COL761-A3

Manshi Sagar (2020CS50429)
Rajat Bhardwaj (2020CS50436)
Rishita Agrawal (2020CS50439)

March 2024

1 Curse of Dimensionality

1.1 Explanation

The distance between the furthest point from the query and the closest point from the query go towards 1 as the dimensions approach infinity i.e. the notion of "closeness" or "similarity" becomes less meaningful, therefore implementing methods based on space partitioning becomes challenging. More over the computational cost also increase with increase in dimensions. If we increase the number of dimensions, every point seems like an outlier.

1.2 Plot

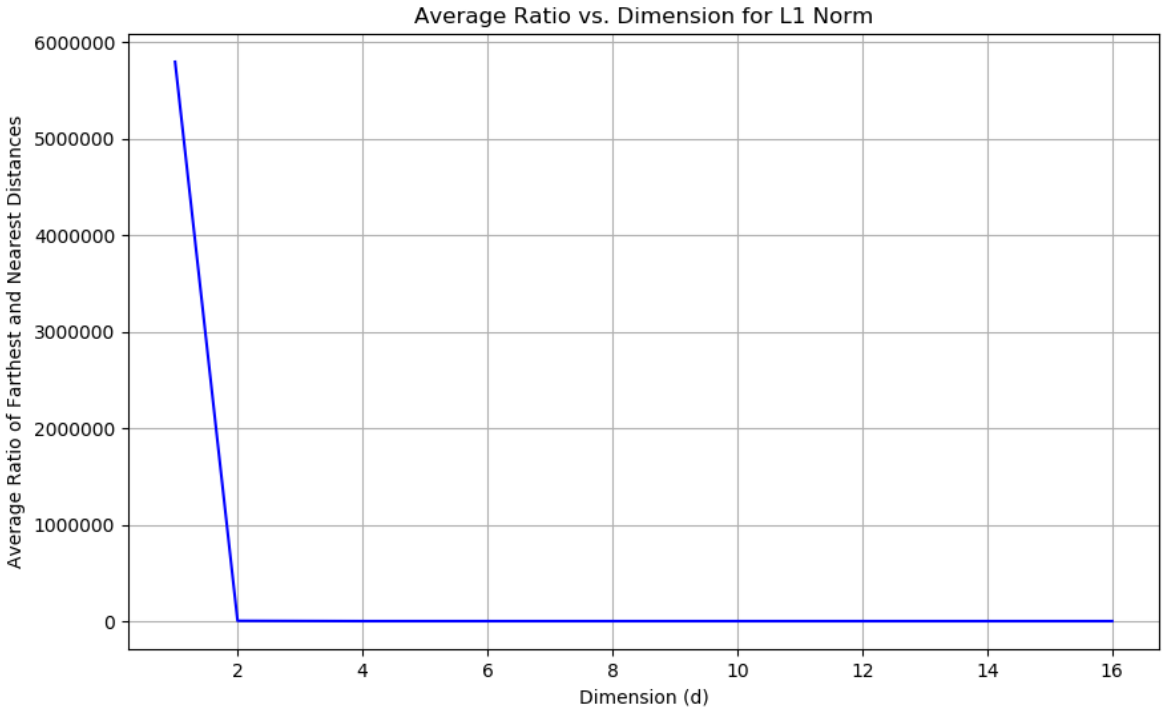

Figure 1: L1 distance ratio plot

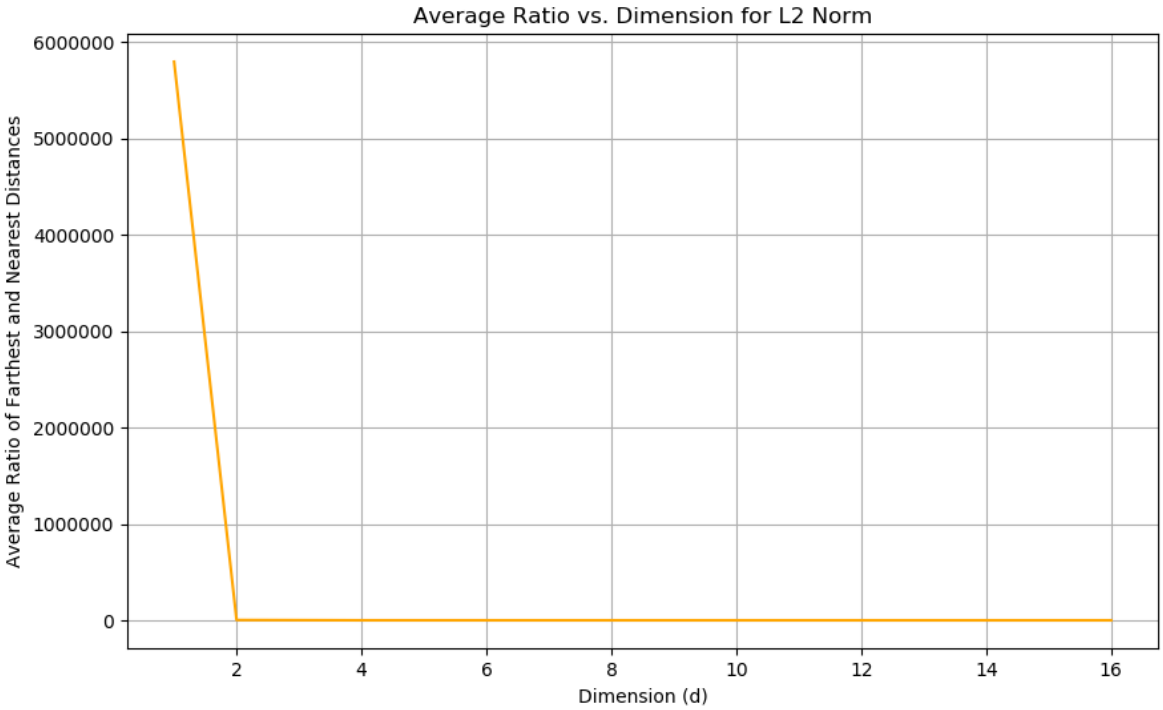

Figure 2: L2 distance ratio plot

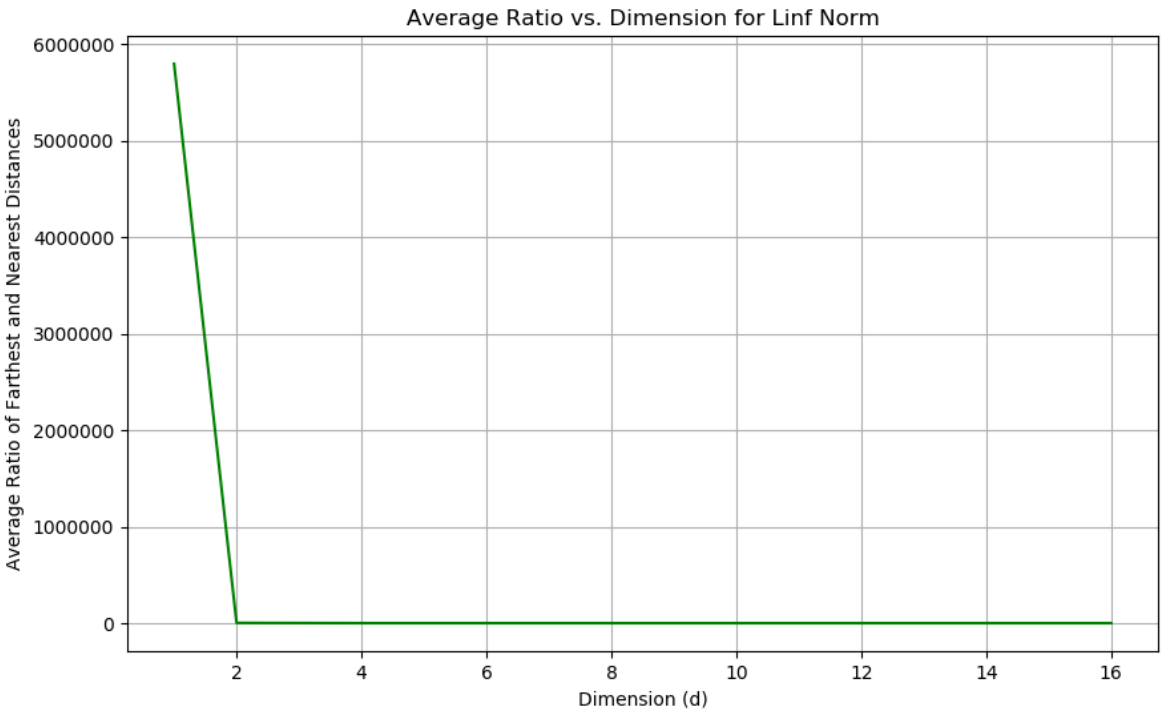

Figure 3: L∞ distance ratio plot

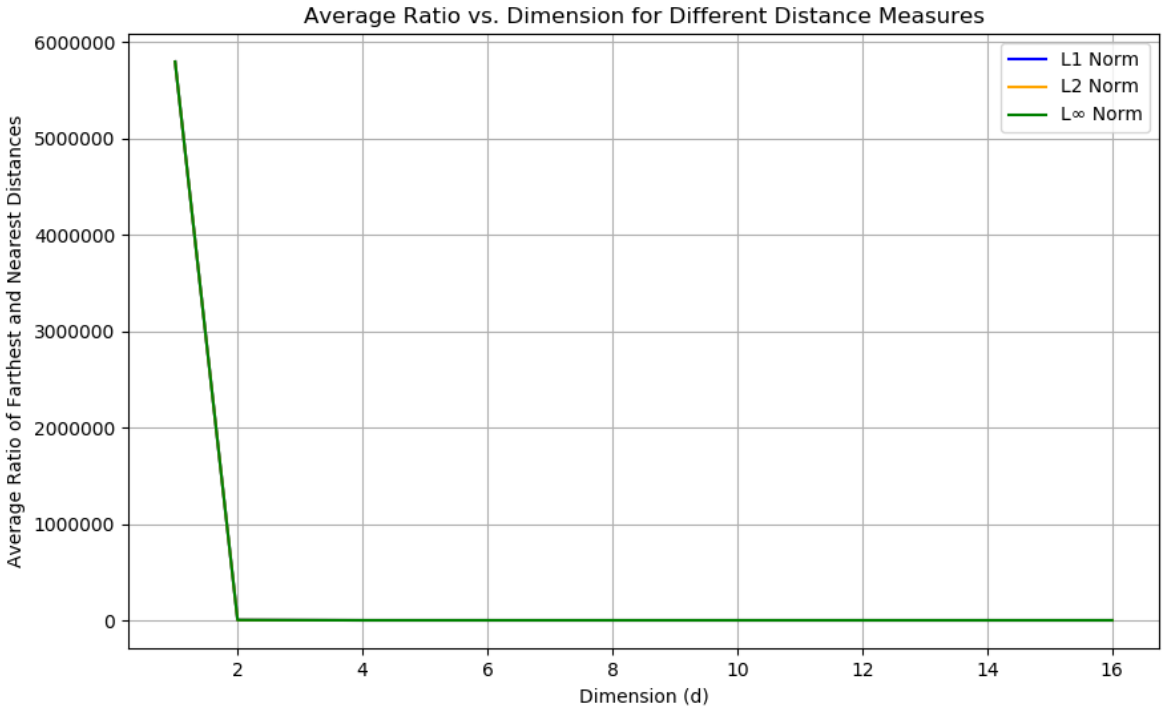

Figure 4: Combined distance ratio plots

2 Index Structures

2.1 No Index (Sequential)

2.1.1 Algorithm:

We calculate the L2 norm of the query with each point. and sort the list based on this norm. We pick top k from this.

2.1.2 Output:

As expected the time increases slightly when the dimension increases as computation increases while calculating the L2-norm but since the dataset is large, the increase is not much.

2.1.3 Plot:

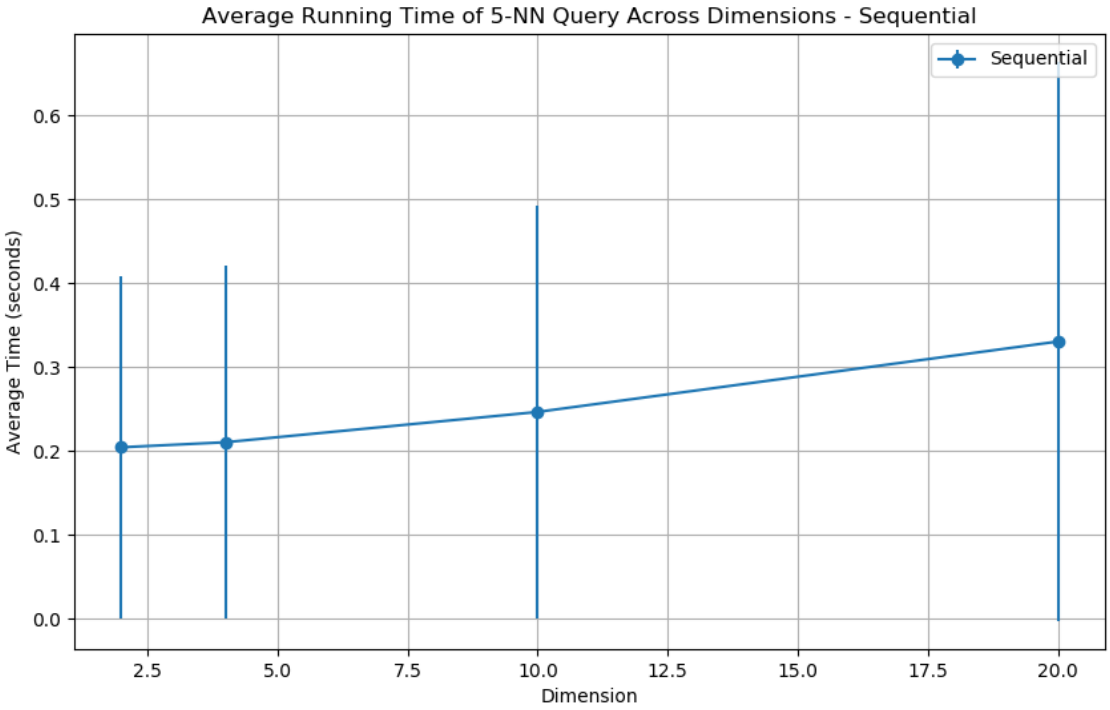

Figure 5: Average Time Plot - Sequential

2.2 KD-Tree

I am using the implementation from Scipy.spatial.

2.2.1 Algorithm:

Build KD-Tree:
Input:
Data (rest of the arguments are kept as default like leafsize = 10 (the number of data points at a single leaf node), balanced_tree = true (should the tree be balanced), compact_nodes = true (this will shrink the hyper-rectangles to the datapoints to avoid empty spaces, this will reduce query time but increase build time), copy_data = false (It copies data to prevent corruption of data)).

Output:
The algorithm outputs the root node of the KD tree. I am considering each node as having a splitting dimension for children, left and right hyper-rectangles. Pointers to left and right child, and data point (along with index) if it is a leaf node.

Algorithm:
We add nodes to the tree in DFS manner. By taking the median at each node splitting the dataset into halves, and using a stack to maintain the list of child nodes yet to be added to the tree, where the parent node is already a part of the tree. (Note that this is the basic implementation of KD-Tree, Scipy adds optimizations and they are described as and when they come. The basic implementation is similar to that of the class and I have referred to the research paper: Maneewongvatana and Mount 1999 as mentioned on the official website of Scipy).

1. Find the bounding hyper-rectangle, by taking the minimum and maximum of the dataset across all dimensions (min/max for each individually).

2. Now to split the data into two halves, Scipy uses a sliding midpoint splitting method. In this method, the splits are created by using the median of the longest dimension (dimension with a large difference between minimum and maximum, ties are broken by choosing the dimension which is more spread). If both the splits are even, we keep the split, but if all the points go to one of the splits, then we slide the boundary of the hyperplane till it touches a point p1 in the split. Then we split the dataset so that the points with the value along the chosen dimension are the same as the value of p1 along that dimension lies in one split, and the rest points go to another split. This avoids creating splits with no data in KD-Tree, thus all the leaf nodes will have some data points.

3. Insert the root node in the TREE (Note that TREE is our final output list, I am therefore writing it in capital letters to distinguish).

4. This was the initialization step of the root node. A stack is to be maintained, of the children nodes yet to be added to the tree. So we push the right child and left child to the stack (Note the order to maintain DFS (Depth first search) ordering). Each stack element is a 5 tuple - (dataset-split, parent index in the TREE, is left node?).

5. Now we recursively pop the elements from the stack till the stack is empty and we have all the nodes in the TREE.

6. For each node popped from the stack, we create the node by forming the left hyper-rectangle and right-hyper-rectangle, we update the parent node, with the left or right child pointer based on is the left node? flag with the current TREE length as we are pushing the node here.

7. If the data size is less than equal to the leaf size we create a leaf node, where there are no children hyper-rectangles or children pointers. just the data. Nothing is pushed to the stack.

8. Else we will calculate the children's hyper-rectangle for this node using the self hyper-rectangle (We use the same sliding mid-point splitting method described above). Then push the child elements to stack, and the node to the TREE.

9. TREE is returned.

KNN-Query using KD-Tree Index:
Input:
KD-Tree, query-point, K (number of nearest neighbors needed)
Output:
knn (it is the list of distances, list of indices of the neighbor data points))
Algorithm:

1. KNN implementation is a basic recursive algorithm. With sequential search at leaf node. And nodes are traversed based on the closeness of the hyperplane with the query point. The closest branches are explored first, and further exploration occurs only if the minimum distance is less than the K_th closest neighbor found so far. (Below is the description of this algorithm, mostly based on what was told in the class).

2. A stack is used to maintain the nodes to be explored. Starting with only one node, that is the root node.

3. We start exploring the nodes till the stack becomes empty.

4. Exploring each node involves popping the node, if it is a leaf node, we insert the distance (L2 norm) of this point and its index to the knn (output lists) for the top k closest data points in the leaf node done in a sequential manner.

5. If the node is not a leaf node, we pop the node and push its children to the stack. Which child node to push (left, right, or both) is based on if the node intersects (with min distance less than k^{th} smallest distance in knn currently) with the hyper-rectangle of the child, the child node is pushed to the stack else not.

6. The intersection can be checked by if the L2-norm distance of the query point and from the point which takes the min value for indices where the query point value ¡ min value of the bounding hyper-rectangle, the max value for indices where the query point value ¿ max value of the bounding hyper-rectangle and point value itself otherwise. and compares it with k^{th} smallest distance. Return true if the distance is less else false. Here we take this specific point to find the minimum distance between the point and the hyper-rectangle. (closest

value possible for each dimension). (We can use this distance to compare and push the one with more distance first. We are using this specific method because this will give us the minimum possible distance from a point in hyper-rectangle to the query point.)

7. knn is returned.

2.2.2 Pseudo Code:

Build KD-Tree:

```
def KDTree(data):

    # find bounding hyper-rectangle hrect
    hrect[0] = min over each dimension
    hrect[1] = max over each dimension

    # splitting the data
    Sort(data) based on longest dimension
    Create split1 and split2 based on median point(<median point value for that dimension to
    if(any split == null):
        find the value v along the dimension, closest to the splitting midpoint (the midpoint

        left_split = Data[where Data[d] = v]
        right_split = Rest

        (If v is the max then left and right will interchange.)


    #initializing hyper-rectangle for children
    left_hrect = hrect
    right_hrect = hrect

    # updating values along the 1st dimension
    left_hrect[1, 0] = v
    right_hrect[0, 0] = v

    # Initialize tree using rootnode
    tree = [(None , left_hrect, right_hrect, None, None, splitting_dimension)]
    #tree = [(No data point as data is present only at leaf nodes , left_hrect, right_hrect,

    stack = [(first half data, Parent at 0, left = True), (second half data, Parent at 0, lef

    # recursively split data in halves using hyper-rectangles:
    while stack is not empty:

        pop element off the stack
        nodeptr = len(tree)

        update parent node with the child node pointer left child is left = true else right c

        if data_size <= leafsize: #We create a leafnode
            leaf = (element[1], element[0], No left child, No right child, 0, 0, -1(No split
            tree.append(leaf)

        else: #If not a leaf node
            Splitting is done the same as above.
            stack.append(left child data with left half)
```

```
            stack.append(right child data with right half)
            Copy the hrect from parent node, copy left_hrect if left_branch else right_hrect
            # Update the child h_rect with splitting_value
            left_hrect[1, splitdim] = splitting_value
            right_hrect[0, splitdim] = splitting_value
            # append node to tree
            tree.append((None, None, left_hrect, right_hrect, None, None, splitting_dimension
    return treec
```

KNN-Query using KD-Tree Index:

```
def tree.query(query, K):
    stack = [tree[0]]
    knn = []
    while stack:
        leaf_data, left_hrect, right_hrect, left, right = stack.pop()

        # if leaf node
        if leaf_data is not None:
            _knn = sequential_top_k_neighbors(leaf_data)
            knn = sorted(knn + _knn)[:K]

        # if not a leaf
        else:
            # check left branch
            l_min = mindistance(left_hrect, query):
            r_min = mindistance(right_hrect, query):
            If(l_min < kth distance and r_min < kth distance):
                stack.append(tree[left])
                stack.append(tree[right])
                (Reverse the append if l_min>r_min)
            else if(r_min < kth distance):
                stack.append(tree[right])
            else if(l_min < kth distance):
                stack.append(tree[left])
    return knn
```

2.2.3 Plot:

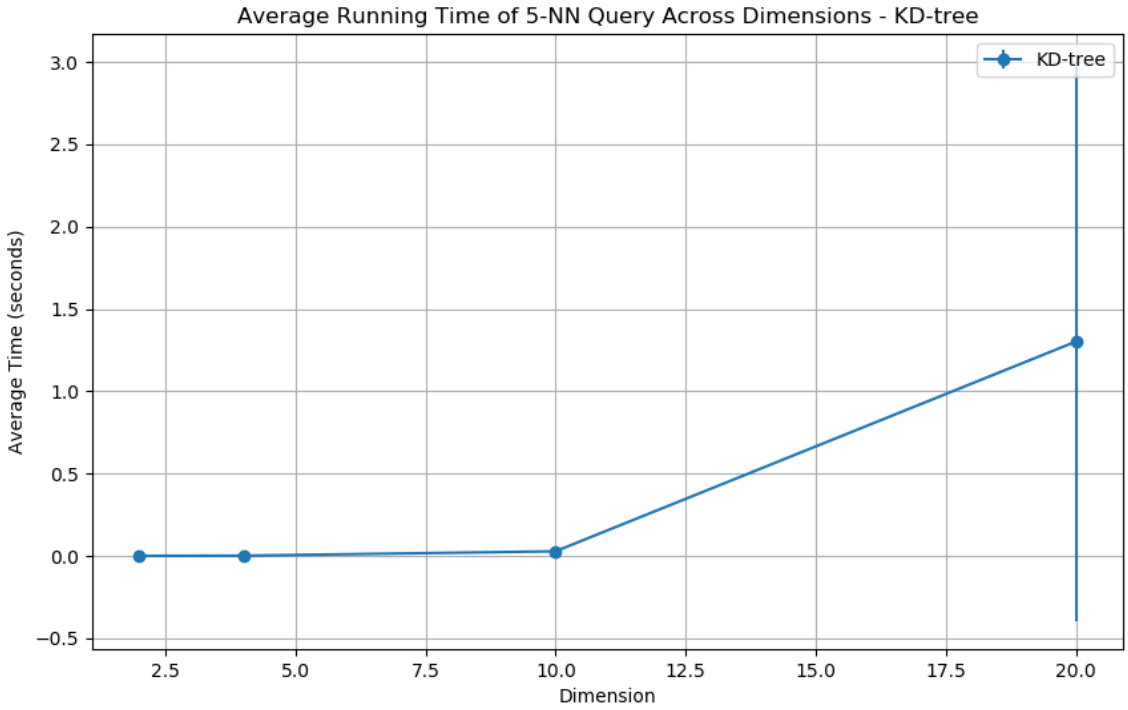

Figure 6: Average Time Plot - KD-Tree

2.2.4 Plot Explanation:

It is visible that the query time increases very fast as the dimension goes to 20. and also the standard deviation is more at dimension = 20. This is due to the curse of dimensionality. When the dimensions are lower, KD-Tree performs way better than sequential search, but this difference tends to reduce as we increase the dimensions. Sequential beats KD-Tree.

2.3 LSH

Our aim is that we should be able to preprocess the data in a fast manner to answer the k-nearest neighbor queries in sub-linear time. LSH randomly partitions the d-dimensional space into cells. The closer points are more likely to collide. We have used the FALCONN library provided to us in the assignment. The parameters that we have used are as follows

1. number_of_tables = 50
2. distance_function = EuclideanSquared
3. lsh_family = Hyperplane
4. storage_hash_table = LinearProbingHashTable

Since I have used Hyperplane as the lsh family, we will explain the algorithm related to it.

2.3.1 Algorithm for Index building

1. First we must center all the points, i.e. make the mean of the points equal to origin, so that we can add hyperplanes passing through the origin and partitioning the points

2. We choose k, L, where k = Number of HashFunction per table and L = Number of Hash tables

3. First we iterate through all the points in our dataset. For each point, we hash the point for every table. i.e. For for a point x and for a table $1 \leq i \leq L$ we have $H_i(x) = (h_{i1}(x), \ldots, h_{ik}(x))$. Now this k-tuple specified the bucket for the point x in the table i.

4. The hash function is used from [https://www.cs.princeton.edu/courses/archive/spr04/cos598B/bib/CharikarEstim.pdf]. Here r is the unit normal of the random hyperplane passing from the origin

$$h_r(u) = \begin{cases} 1 & \text{if } r.u > 0 \\ 0 & \text{if } r.u \leq 0 \end{cases}$$

Meaning - We check on which side of the plane the point lies. On one side we make all the points have a value = 1 and on the other side we make all the points have a value = 0. We do this for k hyperplanes for each table. We can have a total of 2^k different buckets for each table.

2.3.2 Algorithm for query

1. For the query point q we find the hash of it for each table i, i.e $H_i(q) = (h_{i1}(q), \ldots, h_{ik}(q))$. Thus we find all the values corresponding to the keys = $H_1(q), H_2(q), \ldots, H_L(q)$ in each hash table and union them. These values are nothing but the points that lie in the same bucket as the query point for each table.

2. This union set contains all our candidates for k-closest neighbors, now we take the L2 distance of all these candidates with the query point and find the k-closest points and return.

2.3.3 Probing

Since Falconn also uses probing therefore I will explain the algorithm in brief
Instead of checking each table only once for the query we try to perturb the Hash value of the query and check the table multiple times.
Formally for each query and for table i, we gather all the nearby candidates having hash values =

$$H_i(q) + \Delta$$

here $\Delta \in \{-1, 0, 1\}^k$. Where k = number of hash functions. We will also set the minimum number of zeros in the Δ equal to s.

2.3.4 Pseudocode

Algorithm 1 Index Building

```
procedure INDEX_BUILDING(data, k, L)
    Center all points in data around the origin
    for each point x in data do
        for i = 1 to L do
            Compute H_i(x) = (h_i1(x), ..., h_ik(x))        ▷ Hash point x for each table
        end for
    end for
end procedure
```

Algorithm 2 Query

```
procedure QUERY(q, k, L)
    Initialize an empty set candidates
    for i = 1 to L do
        Compute H_i(q) = (h_i1(q), ..., h_ik(q))
        Add points in hash table corresponding to H_i(q) to candidates
    end for
    Compute k-nearest neighbors from candidates based on L2 distance
end procedure
```

Algorithm 3 Probing

procedure PROBING(q, k, L, s)
    Initialize an empty set *candidates*
    for $i = 1$ **to** L **do**
        for each $\Delta \in \{-1, 0, 1\}^k$ with at least s zeros **do**
            Compute $H_i(q) + \Delta$
            Add points with hash value $H_i(q) + \Delta$ to *candidates*
        end for
    end for
    Compute k-nearest neighbors from *candidates* based on L2 distance
end procedure

2.3.5 Graphs and Explanation

LSH running time graph

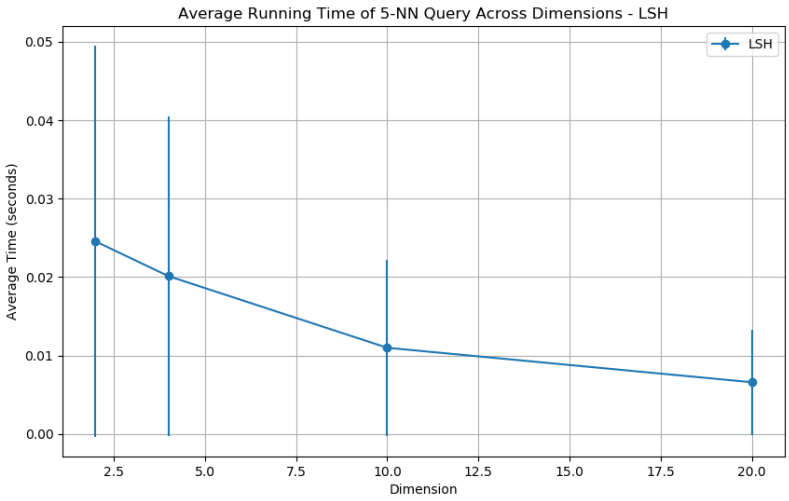

Figure 7: LSH running time graph

Explanation of query-time Decreasing with increasing dimension - We can prove this phenomenon for our hyperplane query. Let us imagine a 2-D space partitioned by 3 lines passing through the origin. We can see that it can partition the space into 6 regions (or buckets). Now let us go to 3-dimensions, and use 3 planes passing through the origin perpendicular to each other, we notice that the space is divided into 8 regions, these we can argue that higher the dimensions, the more number of partitions are created by fixed number of hyperplane for higher dimensions. Although there will be 2^k total possible hash-values but some hash-values would be impossible to have for lower dimensions Thus keeping this in mind we can see that the probability of collision for lower dimensions would be higher than the probability of collision in the higher dimensions. Therefore for the same hash value we will have more number of candidates in the lower dimension and thus the query-time will increase, since we would have to process more number of candidates.

LSH Accuracy vs dimensions

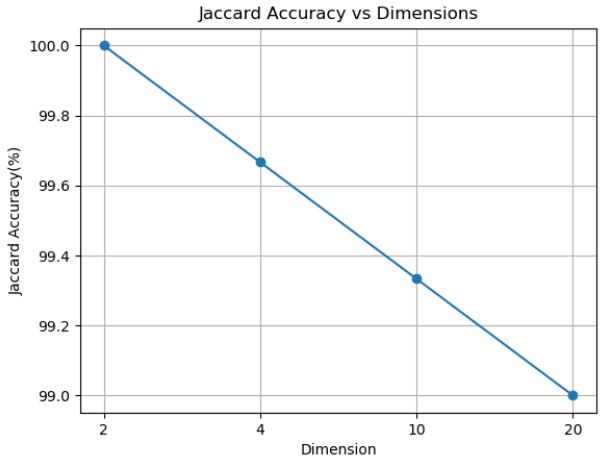

Figure 8: Jaccaard Accuracy vs dimensions

Explanation- Firstly it was not intuitive that the accuracy increase with increase in dimension. Then we realised that after PCA, few points were getting mapped to same point (when we go from high to low dimension). Linear scan picked the point having different index(ID) but same coordinate but the LSH picked up point with different ID. There for for lower dimension, the accuracy is lower. Then it increases as the effect of PCA decreases. Then decrease of accuracy is intuitive, since we have already argued that the points get mapped to different buckets, and there are more number of possible buckets for higher dimensions. Therefore since there are more number of possible buckets for the points, lesser collision happen and it might be the case that the point that are actually closer get mapped to different bucket.

K (top -k) vs accuracy

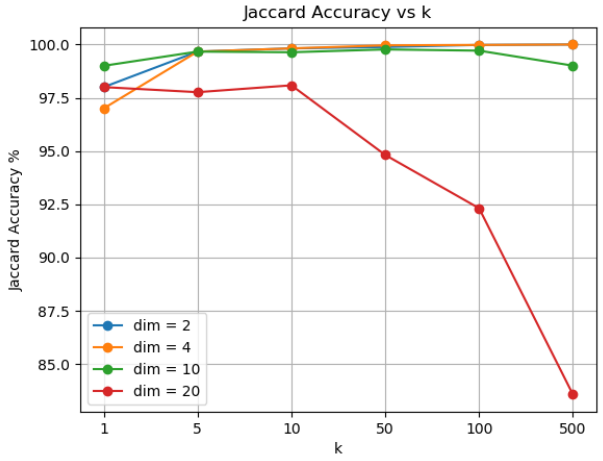

Figure 9: K vs accuracy

Explanation - First we can see that for lower k value, the accuracy is low and keeps on increasing in the starting (and keeps increasing for lower dimension). This can be explained with the same logic of PCA, for lower dimensions, different points were being picked that have the same coordinate but different index by Linear scan and by LSH. Now as k increase, we have to choose more number of points that are closer to the query point.
The accuracy increase because now we are picking more number of points, thus the output of linear scan may look something like [a,b,c,d] and LSH may look something like [b ,a , d,c]. Therefore the ordering may be different but they are the same set. The probability of linear scan and LSH having more number of similar items increase as we

increase the size of the set (i.e. k). Thereofore, the accuracy increases.
The drop of accuracy for k = 500 can be explained as follow.
We are choosing a lot of data points that are closer to our query and this space is highly partitioned, because of high dimensionality. Therefore, a lesser number of points lie in the same bucket. Hence, when we require a lot of points as nearest neighbour for the same query, for higher dimensions, the probability that the further point will get picked over the closer point will become higher, hence with increasing dimensions, and with increasing k, the accuracy drops.

2.4 M-Tree

We are using BallTree from python - sklearn

2.5 Algorithm

2.5.1 BUILD M-TREE

As in k-d trees we divide point clouds into hyper-rectangles, in metric-tree or m-tree or ball-tree we divide the cloud points into hyper-spheres. Formally, a ball tree recursively partitions the data set by enclosing subsets of points within hyperspheres.
Input X: array of points: shape = (nSamples, nFeatures)
nSamples is the number of points in the data set, and nFeatures is the dimension of the parameter space. Note: if X is a C-contiguous array of doubles then data will not be copied. Otherwise, an internal copy will be made. We are using floats, so copy will not be made.
LeafSize: default=40
Number of points at which to switch to brute-force. If number of points is very less, construction of tree and finding neighbours using the tree will take more time than sequential scan, so we switch to sequential scan at leafSize. We are using default leafSize.
DistanceMetric: We are using Eucledian distance.
Output: RootNode of metric tree

ALGORITHM for building M-Tree
Initially S = set of all points in dataset

1. Find diameter: Pick a random point, say p0. Find the point farthest to p0, say p1. Now, find the point farthest to p1, say p2. The line joining p1 and p2 is the diameter.
2. Project points on this diameter. Find the median of these projections. Divide the points into two sets (on different sides of the median)
3. In each space, find the centroid, say c. Then, find the point farthest to the centroid, say p4. Draw a hypersphere with radius = dist(c, p4). Repeat for other space. These hyperspheres are called balls.
4. The initial set of points is divided into two hyperspheres (two sets of points). Repeat the process for each hypersphere(set of points).

TIME COMPLEXITY = $O(Nlog^2N)$ for N points.

PSEUDOCODE for building M-Tree

```
constructBallTree(S):
    if |S| = 1                //leaf node
        return tree with root = point in S, left_child = null, right_child = null
    pick random p0 from S
    find p1 from S farthest to p0
    find p2 from S farthest to p1
    for every pi in S:
        qi = (p1-p2)T.pi               //project points on diameter
    m = median(q1, q2, ...)
    S_1 = { pi | qi< m}
    S_2 = { pi | qi>= m}
```

```
    find center c = mean(S)
    find radius r = max(dist( pi, c))) where pi belongs to S
    C_1 = constructBallTree(S_1)
    C_2 = constructBallTree(S_2)

    return tree with root = c, radius = r, left_child = C_1, right_child = C_2
```

2.5.2 KNN query using M-Tree index

Input
query point: q
number of neighbours required: k
metric tree: T
Min-first priority queue: PQ

Output
Min-first priority queue: PQ containing k nearest points to q, with first being closest

ALGORITHM

1. Start with an empty priority queue PQ to store the k nearest points encountered so far. Begin the search from the root of the ball tree.
2. At the current node, calculate its distance from the query point q.
3. If this distance is greater than the distance to the furthest point in PQ, return PQ.
4. Else if the current node is a leaf: Scan through every point in the leaf node and update PQ if any point is closer to q than the furthest point in PQ. Return the updated PQ.
5. Else if the current node is an internal node: Recursively call the algorithm on its two children. Choose the child node whose center is closer to q and search it first. Update PQ after each recursive call. Return the updated PQ after both children have been searched.

Priority Queue Updates:

The priority queue PQ is modified in place during the search. It maintains the k nearest points encountered so far, with the farthest point being at the end of the queue. When updating PQ, only points closer to q than the current farthest point are kept in the queue.

Termination:

The search terminates when the entire tree has been traversed or when the distance from q to the current node is greater than the distance to the furthest point in PQ. At the end of the search, the updated PQ contains the k nearest neighbors to the query point q. The priority queue PQ is modified in-place.

PSEUDOCODE - KNN query using M Tree

```
knnMtree(query q, k, priority queue PQ, tree T)
    PQ: max-first priority queue containing at most k points

if (distance(q, T.root) - T.radius)  distance(q, Q.last) then
    return Q
else if node is a leaf then
    for each point p in T do
        if distance(q, p) < distance(q, PQ.first) then
            add p to Q
            if size(PQ) > k then
                remove the furthest neighbor of q from PQ
            end if
```

```
            end if
        repeat
    else
        let child1 be the child node closest to q
        let child2 be the child node furthest from q
        knnMtree(q, k, PQ, child1)
        knnMtree(q, k, PQ, child2)
    end if
    return Q
end function
```

2.5.3 Plot:

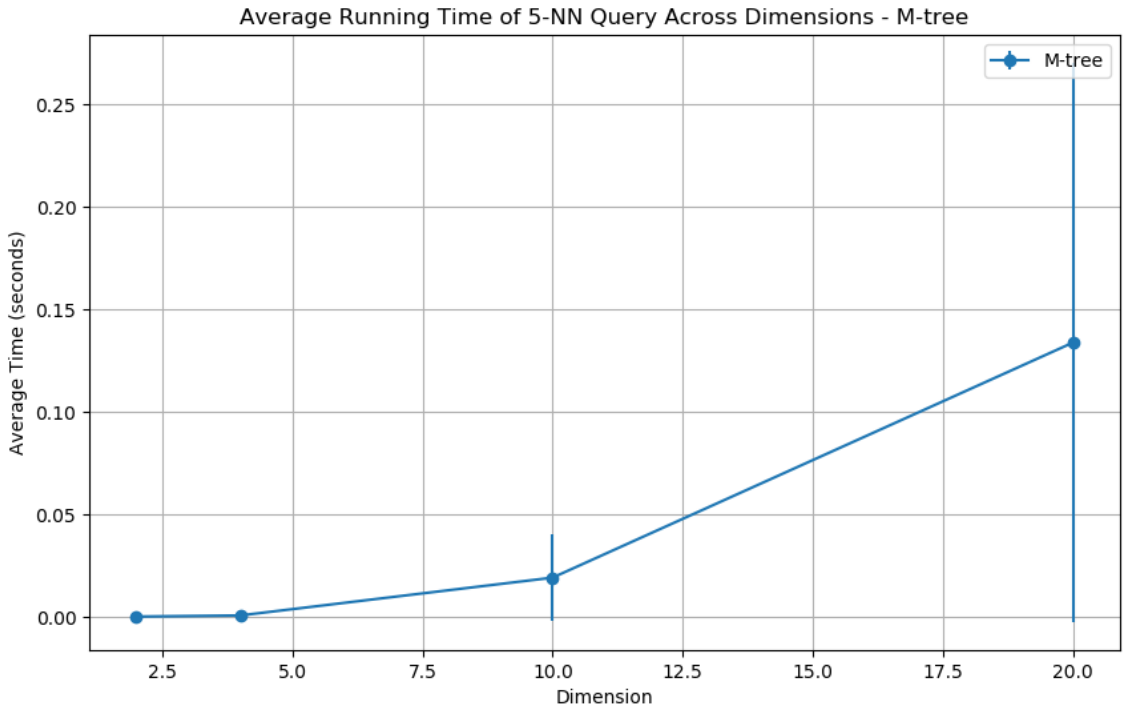

Figure 10: Average time graph of M-Tree

ANALYSIS

- As the dimensionality of the dataset increases, the average query time for KNN search also increases.
  Curse of Dimensionality: As the dimensionality increases, the data points become more spread out in the high-dimensional space. This leads to an increase in the search space and hence the time taken to find nearest neighbors.
  Tree Complexity: As the dimensionality increases, the number of partitions and the complexity of the tree structure also increase, leading to longer query times.
- **Dimension Reduction**: Although dimensionality reduction techniques like PCA help reduce the dimensionality of the dataset, the reduced dimensionality comes at the cost of losing some information. This loss of information may result in less accurate nearest neighbor search

2.6 Combined plot:

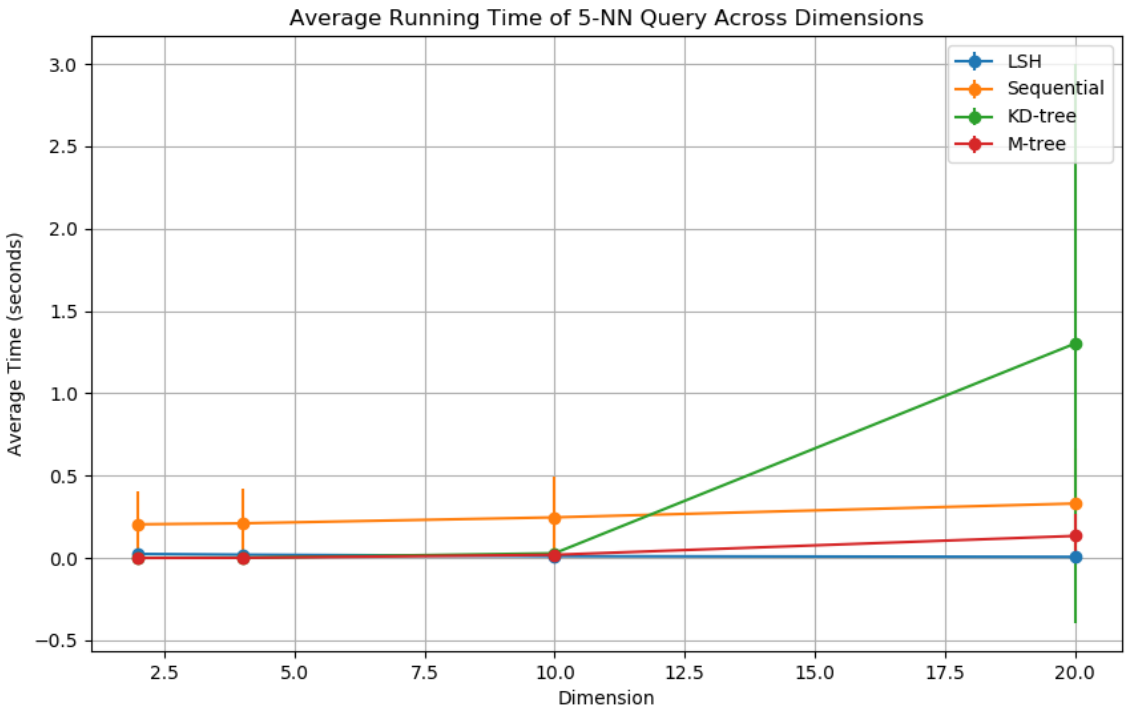

Figure 11: Average Time : Combined

3 Outputs

3.1 Q1:

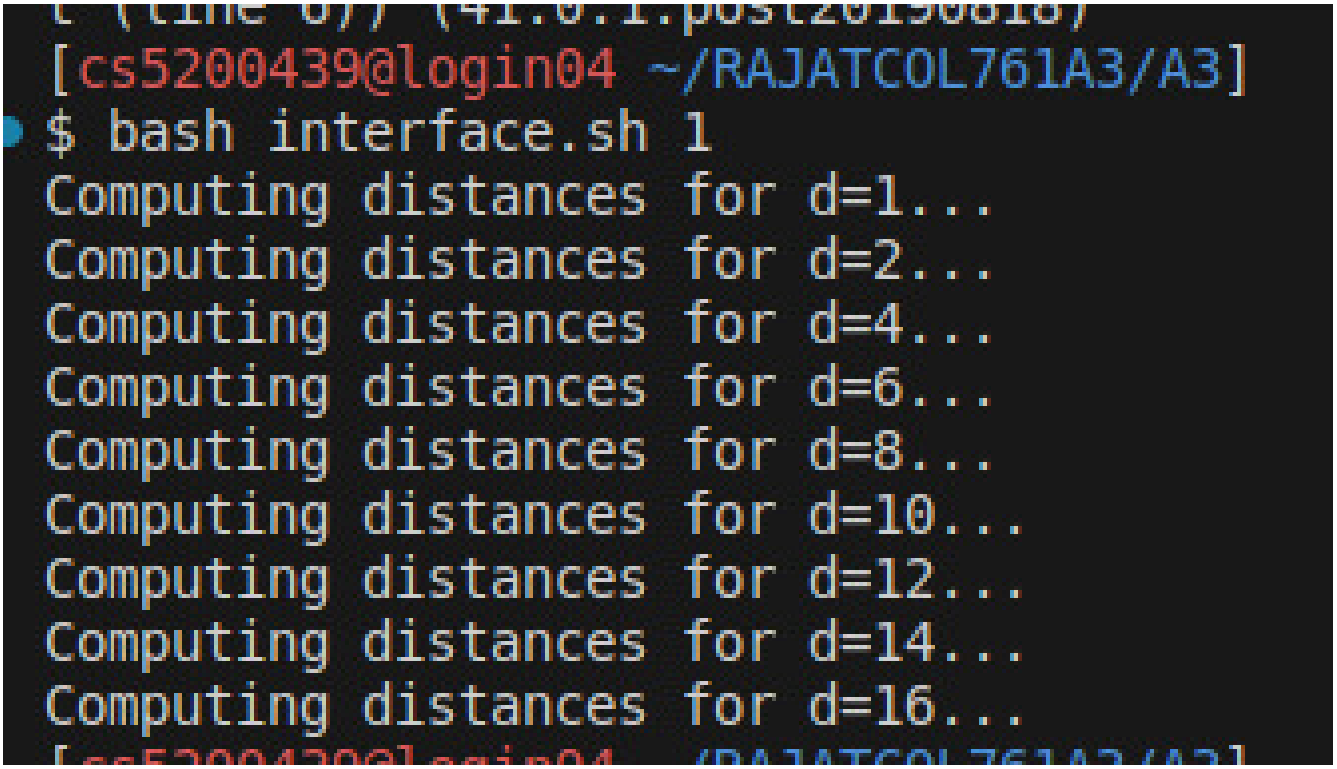

Figure 12: Q1 - Terminal Output

3.2 Q2(c):

Figure 13: Q2(c) - Terminal Output

3.3 Q2(d):

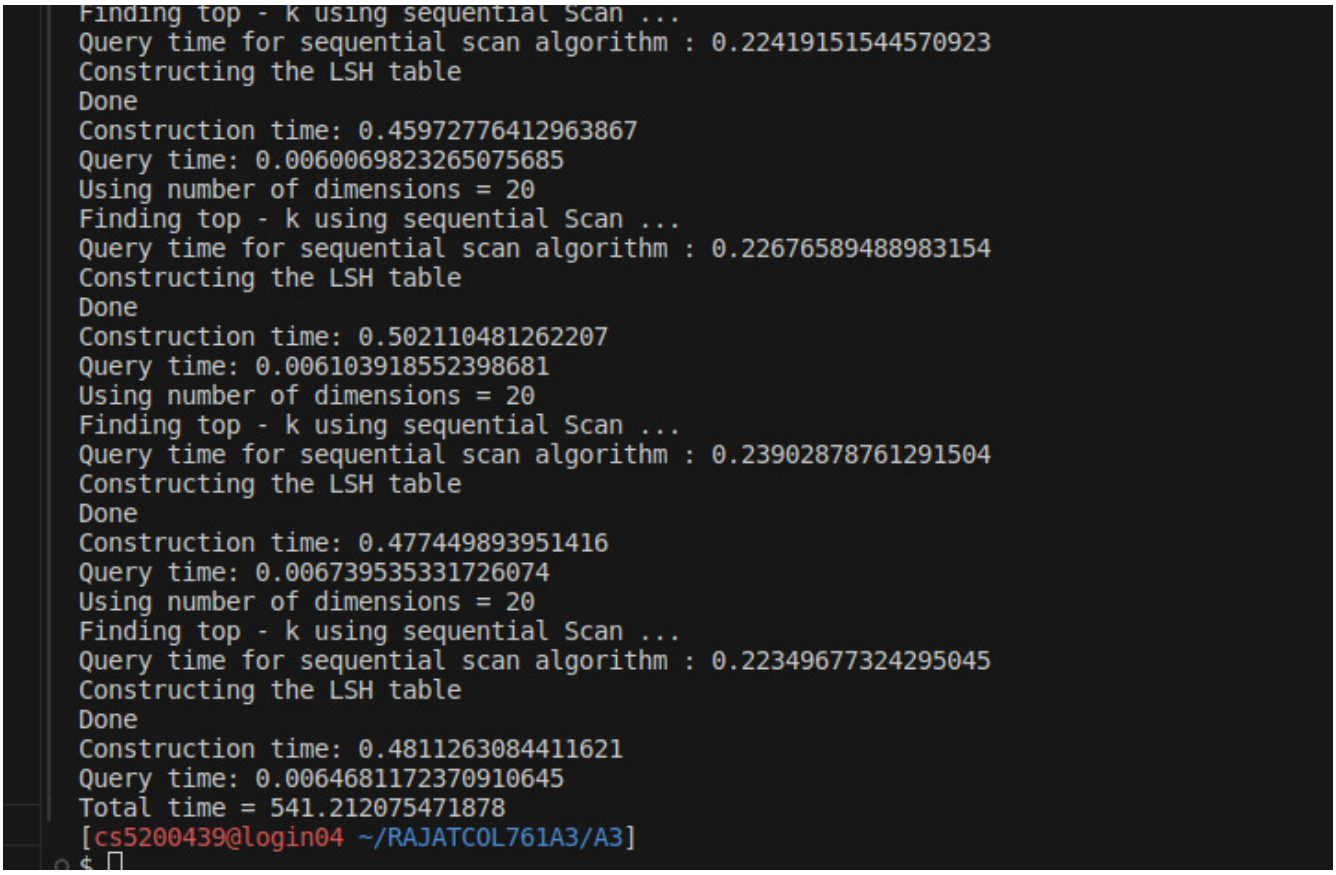

Figure 14: Q2(d) - Terminal Output

4 Graph File names and their meanings:

1. q1_combined.png : combined Graph for question 1
2. q1_L1.png : L1 Graph for question 1
3. q1_L2.png : L2 Graph for question 1
4. q1_Linf.png : L_∞ Graph for question 1
5. Q2_c.png : combined graph for all the index structure and sequential for question 2 part c
6. Sequential.png : Sequential graph for question 2 part c
7. LSH.png : LSH graph for question 2 part c
8. KD-tree.png : KD-tree graph for question 2 part c
9. M-tree.png : M-tree graph for question 2 part c
10. M-tree.png : M-tree graph for question 2 part c
11. Q2_c_DimVsAcc.png : LSH graph for question 2 part c - Accuracy Vs Dimension
12. Q2_d_KvsAccuracy.png : LSH graph for question 2 part d - Accuracy Vs K in Knn