank of
the nodes
    def apply_union(self, parent, rank, x, y):
        xroot = self.find(parent, x)
        yroot = self.find(parent, y)
        if rank[xroot] < rank[yroot]:
            parent[xroot] = yroot
        elif rank[xroot] > rank[yroot]:
            parent[yroot] = xroot
        else:
            parent[yroot] = xroot
            rank[xroot] += 1
```

```python
    # construct the MST
    def kruskal_algo(self):
        mst = []
        # index counter for the sorted edge list
        i = 0
        # index counter for the mst edges list
        e = 0
        # Sort the graph data based on edge weight, ascending
        self.graph = sorted(self.graph, key=lambda item: item[2])
        parent = []
        rank = []


        # creation of subsets
        for node in range(self.vertices):
            parent.append(node)
            rank.append(0)
        #  edge number should be |V| - 1
        while e < self.vertices - 1:
            # start with the smallest edge
            p, q, weight = self.graph[i]
            # increment to the next smallest edge
            i = i + 1
            # see if a cycle is created
            x = self.find(parent, p-1)
            y = self.find(parent, q-1)
            if x != y:
                e = e + 1
                mst.append([p, q, weight])
                self.apply_union(parent, rank, x, y)
        # print out the MST
        for p, q, weight in mst:
            print("%d - %d: %f" % (p, q, weight))
```

- **Main.py** - Finds the MST of the final centroids.

Main.py

```python
import CSV_Parser as csv
import kruskals_mst as mst
import elbow_method as elbow
import math
import pandas
```

```
def retrieve_edge_weight(adj_matrix, row, column):
    #row is outer index. column is inner index
    weight = 0
    weight = adj_matrix[row][column]
    return weight

def main():
    # Fetch Data from CSV file and create node objects
    # These objects have a .id, .xcoord, and .ycoord value (more can be added).

    '''
    Node_List = csv.createNodes("5800_dataset.csv")
    print(Node_List[4].xcoord)
    print(Node_List[4].ycoord)
    '''

    Node_List = []
    centroids_dataset = pandas.read_csv("Centroids.csv")
    for i in range(len(centroids_dataset)):
        new_centroid = csv.Node(i,centroids_dataset["Lat"][i],centroids_dataset["Long"][i])
        #print(str(centroids_dataset["Lat"][i]) + " , " + str(centroids_dataset["Long"][i]))
        Node_List.append(new_centroid)

    matrix = open("edges.csv", 'r')
    #print(matrix)
    adj_matrix = []
    #create adjacency matrix of centroids
    for i in range(7):
        str = matrix.readline().strip()
        if i == 0:
            continue

        row = str.split(',')
        formatted_row = row[1 : len(row)]
        #print(type(formatted_row[0]))

        for j in range(len(formatted_row)):
            string_version = formatted_row[j]
            integer_version = int(string_version)
            formatted_row[j] = integer_version

        #print(formatted_row)
        adj_matrix.append(formatted_row)
```

```
    matrix.close()

    # For each node in the Node_List array add it to a graph and create edges
    # run MST on these values, we will then have MST for ALL points.
    graph = mst.Graph(len(Node_List))
    for i in range(len(Node_List)):
        for j in  range(i+1,len(Node_List)):
            #print("[" + str(i) + ", " + str(j) + "] " +str(Node_List[i].xcoord) + ", " +
str(Node_List[j].ycoord))
            weight = retrieve_edge_weight(adj_matrix, i, j)
            #print(weight)
            graph.add_edge(i, j, weight)

    graph.kruskal_algo()

    # Use elbow Method to figure out the best k value
    #elbow.create_elbow_graph()

    # Run all points that live in Node_List through the k-means clustering algorithm
    # these outputs we can write to a new csv file.
    # TO DO: how do k-clustering?

    # once the new cluster csv file exists, we will run it through the MST process as well.

if __name__ == "__main__":
    main()
```

- **ScatterMap2.py -**

Scattermap2.py

```
import plotly.express as px
import pandas as pd
from urllib.request import urlopen
import json

# df = data frame

df =
pd.read_csv('https://raw.github.khoury.northeastern.edu/acroak/5800_group_project/main/
5800_dataset.csv?token=GHSAT0AAAAAAAAANP22HTESRZ2M72YQX4QWZCCT4CA')


print(df.head(10))
print(df.tail(10))
```

```
fig = px.scatter_mapbox(df,
                        place = df['Place'],
                        lon = df['long'],
                        lat = df['lat'],
                        zoom = 7,
                        width = 1200,
                        height = 900,
                        title = 'Test Map')


fig.update_layout(mapbox_style="open-street-map")
fig.update_layout(margin={"r":0, "t":50, "l":0, "b":10})
fig.show()
```

- **Point.py** - Helper class of the Coordinate Class. For some operations, Coordinates had to be converted into (x,y,z) points.

Point.py

```
from math import *


'''
Method to return a new Centoid (average/center) point for an input set of points.
Paremeters:
   - points set populated by only Point objects, and with a size > 0
Returns:
   new Point object
'''
@staticmethod
def average_of(points):


   num_points = len(points)


   if (num_points == 0):
      #ensure the input set is not empty
      raise ValueError("Must input a set of Point objects with at least 1 object")
   elif (not(isinstance(list(points)[0],Point))):
      #ensure the input set contains Point objects
      raise ValueError("The objecst in the set must be Point objects")
```

```
    #get average x coord
    average_x = 0
    for pt in points:
        average_x += pt.x
    average_x = average_x/num_points


    #get average y coord
    average_y = 0
    for pt in points:
        average_y += pt.y
    average_y = average_y/num_points


    #get average z coord
    average_z = 0
    for pt in points:
        average_z += pt.z
    average_z = average_z/num_points


    return Point(average_x,average_y, average_z)



'''
Class representing a cartesian point


Attributes:
    - x coordinate
    - y coordinate
    - z coordinate
'''
class Point():
    def __init__(self, x, y, z):
        self.x = x
        self.y = y
        self.z = z


    def __str__(self):
        return "[ x = " + str(self.x) + ", y = " + str(self.y) + ", z = " + str(self.z) + ']'
```