COMP9318 Project

Part 1: Abstract

In this project, we are required to write K-means clustering algorithm with L1 distance as the distance function, which can store data in a more space-efficient way.

Part 2: Methodology

problem 1:

a) In order to cluster data, we use K-means algorithm to find K centroids and calculate the L1 distance from each point to each centroid. Points will be clustered into the nearest centroid. After that, we no longer need to store every original point, but store the centroid of the cluster it belongs to.

b) When all points are classified into their cluster, we will find the median number and create the new centroid of every cluster. For empty cluster, we keep the original centroid. We will repeat above steps several times.

c) However, K-Means algorithm doesn't work efficiently when dealing with high-dimensional data. So we divide the data into P parts, and use K-means algorithm on each part.
In function pq, the data will be divided into P parts and each part will be passed into function K_star. Function K_star accepts three arguments which are data, centroids and max_iter.
In K_star, we initialise an empty array with shape (N,1) and N is the number of vectors in the data. This array is the codes for each part. Then we do a while loop which will loop k times (k is the iteration times).

In this while loop, we have two for loops. In the first for loop, we calculate the distance from each data point to all centroids using the module distance from scipy.spatial and the distance here is L1 distance. After that, we use argmin to get the index of the smallest distance centroid and store the index number into the corresponding position of codes.
In the second for loop, we will calculate the new centroids. First, using argwhere in codes to get a list of index numbers when its value is equal to the index number of centroids, we then do a if statement to see if this list is empty or not. If the list is not empty, we will get all data points which belong to the same cluster and using np.median to calculate and update the new centroid in codebooks.
When finish the for loop, we will do a update of codes again using a for loop to make sure all points being able to cluster into the nearest centroid. At the end of this function, it will return codes and codebooks for each part.

d) Back to function pq, because we do K_star on each partition, the returned codes and codebooks only belong to each partition. So at the end we concat all codes and all codebooks together into the required format . Finally, the function will return the required codes and codebooks.

The change when using L1 as the distance method is using median instead of mean to get the new centroid.

problem 2:

a) In this part, we want to implement a query method which can return a list of sets for all queries, which contains the nearest data points to each query. First, we have a function called query which take four arguments. In this function, we first create a dictionary with each row in codes as key and the set of corresponding indices as value.

b) Then we create a distance matrix which has shape (P,Q,K) and this has the sorted distance of each query to every centroid. Also, we create a distance index matrix which has shape (P,Q,K) and this has the indices of the sorted distance of each query to every centroid.

c) Then we pass the above two matrices, the code dictionary and T into another function called multi_seq to do further steps.

d) In multi_seq, it will do a for loop on the number of queries, for each query, it creates a traverse tuple (check the traverse step), a priority queue and the points set (store the indices of data points). Then it will go to a while loop and the stop condition is we get a enough number of data points. In this while loop, first it will get an item from the priority queue which has the smallest distance as well as the indices tuple. We add this indices tuple into traverse tuple and get a tuple of indices of centroids. Then check if this centroids tuple is in code dictionary and update the data points set with its value.

e) The next step is to do a set of if statements. In this part, instead of enumerating all possible combinations. I do a for loop in a range P. In this loop, first do if the number in the p position of the indices tuple is smaller then the number of centroids. Then under the if statement, I transform the indices tuple to a list and add 1 to the value in position p and also use the new list to create a new indices tuple which can be used later.

f) Then we pass traverse tuple, p, indices tuple, new list and the number of partitions into another function which will return True or False.

g) In this boolean function, first the condition will be set to True. Then we do a for loop which check the index at each position except position p to see if that is equal to 0 or the tuple when the index at

each position minus 1 is in traverse tuple or not. If all the conditions are satisfied, we return True otherwise we return False.

h) When the condition satisfies, we push an item into our priority queue with the sum of distance from all dimensions and a tuple which contains a next indices tuple.

The efficient part I did is first create a dictionary of codes which has the tuple of centroid indices as key and the points around that centroid as value. This part can be passed to the query part at once. All we need to do is to find if a combination is in this dictionary or not. If it is, then we get the value which is the indices of data points which can greatly save us time.
Another one is using tuples as much as possible which can save time when doing indexing compare to using list.