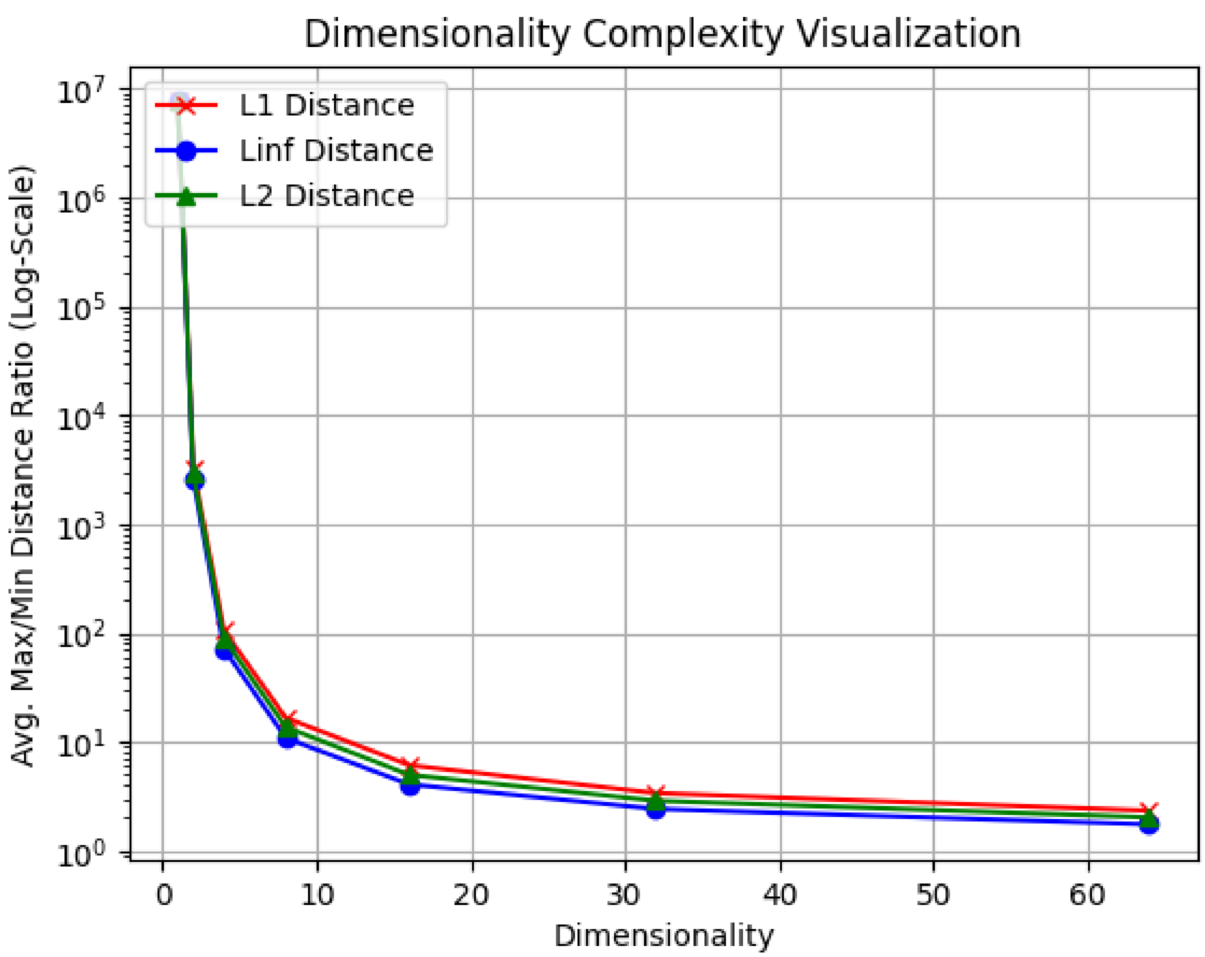
COL -761 Data Mining
Assignment 3
Group Name - Onlytwo

Team Members:
Varada Vinay Bhaskar 2020CS10405 - 50%
Chinmayee Behera 2020CS10337 - 50%

Question1:



	L1 Max/Min	L2 Max/Min	L inf Max/Min
Dimension 1	7635138.65980	7635138.65980	7635138.65980
Dimension 2	3297.53779	2945.95150	2593.75200
Dimension 4	104.87562	88.36888	71.33316
Dimension 8	16.68896	13.68760	10.86172
Dimension 16	6.11817	4.99350	4.07880
Dimension 32	3.39141	2.88666	2.42215
Dimension 64	2.33832	2.03129	1.75241

The distances of uniformly sampled points in high-dimensional spaces are the subject of the question. Thus, the metrics have distinct effects on the data-mining algorithms, such as clustering, which is based on distance metrics.

Distance is easy for us to think about intuitively in low-dimensional spaces, but it gets harder as we go higher. The data points are sparse because of the space's rapid volume expansion. This sparsity makes it difficult to infer meaningful relationships between any two randomly selected points because of the likelihood of a larger distance between them.

Because of the sparsity, nearer or farther cease to make sense as we go towards higher dimensions. This is supported by the fact that average ratios in higher dimensional spaces rapidly decrease.

As the graph illustrates, the sparse nature in higher dimensions has an impact on all three distance metrics: the Euclidean, Manhattan, and Chebyshev.
But L1 appears to be less vulnerable than L2, which is even less vulnerable than Linf.
This is due to the fact that while L1 uses distances in all dimensions, L2 uses them by squaring them, which makes it more effective in lower dimensions but ineffective in higher ones.
Linf is particularly vulnerable to increasing dimensionality since it uses distance along a single dimension, losing a lot of information in the process.

Even though the higher dimensionality makes it challenging for us to use distance metrics to discriminate between closer and farther points, we can still make some choices and use them.

Question2:

Reading Data:

We used pandas to read the input data

```
input_data = pd.read_csv(inputfile,header=None,sep=" ")
input_data = np.array(input_data)
input_data = input_data[:,:-1]
```

Primary Step:

We used the sklearn.decomposition.PCA library for reducing the dimension of given data.

Dimensions = [2,4,10,20]

```
for alpha in dimensions:
    pca = PCA(n_components=alpha,random_state=42)
    red_data = pca.fit_transform(input_data)
```

Part A:

We have used sklearn.neighbors.KDTree library for implementation of KDTree And sklearn.neighbors.BallTree library for implementation of MTree.
Falconn library for implementation of LSH.

KDTree Indexing:

```
kdtree = KDTree(red_data, metric='euclidean')
```

Mtree Indexing:

```
class MTree:
    def __init__(self, input_data):
        self.tree = BallTree(input_data, metric='euclidean')

    def find_k_nearest_neighbors(self, query_point, k):
        distances, indices = self.tree.query([query_point], k=k)
        return indices[0]
```

LSH Indexing:

```
params_cp=falconn.get_default_parameters(m,alpha,falconn.DistanceFunction.EuclideanSquared)
lsh = falconn.LSHIndex(params_cp)
```

Algorithms:

KDTree:

Input for KDTree:
Query Point(Q),k(number of nearest neighbours to be found,Root Node of KDTree(root)

Initially ,We will have a priority queue (PQ) which we use to store the closest neighbours. And a variable Dk (distance from neighbour k) = Infinity.
Now we will start the KNN search from the root node.KNN(root,0)

KNN Query using KDTree:

```
KNN(Node a,int dimension):
        Calculate the distance between the Q and Node a using euclidean metric(in this case)
        D = Dist(Q,a).
        If D<Dk :
                PQ.insert(D,a)
                update_value(Dk) (Dk=max  key value in the PQ)
        Now we Recursively traverse the tree:
                If Q[dimension]<=a[dimension]:
                        newdim = (dimension+1)%D
                        KNN(a->left,newdim)
                        If intersection_Distance(Q,a)<=Dk):
                                KNN(a->right,newdim)
                Else:
                        Newdim = (dimension+1)%D
                        KNN(a->right,newdim)
                        If intersection_Distance(Q,a)<=Dk
                                KNN(a->right,newdim)
```

Here the function intersection_Distance(Q,a) calculates the minimum distance of the divider at node a from the query point Q.

Here we always keeps the priority queue max size as k. So we keep on popping the minimum values when there is a need to push and the size of PQ is already k.

MTree(Ball Tree)

Let Or represent an entry in node N of the M-tree.
Op is the parent of node N.
d(Oj,Q) is the distance between object Oj and query point Q.
dk - maximum distance to a nearest neighbour found so far(initially infinity)
r(Or) is the maximum distance of an object in the covering tree from its root.
T(Or) is the covering tree associated with entry Or.

KNN Query using Mtree:
At each Node N, we check if it's a leaf node or not
If N is a leaf node:

- Iterate through all objects Oj in N.
- If the difference between the distance of Oj to Q and the distance of Oj to Op is within dk, compute the distance of Oj to Q.
- If this distance is less than or equal to dk, insert Oj into PQ and update dk.

If N is not a leaf node:

- Iterate through all entries Or in N.
- If the difference between the distance of Op to Q and the distance of Or to Op is within dk+r(Or), compute the minimum distance from the root of the covering tree T(Or).
- If this minimum distance is less than or equal to dk, insert the pointer to the root of T(Or) into PQ and update dk.

And the update dk and the priority_queue utilisation is similar to KDTree.

LSH:

Input:

Query Point (Q): Point for which nearest neighbours are to be found.
k: Number of nearest neighbours to retrieve.
Dimensions:dim
Number of hash tables.
LSH Index: Index structure created using LSH.

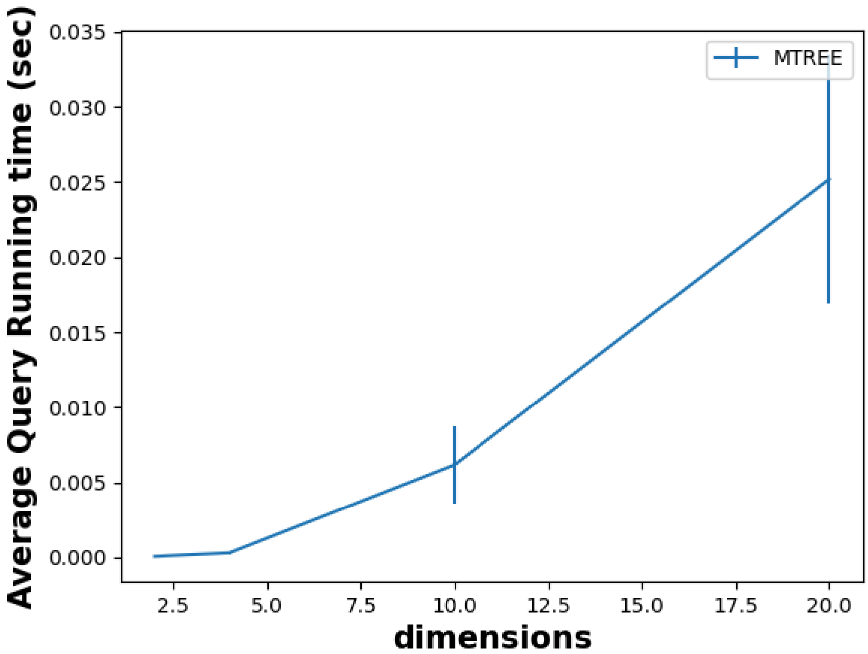
KNN Query using LSH:

Hash the query point Q using LSH.
Retrieve the buckets corresponding to the hashed value.
If necessary, perform a brute-force search within the retrieved buckets to find the nearest neighbours.

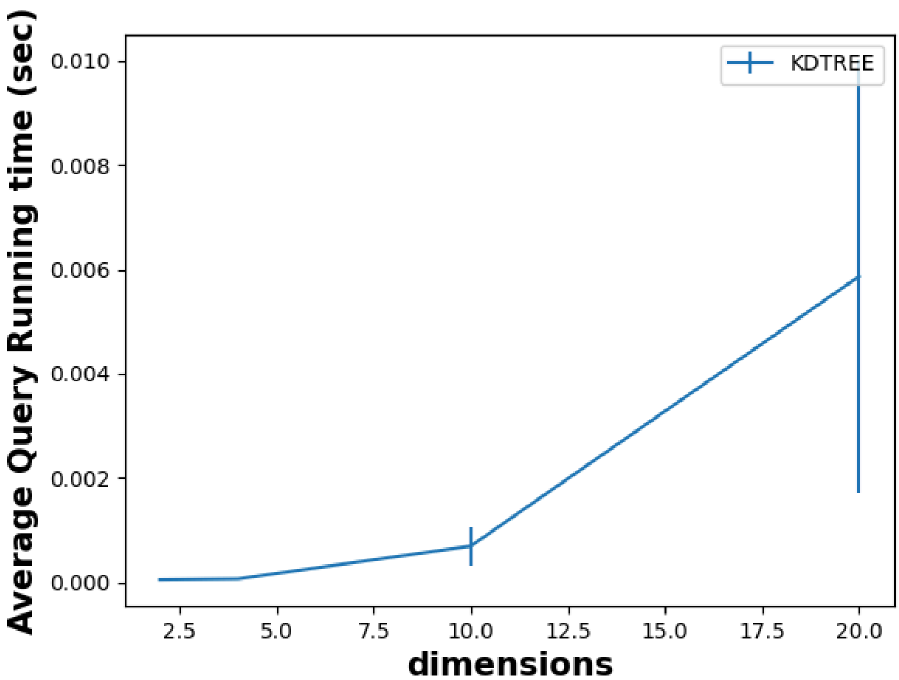
LSH doesn't guarantee exact nearest neighbours but provides approximate results efficiently.So we calculate jaccard_score = LSH values/ground truth.

Part C:

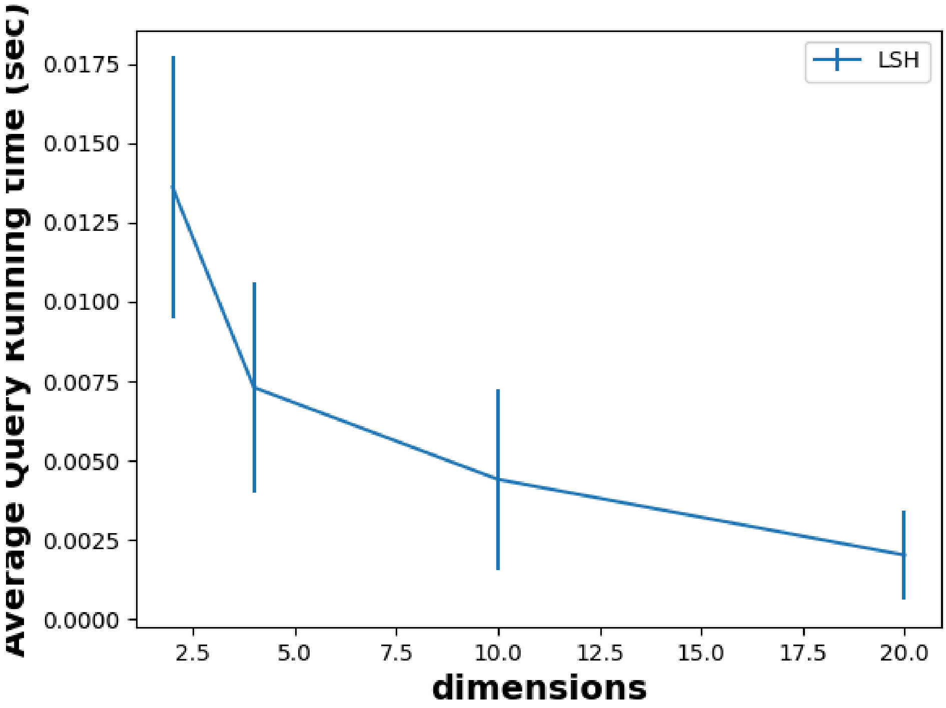
Graph for Mtree:



Graph for KDTree:

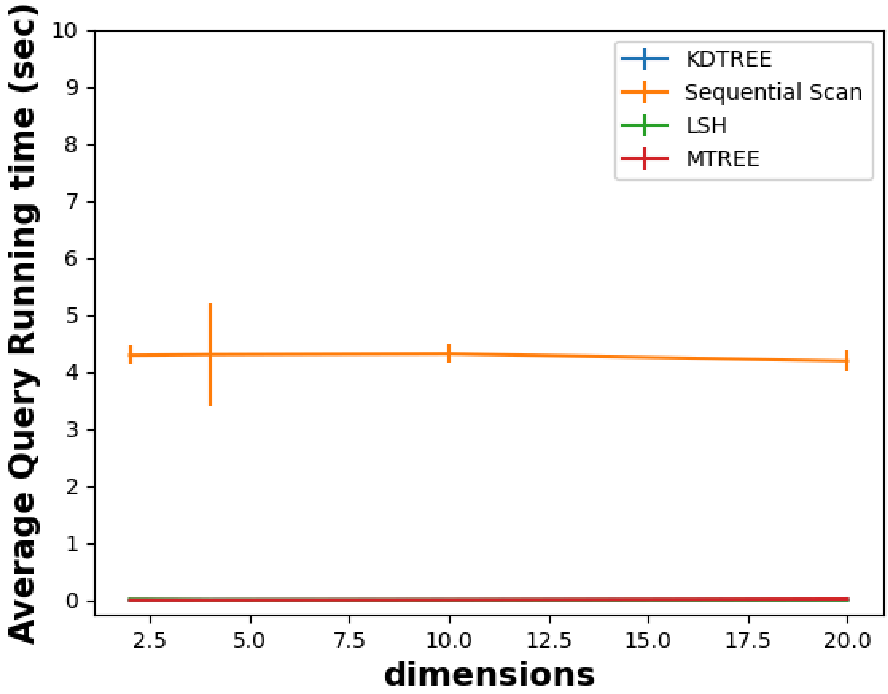


Graph for LSH:



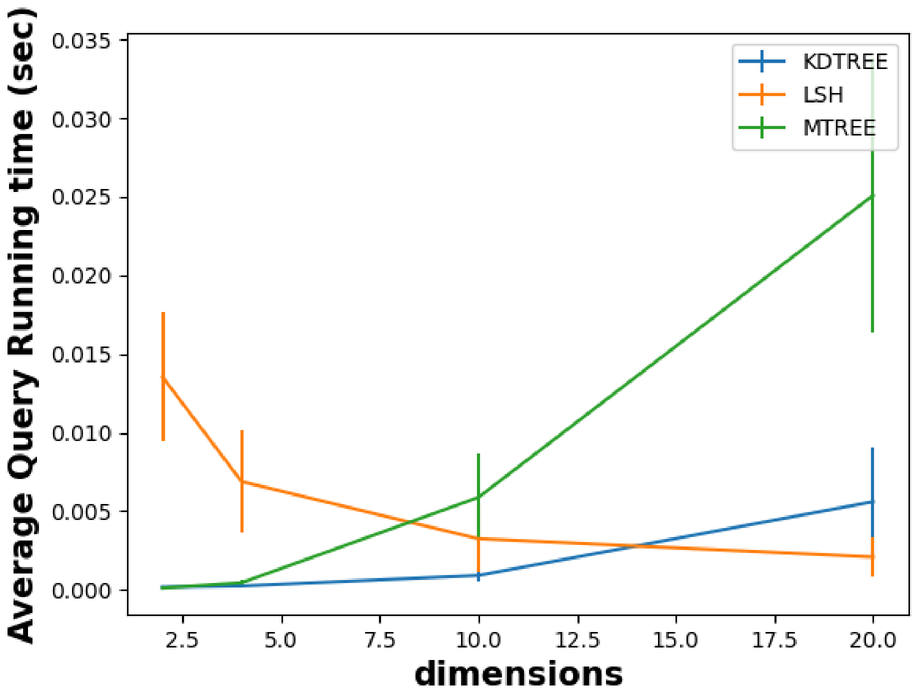
Now plotting along with sequential Scan:
For dimensions [2,4,10,20]
The mean and std for all methods are:

	Dimension 2	Dimension 4	Dimension 10	Dimension 20
KD_Mean	0.0003020668	0.000317935	0.001150622	0.006065020
KD_Std	0.0000423792	0.00008.1296	0.0005293753	0.0041803751
Seq_Mean	4.2934283590	4.3066411066	4.3201504945	4.1927282524
Seq_std	0.1597239834	0.8960018377	0.1644328328	0.1792985858
MTree_Mean	0.0001667618	0.000523931	0.006544611	0.022033991
Mtree_Std	0.0000377129	0.000523931	0.002683611	0.0087621256
LSH_Mean	0.0155205082	0.0055186772	0.0036373233	0.0019315862
LSH_std	0.004149263	0.0033096239	0.0023411851	0.0012189690



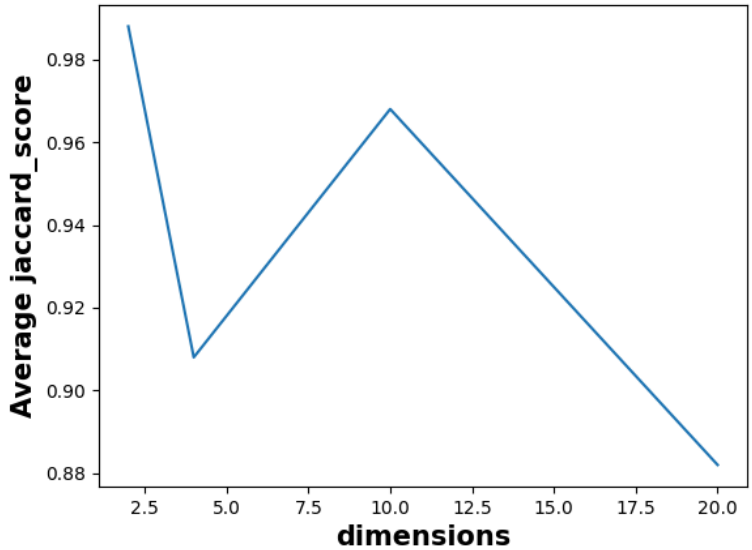
Here in the above graph all the remaining 3 with indexing get merged as all of them have Average time in between 0 to 0.04 secs whereas the sequential Scan has time in range of 4-5 secs.

We can see the comparison among LSH,KDTree,Mtree in the below Graph:



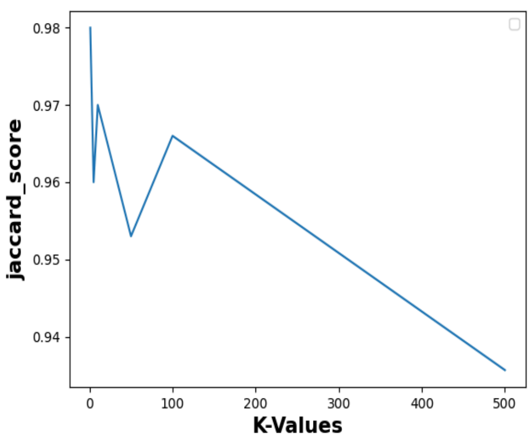
From the above graph we can see for higher dimensions LSH performs better and for smaller dimensions KDTree and Mtree performs better.
From Graph MTree performs almost perform better than KDTree in all cases.

The accuracy plot against Dimension for LSH:

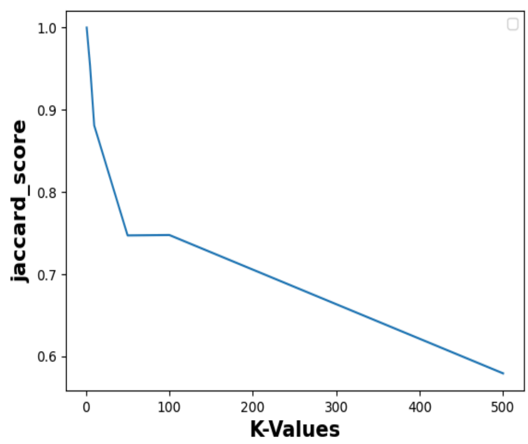


Part D:

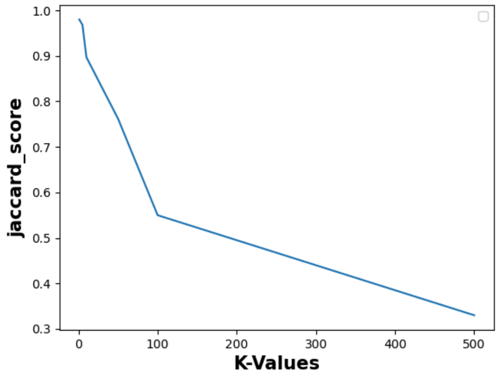
K_list = [1,5,10,50,100,500]



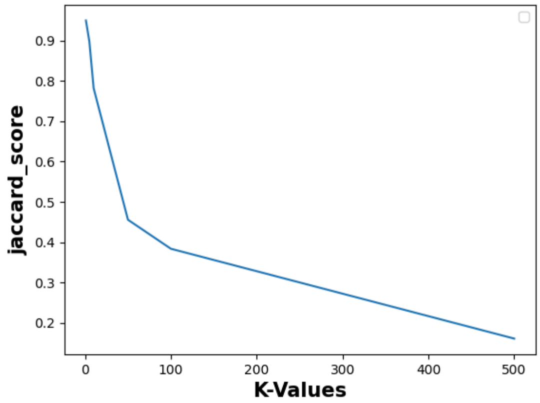
Dimension 2



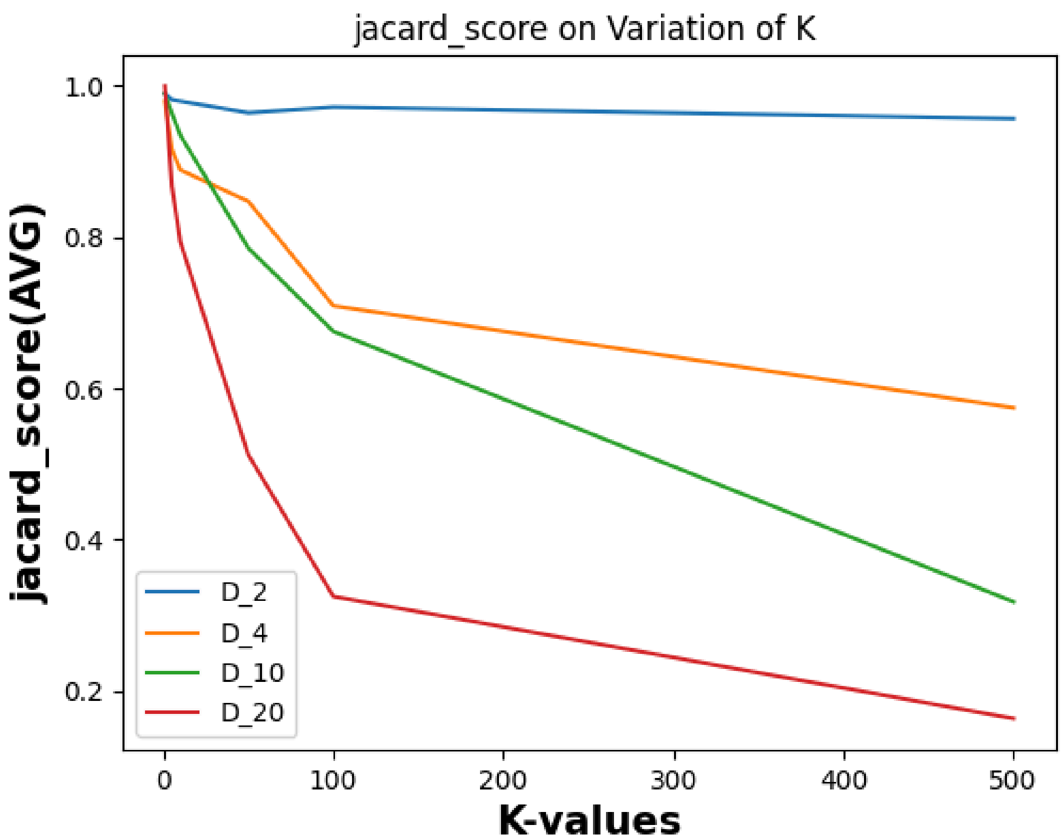
Dimension 4



Dimension 10



Dimension 20



From the Graph we can see for any particular k the accuracy(jaccard_score) is good at lower dimensions. As the dimension increases the jaccard_score decreases in most cases.

And also for a particular dimension the accuracy decreases as the K value increases.