K Mean Clustering Assignment

Introduction

Data

In this assignment, a dataset of 800,000 instances of 8 features was subjected to a k means clustering algorithm. The ranges of the features are from 20-25 to -15-20 approximately in value so it was decided that rescaling was not necessary. There were no missing values. There is some evidence of duplicated data with 640,000 unique rows compared to the 800,000 in total but it is entirely possible that rows could be the same and be valid. All data was included in the algorithm.

Implementation

Creating the Initial Centroids

A K means++ algorithm was used for the choosing of the initial centroids in order to promote dispersion and improve performance. To do this, a function called *create_centroids()* was written. The function uses numpy's random integer function to select a random row, which becomes the initial centroid. It then calculates the squared difference between each row and that centroid and stores it in an array of length 800,000. Each square distance is then divided by the total squared difference (all the row differences added up) and saved to a probability array. By dividing each row distance by the total, the value is between 0 and 1 and the sum of all of them is 1. This array is used in the numpy random choice function to select the next centroid. The further away the point is from the previous centroid, the higher the probability of being selected as the next centroid.

Calculating the Distance Metric

To find the distances between points and centroids a function was written called calculate_distance(). This function used the subtract function in numpy to compute the differences between each point and a given centroid. This allows for avoiding using a for loop to go through each row individually. The function "recognises" that if the feature data has shape of (800k,8), and the centroid has shape (1,8) that the centroid should be subtracted from each row in feature data and provides an (800k,8) array with the differences. The sum of the absolute values of these elements were computed using the numpy.absolute() and the numpy.sum() functions. This produced an array of (800k,1) of Manhatten Distance metrics for each point and the given centroid. In the numpy.sum() function the "axis=1" term tells numpy to sum by row (as opposed to by column, if axis=0 or sum all values into one if axis=None).

Assigning Points to their Centroids

In order to run the algorithm, each point has to be grouped with its own centroid (the closest one to it according to the distance metric). To perform this, the function *assign_centroid()* was written. This function fills an array of shape (K,800k), where K is the number of centroids, with the distance between each centroid and each point. For example, the first row contains the distance metric between each point and the first centroid. A for loop is used in this case, however it only has to loop K times. The *numpy.argmin()* function with axis=0 produces an array of shape (1, 800k) with the index of the minimum distance from each column which is also the assigned centroid.

Moving the Centroid to their Means

Having found the “Clusters” (the rows that belong to their respective centroids), the new centroids are calculated by finding the mean of all the rows that belong to that centroid. The function *move_centroids()* was written for this purpose. The function loops K times, and for each centroid, finds the rows in the feature data that were assigned to that centroid and computes the mean of each column to produce the next centroid. The following line of code was used:

centroids[i]=data[cluster_indices==i].mean(axis=0)

Boolean indexing was used to find the relevant feature rows using the cluster indices generated by the *assign_centroid()* function. The mean function with axis=0 finds the mean by column.

The output is the new centroids having been moved to their means.

Rating a Set of Centroids

Having found new centroids, the next step was to compute how well they clustered the data. To do this, the *distortion_cost()* function was written. This function found the squared differences between each row and its centroid and summed up all of them to produce one number (the distortion cost). This distortion cost describes how well the clusters are fitted. Better centroids should have lower distortion costs.

Iterating the Process

Number of Iterations (Inner Loop)

To implement the algorithm, the process of finding the means and moving them must be iterated a certain number of times. In this assignment the number of iterations is set to 10. This was part (but not all of) the *restart_KMeans()* function. This iteration was done by the

inner for loop. In each iteration of the loop, the centroid clusters are found by the *assign_centroid()* function. These were then passed to their respective row in an array holding the clusters for each iteration (each iteration is a cluster). There is a row for each iteration. The *distortion_cost()* function then computed the distortion value for each cluster. The *numpy.argmin()* function was used to find the index of the lowest distortion value and this is used to select the optimum cluster from the particular set of centroids. The optimum cluster is then passed to an array that has a row containing clusters for each time the outer for loop runs.

Restarting with New Centroids

As K Means is sensitive to initial conditions i.e. different chosen initial centroids can produce different final clusters, the process is best restarted with new centroids several times. In this project, the number of restarts was set to 10. The above described inner loop which picks the optimal cluster from 10 iterations is run by an outer loop ten times. Each time with a new set of centroids. Each iteration of the inner loop adds a cluster and a distortion value to their respective arrays. When the array is filled, the *numpy.argmin()* function finds the index of the minimum distortion cost and this index is used to select the best cluster from all the restarts. The numpy.hstack() function was then used to put the distortion value at the end of the chosen clusters so as to return everything in one array.

Finding the Output

The final output is taken from the above array. The final value of the array is the distortion cost and the other values are the clusters. These correspond to the best iteration from the best restart.

Optimising the Number of Centroids

To optimise the number of centroids, a plot was produced of the distortion costs and k values from 1 to 10. The following graph is displayed below. From the graph we see that after six clusters, the reduction in distortion cost for each increment of k is relatively small. For this reason, it was concluded that six clusters was optimal for this dataset as.

Graph of Distortion Cost against k
Distortion costs are shown on y-axis and k is shown on the x-axis