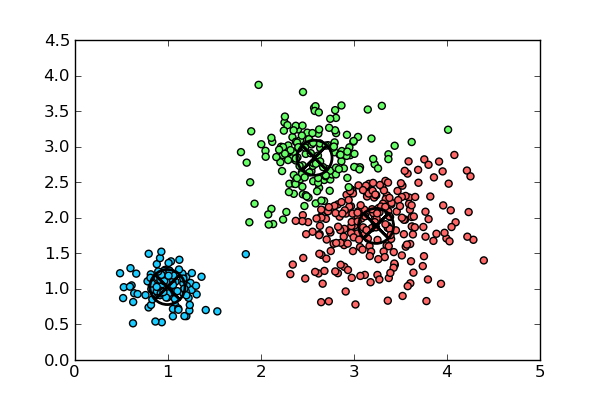
K-Means Clustering

Assignment #1



by Jihoon Oh

TAble of  
Contents

Report Questions 1

Randomly Generated Centroids 1

Manually Picking Centroids 2

Important Details 3

Table of Centroids, IV, EV, and IV/EV 4

Discussion of Results 6

# Report Questions

## Randomly generated centroids

*“Explain how you randomly generate the starting centroids in step 2.”*

When generating the centroids randomly, there were two options given:

1. Generate a “truly random” value that exists between the min and max of that property.

For example, if the parameters of the area given in the data set were between 10 and 21, generate a random value between 10 and 21 for the area property of the centroid.

1. For every property of the centroid, pick a random data point and use that as the starting centroid.

For example, the area of the centroid is the area of row 21, the perimeter of the centroid is the perimeter of row 57, etc.

It is more logical to choose the latter method of generating the initial centroid because if the data set has a heavy bias towards a specific range, it will create the centroids and clusters based on that bias. In other words, it is dynamically adjusting the centroids based on the actual data points, instead of doing it later in the program.

## Manually picking centroids

*“Explain how you manually pick the starting centroids in step 6.”*

|  |  |  |
| --- | --- | --- |
|  | Starting | Update once |
| IV | 51812.2213 | 689.9145 |
| EV | 0 | 0 |
| IV/EV | Infinity | Infinity |

Out of sheer curiosity, I chose the three starting centroids to be 0 for all the dimensions of each centroid. Not surprisingly, code terminated prematurely. Because every cluster started with the same centroids, every data point was assigned to the same cluster, resulting the code to update the centroids once and terminate.

It is also important to note the values of IV, EV, and IV/EV in this situation. Since every data point was assigned to the same cluster in the beginning (before the update), the IV had a value of ~51812, which is an extremely high value for this data set. After the first update, the IV has a value of ~689, which is more reasonable and realistic.

The EV has a value of 0 for both before and after the update, and in result of that the IV/EV is infinite for both before and after the update as well. IV/EV has a value of “Infinity” because Java returns “Infinity” when dividing by a double that has a value of zero. The reasoning behind EV being zero is a little more interesting to analyze.

By definition, EV measures the variability of every data point in distinct clusters. In this case, since every data point is in the same cluster, there is no variability between clusters. Therefore, the EV holds a value of zero.

## Important details

*“Describe any details of your implementation that might be important.”*

Although time complexity wasn’t much of an issue for this assignment, I tried to achieve the perfect balance between time efficiency and space efficiency. To reduce the time complexity on most of the methods, I’ve utilized three arrays of the following:

1. **double[][] dataSet**

To prevent opening and reading the same data file repeatedly, I’ve used a two dimensional array to represent the data set, which makes the program more time-efficient. Rows and columns are represented the same way it’s represented in the data set: rows represent each data point and columns represent the dimensions of the data (area, perimeter, etc.)

1. **double[][] centroidSet**

Due to the nature of the data set having 7 dimensions, it is crucial to use a two dimensional array to represent each centroid. The representation of rows and columns is similar to that of the data set. Each row represents a centroid that belongs to a distinct cluster and each column represents the dimensions (area, perimeter, etc.)

1. **ArrayList<Integer>[] clusterSet**

As opposed to using a static two dimensional array for the cluster set, I wanted to use a more dynamic data structure because the program required the cluster set to constantly change its members. This is why I chose to use an array of array lists.

The index represents a distinct cluster; Hence the array is of size three. Furthermore, each array element (or cluster) contains the row numbers of its members. For example, if cluster 0 has data points 3, 25, and 112, index 0 of clusterSet would have an array list of size three, containing 3, 25, and 112.

Another important detail to note is the use of the variable “row”. In most of the for-loops in the program, “row” is used to represent each data point. Therefore, if one were to iterate through all the data points in a cluster, it would’ve looked like this:

*for(int row : clusterSet[cluster]) {*

*// Do something*

*}*

## Table of centroids, iv, ev, and iv/ev

*“Report all sets of starting centroids, final centroids, IV, EV, and IV/EV in a table.”*

The following pages contain the tables required.

## final iv, ev, and iv/ev table:

|  |  |  |  |
| --- | --- | --- | --- |
| Trial | #1 | #2 | #3 |
| Final IV | 313.7343 | 313.2167 | 313.7343 |
| Final EV | 389.3652 | 390.1334 | 389.3652 |
| Final IV/EV | 0.8058 | 0.8028 | 0.8058 |

## Initial Centroid table (first Trial):

|  |  |  |  |
| --- | --- | --- | --- |
| Centroid | #1 | #2 | #3 |
| Area | 20.16 | 11.19 | 11.19 |
| Perimeter | 13.0 | 13.41 | 16.23 |
| Compact | 0.8558 | 0.8891 | 0.8859 |
| Length of Kernel | 5.108 | 5.388 | 5.585 |
| Width of Kernel | 3.512 | 3.684 | 3.796 |
| Asymmetry | 3.128 | 4.217 | 3.252 |
| Groove Length | 5.185 | 6.238 | 5.049 |

## Final centroid Table (first trial):

|  |  |  |  |
| --- | --- | --- | --- |
| Centroid | #1 | #2 | #3 |
| Area | 18.72 | 11.99 | 14.82 |
| Perimeter | 16.30 | 13.28 | 14.54 |
| Compact | 0.8851 | 0.8527 | 0.8805 |
| Length of Kernel | 6.209 | 5.227 | 5.591 |
| Width of Kernel | 3.723 | 2.880 | 3.299 |
| Asymmetry | 3.604 | 4.584 | 2.707 |
| Groove Length | 6.066 | 5.074 | 5.218 |

Please note that for this section, I was only able to make the initial and final centroids table for the first trial because there is too much data to write down for all five trials. However, I have kept track of which starting centroids produce which IV, EV, and IV/EV, which will be discussed in the next section.

## discussion of results

As you can see in the first table with the IV, EV, and IV/EV values, the results actually repeat between two options for each parameter. This is particularly interesting because, for example, IV would only produce 313.7343 or 313.2167 even when the starting centroid values are completely random.

One peculiar result to note is how, on a rare occasion, IV would return 421.819. When this happens, there is always one cluster that existed in the beginning, but not in the end. In other words, the maximum IV for my algorithm is reached when there are only two clusters in the end, even though K = 3. This makes sense to a certain degree because having two clusters results in a higher sum between the individual points and its centroid, as opposed to having three clusters. Admittedly, I have yet to figure out why this anomaly happens in the first place.

When the program isn’t spitting anomalies, it shows a pattern with respect to the starting centroids and the final IV, EV, and IV/EV values. Although the final values for IV/EV were either 0.8058 or 0.8028, there was a pattern between which starting centroids produced these values. It turns out, the sparser the initial centroids, the lesser the IV. Because the task of K-Means Clustering is to minimize IV/EV, this is a very crucial piece of observation to take note of.

With this in mind, it should be obvious that the starting centroids should not be randomly chosen. To construct a program that performs this clustering at the highest level of efficiency, it is necessary to design the initial centroid choosing method in such a way that they are scattered as much as possible.