Project Assignment

K Nearest Neighbour Classification Algorithm Based on KD-Trees Data Structure

Student(s): Siddharth Gupta-20200620

Atul Kumar Singh-20200619

Lecturer: Dimitrios Tselios

Module: COMP47500 Advanced Data Structures in Java

Course: Master's in Computer Science by Negotiated learning

Date Due: 18th December 2020

ABSTRACT

KNN (K Nearest Neighbour) is the supervised Machine Learning Algorithm in which we try to classify the unseen test data based on the class labels of the trained data sets. We also utilize it to predict the class labels of the test data points. Unlike other supervised Machine learning algorithms, KNN does not have any Mapping function through which a classification algorithm tries to learn from the training set. Instead, it retains the information of the full training data Hence, called the Lazy Learning Approach. The prediction of the labels of the test data points happens at the Run-time.

This classification technique works on the minimum distance approach. The distance metric often used is the Euclidean distance. The algorithm tries to find out the nearest neighbour (s) in the entire training data set. The main part here is that the time taken for the prediction of the nearest neighbour takes a significant amount of time. Here, we propose an efficient data structure called KD-Trees to store the training data sets to improve search operations. at a later stage, we will be concluding that KD-Trees can be used as a Data Structure in KNN Machine Learning Classification Algorithm to improve the search performance while finding out the nearest neighbour(s).

Table of Contents

1. INTRODUCTION TO KD-TREES

The KD-Tree Data Structure was invented in the 1970s by Jon Bentley. It was primarily developed to denote dimensions at each level of the tree, like 5d trees will have 5 dimensions at different levels of the tree. Considering that the KD-Tree has L levels, then for all levels starting from {1 to L}, a particular dimension is denoted by dim=level mod k. This dimension is commonly known as the splitting dimension. During traversal of the KD-Tree at a particular level, a comparison between the splitting node happens. If the comparison at the splitting node is less, we further move to the left branch otherwise we move to the right branch. This is how it is based on the underlying concepts of Binary Search Tree.

To Illustrate this with an example, we have a set of 2-Dimensional arbitrary space called Enquiry set. The set contains elements as: {(9, 5), (10, 2), (4, 8), (6, 10), (5, 2), (5, 7), (2, 1), (7, 3), (6, 2), (5, 3)}

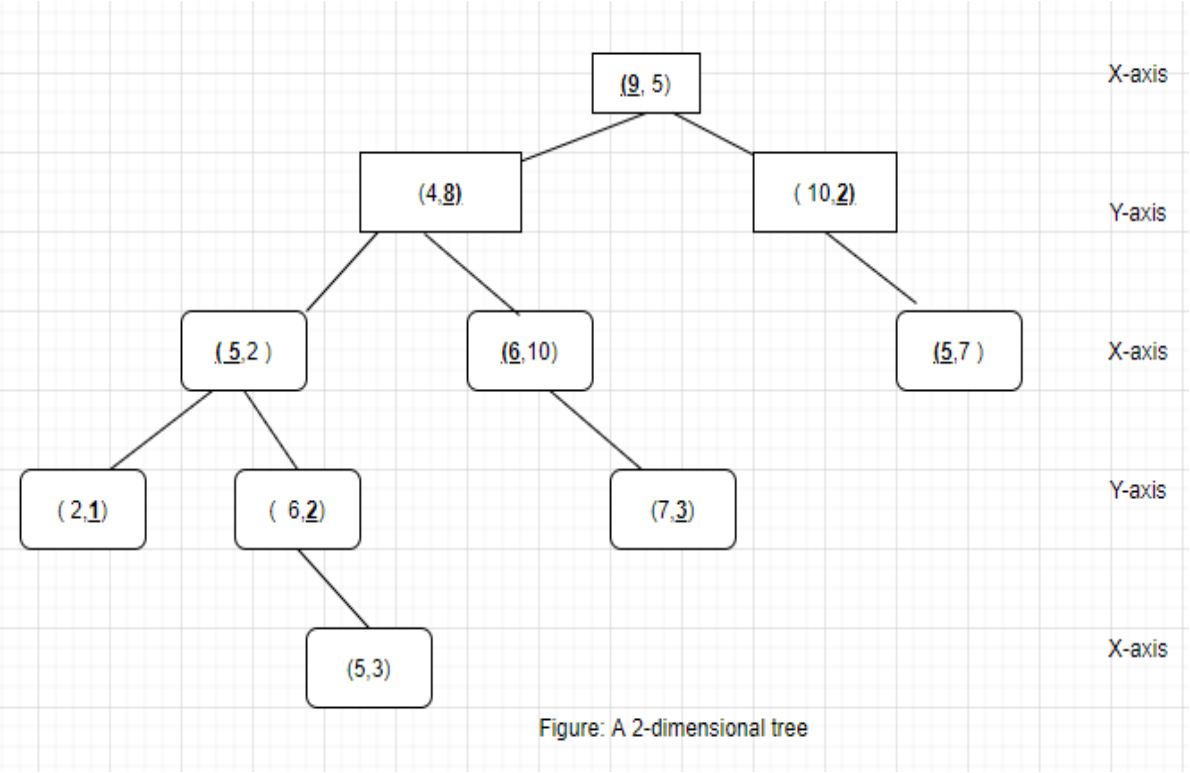

Figure: A 2-dimensional tree

For the first level, X-axis is the spitting dimension. (9,5) serves as the root node. (4,8) and (10,2) are compared with the root node. (4,8) compared less in the x-axis. (10,2) compared higher with the root node. For the level-2, Y-axis is the splitting dimension. (5,2) branches left of (4,8) because it compared less on the y-axis. Similarly, (6,10) branches right of (4,8) because it compared higher on the y-axis.

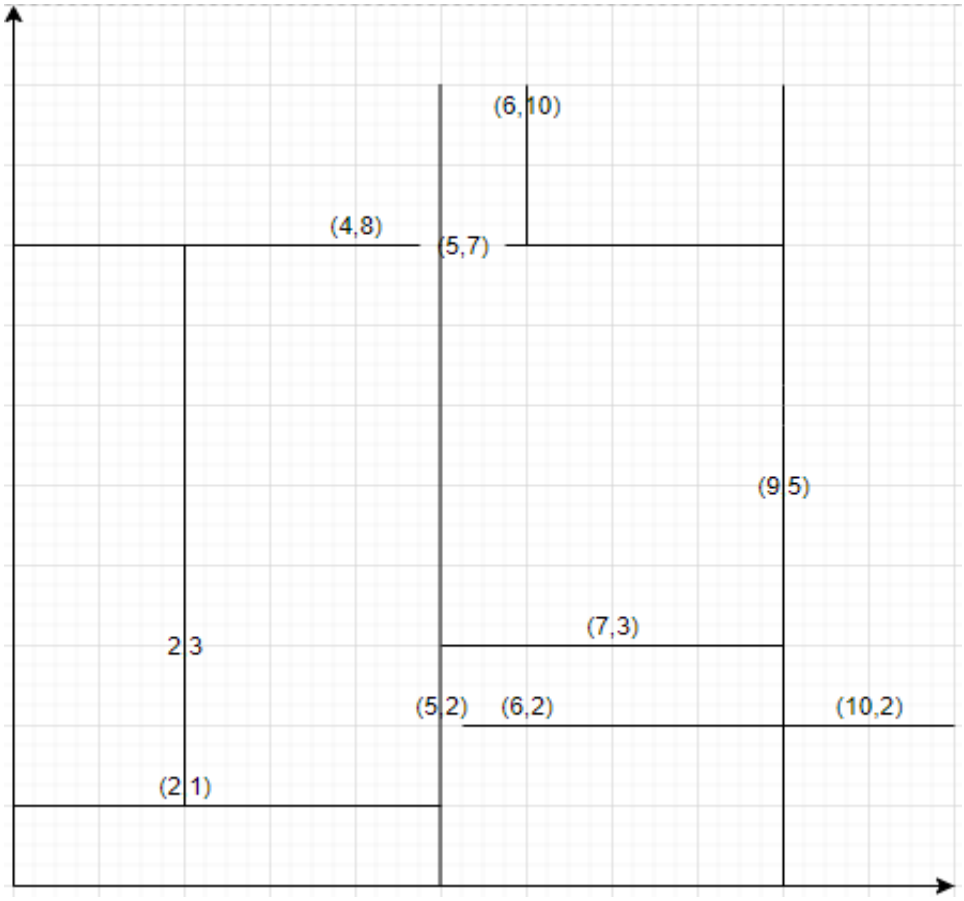

Figure: Set Visualization in 2-Dimensional Space

The figure shows that the 2-D space has been partitioned by the lines in X and Y direction. Hence the KD-Trees are also knowing as space-partitioning algorithm. The 2-dimensional tree is easy to visualize. However, as the dimensionality increases, visualization becomes difficult. The KD-Trees can be useful in applying higher level of dimensions.

---

2. PROBLEM SOLVED

2.1 K NEAREST CLASSIFICATION USING KD-TREES

KNN is the classification machine learning technique where we train a set of input data. All the input data in the training set is then utilized to classify data available in the test set. The classification of the new input data happens at the run-time. Hence, it is classified under the lazy learning approach.

Since we calculate the distance of the new sample data with all the data in the training set, the time complexity of the KNN algorithm will be proportional to the number of data points in the training set. The time complexity is O(N). For larger data sets this process will be a bit time-consuming. It then becomes a crucial step to consider reducing the time complexity of this algorithm. [2] KD-Trees can be utilized to efficiently store the training data sets. It is the representation of the Binary search tree and hence can significantly increase the search performance. KD-Trees has the power to efficiently search in a fast neighborhood in a huge data environment.

A method has been proposed to build the KNN Classification based on the KD-Trees. We start at the root node by informing the metric to partition the dimension. The data is then arranged into left and right sub-tree and obtained recursively until we obtained a balanced KD tree [2]. Apart from this, an index table is maintained to gather the label information of the nodes.

SEARCH AND CLASSIFICATION USING KD-TREE

The forward search is basically derived from the BST implementation that says if the node value is less than the axis-data then the forward search will go into the left subtree otherwise it will go to the right sub-tree. The key step here is to note down the search path compared to the new node since it will also be used for backtracking purposes. We keep on updating the point until we find the nearest to the node provided. For labelling purpose, we provide the label of the nearest point to the test data.

Algorithm: KNN Classification based on KD-Tree

INPUT: KDroot [root of the KD-Tree], test_data

OUTPUT: To find out the label of the test data

=> While the KDroot is not empty we start by adding root of the KDTree into the search path.

=> If KDroot [dimension] > test_data[dimension] then Search(right-subtree)
Else Search (left subtree)

=>Calculate nearest distance for all the points in the forward search path with the given test_data.

Distance_nearest= distance between (point, test_data)

If (point[dimension] – test_data[dimension]) > (test_data[dimension] - KDroot[dimension])
Then we travel to next child of the KDroot.

We then calculate the distance between child and test_data.

Distance = distance between (child, test data)

If this Distance is less than the Distance_nearest then update the nearest distance.

=>Assign the label of the test data with the label of the nearest point.

The effective time complexity of the KNN algorithm by utilizing the KD trees comes out to be O (N raise to(1-1/d) + s). Here, s is the no. of nodes and d is the number of dimensions of the KD-Tree. Experiment shows that in the Blood dataset with 5 dimensions and 700 data points, the search operation with the KNN-KD tree combination was $1/4^{th}$ of KNN.

A significant advantage is there when the KD-Tree is balanced. It offers a time complexity of O(sqrt(n) + k) which is much better than O(n) when the tree is un-balanced.

2.2 OTHER APPLICATIONS

The KD-Tree Algorithms has its applications in enhancing the real-time performance of the Graphical Processing Units (GPU). In this algorithm, the nodes of the KD-Trees are constructed in the Breadth-First Search Order [3]. The strategy involves utilizing larger nodes that are placed at the upper level of the tree and achieve the parallelization in the computation through geometric measures. The KD-Tree balancing Quality has been maintained by faster retrieval of Node Split criteria.

This KD-Tree algorithm built on the GPUs has significant advantage over single-core CPU Algorithms. The reason is because this algorithm deals with applications that involves dynamic nature of the scenes. This involves GPU ray tracing, interactive photon mapping, and point cloud modeling. [3]

3. IMPLEMENTATION

The KD-Trees are implemented in such a way that each node corresponds to a binary search tree and the node denotes a k-dimension point in the space. Each node should have a key value and a measure to split the dimension. The algorithm is very well versed in organizing dimensional point in the space. An Internal node divides the space into left and right sub-trees. Points that occur in the left part of the space are represented by the left subtree whereas, points on the right of the space are represented by the right subtree. The recursive approach is then followed for all the subsequent sub-trees for the space partition till an empty node in the tree is reached.

3.1 INSERT OPERATION

The splitting dimension measure stores the information at each node in the given level of the KD-Trees, this information is essential in notifying the appropriate usage of the dimension at each level of recursion. The insertion procedure involves the addition of the node with a key and a value into the KD-tree. The insertTree (Ktree,elem,value) will insert the node into the kd-tree. Initially the tree is empty. The insertNode (elem, value) will insert the node in the current leaf position.

Algorithm: kd-trees Insertion

Process insertTree(Ktree, elem, value)

```
If Ktree = Null then insertNode(elem, value)
pointer ← Ktree.pointer; k ← T.splitFunction
if elem[k]< pointer[k] then insertTree(Ktree.left, elem, value)
else insertTree(Ktree.right, elem, value)
```

The figure below shows the insertion of elements in a 2d Tree [1]. The range for X and Y in the region is from 0 to 10. We have an assumption for the insertion scenario that all the points mentioned in the below figure will lie inside the range of X and Y. Root (6,4) insertion divided the plane into two regions. The second node insertion i.e., (5,2) affected X values from 0 to 6 but Y values still range from 0 to 10. The third node (4,7) impacts the X region from 0 to 6, and the Y region from 2 to 10. In this way, nodes represent a bounding box for themselves.

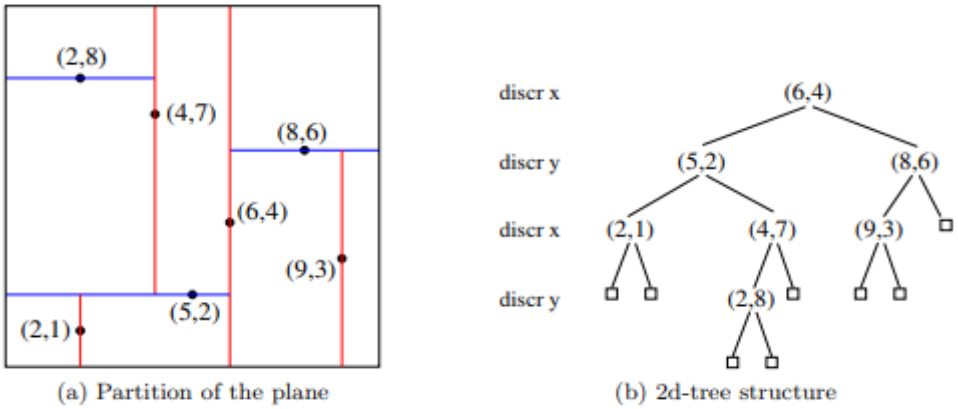

Figure: Insertion of elements in the 2d-Tree

3.2 SEARCH OPERATION

The search operation in KD-Tree is more like that of the binary search tree. It returns if a certain element is present in the KD-Tree, otherwise, it returns not found. Another parameter called depth is used to locate the current axis as the search is depth-first**.**

Algorithm: KD-Trees Search operation

Process SearchOperation(Ktree, elem)

If Ktree = Null, then return no such element found

pointer ← Ktree.pointer; k ← T.splitFunction

if elem = pointer then return element found

if elem[k]< pointer[k] then SearchOperation(Ktree.left, elem)

else SearchOperation(Ktree.right, elem)

We search the (2,8) node by following the path provided in the below tree.

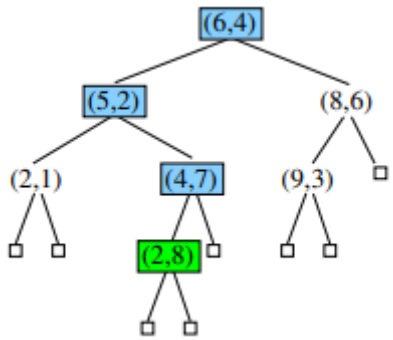

Figure: Search Operation (2,8)

3.3 ORTHOGONAL RANGE SEARCHING

The orthogonal range searching fetches all the nodes lie within the lower bound and upper bound. Usually, it is represented as a triangle with a lower and upper bound. All the points that lie inside the triangle are returned. [1]

Rectangle lower bound[k] <= elements[k]<= Rectangle upper bound[k]

We are utilizing Internal () function that will check whether the node points lie inside the triangle or not given that we have lower and upper bound [1]

Algorithm: Orthogonal range Search operation

Process SearchOperationOrthogonal(Ktree, Lbound, Ubound, set)

If Ktree = Null, then return

pointer ← Ktree.pointer; k ← T.splitFunction

if Internal (pointer,Lbound, Ubound) then pointer belongs to set

if LBound[k] < elem[k] then SearchOperationOrthogonal (Ktree.lef t, Lbound, Ubound, Set)

if Ubound[k] ≥ elem[k] then SearchOperationOrthogonal (Ktree.right, Lbound, Ubound, Set)

In the below figure, we have defined the lower bound as 1 to 5, and the Upper bound as 5 to 9 of the rectangles. The nodes that are internal to the given orthogonal query are (4,7) and (2,8), highlighted in Green. The traversal started from the root node (level 0), it then visited the (5,2) and discarded (8,6) at level 1. In the second level, the left subtree of the (5,2) is (2,1). It is discarded and the traversal continues to the node (4,7). This node is marked in the set as found. Later, the level-3, traversal continues to point (2,8). This point is also included in the set as found. On the level 4, both the right and left subtree are empty.

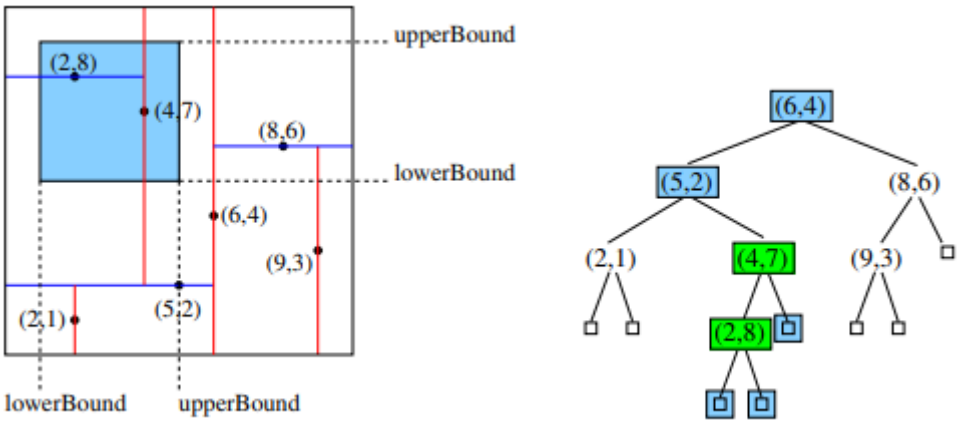

Figure: Orthogonal Range search

3.4 SEARCH OPERATION: NEAREST NEIGHBOUR

In this type of search criteria, the algorithm returns the nearest point(s) concerning some given point in space. Usually, a distance function such as Euclidean is used to find the distance between the neighboring points. When we are searching the nearest neighbor concerning a node N. The algorithm starts with a certain node in the KD-Tree, it then calculates the distance between this node and the node N. If this distance is minimum, the distance value is stored. Otherwise, it is discarded. The left and right subtree are visited to find the potential minimum distance until now. This algorithm utilizes a priority queue to keep the track of the node values of the sub-trees. Every node has the information of its root node and its distance from Node N. We have the priority queue in the sorted order of their potential distance from Node N. The algorithm will try to look for the nearest neighbor in the sorted order of the sub-trees.

At each loop of this algorithm, the calculation between the root of the subtree and Node N is done. If this distance comes out to be smaller than we have found till now, we update the information. The information related to potential distances between the two sub-trees is stored only if it is smaller than the lowest distance found. Otherwise, if the distance is greater than the lowest distance found, the whole subtree is discarded.

The exit criteria of the loop are defined as when the priority queue gets empty or when the top element of the priority queue has a distance larger than the minimum distance found so far. This situation claims that we got the nearest neighbor of the Node N.

Based on the Euclidean distance, the below picture shows that the node closest to point (9,8) is (8,6). The algorithm first visited all the nodes in blue, (5,2) is still stored in the priority queue and yet to be visited. Later, the node (5,2) will be discarded since its bounding box has range values of X as $(-\infty, 6)$ and Y and $(-\infty +\infty)$. Hence, the potential distance with the point (9,8) has become 3 which is larger than the initially found $\sqrt{5}$. [1] which is the distance between the points (9,8) and (8,6).

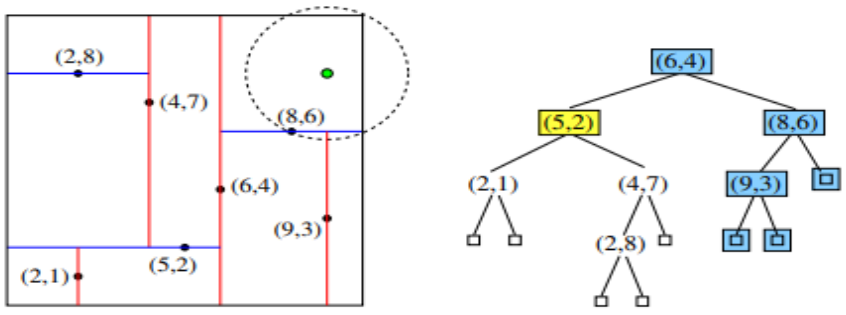

Figure: Nearest Neighbor in KD-Tree

4. Conclusion and future work

In this paper we have discussed a variation of Binary Search tree called KD-Trees, its implementation, and its beneficial usage in finding Nearest Neighbours by using the KNN Classification algorithm. We have seen that KD-Tree algorithm can be useful in reducing the Time Complexity as the earlier brute force method takes O(N) and may not be practical for large applications. The KD-Trees serve as an alternative for the Search algorithms.

We utilized the KD-Trees for the KNN Classification algorithm and were successful in reducing the time complexity from **O(N)** to **O (N raise to(1-1/d) + s).** This survey shows the improvement in the search efficiency. For the future work, we will be focusing on improving the Space complexity of the KNN-KD Tree algorithm. Also, all the dimensions in the KNN Tree Classification have been given equal weightage, we will then try to give higher weightage to the more important dimensions to have a better score on Accuracy.

5. References

[1] MARIA MERCE PONS CRESPO

Design, Analysis, and Implementation of New Variants of Kd- Trees, Universitat Politecnica de Catalunya Departament de Llenguatges I Sistemes Informatics Master en Computacio

[2] Wenfeng Hou, Daiwei Li, Chao Xu, Haiqing Zhang, Tianrui Li
An Advanced k Nearest Neighbor Classification Algorithm Based on KD-tree,
College of Software Engineering, Chengdu University of Information Technology

[3] Kun Zhou Qiming Hou_ Rui Wang† Baining Guo

Real-Time KD-Tree Construction on Graphics Hardware,

Microsoft Research Asia _Tsinghua University †Zhejiang University

[4] KD-Trees, **CMSC 420**

[5] KD-Trees, **OpenDSA, CS3 Data Structures and Algorithms**

[6] E. Horowitz, S. Sahni, Fundamentals of Data Structures, **Computer Science Press**