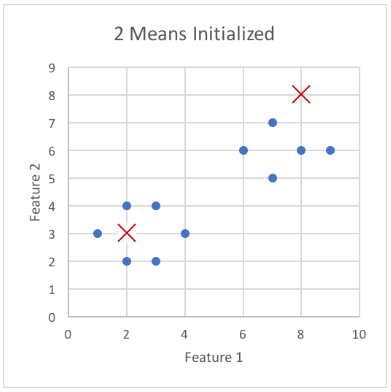
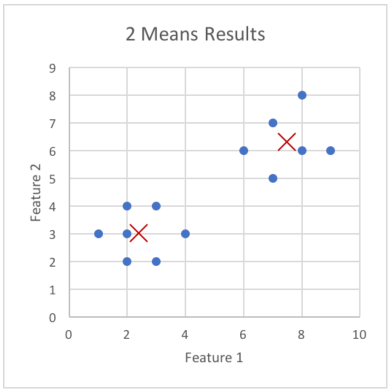
Katherine Archer
CSCI 4350
OLA-4
December 5, 2017

Unsupervised Learning with K-means Clustering

Unsupervised learning is a learning algorithm that tries to extrapolate correlations and patterns from unclassified data. Through the method of k-means, the algorithm attempts to do this by finding clusters of data that share common attributes, inferring that a given cluster is of the same classification since the data has seemingly grouped together in a given instance. A very simple example of k-means in action is below:



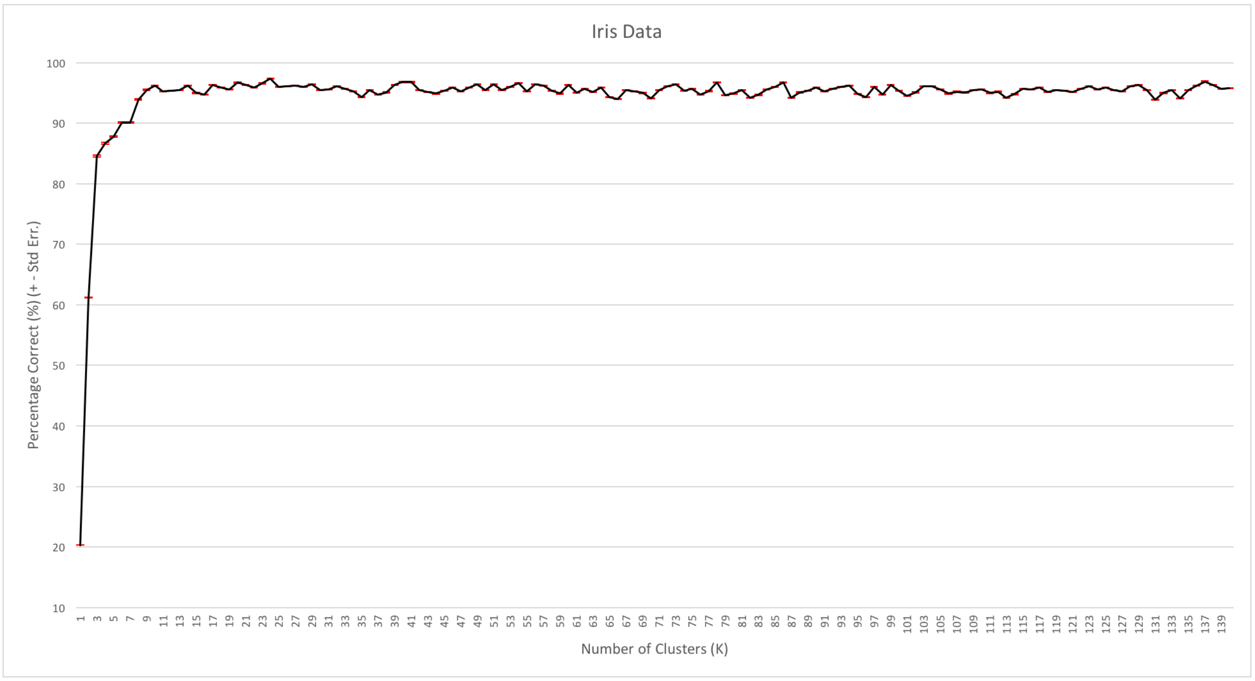


In this example, the data set has 2 features to evaluate, and it is easy to see that there are two apparent clusters. How the algorithm determines these cluster centers starts with initializing the mean points at random locations within the data set (indicated by red X's). The means then go through a series of iterations, updating their location based on their closest associated points and the mean of those points. When the means have ceased updating, this indicates that the means have found the center of the clusters they were seeking.

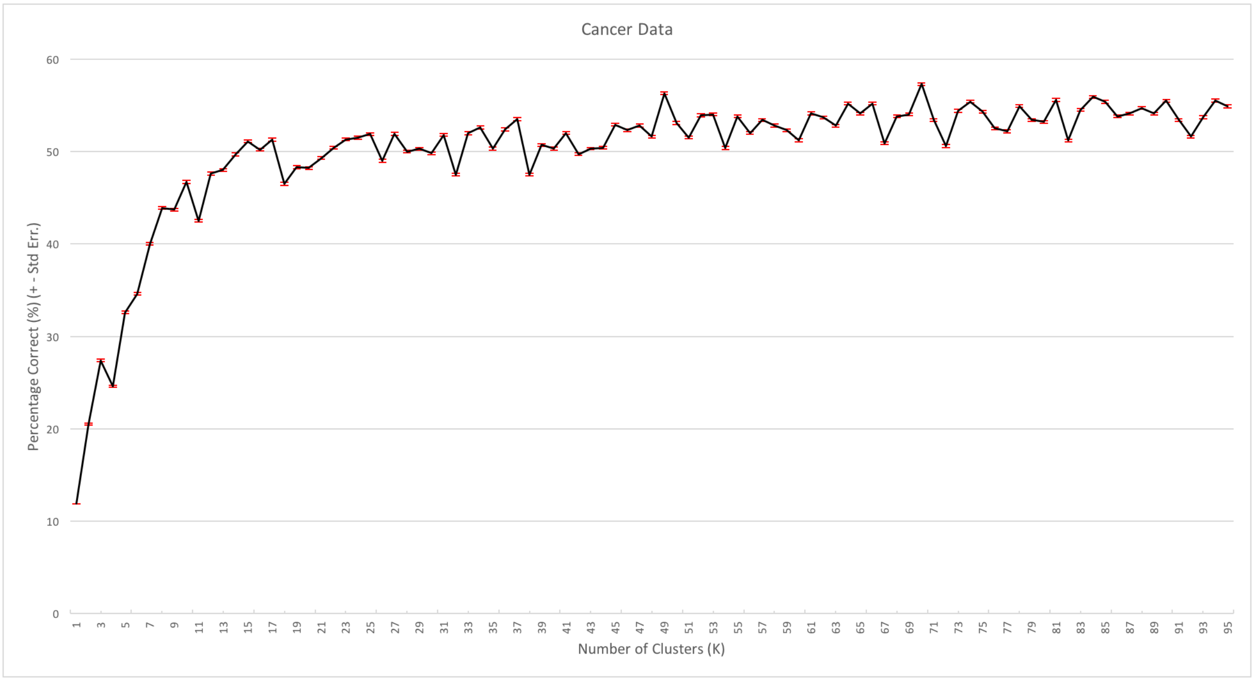
The example above is easy to visualize, the data can be viewed in two dimensions, and we can clearly see two clusters. A program does not have the benefit of being able to *see* that though. If we have chosen 1 mean or 3, there would have been very different results, and the resulting classification structure's accuracy could be better or worse. So how do we test this? I used cross-validation to assess the accuracy of a number of possible means for the data sets used in testing.

For example, the iris data had 150 data points. I set aside 10 points for testing, leaving 140 for training. With 140 training points, the algorithm could have anywhere from 1 to 140 unique means identifying clusters in the data. Deciding how many clusters there actually were in the data, what number would provide a consistent, accurate result, meant testing every one of them and evaluating the results from testing.

For the iris data, I found that beyond ~8 clusters, the accuracy of classifying the data was good, around 95%, and did not improve after that point. This indicates that the smallest number of clusters is around 8 for an accurate result, and so the data set could be compressed from 150 points to 8-10. Below is the plot of the learning curve from my tests, along with (very small, red) standard error bars:



The cancer data provided a less clear learning curve. Again, I set aside 10 data points for testing and used the remaining 95 to train and test 1-95 means. What became apparent in the testing portion was that there did not appear to be a number of clusters that provided an accurate and consistent result. Below is the plot of results from the cancer data, again with standard error bars:



What could be drawn from these results is that the features that are being used to evaluate whether or not a type of breast cancer is present are not correct indicators of the disease. Perhaps different types of data or measurements would better serve to classify the disease.

Kmeans.cpp

Kmeans.cpp accomplishes unsupervised learning with k-clusters, with the input of a random seed, K-number of clusters, the number of real-valued features in the data set, as well as files for training and testing provided from the user as command line arguments. The program is divided into two sections, training and testing. Before beginning, all of the training data is read into a vector, and for testing purposes a random seed is set. A vector of indices is also set-up for assisting in selecting random, but unique, initial locations for the k-number of means. This will be discussed further in the performance section.

Once the data and the means have been initialized, the program iterates through a system of determining the closest k-mean to each data point. These points are then used to update the location of the mean. By doing this, the means migrate through the data, each eventually finding a resting place that will not change on the final iteration. This indicates that for each given mean, it has found the center of its cluster. In my implementation, this loop is also iterated through one final time in a "classifying" mode. This is for the purpose of re-using the code that determined which points belonged to which mean. In this state, the classes of each point are collected so that each k-mean can be labelled with its majority classification.

This primary loop that accomplishes the training portion of my program relies on the way I chose to structure the data. I decided to create a struct for each k-mean that helped keep track of the sum of its associated points, the total number of associated points (both for the "updating" mode), and its classification. This made it possible to have less data to keep track of. Instead of storing all of the associated points for each k-mean separately, I summed each feature of a given data point as I determined which mean it belonged to, and then updated the mean directly from that sum of coordinates and the total number of points that were added to create it.

When training is complete, the program reads in testing data. For each point of testing data, again, the closest k-mean is found. The program checks the classification of the mean against the classification of the testing data, and increments a counter if the test was correctly classified. The program only outputs the number of correct classifications that were done in the testing phase before terminating.

Challenges

My first few implementations of this program had rather large flaw. As mentioned earlier, when initializing the k-means for the first time, they are randomly chosen from the training data set. On my first attempt, for each feature, I chose a random feature from the entire data set. For example, if the training data set had (x, y, z) real-valued features, for one k-mean I chose a random x out of all the x's in the training data, a random y, and so on. This had a tendency of creating the undesired effect of a k-mean being associated with *no* points. It had the possibility of being far outside of the range of any given cluster, and therefore the data would not be accurately divided into that many means.

My next attempt had a similar, but slightly different issue as well. For each k-mean, I chose a random, complete, data point from the set, making sure that that k-mean would start-off within the bounds of the data this time. What this approach did not account for was the possibility of choosing the same random data point more than once. I still found that some points had no points associated with them, and therefore the data was still not being divided up in the way it was meant to be. This was not immediately apparent when testing the program with the

Iris data. When using cross-validation to verify the accuracy of k-means clusters with the Iris data, it is apparent that a beyond ~8 clusters, the accuracy does not improve. When choosing seven random point out of one hundred and forty, the probability of choosing duplicates is very low, so in higher cluster tests, duplicate k-means does not have as much of an impact on the results.

Upon testing the Cancer data, the flaw became more apparent. According to the results provided upon assignment, the cancer data for this project appeared to have less accurate results. Since the data did not provide a clear and well-defined learning curve like the iris data, it did not have the benefit of consistency after a certain number of clusters. The results due to this bug were very unreliable, particularly as the number of clusters increased, since there was a higher likelihood that duplicates were being selected to initialize the means.

I was able to remedy this by creating the vector of indices at the beginning of the program. It is a quick-fix, and I don't believe the most efficient, but for the purposes of presenting the correct implementation, I added the vector of indices, which is simply a range from zero to the number of data points. I then had the vector randomly shuffled, and therefore was able to choose an index at random from the data. When an index is chosen, it is removed from the vector so it will not be selected again.

Overall, I would have like to make the program cleaner and more elegant. I could have implemented functions to assist with re-using code, and reviewed my data structures to see if they could be improved and modified to a simpler solution. I plan on re-visiting this program at semester's end to make these changes and try to apply it to other situations as well!

Thanks for a great semester!