IE 599 – Introduction to Data Mining:

Project 1
Bisecting K-Means

Benjamin Fields

931-707-381

October 26, 2016

Introduction:

In this project, we were to explore the basic concepts and algorithms used in cluster analysis. Cluster analysis is described as the dividing of data into groups(clusters) that are meaningful, useful, or both (Tan, Steinbach, & Kumar, 2005). This can be accomplished with many different algorithms, but the two that are significantly important for this project are the Bisecting K-Means algorithm and the standard k-means algorithm. Both can be applied to achieve similar results however the bisecting k-means algorithm uses the concepts of the standard k-means where the standard k-means algorithm is independent of the bisecting implementation. The standard k means algorithm can be described with the following pseudocode:

```
1. Select K points as initial centroids
2. Repeat
3.     Form K clusters by assigning each point to its closest centroid.
4.     Recompute the centroid of each cluster.
5. Until Centroids do not change
```

(Tan, Steinbach, & Kumar, 2005)

This algorithm contains multiple steps that all contribute to the solution obtained in the end. In the very beginning one must select appropriate centroids such that they are positioned optimally or one could select random centroids. The algorithm must also incorporate distance measurements such as Euclidian distance and sum of squares error. With all these parts working together in the algorithm, accurate results can be obtained. The standard K means algorithm is the foundation of the Bisecting K-means algorithm. For this project the focus was on implementing the bisecting version. The algorithm can be described by the following pseudocode:

```
1. Initialize the list of clusters to contain the cluster consisting of all points
2. Repeat
3.     Remove a cluster from the list of clusters.
4.     {Perform several "trial" bisections of the chosen cluster.}
5.     For I = 1 to number of trials do
6.         Bisect the selected cluster using basic K-means
7.     End for
8. Select the two clusters from the bisection with the lowest total SSE.
9. Add these two clusters to the list of clusters.
10. Until the list of clusters contains K clusters.
```

(Tan, Steinbach, & Kumar, 2005)

This implementation uses multiple iterations of the standard k-means technique and must create new centroids each time a new cluster is to be split by the k-means algorithm. The algorithm also checks the objective function which is the SSE obtained by summing over all clusters the square of the Euclidian distance. The highest SSE is chosen each time in order to get the best results for each cluster. Once the number of clusters desired is obtained the algorithm terminates. One can also use preprocessing techniques to obtain different results. In the case of this project visual preprocessing was performed to obtain initial relationships that may be within the data. The implementation used for this project was in the C/C++ programming language.

Algorithm Implementation

The process of creating the full implementation required a series of steps that each built on one another. In the initial phase of the project I created a series of classes to allow for easy handling of data elements within the program. When I began the implementation, I focused on creating the basic implementation of the k-means algorithm as this is the main foundation of the bisecting k-means algorithm. As described in the previous section one of the critical steps in the algorithm is to obtain initial centroids. To accomplish this I created a method to extract the Max and min values for the A data points and the B data points. Once I had these values I could then create a random centroid within this range. I also had to check after each centroid was created to ensure that there were no previous centroids that were of the same value. Once the centroids were chosen I could then create empty vectors to represent the clusters and begin assigning the data points. This was accomplished by comparing each data point's Euclidian distance to the cluster's centroid and assigning it to the closest value. Once the initial assignment was complete I could then recompute the centroids by taking the average of the points in each cluster. With this average, I can then calculate the SSE value for each cluster and the clustering at the end of the iteration. This then begins a loop of iterations until the centroids do not change. After each iteration, I was comparing the initial centroids to the final centroids to determine when the algorithm had converged.

With a complete standard k-means algorithm I could then begin creating the bisecting k-means implementation. This involved some substantial restructuring in order to allow for the new logic. In the standard k-means I was able to use one data set throughout but in the bisecting version the data set being focused on shifts as each new cluster is identified. To begin the algorithm, I simply create one vector and assign all the data points to it. Once this is complete I can run the standard k-means algorithm to obtain two clusters from this initial data set. This process uses the standard k-means algorithm created previously with some modification to what data is being focused on. By passing the cluster of focus in each iteration I can ensure that the algorithm runs on the correct data set. In order to identify this selected cluster there are first some steps that must be taken. The first pass does not require any selection but after the first pass the clusters must be selected appropriately.

In order to split the optimal cluster after each iteration the SSE must be calculated for each cluster. This technique uses the same technique as the traditional k-means algorithm and uses the sum of the squares of the Euclidian distance. Once the SSE for each cluster are selected then the cluster with the highest SSE can be selected. The index of this cluster can then be tracked as the algorithm progresses to ensure the correct cluster is removed and replaced with new split clusters. This process continues until the number of clusters is met.

As mentioned the core equations used to drive this algorithm are the Euclidian distance which is represented by the following equation:

Euclidian distance $d(x, y) = (\sum_{n=0}^{n} |x_k - y_k|^r)^{1/r}$
Where r = 2

Similarly, the objective function used to select the proper clusters during each iteration is selected using the square of this Euclidian distance. The formal term used to represent this error value is the Sum of Squares Error (SSE) and is represented by the equation below:

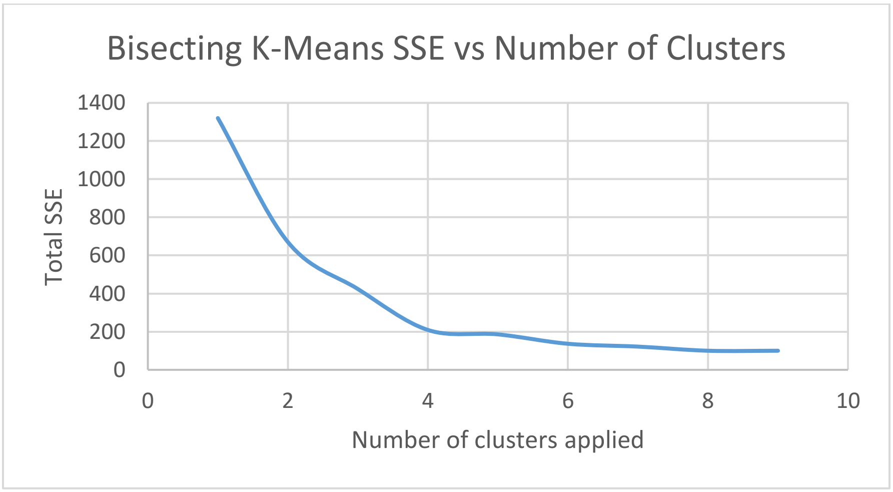
$$SSE = \sum_{i=1}^{n} (x_i - \bar{x})^2$$

The objective function that is optimized throughout the execution of the program is to maximum SSE among each cluster so that we can select and split that cluster.

It is also important to note the behavior of the algorithm as the number of clusters changed. By the nature of the calculations one would expect that as the number of clusters gets larger the SSE of the set is going to get smaller, but this relationship should only progress to a certain point after which the change in SSE should level as the number of clusters becomes larger. After running a trial of 9 iterations I was able to validate this relationship. The chart below shows the SSE for each run with the correlating number of clusters. We can also observe this relationship graphically in the graph shown.

Number of Clusters	SSE
1	1320.12
2	670.24
3	423.72
4	209
5	186.01
6	136.73
7	121.82
8	100.2
9	100.2

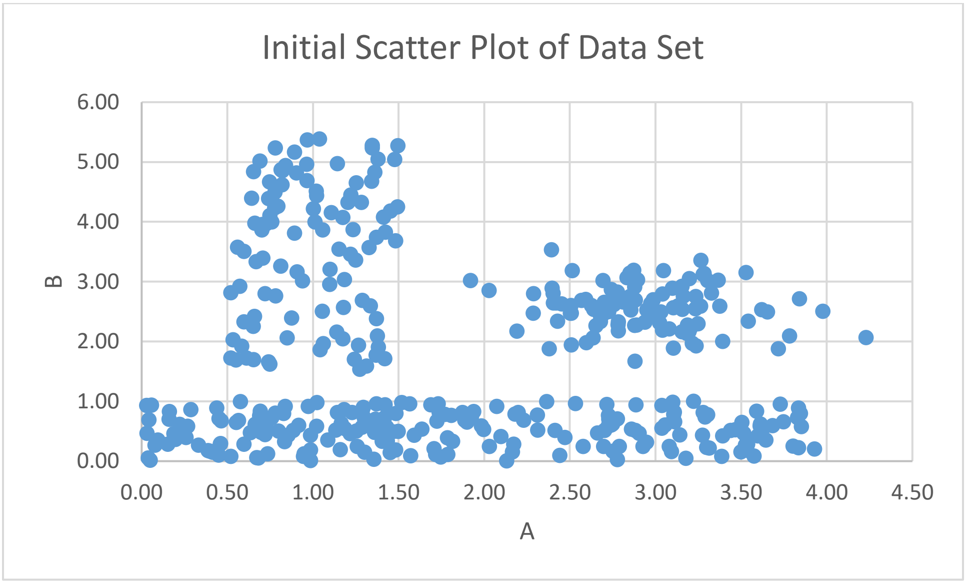
Table 1: Total SSE for the Bisecting K-means algorithm for cluster sizes 1-9



Graph 1: Plot of SSE vs number of Clusters for the Bisecting k-means algorithm

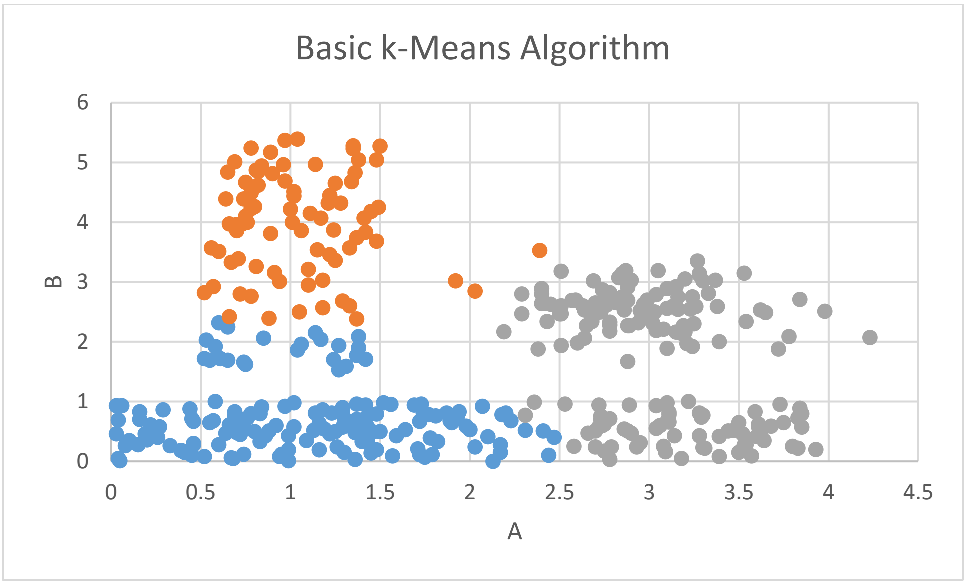
Results:

The results of this implementation are quite promising and give us some insight into how these different algorithms actually perform. To start I wanted to get an understanding of the initial data set to check if there were any apparent relationships that can help us in determining cluster size or other related parameters. From an initial scatter plot of the original data set we can see that there are three distinct groups within the data set already. This then allows us to test out a cluster number of three to verify whether or not these groups are unique.



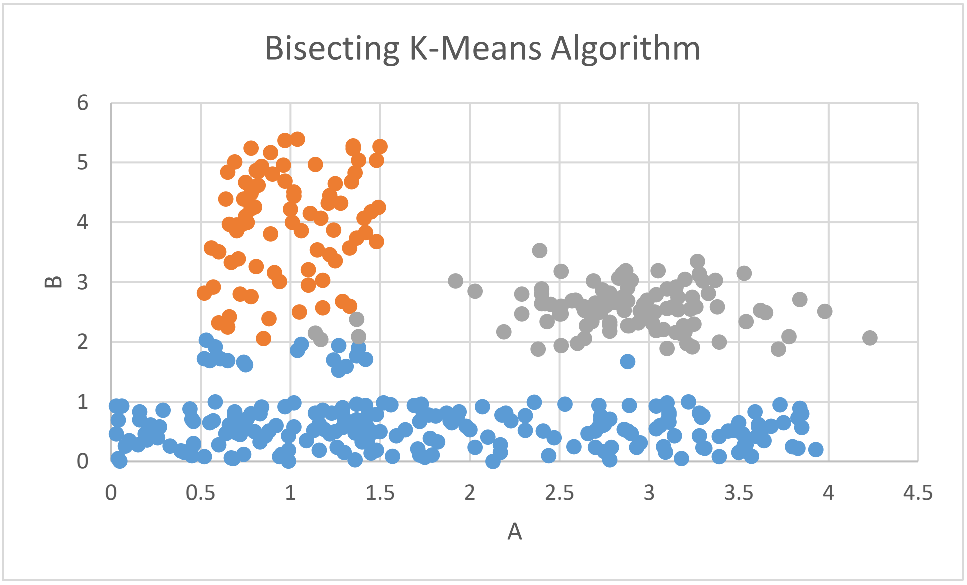
Graph 2: Initial plot of the data

Once I had complete the initial standard k-means algorithm I wanted to measure how well it would cluster the data set for future comparison against the bisecting K-Means algorithm. After running the algorithm and visualizing the results it becomes apparent that the algorithm does a poor job of accurately separating each individual grouping, and in the case of the test case show it actually split one of the major groupings. The performance of the standard K-means is definitely average, but it does identify chunks of the groupings. One reason for this is that the data set has rectangular characteristics, which the k-means algorithm is not particularly good at dealing with. As far as the SSE is concerned this trial returned an SSE of 381.80 after converging with 9 iterations. As I will mention briefly this is actually a better SSE than that of the bisecting k-means but basing information solely on this value can be misleading. We can see in the graph for the standard K-Means algorithm that each group was clustered in such a way that it makes sense to see the SSE be reduced to a smaller value. The graph on the following page shows the clustering results for the standard k-means algorithm.



Graph 3: Cluster results for the standard K-Means algorithm after 9 iterations with a total SSE of 381.80.

After the final implementation of the bisecting K-Means algorithm I was then able to test the performance against the originally plotted data and the plot produced by the standard k-means algorithm. A plot of the cluster results from a trial of the Bisection K-Means algorithm with three clusters can be seen in the graph below.



Graph 4: Cluster results of the Bisecting K-means algorithm with three clusters. This run produced an SSE of 423.72 after 2 iterations.

From viewing Graph 4 we can clearly see that the algorithm was much more effective at grouping the data into relevant clusters than the traditional k-means approach. The original structure as seen in the original plot holds and the data points were placed in the clusters that represent the relationships originally seen. As mentioned previously we can also see that the SSE obtained from the bisecting k-means algorithm is higher than that of the standard SSE. Without actually visualizing the data it might be considered that the standard algorithm is more effective, but after graphing the data we can see that this is a result of the traditional approach poorly clustering data points.

Based on the results obtained we can clearly see that the bisecting k-means algorithm is far more effective in clustering techniques. We can also see from the graph one that the data is best represented using a grouping three clusters. Anything past 3 and the relevance of the grouping begins to lose strength. Eventually the SSE does not change as the number of clusters no longer effects the resulting clusters. When choosing the optimal number of clusters, it is usually best to select the value where the SSE begins to level out. By selecting this value, we can be sure that the SSE is minimized but the results obtained still provide strong clusters.

When considering improvements, there are a few areas that my implementation can focus on. Once such area is in the selection of initial centroids. Currently this is done pseudo randomly with some slight enhancements but this method could be improved upon to always get better results. Towards the end of the programming the solid structure that was used to support the traditional k-means ended up begin problematic for the bisecting k-means, and as such the code could be restructured to optimize the bisecting k-means approach. This algorithm is not necessarily scalable to very large data sets, and would be a good candidate for parallel computing. There are also variations on the algorithm that incrementally update the clusters and averages. This can allow for enhanced results but at the cost of computing power and the introduction of new dependencies on the order of execution.

This project allowed me to learn the ins and outs of these basic clustering techniques and showed me the strength of more advanced algorithms. I hope to continue to learn more about these topics and look forward to the projects to come. In the following pages, you will find a listing of all the code used to implement the clustering algorithms.

Code Listing:

```
Author: Benjamin Fields
Program: K means clustering
Description: Program that reads in a data set and performs either standard k-means
clustering
or bisecting k-means clustering. The results are then written to a .CSV file for plotting
*******************************************************************************************
*******************/
#include <stdio.h>
#include<string>
#include "Point.h"
#include<vector>
#include "DataSet.h"
#include <stdlib.h>
#include <time.h>
#include<iostream>
#include<fstream>
#include<iomanip>


//used to populate arrays with the data set from external text files
void populateArrays(float *a, float *b) {
	FILE *inputFile;
	inputFile = fopen("InputData_A.txt", "r");
	if (inputFile == NULL) {
		throw "fileNotFound";
	}
	else {
		printf("FILE A FOUND\n");
		for (int i = 0; i < 400; i++) {
			fscanf(inputFile, "%f", &a[i]);
		}
		fclose(inputFile);
	}
	inputFile = fopen("InputData_B.txt", "r");
	if (inputFile == NULL) {
		throw "fileNotFound";
	}
	else {
		printf("FILE B FOUND\n");
		for (int i = 0; i < 400; i++) {
			fscanf(inputFile, "%f", &b[i]);
```

```
                }
                fclose(inputFile);
        }
}


//assign the data points to the point class for easier data management
 Point *createPoints(int size,float *a,float *b) {
        Point *myarray = new Point[size];
        for (int i = 0; i < size; i++)
        {
                myarray[i].setX(a[i]);
                myarray[i].setY(b[i]);
        }
        return myarray;
}


 //variation for Bisection k-means as only two centroids are
 //needed for each iteration and this version takes in the
 //cluster of focus instead of the whole set
 Point* createTwoCentroidsB(std::vector<Point> initCluster) {
         //find min and max for best centroids range
         float largestX = initCluster[0].getX();
         for (int i = 0; i < (int)initCluster.size(); i++)
         {
                 if (initCluster[i].getX() > largestX) {
                         largestX = initCluster[i].getX();
                 }
         }


         float largestY = initCluster[0].getY();
         for (int i = 0; i < (int)initCluster.size(); i++)
         {
                 if (initCluster[i].getY() > largestY) {
                         largestY = initCluster[i].getY();
                 }
         }


         float minX = initCluster[0].getX();
         for (int i = 0; i < (int)initCluster.size(); i++)
        {
                if (initCluster[i].getX() < minX) {
                        minX = initCluster[i].getX();
                }
        }
        float minY = initCluster[0].getY();
        for (int i = 0; i < (int)initCluster.size(); i++)
        {
                if (initCluster[i].getY() < minX) {
                        minX = initCluster[i].getY();
                }
        }



        Point *centroids = new Point[2];
        int index = 0;
        //loop till both centroids selected
        while (index < 2) {


                float x, y;
                x = minX + static_cast <float> (rand()) / (static_cast <float> (RAND_MAX /
(largestX - minX)));
                centroids[index].setX(x);


                y = minY + static_cast <float> (rand()) / (static_cast <float> (RAND_MAX /
(largestY - minY)));
                centroids[index].setY(y);


                if (index == 0) {
                        index++;
                        continue;
                }
                bool accept = true;
                //check each index against previous centroids to ensure unique values
                for (int i = 0; i < index; i++) {

                        if (abs(centroids[i].getX() - centroids[index].getX()) >
.0001&&abs(centroids[i].getY() - centroids[index].getY()) > .0001) {
```

```
                                continue;
                        }
                        else {
                                accept = false;
                                break;
                        }
                }
                if (accept == true) {
                        index++;
                }
        }
        return centroids;
}

//version for standard k-means to create centroids for
 //any number of clusters
Point* createCentroids(int num, DataSet &set) {
        Point *centroids = new Point[num];
        int index = 0;
        while (index < num) {

                //assign random floats between the mins and maxs of each data set
                float x, y;
                x = set.getMinX() + static_cast <float> (rand()) / (static_cast <float>
(RAND_MAX / (set.getMaxX() - set.getMinX())));
                centroids[index].setX(x);


                y = set.getMinY() + static_cast <float> (rand()) / (static_cast <float>
(RAND_MAX / (set.getMaxY() - set.getMinY()+.01)));
                centroids[index].setY(y);


                if (index == 0) {
                        index++;
                        continue;
                }
                bool accept = true;
                //compare each set against previous values to ensure unique set
                for (int i = 0; i < index; i++) {
```

```
			if (abs(centroids[i].getX() - centroids[index].getX()) >
.0001&&abs(centroids[i].getY() - centroids[index].getY()) > .0001) {

				continue;
			}
			else {
				accept = false;
				break;
			}
		}
		if (accept == true) {
			index++;
		}
	}
	return centroids;
}


//assign each point in the set to a correlating cluster based on euclidian distance
void assignPointsToCentroids(DataSet set,std::vector<std::vector<Point> > &myClusters,
Point* myCentroids,int numKs) {
	for (int j = 0; j < 400; j++)//j represents each point
	{
		int smallest = 0;
		for (int i = 0; i < numKs; i++)//i represent each cluster
		{
			//call the class function to calculate the euclidian distance to a
point and check if it is the smallest
			//if it is the smallest then assign it to tracking variable
			if (set.getPoint(j).EuclidianDistanceTo(myCentroids[i]) <
set.getPoint(j).EuclidianDistanceTo(myCentroids[smallest])) {
				smallest = i;
			}
		}

		set.assignCluster(j, smallest);

		myClusters[smallest].push_back(set.getPoint(j));
	}
}
```

```
//variation for bisecting algorithm that uses a single cluster instead of the whole data
set same logic as the k-means version
void assignPointsToCentroidsB(std::vector<Point> &initCluster,
std::vector<std::vector<Point> > &myClusters, Point* myCentroids, int numKs) {
        for (int j = 0; j < (int)initCluster.size(); j++)
        {
                int smallest = 0;
                for (int i = 0; i < numKs; i++)
                {
                        if (initCluster[j].EuclidianDistanceTo(myCentroids[i]) <
initCluster[j].EuclidianDistanceTo(myCentroids[smallest])) {
                                smallest = i;
                        }
                }

                initCluster[j].setCluster(smallest);

                myClusters[smallest].push_back(initCluster[j]);
        }
}


//calculate the averages of each cluster and assign them to an array
Point* CalculateNewAverages(std::vector<std::vector<Point> > &myClusters,int numKs) {
        Point *newCentroids = new Point[numKs];


        //loop through each element in each cluster and sum the values for calculating the
average
        for (int i = 0; i < numKs; i++) {
                float xSum = 0.0f;
                float ySum = 0.0f;
                for (int j = 0; j < (int)myClusters[i].size(); j++) {
                        xSum += myClusters[i][j].getX();
                        ySum += myClusters[i][j].getY();
                }
                int sizeofCluster = myClusters[i].size();
                if (sizeofCluster == 0)sizeofCluster = 1;
                float averageX = xSum / (float)sizeofCluster;
                float averagey = ySum / (float)sizeofCluster;
                newCentroids[i].setX(averageX);
                newCentroids[i].setY(averagey);
        }
        return newCentroids;
}


//function to calcualte the SSE for each cluster and assign the values to an array
float *calcSSEofClusters(std::vector<std::vector<Point> > &myClusters,Point *newCentroids,
int numKs) {
        float *SSEofClusters = new float[numKs];
        for (int i = 0; i < numKs; i++) {
                float sum = 0.0;
                for (int j = 0; j < (int)myClusters[i].size(); j++) {
                        sum +=
(myClusters[i][j].EuclidianDistanceTo(newCentroids[i]))*(myClusters[i][j].EuclidianDistance
To(newCentroids[i]));
                }
                SSEofClusters[i] = sum;
        }


        return SSEofClusters;
}


//loop throught the array of SSEs and sum them for the total SSE
float calculateTotalSSE(float *SSEofClusters, int numKs) {
        float total = 0.0;
        for (int i = 0; i < numKs; i++) {
                total += SSEofClusters[i];
        }


        return total;
}


//write perforance data to the .csv file for visual plotting
void writeToFile(std::vector< std::vector<Point> > &myClusters, int numKs,int
iteration,float * SSE,float totSSE) {
        std::ofstream myFile;
        myFile.open("Results.csv");
        myFile << "Clustering Results after "<<iteration<<" iterations:\n\n";
        for (int i = 0; i < (int)myClusters.size(); i++) {
                myFile << "SSE of cluster ," << i + 1 << " : " << SSE[i] << "\n";
        }
        myFile << "Total SSE: " << totSSE << "\n";
        for (int i = 0; i < numKs; i++) {
                myFile << "Cluster: " << i + 1 << "\n";
                myFile << "Data:\n";
                for (int j = 0; j < (int)myClusters[i].size(); j++) {
                        myFile << myClusters[i][j].getX() << ",";
                }
                myFile << "\n";
                for (int j = 0; j < (int)myClusters[i].size(); j++) {
                        myFile << myClusters[i][j].getY() << ",";
                }
                myFile << "\n";
        }
        myFile.close();


}



//looping logic for the standard k-means algorithm
//terminates once the averages no longer change
std::vector< std::vector<Point> > loopKmeans(DataSet set, int numKs, Point
*initialCentroids,int &enditeration){
        int iteration = 1;//track how many iteration
        bool stop = false;
        std::vector< std::vector<Point> > myClusters(numKs);//vector used for represnet each
cluster

        //loop till converged
        while (stop == false) {
                iteration++;
                assignPointsToCentroids(set, myClusters, initialCentroids, numKs);
                Point *newCentroids = CalculateNewAverages(myClusters, numKs);
                float *SSEofClusters = calcSSEofClusters(myClusters, newCentroids, numKs);
                float totSSE = calculateTotalSSE(SSEofClusters, numKs);
                printf("\niteration %d\n", iteration);//print to cmd line for tracking
                for (int i = 0; i < numKs; i++) {
                        printf("The SSE of cluster %d is %.2f\n", i + 1, SSEofClusters[i]);
                }
                printf("The total SSE is %.2f\n", totSSE);
                stop = true;
                //check each new average to see if the values are no longer changing
                for (int i = 0; i < numKs; i++) {
                        if (newCentroids[i].EuclidianDistanceTo(initialCentroids[i]) < .0001)
{
                                continue;
                        }
                        else {
                                stop = false;
                        }
                }
                //if loop is to continue clear data for next iteration
                if (stop == false) {
                        for (int i = 0; i < numKs; i++) {
                                initialCentroids[i] = newCentroids[i];
                                myClusters[i].clear();
                        }
                }
        }

        printf("Kmeans has converged afer %d iterations\n", iteration);
        enditeration = iteration;
        return myClusters;
}


//bisecting version of looping k_means logic to use a single cluster instead of the entire
data set.
//logic holds between versions
std::vector< std::vector<Point> > loopKmeansB(std::vector<Point> init, int numKs, Point
*initialCentroids) {
        int iteration = 1;
        bool stop = false;
        std::vector< std::vector<Point> > myClusters(numKs);

        while (stop == false) {
                iteration++;
                assignPointsToCentroidsB(init, myClusters, initialCentroids, numKs);
                Point *newCentroids = CalculateNewAverages(myClusters, numKs);
                float *SSEofClusters = calcSSEofClusters(myClusters, newCentroids, numKs);
                float totSSE = calculateTotalSSE(SSEofClusters, numKs);

                stop = true;
                for (int i = 0; i < numKs; i++) {
                        if (newCentroids[i].EuclidianDistanceTo(initialCentroids[i]) < .0001)
{
                                continue;
                        }
                        else {
                                stop = false;
                        }
                }
                if (stop == false) {
                        for (int i = 0; i < numKs; i++) {
                                initialCentroids[i] = newCentroids[i];
                                myClusters[i].clear();
                        }
                }
        }
        printf("Kmeans has converged afer %d iterations\n", iteration);

        return myClusters;
}


//initial kmeans logic used to set centroids and launch looping iterations
void kmeans(DataSet set, int numKs) {
        printf("\nIteration 1\n");
        std::vector< std::vector<Point> > myClusters(numKs);
        Point* myCentroids = createCentroids(numKs, set);
        assignPointsToCentroids(set, myClusters, myCentroids, numKs);
        Point *newCentroids = CalculateNewAverages(myClusters, numKs);
        float *SSEofClusters = calcSSEofClusters(myClusters,newCentroids,numKs);
        float totSSE = calculateTotalSSE(SSEofClusters,numKs);
        for (int i = 0; i < numKs; i++) {
                printf("The SSE of cluster %d is %.2f\n", i + 1, SSEofClusters[i]);
        }
        printf("The total SSE is %.2f\n", totSSE);
        int count;
        myClusters = loopKmeans(set, numKs, newCentroids,count);
        newCentroids = CalculateNewAverages(myClusters, numKs);
        SSEofClusters = calcSSEofClusters(myClusters, newCentroids, numKs);
        totSSE = calculateTotalSSE(SSEofClusters, numKs);
        //writeToFile(myClusters, numKs, count, SSEofClusters, totSSE);

}
```

```
//bisecting version used to lauch the kmeans logic that uses a single cluster instead of
the
//entire data set
void kmeansB(int numKs, std::vector<Point> &InitCluster, int posRemove, std::vector<
std::vector<Point> > &myClusters) {
	printf("\nIteration 1\n");
	std::vector< std::vector<Point> > newClusters(2);
	Point* myCentroids = createTwoCentroidsB(InitCluster);
	assignPointsToCentroidsB(InitCluster, newClusters, myCentroids, 2);
	Point *newCentroids = CalculateNewAverages(newClusters, 2);
	float *SSEofClusters = calcSSEofClusters(newClusters, newCentroids, 2);
	float totSSE = calculateTotalSSE(SSEofClusters, 2);
	for (int i = 0; i < numKs; i++) {
		printf("The SSE of cluster %d is %.2f\n", i + 1, SSEofClusters[i]);
	}
	printf("The total SSE is %.2f\n", totSSE);
	newClusters = loopKmeansB(InitCluster, numKs, newCentroids);


	//remove the selected cluster and replace with new
	myClusters.erase(myClusters.begin() + posRemove);
	for (int i = 0; i < 2; i++) {


		myClusters.push_back(newClusters[i]);
	}

}


//initial logic to begin the bisecting k-means algorithm
//create first cluster then select the first one to be split
//call k- means logic for algorithm
std::vector< std::vector<Point> > bisectingKmeans(DataSet &set,int numKs) {
	std::vector< std::vector<Point> > myClusters(1);


	for (int i = 0; i < 400; i++) {
		myClusters[0].push_back(set.getPoint(i));
	}
	int posRemove = 0;
	printf("Cluster size is: %d\n", myClusters.size());
	bool stop = false;
	int iteration = 0;
	while ((int)myClusters.size() < numKs) {
		iteration++;
		kmeansB(2, myClusters[posRemove],posRemove,myClusters);//k-means logic
		Point *averages = CalculateNewAverages(myClusters, myClusters.size());
		float *SSEofClusters = calcSSEofClusters(myClusters, averages,
myClusters.size());

		//select the biggest SSE
		int biggest = 0;
		for (int i = 0; i < (int)myClusters.size(); i++) {
			if (SSEofClusters[i] > SSEofClusters[biggest]) {
				biggest = i;
			}
		}
		posRemove = biggest;//identify the largest SSe cluster to be used in next
split

	}
	//print out performance results to cmd line
	Point *averages = CalculateNewAverages(myClusters, myClusters.size());
	float *SSEofClusters = calcSSEofClusters(myClusters, averages, myClusters.size());
	float totSSE = calculateTotalSSE(SSEofClusters, myClusters.size());
	printf("Bisecting K-means complete with %d clusters after %d iterations\n\n",
myClusters.size(),iteration);
	for (int i = 0; i < (int)myClusters.size(); i++) {
		printf("The SSE of cluster %d is %f\n", i + 1, SSEofClusters[i]);
	}
	printf("The total SSE is : %f\n", totSSE);
	writeToFile(myClusters, numKs,iteration, SSEofClusters,totSSE);//write to external
file for plotting
	return myClusters;
}


//main entry point to program
//creates the initial data
//and calls either the standard k-means
//or the bisecting k-means
int main(int argc, char **argv) {
	unsigned int seed = (unsigned int)time(0);
	srand(seed);
	float *a,*b;
	a = (float*)malloc(400 * sizeof(float));
	b = (float*)malloc(400 * sizeof(float));//allocate memory for the data
	try {
		populateArrays(a, b);//fill array
	}
	catch (std::string) {
		printf("DATA FILE COULD NOT BE FOUND!\n");
		exit(EXIT_FAILURE);
	}

	Point *myPoints= createPoints(400, a, b);//create the points from data
	DataSet mySet(myPoints, 400);//create the complete set for data analysis


	//kmeans(mySet, 3);
	for(int i = 0;i<10;i++)//loop through each number of clusters on the bisecting k-
means
	std::vector< std::vector<Point> >myClusters = bisectingKmeans(mySet, i);

	//free the data memory
	free(a);
	free(b);


	system("PAUSE");
	return 0;
}
/*******************************************************************************************
**************************
Author: Benjamin Fields
file :header file for point class
Description :contains structure for class
*******************************************************************************************
******************** */
#ifndef POINT_H
#define POINT_H


class Point {
	private:
		float m_X;
```

```
                float m_Y;
                int cluster;
        public:
                Point();
                Point(float x, float y);
                void setX(float x);
                void setY(float y);
                float getX();
                float getY();
                float EuclidianDistanceTo(Point p);
                void setCluster(int num);
                int getCluster();
};
#endif
Author: Benjamin Fields
file :point implementation file for functions
Description :contains functions to act on a point
*****************************************************************************************
******************* */
#include "Point.h"
#include<cmath>


//constructor
Point::Point() {
        setX(0.0);
        setY(0.0);
        setCluster(0);
}


//contructor used to create from data
Point::Point(float x, float y) {
        setX(x);
        setY(y);
        setCluster(0);

}


float Point::getX() {
        return m_X;
}
```

```
float Point::getY() {
        return m_Y;
}


void Point::setX(float x) {
        m_X = x;
}
void Point::setY(float y) {
        m_Y = y;
}


//function used to return the euclidian distance from calling point to passed point
float Point::EuclidianDistanceTo(Point p) {
        float xterm = (this->getX() - p.getX())*(this->getX() - p.getX());
        float yterm = (this->getY() - p.getY())*(this->getY() - p.getY());


        return sqrt(xterm + yterm);
}


void Point::setCluster(int num) {
        cluster = num;
}


int Point::getCluster() {
        return cluster;
}
Author: Benjamin Fields
file :header file for data set class
Description :contains structure for class
*******************************************************************************************
******************* */
#ifndef DATASET_H
#define DATASET_H
#include "Point.h"


class DataSet {
        private:
                Point *mySet;
        public:
                DataSet(Point *array, int size);
                void PrintSet();
                ~DataSet();
                float getMaxX();
                float getMaxY();
                float getMinX();
                float getMinY();
                Point getPoint(int num);
                void assignCluster(int element, int cluster);
};


#endif
Author: Benjamin Fields
        file :data set implementation file for functions
        Description :contains functions to act on data set
***********************************************************************************************
******************* */
#include "DataSet.h"
#include<stdio.h>


//constructor
DataSet::DataSet(Point *array, int size) {
        mySet = new Point[size];
        for (int i = 0; i < size; i++) {
                mySet[i] = array[i];
        }
}
//print points
void DataSet::PrintSet() {
        for (int i = 0; i < 400; ++i) {
                printf("Set Point %d: X = %.2f  Y = %.2f\n", i + 1, mySet[i].getX(),
mySet[i].getY());
        }
}


//get max value from a
float DataSet::getMaxX() {
        float largest = mySet[0].getX();
        for (int i = 0; i < 400; i++)
        {
                if (mySet[i].getX() > largest) {
                        largest = mySet[i].getX();
                }
        }
        return largest;
}


//get max from b
float DataSet::getMaxY() {
        float largest = mySet[0].getY();
        for (int i = 0; i < 400; i++)
        {
                if (mySet[i].getY() > largest) {
                        largest = mySet[i].getY();
                }
        }
        return largest;
}
//get minimum value for a
float DataSet::getMinX() {
        float smallest = mySet[0].getX();
        for (int i = 0; i < 400; i++)
        {
                if (mySet[i].getX() < smallest) {
                        smallest = mySet[i].getX();
                }
        }
        return smallest;
}
//get min value for b
float DataSet::getMinY() {
        float smallest = mySet[0].getY();
        for (int i = 0; i < 400; i++)
        {
                if (mySet[i].getY() < smallest) {
                        smallest = mySet[i].getY();
                }
        }
        return smallest;
}
```

```
//return a point to use
Point DataSet::getPoint(int num) {
	return mySet[num];
}


//assign a cluster to a point for tracking
void DataSet::assignCluster(int element, int cluster) {
	mySet[element].setCluster(cluster);
}


//destructor
DataSet::~DataSet() {

}
```

References:

Tan, Pang-Ning, Michael Steinbach, and Vipin Kumar. *Introduction to Data Mining*. Boston: Pearson Addison Wesley, 2005. Print.