CKD-TREE: AN IMPROVED KD-TREE CONSTRUCTION ALGORITHM

Y Narasimhulu[a], Ashok Suthar[a], Raghunadh Pasunuri[b] and V China Venkaiah[a]

[a]SCIS, University of Hyderabad, Prof. CR Rao Road, Gachibowli, Hyderabad, 500046, India
[b]CSE, Malla Reddy Engineering College(Autonomous), Maisammaguda(H), Gundlapochampally Village, Medchal Mandal, Medchal-Malkajgiri District, Telangana State, 500100, India

Abstract

Data structures such as VP-Tree, R-Tree and KD-Tree builds an index of all the data available in the offline phase and uses that indexed tree to search for and answer nearest neighbor queries or to classify the input query. We use a Lightweight Coreset algorithm to reduce the actual data size used to build the tree index, resulting in a faster index building time. We improve on already available Nearest Neighbor based Classification techniques and pit our classification method against the widely accepted, state of the art data structures such as VP-Tree, R-Tree and KD-Tree. In terms of speed the proposed method out performs the compared data structures, as the size of the data increases.

Keywords

KD Tree, Coresets, Nearest Neighbor, Classification.

1. Introduction

k-Nearest Neighbor (kNN) problem refers to the problem of finding k points or samples in the data which are closest to the query point. Nearest Neighbor algorithm finds its use in several machine learning areas, such as classification and regression and is also the most time-consuming part of these applications. In different use cases such as in recommendation systems, computer vision and robotics etc, fast response times are critical and using brute force approaches such as linear search is not feasible. Hence there are several approaches to solve these Nearest Neighbor problems which are based on Hashing, Graphs or Space-Partitioning Trees. Space-partitioning methods are generally more efficient due to less tunable parameters.

One such algorithm is KD-Tree. It is a space partitioning algorithm which divides space recursively using a hyper-plane based on a splitting rul. It reduces the search space by almost half at every iteration. Another space partitioning algorithm is Vantage Point Tree (VP-Tree)[1], which divides the data in a metric space by selecting a position in the space called vantage point and partitions the data into two parts. The first part contains data that are closer to vantage point and the other part which are not closer to the point. The division process continues until there are smaller sets. Finally a tree is constructed such that the neigbors in the tree are also neigbors in the real space. R-Tree[2] is another data structure that is most commonly used to store spatial objects such as location of gas stations, restaurants, outlines of agricultural lands and etc.

In this paper we consider kNN for classification, where nearest neighbors of a query point in the dataset are used to classify the query point. Nearest neighbor in essence is a lazy learning algorithm, i.e. it memorizes the whole training dataset to provide the nearest neighbors of an incoming query point. Consequently, though the algorithms provide very efficient solutions to the nearest neighbor problem, they might run into problems. This is because data size becomes too large due to the high magnitudes of data available today to process. In critical systems where time is of essence, loosing even a few seconds while processing all that data might cause issues. The author in [3] uses SVM to tackle a similar problem by reducing the size of data on which Nearest Neighbor algorithm runs. We use coresets for a similar effect, but on very large datasets.

The concept of coresets follows a data summarization approach. Coresets are small subsets of the original data. They are used to scale clustering problems in massive data sets. Models trained on Coresets provide competitive results against a model trained on full original dataset. Hence these can be very useful in speeding up said models while still keeping up theorit-

ISIC 2021: International Semantic Intelligence Conference, February 25–27, 2021, New Delhi, India
narasimedu@gmail.com (Y. Narasimhulu);
ashok.suthar.sce@gmail.com (A. Suthar);
raghupasunuri@gmail.com (R. Pasunuri); venkaiah@hotmail.com (V.C. Venkaiah)
https://github.com/Narasim (Y. Narasimhulu);
https://www.linkedin.com/in/ashok-suthar (A. Suthar);
http://cse.mrec.ac.in/StaffDetails?FacultyId=3072 (R. Pasunuri);
https://scis.uohyd.ac.in/People/profile/vch_profile.php (V.C. Venkaiah)
0000-0001-5440-0200 (V.C. Venkaiah)

ical guarantees upto a level. Coresets are often used in clustering algorithms to improve their speed even further. To achieve this, first construct a coreset — usually in linear time — and then use an algorithm that works on coreset to solve the clustering problem. As the coreset size is very small compared to the actual data size, this can provide significant speed in the said algorithms.

We use a state of the art lightweight coreset construction algorithm to improve time in the case of solving Nearest Neighbor problem using KD-Tree space partitioning algorithm. We use the end result of Nearest Neighbor query to classify our input query point based on its nearest neighbor points found.

2. Related Work

As foretold there are a number of approaches to solve a nearest neighbor problem based on hashing, graphs, space-partitioning etc. We focus mainly on KD-Tree, a widely accepted, widely used and fast space partitioning technique for Nearest Neighbors or Classification problems than VP-Tree and R-Tree.

2.1. KD-Tree (K Dimensional-tree)

KD Tree is a space-partitioning algorithm. Its main work flow can be divided in two parts, called offline phase and online phase. It builds the index(tree) in the first phase called, offline phase, so that it can answer the queries later on in the other phase called online phase. During the offline phase it subdivides space into rectangular cells through the recursive use of some splitting rule. In the *standard split* the splitting dimension is chosen based on the maximum spread along a dimension in the data. Whereas in *midpoint split* the splitting hyper-plane bisects the longest side of the cell and passes through the center of the cell. Most widely accepted KD-Tree implementation in work today is based on the paper written by Songrit Maneewongvatana and David M. Mount[4].

They consider a variant of the midpoint splitting rule called sliding-midpoint to overcome the problem of having a data set in which points might be clustered together. The example of sliding-midpoint rule is presented in Figure 1 and Figure 2 depicts the working of other splitting rules.

In this approach data is first split at midpoint, by considering a hyper-plane which passes through the center of the cell, dividing the cell's longest side in two parts. Then they check if data points exist on both sides of the splitting plane. If so, the splitting plane is not moved. However, if one side of the splitting plane is empty, i.e. without any point, then they *slide* the splitting plane towards the data points until it encounters at least one of the points. That point is then a child or leaf cell containing single point, while the algorithm continues to work recursively on the remaining points.

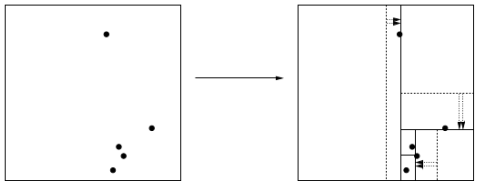

Figure 1: Example of the sliding-midpoint rule.

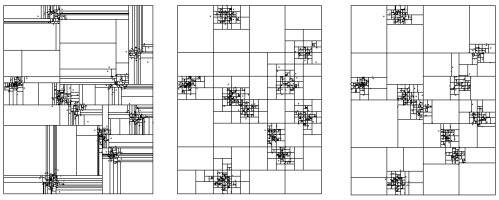

Figure 2: Examples of splitting rules on a common data set.

Once the tree is built completely, and a query comes in, we trace the query point down along the tree, by comparing it's dimension value with the cut dimension value at each level of the tree. Continuing this process leads to a leaf node. This leaf node to which the query point reaches in the end is called its nearest neighbor. A query point can have more than one nearest neighbors.

2.2. KNN Classification Based on KD-Tree

The authors of [5] talks about how KD-tree is a special storage structure for data and how it represents the training data efficiently. They propose a *k*NN-KD-tree classification algorithm utilizing the advantages offered by the kNN and KD-tree algorithms. They use the proposed method on eleven datasets and show that their *k*NN-KD-tree algorithm reduces time complexity while significantly improving search performance. Another nearest neighbor search algorithm that uses random projection and voting is discussed in [6].

2.3. SVM Based reduced NN Classification

The authors in [3] propose a novel SVM based instance selection method. Here Support Vector Machine (SVM) is used to form an instance space for instances of a particular pattern classification problem. A wrapper-based classification performance validation technique is used to find the best hyperparameters which identify the support vector. Then they identify and select informative support vectors (instances) lying on the margin hyper-plane in the instance space. Thus deriving a reduced training set for reduced nearest neighbor classification. This reduced training set is then used to classify new instances.

They demonstrate the performance of the proposed instance selection method on some datasets. Although their method maintained or even increased classification accuracy with a small number of training instances, all the datasets chosen are very small in nature. The highest number of instances in a dataset being 699 in Breastw dataset. We focus more on datasets with very large number of instances.

2.4. Lightweight Coresets

Coresets are compact representations of original data sets. They provide provably competitive results with models trained on the full data set. As such, coresets are successfully used to scale up different clustering models to very large data sets.

There are several existing Coreset building methodologies[7]. O. Bachem, and M. Lucic[8] proposed a notion of lightweight coresets that allowed for both multiplicative and additive errors. They provided an algorithm to construct lightweight coresets for k-means clustering and soft and hard Bregman clustering. Their algorithm is substantially faster than the other existing coreset construction algorithms. It's parallel in the sense that the data can be divided between several threads for parallel computation. Also as the name suggests, it results in smaller coresets.

Lightweight coreset algorithm uses variance as a means to create a probability distribution for the points in data set. Points farther from the mean have higher probability when compared to points which are closer to the mean. A small subset of points can then be sampled based on the derived probability distribution. As points are sampled based on the variance in data, it helps keep classification or clustering properties of datasets. They have showed that their approach naturally generalizes to statistical k-means clustering. They also performed extensive experiments, demonstrating that the proposed algorithm outperformed other existing practices.

3. Construction of Coreset KD-Tree (CKD-Tree)

Though KD-Tree for classification is a pretty fast algorithm in itself, it may not be so for very large datasets. To improve on the already fast KD-Tree classification algorithm, and to create an even faster version of KD-Tree we use similar approach as in the case of clustering algorithms, i.e. make use of Coresets. We first use a Coreset algorithm to create a representative set of points from the original data set. This representative set is then fed to the KD-Tree algorithm to build a tree index (offline phase) based on the representative set. When a query point arrives, we feed it into the tree, where it traces down to one of the leaf nodes in the tree index. At this point any suitable search method can be used to find nearest neighbors to the query point in the leaf node.

Algorithm 1 Lightweight Coreset Construction

Require: Set of data points X, coreset size m

1: $\mu \leftarrow$mean of X
2: **for** $x \in X$ **do**
3: $\quad q(x) \leftarrow \frac{1}{2}\frac{1}{|X|} + \frac{1}{2}\frac{d(x,\mu)^2}{\sum_{x' \in X} d(x,\mu)^2}$
4: **end for**
5: $C \leftarrow$ sample m weighted points from X where each point x has weight $\frac{1}{m.q(x)}$ and is sampled with probability $q(x)$
6: **return** lightweight coreset C

We use Algorithm 1, Lightweight Coreset Construction [8] (LWCS) to create the set of representative points from the actual dataset. This algorithm takes as input a dataset X and the coreset size m, i.e. the number of representative points in the coreset. It creates a probabilty distribution based on a point's distance from the mean, w.r.t. the total of all such distances. Distance metric used here is euclidian distance. Once every point has a probability assigned to it, we sample m points with weight $\frac{1}{m.q(x)}$ and probabiltiy $q(x)$.

Algorithm 2, CKD-Tree Algorithm for classification, uses Algorithm 1 Lightweight Coreset Construction (LWCS), to process and get a compact version of the original large dataset $repData$. This coreset $repData$ is then used to build the $tree$ index at line 2 of the algorithm. To build the tree index we use sliding-midpoint[4] technique. The $tree$ index can then be used to $query$

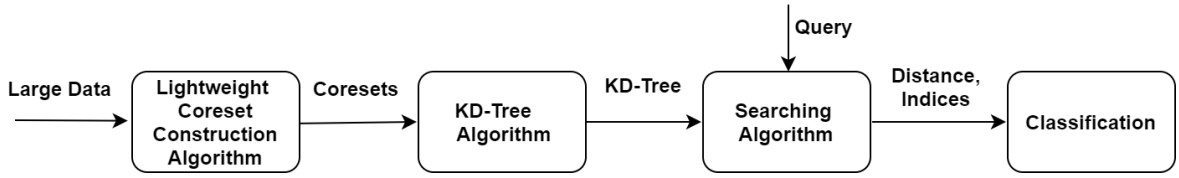

Figure 3: Flow diagram of CKD-Tree Algorithm for Classification

Table 1
Datasets Used

Dataset	Number of Instances	Dimensions/Attributes
bio_train	145,751	74
MiniBooNE Particle	130065	50
default of credit card clients	30,000	24
HTRU2	17898	9
spambase	4601	57

the index with a query point. Query requires you to specify k i.e. number of nearest neighbors required along with the point to query with, i.e. $queryPoint$. The flow diagram of the entire work is presented in the Figure 3.

In our specific use case, we use nearest neighbors to classify the query point into a class. This can be done easily based on the majority class in nearest neighbors returned.

Algorithm 2 CKD-Tree Algorithm For Classification

Require: Large dataset X, coreset size m

repData $\leftarrow lightweightCoresetAlgorithm(LargeDataset\,X, coresetsize\ m)$
$tree = KDTree(repData)$
$dist, NNIndices = tree.query(queryPoint, k = numOfNeighbors)$
for $index \in NNIndices$ **do**
    print point at index in repData i.e. Nearest Points
end for
queryPointClass $\leftarrow$ Majority class of Nearest Neighbor Points.

4. Experiments and Results

We implement the CKD-Tree using the above methodology and compare it against KD-Tree[4] [9] to see the performance difference it can provide and the cost.

All of the datasets [10] in table 1 have two target classes. While datasets *bio_train* and *MiniBooNe Particle* are both very large datasets, *HTRU2* and *spambase* are relatively very small. This helps in showing the relative performance of CKD-Tree algorithm on different types of datasets. Dataset *default of credit card clients* is a more balanced dataset in terms of sample size and dimensionality.

We kept 1000 samples from each dataset as test dataset for testing the models. These samples are used to check the accuracy of the prediction made by the algorithm. While testing the VP-Tree and R-Tree, we considered test sample sizes to 10, 50, 100, 200, and 500. Later we calculated the average times for them. While building the tree index for KD-Tree, $leafSize$ was kept same as the number of nearest neighbors queried (k). i.e., $leafSize = k$. Here $leafSize$ is the number of points in each leaf node of the tree index.

We measure the performance based on three factors, Accuracy of the results, average time(in seconds) taken in building the tree index and average time(in seconds) taken in answering the query. Each of these factors are compared and tabulated separately for all the data structures that were used and also for the proposed work.

Table 2 shows the results of CKD-Tree for $k = 10$. We use 3 different coreset sizes $m = 1000$, $m = 2000$ and $m = 5000$ and find the average of all of them(Avg. 1). For spambase dataset coreset size $m = 5000$ is not generated as data size itself is only 4601. Consider the bio_train dataset, here in table the average value of 'Indexing time' is calculated by adding the Indexing time of $m = 1000$, $m = 2000$ and $m = 5000$ and finally dividing it by 3. The same is applied for Querying time

Table 2
Proposed work comparison among various coreset sizes for k = 10

	m=1000			**m=2000**		
k = 10	Indexing time	Querying time	Accuracy	Indexing time	Querying time	Accuracy
spambase	0.131846189	0.002161955	75.1	0.14062953	0.002655576	75
bio_train	9.025853157	0.003039943	97.6	9.060353756	0.004280325	97.8
HTRU2	0.393140554	0.002060544	98.9	0.328070402	0.001718247	98.7
credit card	0.738107204	0.002901038	73.8	0.593619585	0.002796253	74.2
MiniBooNE	4.820586681	0.001834114	94.8	4.866789103	0.001765214	94.8
	m=5000			**Avg. 1(of m = 1000,2000,5000)**		
	Indexing time	Querying time	Accuracy	Indexing time	Querying time	Accuracy
spambase	N/A	N/A	N/A	0.13623786	0.002408766	75.05
bio_train	9.396525383	0.006560953	98.4	9.160910765	0.004627074	97.93333333
HTRU2	0.312451839	0.002077646	98.7	0.344554265	0.001952145	98.76666667
credit card	0.7029984	0.003421038	75.1	0.67824173	0.003039443	74.36666667
MiniBooNE	4.655207396	0.002046397	94.8	4.78086106	0.001881908	94.8

Table 3
Proposed work comparison among various coreset sizes for k = 50

	m=1000			**m=2000**		
k = 50	Indexing time	Querying time	Accuracy	Indexing time	Querying time	Accuracy
spambase	0.159596443	0.009064549	73.6	0.156191826	0.010310147	71.4
bio_train	9.68689847	0.018128316	97.6	9.013507843	0.013137584	97.6
HTRU2	0.346904278	0.006985356	98.7	0.312402248	0.007482661	98.6
credit card	0.879361391	0.006123379	75.4	0.609235764	0.007201368	75.2
MiniBooNE	4.948148489	0.011002789	94.8	4.764508009	0.008747934	94.8
	m=5000			**Avg. 2(of m = 1000,2000,5000)**		
	Indexing time	Querying time	Accuracy	Indexing time	Querying time	Accuracy
spambase	N/A	N/A	N/A	0.157894135	0.009687348	72.5
bio_train	9.372775555	0.015574522	97.6	9.357727289	0.015613474	97.6
HTRU2	0.328086615	0.007748176	98.8	0.329131047	0.007405398	98.7
credit card	0.656133652	0.008404238	75.2	0.714910269	0.007242995	75.26666667
MiniBooNE	4.717645407	0.00815424	94.8	4.810100635	0.009301654	94.8

and Accuracy of Avg. 1.

Table 3 show the results of CKD-Tree for k = 50. We use 3 different coreset sizes m = 1000, m = 2000 and m = 5000 and find the average of all of them(Avg. 2). For spambase dataset coreset size m = 5000 is not generated as data size itself is only 4601. Consider the bio_train dataset, here in table the average value of 'Indexing time' is calculated by adding the Indexing time of m = 1000, m = 2000 and m = 5000 and finally dividing it by 3. The same is applied for Querying time and Accuracy of Avg. 2.

Table 4 presents the average of Avg. 1 and Avg. 2. This average is called 'Overall Avg'. Indexing time of 'Overall Avg' is obtained by averaging the 'Indexing time' of Avg. 1 and Avg. 2. The same is applied for 'Querying time' and 'Accuracy'.

Table 5, given below, is the final comparison table. The table presents the comparison among R-Tree, VP-Tree, KD-Tree and the proposed work.

It is observed from Table 5 that the proposed work out performs all the data structures in Querying time. Considering the Indexing time, the proposed work also performed better than that of VP-Tree and R-Tree. The accuracy of the proposed work is approximatley close to other data structures.

The Figures 4 and 5, given below, show the comparison of Indexing time and Querying time respectively among R-Tree, VP-Tree,KD-Tree and Proposed Work.

Among the data structures that were used for comparison, KD-Tree is considered to be the best. So we concentrated mostly on KD-Tree. Here in Table 6 we present the indexing time comparison of the KD-Tree and proposed work. As the size of the data increases the performance of the proposed work increases and at a point of time it even starts performing better than the KD-Tree. So, for large datasets the proposed work

Table 4
Overall Average of proposed work

Cumulative	Overall Avg.((Avg. 1+Avg. 2)/2)		
	Indexing time	Querying time	Accuracy
spambase	0.147065997	0.006048057	73.775
bio_train	9.259319027	0.010120274	97.76666667
HTRU2	0.336842656	0.004678772	98.73333333
credit card	0.696575999	0.005141219	74.81666667
MiniBooNE	4.795480847	0.005591781	94.8

Table 5
Comparison among R-Tree, VP-Tree, KD-Tree and Proposed Work

	R-Tree			VP-Tree		
	Indexing time	Querying time	Accuracy	Indexing time	Querying time	Accuracy
spambase	46.01833411	0.025816591	66.632	0.9678272	0.00872661	88.216
bio_train	300.3921245	0.674326682	98.6	46.720815	0.027852184	98.6
HTRU2	98.00138087	0.002142198	96.6	4.2660978	0.011163483	93
credit card	151.047457	0.08385841	79.8	6.7166709	0.048709915	79.4
MiniBooNE	250.0596289	0.099793004	99.8	41.5471755	0.021343294	99.8
	KD-Tree			**Proposed Work**		
spambase	0.097132921	0.009621621	71.2	0.147065997	0.006048057	73.775
bio_train	13.37806582	0.079558667	99.3	9.259319027	0.010120274	97.76666667
HTRU2	0.088246346	0.006940414	98.6	0.336842656	0.004678772	98.73333333
credit card	0.799064255	0.012309615	75	0.696575999	0.005141219	74.81666667
MiniBooNE	7.842401028	0.022687735	94.8	4.795480847	0.005591781	94.8

takes less time for creating index.

The breakover point, 30000(credit card dataset), of the proposed work is shown in the figure Figure 6.

5. Conclusion

The above tables show that for at least one value of m each dataset showed competitive or in some cases better accuracy (*default credit card* and *HTRU2*) when used with coresets. In case of larger Datasets such as *bio_train* and *MiniBooNE*, the coreset size is very less compared to the original dataset size. But they still manage to provide almost same results in terms of accuracy as the original dataset. Also KD-Tree built on the coresets of these datasets see a significant speed boost in offline (indexing) and Online (Query) phases. We can also notice that as the dataset size starts to decrease, the gap in indexing and query speeds starts to become smaller and smaller. For smaller datasets (*spambase* and *HTRU2*), this might even lead to higher query or indexing times.

This shows that CKD-Tree is a very efficient algorithm for nearest neighbor based classification in large datasets.

6. Future Work

CKD-Tree gives good results on the experimented datasets. The probability distribution used here is based on the variance of data. Consequently this approach might not perform well on noisy or locality sensitive data. In such cases a better probability distribution approach for data summarization could be based on the locality of data. Locality Sensitive Hashing (LSH) functions could provide more accurate form of data summarization. For image data *vector quantization* based approach might help in data summarization.

In the online phase, implementation of a more robust and faster search technique could also be fruitful. Randomizing the algorithm or having multiple tree indexes to increase the accuracy is another option.

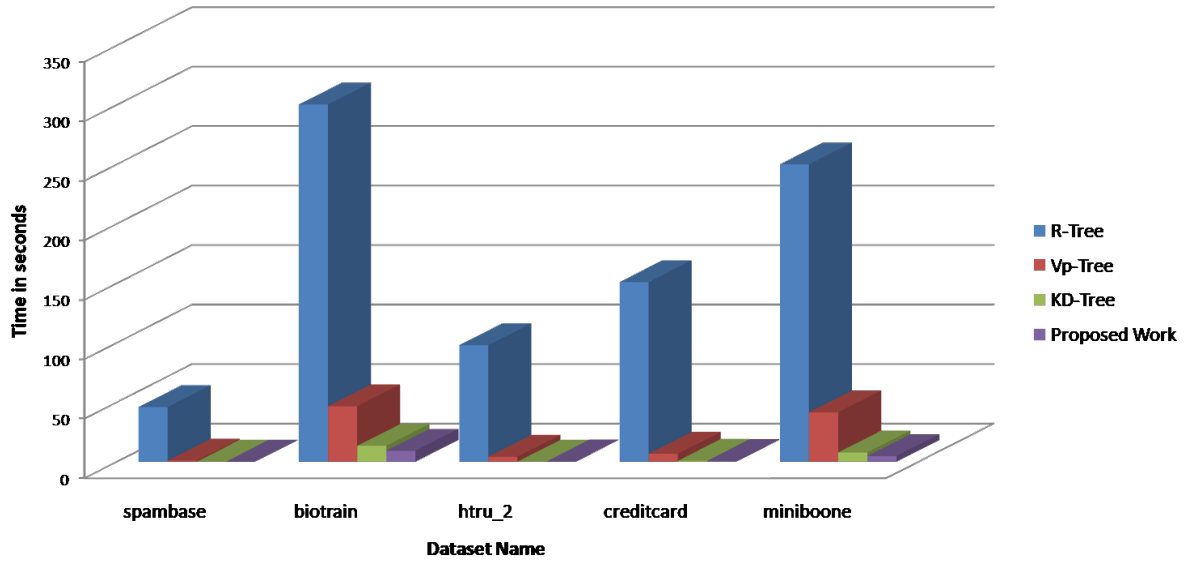

Figure 4: Indexing Time Comparison

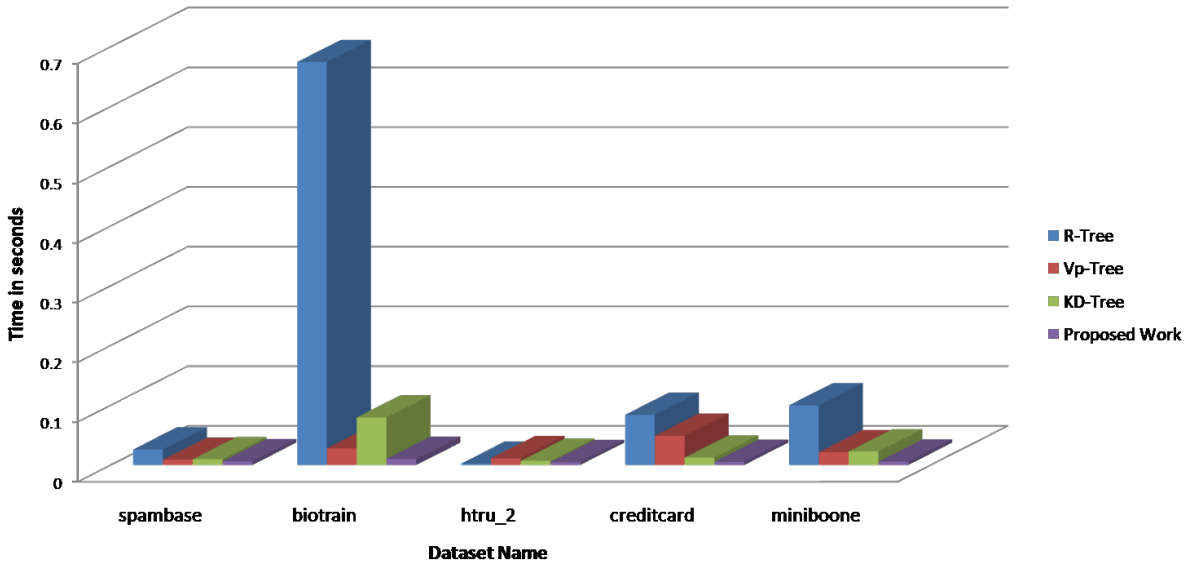

Figure 5: Querying Time Comparison

Table 6
Indexing time comparison between KD-Tree and Proposed Work

Dataset Name	Dataset Size	KD-Tree Indexing time	Proposed Work Indexing time
spambase	4601	0.097132921	0.147065997
HTRU2	17898	0.088246346	0.336842656
credit card	30000	0.799064255	0.696575999
MiniBooNE	130065	7.842401028	4.795480847
bio_train	145751	13.37806582	9.259319027

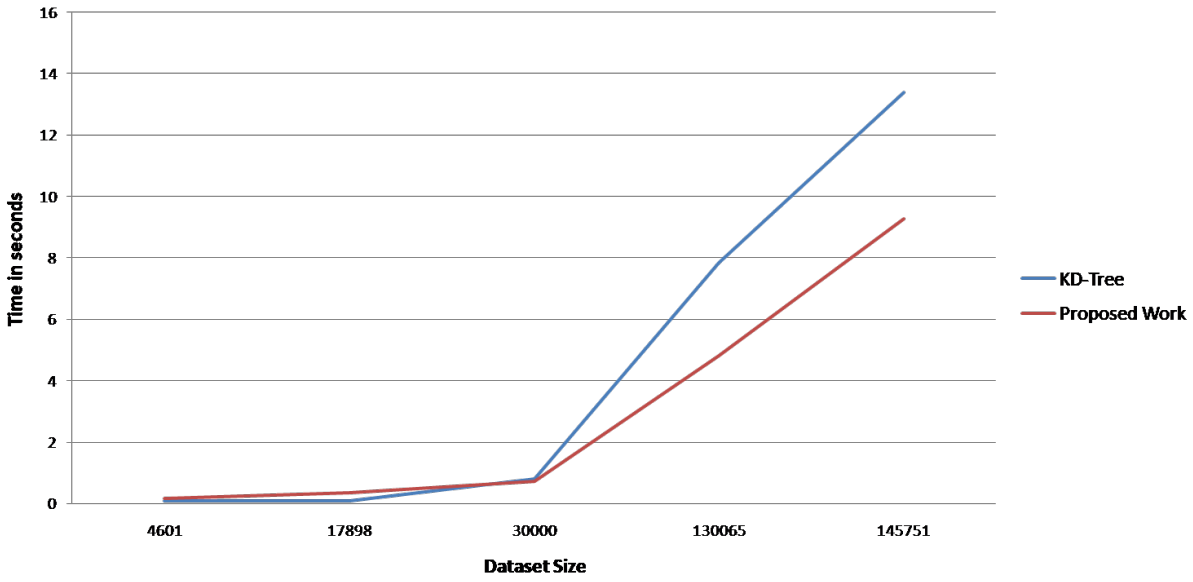

Figure 6: KD-Tree and Proposed Work Comparison

References

[1] Yianilos, Data structures and algorithms for nearest neighbor search in general metric spaces, Fourth annual ACM-SIAM symposium on Discrete algorithms (1993). doi:10.1145/313559.313789.

[2] A. Guttman, R-trees: A dynamic index structure for spatial searching, Proceedings of the 1984 ACM SIGMOD international conference on Management of data – SIGMOD '84 (1984). doi:10.1145/602264.602266.

[3] C.-C. Huang, H.-Y. Chang, A novel svm-based reduced nn classification method., 11th International Conference on Computational Intelligence and Security (2015). doi:10.1109/CIS.2015.23.

[4] S. Maneewongvatana, D. M. Mount, It's okay to be skinny, if your friends are fat., 4th Annual CGC Workshop on Comptutational Geometry (1999). doi:10.1.1.39.8380.

[5] C. X. H. Z. Wenfeng Hou, Daiwei Li, T. Li, An advanced k nearest neighbor classification algorithm based on kd-tree, IEEE International Conference of Safety Produce Informatization (IICSPI) (2018). doi:10.1109/IICSPI.2018.8690508.

[6] S. T. E. J. R. T. L. W. J. C. V. Hyvönen, T. Pitkänen, T. Roos, Fast nearest neighbor search through sparse random projections and voting., IEEE International Conference on Big Data (2016). doi:10.1109/BigData.2016.7840682.

[7] O. B. Mario Lucic, A. Krause, Strong coresets for hard and soft bregman clustering with applications to exponential family mixtures., arxiv.org (2015).

[8] A. K. O. Bachem, M. Lucic, Scalable k-means clustering via lightweight coresets., ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD) (2018). doi:10.1145/3219819.3219973.

[9] scipy.org, Spatial kdtree class, 2020. URL: https://docs.scipy.org/doc/scipy-0.14.0/reference/generated/scipy.spatial.KDTree.html.

[10] c. ics.uci.edu, Datasets used, 2007. URL: https://archive.ics.uci.edu/ml/datasets.php, osmot.cs.cornell.edu./kddcup/datasets.html.