K MEANS: REPORT

Algorithm

The *K*-means clustering algorithm uses iterative refinement to produce a final result. The algorithm inputs are the number of clusters *K* and the data set. The data set is a collection of features for each data point. The algorithms starts with initial estimates for the *K* centroids, which can either be randomly generated or randomly selected from the data set. The algorithm then iterates between two steps:

1. Data assignment step:

   Each centroid defines one of the clusters. In this step, each data point is assigned to its nearest centroid, based on the squared Euclidean distance. More formally, if c_i is the collection of centroids in set *C*, then each data point *x i*s assigned to a cluster based on

   $$\underset{c_i \in C}{\arg\min} \; dist(c_i, x)^2$$

   where *dist*(·) is the standard (L_2) Euclidean distance. Let the set of data point assignments for each i^{th} cluster centroid be S_i.

2. Centroid update step:

   In this step, the centroids are recomputed. This is done by taking the mean of all data points assigned to that centroid's cluster.

   $$c_i = \frac{1}{|S_i|} \sum_{x_i \in S_i} x_i$$

3. The algorithm iterates between steps one and two until a stopping criteria is met (i.e., no data points change clusters, the sum of the distances is minimized, or some maximum number of iterations is reached).

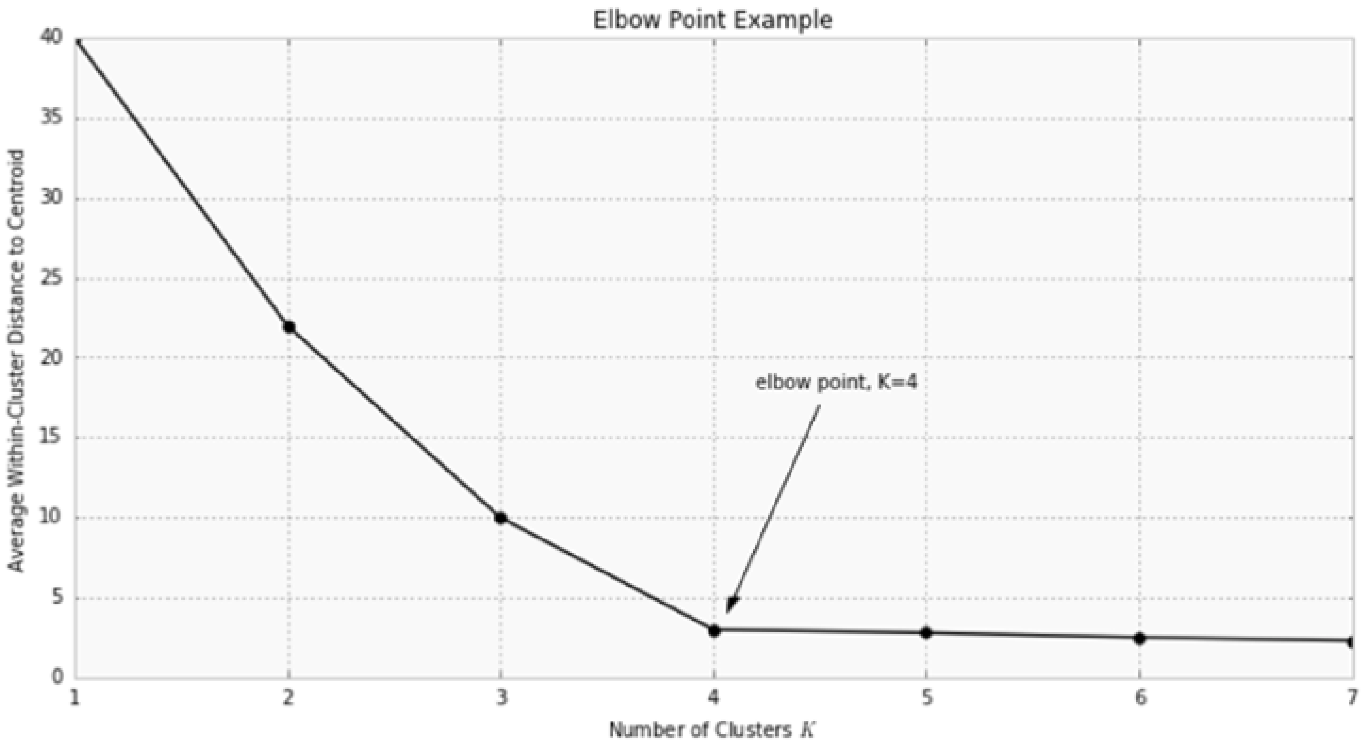
This algorithm is guaranteed to converge to a result. The result may be a local optimum (i.e. not necessarily the best possible outcome), meaning that assessing more than one run of the algorithm with randomized starting centroids may give a better outcome.

Choosing *K*

The algorithm described above finds the clusters and data set labels for a particular pre-chosen *K*. To find the number of clusters in the data, the user needs to run the *K*-means clustering algorithm for a range of *K* values and compare the results. In general, there is no method for determining exact value of *K*, but an accurate estimate can be obtained using the following techniques.

One of the metrics that is commonly used to compare results across different values of *K i*s the mean distance between data points and their cluster centroid. Since increasing the number of clusters will always reduce the distance to data points, increasing *K* will *always* decrease this metric, to the extreme of reaching zero when *K* is the same as the number of data points. Thus, this metric cannot be used as the sole target. Instead, mean distance to the centroid as a function of *K* s plotted and the "elbow point," where the rate of decrease sharply shifts, can be used to roughly determine *K*.

A number of other techniques exist for validating *K*, including cross-validation, information criteria, the information theoretic jump method, the silhouette method, and the G-means algorithm. In addition, monitoring the distribution of data points across groups provides insight into how the algorithm is splitting the data for each *K*.



Example: Applying K-Means Clustering to Data

As an example, I will demonstrate how the *K*-means algorithm works with a sample dataset (location : “/example/input.txt”). For the sake of simplicity, we'll only be looking at the two attributes.

Step 1: Clean and Transform Your Data

For this example, we've already cleaned and completed some simple data transformations.

This involves all the pre-processing to get the data into a format in which the program can work with. Includes strip commas, new line .

Step 2: Choose K and Run the Algorithm

The K value as well as input (text file contains dataset) should be specified in calling argument to the program

Sample:

If using stack use guideline to generate run command arguments:

stack exec k-means-exe /dir/to/dataset NumOfClusters /*dir*/to/store/output

`Note:`

NumOfClusters should be of type integer and greater then 3

Snippet of program showing actual implementation

Sample : Shows request for input of k when call program from command line

```
-- User-friendly version - to be called from GHCI or WinGHCI
kmeans :: String -> Int -> String -> IO ()
kmeans inFilename k outFilename = withArgs [inFilename, (show k), outFilename] main
```

Step 3: Review the Results

The program will output the results to the directory specified as the third argument when calling the program.

All the data will be classified into columns (clusters) specified in calling argument.

Example:

Cluster 1:	Cluster 2:	Cluster 3:
1317.67 7518.46	1202.28 -3753.14	1208.23 -5435.23
1317.11 7521.37	1198.98 -3747.28	1270.31 -5418.92
1317.11 7521.37	1199.21 -3741.35	1253.48 -5427.2
1316.49 7530.07	1195.19 -3729.51	1266.66 -5569.08
1320.08 7538.74	1195.19 -3729.51	1267.48 -5693.0

Step 4: Iterate Over Several Values of K

To do this call the program with different value of k and compare results

APPENDIX

Overview

Implementation of a simple but powerful clustering algorithm for 2D ('x' 'y') points . Simple and optimal for most data structures that are crawled iteratively, large list searches are avoided, some calculations are strict, and so on. Additionally, the total time of program operation (including IO operations, input parsing and output formatting), and how many iterations the algorithm has completed is measured. Finally, an intra-cluster sum of squares is calculated - the sum of the squares of the distances between each point in a cluster and the cluster center (for all clusters in total). This amount actually tries to minimize the algorithm, but may not reach a global minimum depending on the selected start centers (NP-hard is the task of finding a global minimum). For convenience, the minimum reached is displayed on the standard output, along with the number of iterations and program runtime.

Arguments are not validated, only their number is tracked (at least three, all after the third are ignored). The only validation of input data is whether the number of desired clusters does not exceed the amount of data.

The file [`input.txt`](./input.txt) serves as an example input and contains 1249 points without any particular internal structure

Runtime

On my non-dedicated laptop clustering in 4 clusters takes an average of ~ 0.3 seconds and in 10 clusters - about 0.7 seconds.

Execution

If you run into errors with stack consider compile and running the code from the command line manually.

Command: ghc main.hs