K-Means Clustering

Isaac G Callison

Middle Tennessee State University

Computer Science 5350: Introduction to Artificial Intelligence

Fall Semester 2019

**Introduction**

The purpose of this report is to discuss the concepts and practical implementation of a K-Means Clustering algorithm to classify simple data-sets. The K-Means algorithm is a relatively simple unsupervised machine learning algorithm used to classify data points. For purposes of this paper two data-sets will be used, the Iris and Cancer data-sets. As stated, K-Means is unsupervised, that means that the algorithm does not have the benefit classification labels when classifying data into different groups. The basic idea is grasped more readily if we think of a two-dimensional space where data points are graphed. If all the data points in the data-set are graphed in an X and Y plane, then we can imagine a two-dimensional space covered in dots. Similar data-points, which may likely represent the same classification, are likely to be graphed close to each other. So assuming that data-points with the same classification labels are likely to be clustered around each other, how do we identify and take advantage of this clustering? Also, if we are dealing with more attributes, we are dealing more dimensions than could be represented in a two-dimensional graph. For the Iris data-set, there are four attributes and thus four-dimensions.

The first step to clustering data for say, the Iris data-set, is to choose how many clusters we want. Much ink has been spilled to determine the right number of clusters to use with a given data-set. Typically, with a low number of clusters, the results are very poor. After all, how can you effectively distribute three different Irises into a single cluster in any sensible way? As the number of centroids, or cluster points, increase, we see an increase in accuracy. At some point, if graphed this will show a sharp acceleration in accuracy before leveling off. Going back to the Iris example, lets say we choose three centroids to start with. These are just some randomly chosen points in some n-dimensional space. We can choose these points by selecting one of our data points and making it into a centroid. So we have three randomly selected data-points that we have labeled centroids. The next step is to calculate the distance between every data-point and these randomly chosen centroids. We then “cluster” the data-point with that particular centroid. In practicality, by clustering the point with a centroid we are just adding it to a list of points associated with that particular centroid. Once the distances have been measured and the points associated with the closest centroid or cluster point, the mean of all the points in each cluster is calculated and the centroid is moved to match the mean. All the distances of each point are then recalculated to see if any points have moved closer to a different centroid. After a few iterations each of the centroid’s locations should have stopped shifting and each of the points will be firmly associated with a particular centroid. In each cluster, a majority of class labels can be tallied to see what class label predominates. Untested data-sets with similar attributes can be run through a similar process to see which centroid they are closest to and what class label they are likely to possess.

**Program Basics & Implementation**

The program created for this project was written in Python 3. There were four command line arguments that the program took. The first was a seed value for random number generation. The second was the number of clusters that the program was to work with. The last two arguments were a randomly shuffled training and testing data-set from either the Iris or Cancer sets. To make data manipulation easier, numpy and pandas were used at various points. It was relatively easy for instance to load the testing and training data-sets into panda dataframes and extract various attributes. The class labels were stripped off for later use. The algorithm is unsupervised so class labels were not used for any classification. Because the centroid is basically the heart of this algorithm, I made a class to instantiate a centroid object with its own attributes. Some number of centroid objects, based upon the second command line argument, were created. A random number generator chose a random line from the training data-set to make each centroid. These were returned to the main function in a list. A fairly straightforward while loop iterated and classified each point with each centroid based on distance. The numpy module has a really useful function called linalg.norm() that made calculating the euclidean distance between each point and each centroid very easy and accurate. There was a check in the while loop to see if the centroids were continuing to shift. If they stopped shifting then a boolean was switched to true and the while loop stopped. Otherwise, an update position function was called and each point was measured against each centroid again, followed by a measure of the mean for the list of data-points and an updating of the centroid’s position.

After the centroids positions were finalized, a majority vote was taken among each of the centroid’s list of associated data points. Each centroid object has an attribute called “name” and this was assigned the majority label for later reference.

The testing set was also imported as a panda dataframe. Classifying was a simple. Each data-point was measured against each centroid in terms of closest distance. If distance indicated that it should be clustered with some particular centroid, then that centroid’s name and the class label of the testing data-point were compared. If they matched, then it was counted as correct, if they did not match then it was counted as incorrect. The total number of correct were then outputted to stdout.

**Results**