COMP9318 Project

Part 1: PQ for L_1 distance Implementation

Given data in shape (N, M), number of partitions P, initial centroids for P blocks and max_iter:

1) Partition data and initial centroids into P blocks, obtaining P data-blocks in shape (N, M/P).
2) Conduct **K-Medians clustering** on each block.
   a) Assign points to clusters by finding "mean" with smallest Manhattan distance (L_1 distance) We used *scipy.spatial.distance.cdist(data,init,metric='cityblock')* to compute Manhattan distance $D(x, y) = \Sigma|x_i - y_i|$.
   b) Update 'means' as median value (dimension-wise) of each cluster
   This is the major change we made to accommodate L_1 distanceBy **adopting median to find new centroids** make the algorithm more reliable for discrete or even binary data sets. In other words, this approach is more robust to outliers.
      i) bad centroids are kept in the project to ensure K=256 clusters returned.
3) Obtain codebooks in shape (P, K, M/P) and re-do a) one time to return codes in shape (N, P).

Part 2: Query using Inverted Multi-index with L_1 distance implementation

Given queries in shape (Q, M), codebooks in shape (P, K, M/P), codes in shape (N, P) and T:

1) Partition query into P blocks.
2) In each block: obtain PQ codebooks for the higher-order inverted multi-indices by calculating L_1 distances between query and codebooks.
3) Querying the higher-order inverted multi-indices to retrieve candidates from codes.
   a) First stage, all q_i, q_j, q_k, q_l are independently matched to corresponding codebooks. Rather than using separate inputs like i, j in Algorithm 3.1 that only applies to P=2, **it would be wise to use an arr[] to combine them.**
   b) Second stage, we follow Algorithm 3.1 to transverse the possible combinations of codewords $[u_i, v_j, w_k, x_l]$ in the order of increasing distances from q. **We use a dictionary to track whether the combination has already been pushed into priority queue. We also avoid cases where K is over 256.**
4) Stop searching when candidates obtained from codes exceeds the predefined length T.

As Algorithm 3.1 is in the project spec, we use the above method. However, we also find a more efficient way of searching rather than using Algorithm 3.1:

In this project, the possible combinations of codewords is fixed at 256^P. However, there may be far less data points. In other words, there will be lots of empty lists that correspond to u_i, v_j, w_k, x_l. that never co-occur together using Algorithm 3.1. To improve efficiency, **instead of identifying a sufficient number of <u>possible</u> combinations of codewords [u_i, v_j, w_k, x_l] that are closest to q and concatenating their lists W_{ijkl} which will lead to a lot of unnecessary empty searches, we use the**

existing combinations from codes to calculate the distances(the PQ codebooks we obtained in 2) is used here) and retrieve answers from the inverted multi-index. By doing so, no empty lists are searched and thus reduce overheads in computational cost.