Clustering Program

Zach Bender, Jacob Everett, Devon Gates, Mitchell Koen

Project Description

Given the name of a user-entered configuration text file—consisting of an input file, a minimum and maximum k-mean value, and some number of k-values with the IDs of points corresponding to initial centroids—this program will open the input file containing the IDs of some number of points and their x y coordinate values and store those IDs and values. It will then perform k-means and hierarchical clustering on that data, and afterwards will calculate the Dunn Index for both k-means and hierarchical clustering.

Part 1 (Devon Gates)

This portion of the project stores data from config.txt and points.txt files. My goals for this portion of the assignment were to (1) make my code very easy to understand, as my group would depend heavily on this portion, and (2) process the data efficiently. I chose to store data for centroids and points in structs. I did this because the structs would be stored in private vectors in class Config, which would allow safe access to the sructs while also having the benefit of ease of access to the public data within the structs. Each config.txt and points.txt file corresponds to one Config object. Each Config stores private data for Centroids in a vector, Points in a vector, the points.txt filename in a string, the minimum number of clusters in an int, and the maximum number of clusters in an int. I chose to store Centroids and Points in vectors because my group would need to access data using iterators and because of how safe vectors are with memory management. Additionally, because the data could effectively be represented by a list, other data structures such as maps seemed unnecessary.

I included two functions to populate my vectors for Centroids and Points: parse_config and parse_points. parse_config returns type bool in order to ensure that code in main.cpp does not execute if the filename obtained from cin does not exist. Data is read in by using fstream. This process was straightforward for reading in the first three lines of each config.txt, but the centroids proved challenging to process. The difficulty with the centroids was that each line required a different number of reads. I overcame this by using sstream, which allowed me to treat each line as its own file. This allowed me to easily detect the end of each line of data in a way that the overloaded >> operator does not allow. parse_points also reads in data using fstream, though the >> operator was sufficient in handling all of the reads in this case.

For adding Centroids and Points to their respective vectors, I chose to declare a single variable for each in their respective functions. I defined their data after each data read and then did a push back by value into the vector. This worked because my function to access the vectors returns each vector by reference. This allows the built-in functionality of the vector library to handle all of the memory allocation, rather than having to manually allocate that memory. This is advantageous because memory allocation in vectors is safe, consistent, as well as time-saving.

Part 2 (Jacob Everett)

The goal for this section was to perform k-means clustering on all the points and, once no points

changed clusters between new centroid calculations, output for each value of k: the cluster label, how many points are the cluster contains, and the ID of the point closest to its centroid for each cluster.

A function called initPoints first gets and copies all the points read-in earlier into a map in order to efficiently find them by their ID using map functions. Once all points are stored, the initial centroids are set as float pairs by using the centroid point IDs read-in from the config file to find the corresponding x y values of the points in the map. Once all initial points and centroids are set, finalizePoints is called, which repeatedly assigns points and recalculates centroids until a bool indicating that two consecutive centroids remained the same is set to false.

I needed to be able to store all the clusters I created as well as be able to know which cluster corresponds to which k-value so that the Dunn Index for each could be calculated later, so I decided to use maps to store the major components I needed to keep track of. Using maps allowed me to use an ID (for points) or value of k (for clusters) as a key, and I could store a vector of clusters or pair of floats as the value.

It took me some time to get used to accessing things that were buried so many levels deep. For instance, if I wanted to get a specific point in a specific cluster for a specific value of k, I would need to access my map that stored all the clusters (allClusters) at the element corresponding to a key of k. That element was a vector of cluster struct pointers, and index 0 of that vector would contain the first cluster for that value of k. Within that pointer to the cluster struct, I would need to access pointsInCluster, which was a map of points with float pair values stored at keys corresponding to their ID. allClusters[2][0] is a pointer to the first cluster for a k-value of 2.

The final values output for k-means are slightly different than the example output for some clusters. I believe this is because I stopped the point assignment and centroid recalculation process when two consecutive centroids remained the same rather than when no points changed clusters.

Part 3 (Mitchell Koen)

The goal of this section is to perform centroid-linkage hierarchical clustering on a set of points brought in by part 1 and manipulate them in a format that part 4 can use. I had to make each point their own cluster, and then find the two closest clusters to each other and merge them with each subsequent hierarchical level. This process would occur once per hierarchical level in order to document each individual merge.

The first step was the easiest since that required a simple cloning of each point into an identical struct called HCluster, these HClusters would contain an ID, for the cluster to be tracked, X and Y coordinates, so that the cluster's individual centroid can be tracked for finding the shortest distance between 2 clusters, and a point list containing all points within that specific cluster. A vector these HClusters would then be used in a specific HLevel. These HLevels would contain a vector of HClusters, and an ID (variable called Level) for tracking. Another variable, shortest_d (short for shortest distance) was used in testing but has been left in. Each HLevel would represent the merging of the 2 closest HClusters in the previous HLevel. So for example if there were 301 HClusters in HLevel 0, then there will be 300 HClusters in HLevel 1 since 2 HClusters were merged to become a single HCluster.

The function closest, requires a vector of HClusters to be passed in by reference, finds the 2 HClusters that are the closest to each other then passes those two HClusters into another function called merge. Merge requires 2 HClusters to be passed by reference as well as the vector of HClusters the 2 HClusters originate from. The process of merging required for the smallest ID of the 2 clusters to be used, the cluster's centroid's (x,y) values to be averaged, and for the vectors containing the point lists of each cluster to be merged together.

The major problems that occured during the project were general syntax issues and properly typing out what I was thinking in my head. There were some occurrences where debugging was needed to find the fault of some segfaults but these were generally problems caused by misusing get functions or vector functions. Overall went pretty smoothly.

Part 4 (Zach Bender)

For part four I had to start with checking to see how the other two parts, those being part 2 and part 3 saved their information. I hoped that they would have saved it the same way but no such luck. I Began by creating a class to contain the index's of each of the different algorithms one named Hdunindex for the hierarchy index's and the other named Cdunindex to contain the k means index's. I then created get and set functions for those index's in a separate .cpp file. The reasoning to do this was to then create a second class called the calculation class that would hold a vector of the first class. In the public section of the second class I would also make a function to return the vectors of the first class. I did this to be able to make functions that could work with and manipulate the information in the vectors (ie: h dunindexes and the c dunindexes). The first major function which would be called in main would take the k value and then within the function I would take all of the points associated with that k value and begin to compare all the points within that k value and find the smallest distance and the largest distance between the points associated with that k value. Then after I have completed finding the smallest distance and the largest distances using the distance formula I would divide the two number and then use a set function for the first class to set it to the c dunindex then push that into my vector of class one.

For the second calculation function I would take in the vector of clusters in main that contained the hierarchy cluster along with the level and the size of the cluster. In the function I would something like part two's but a little different since they were stored differently. I assumed the first cluster to have the smallest distance between to points and the second to have the largest then filtered thru each of the points in a for loop saving the largest and smallest to a temporary variable. I would then divide the smallest distance and the largest distance and then use the set function from the first class to set the index value and use the push back function from the vector in the second class to save it. I also checked to make sure the level was not less than 150 since it seemed that I only had to calculate the numbers between the 150 level and the max level, and it would not print anything if the cluster size was 1.

I then had to create a function to calculate the largest of the dunn indexes for both algorithms. I did this by taking in the vector of indexes for both types of indexes, though I ended up having to make two separate functions because they both depended on different things, but the process were identical. I would assume the first index is the smallest then loop through comparing all the numbers setting the highest number to a temporary variable in the function then print out that temporary variable.