K-Dimensional Trees

Alexander Raschl	*K01556188*
Markus Laube	*K01557290*
Stefan Brandl	*K01555842*

Introduction

- Invented by Jon Louis Bentley in 1975
- Index structure for managing multidimensional objects
- k is the dimensionality of the search space
- Space partitioning is performed via building a binary search tree
- Each node is equal to a k-dimensional point
- Performs only axis aligned splits

Application Domain

- Mostly theoretical
- Can be used for
  - Nearest Neighbor Search
    - Search k-dimensional point in tree nearest to a given point
    - Suffers from curse of dimensionality
  - Range searches
    - Search for ranges of attributes
    - For example query income-ranges for persons
    - Worst complexity O(k * N^(1-1/k))

Complexity

Action	Average Complexity	Worst Complexity
Space	O(n)	O(n)
Insertion	O(log n)	O(n)
Query	O(log n)	O(n)
Removal	O(log n)	O(n)

Examples

- Construction
- Insertion
- Deletion
- Query / Lookup
- NN-Search

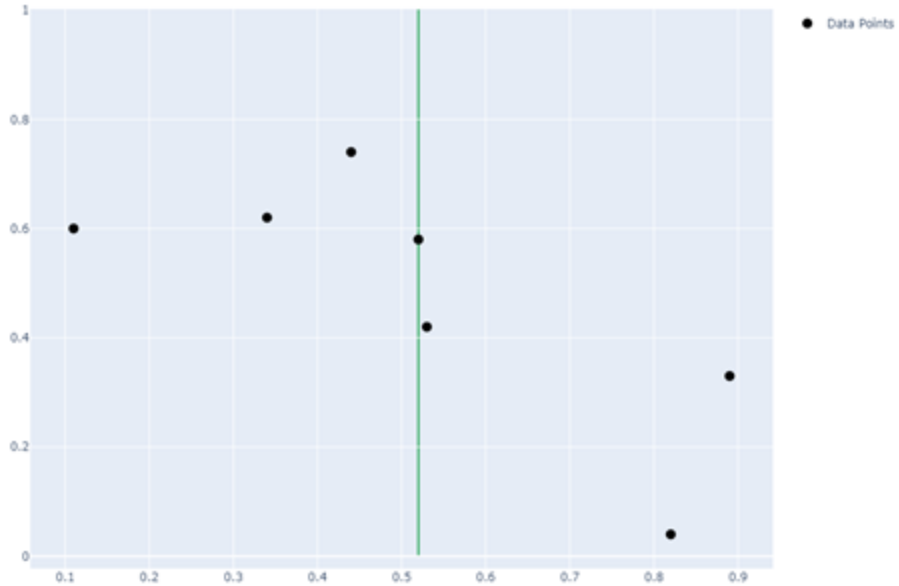
Construction

Construction

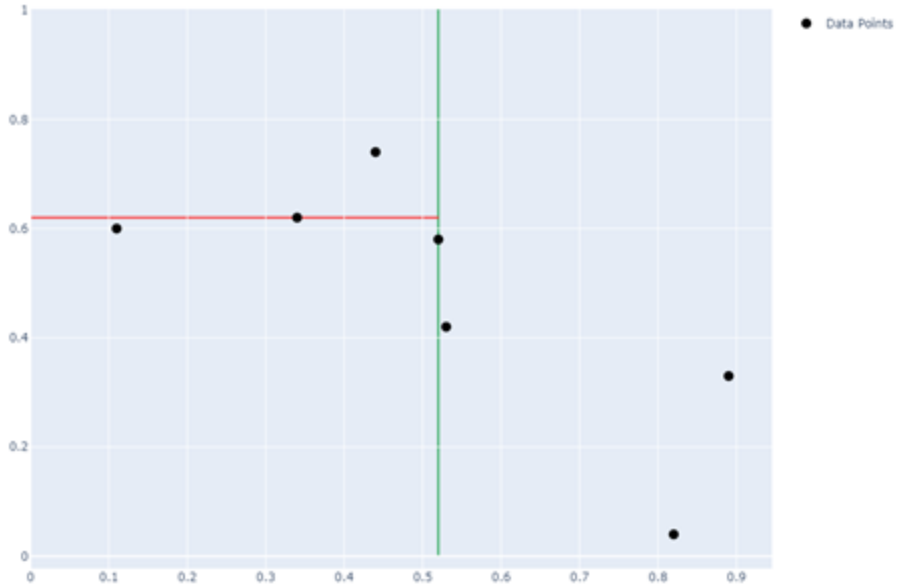
1. as long as >= 1 element remaining:

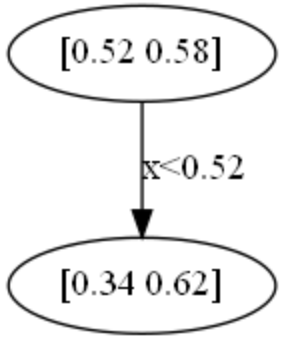
   Pick a split dimension (f.e. x or y)

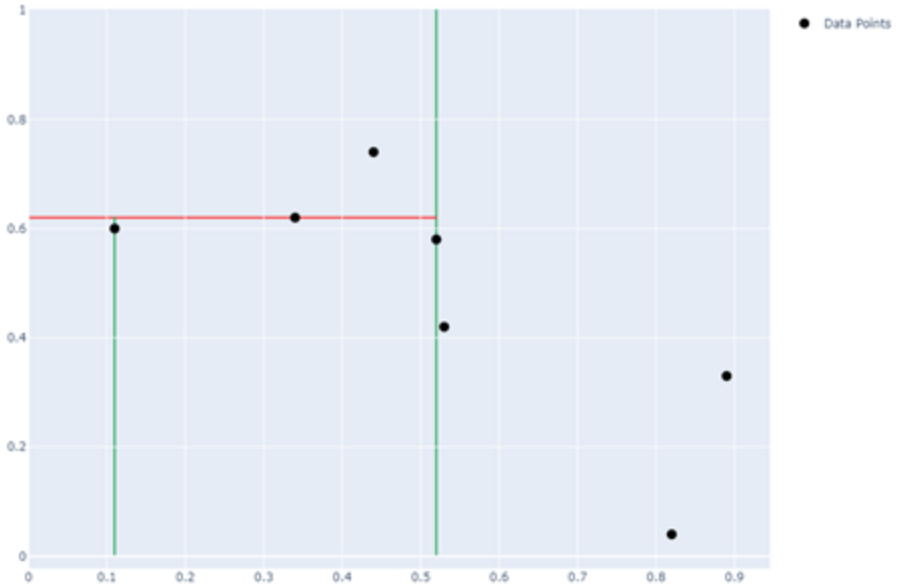
   - canonical (increasing)
   - random
2. Compute the median
3. Axis aligned split into left and right subtree at median
4. repeat 1

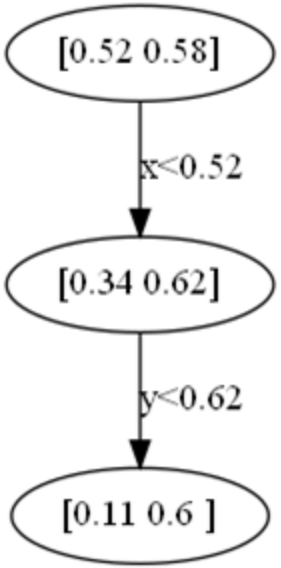


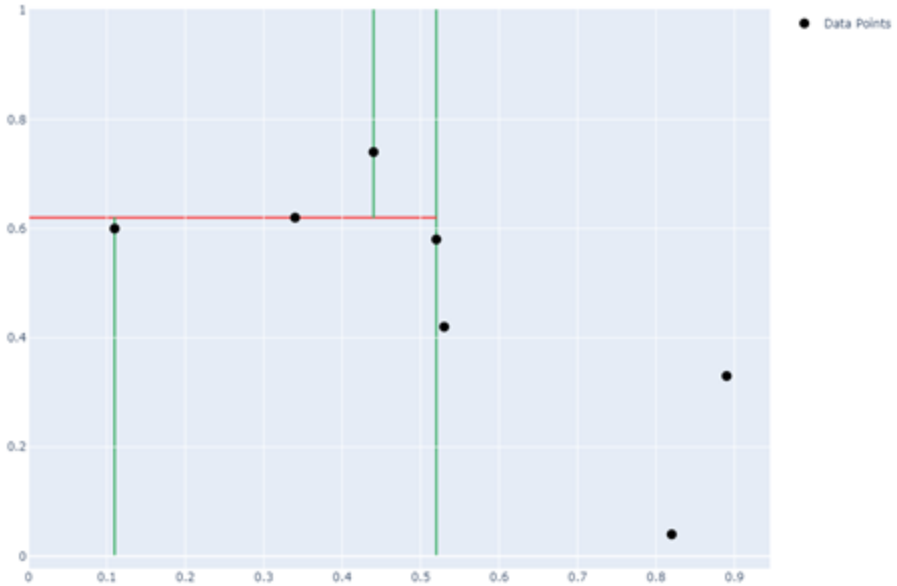
[0.52 0.58]

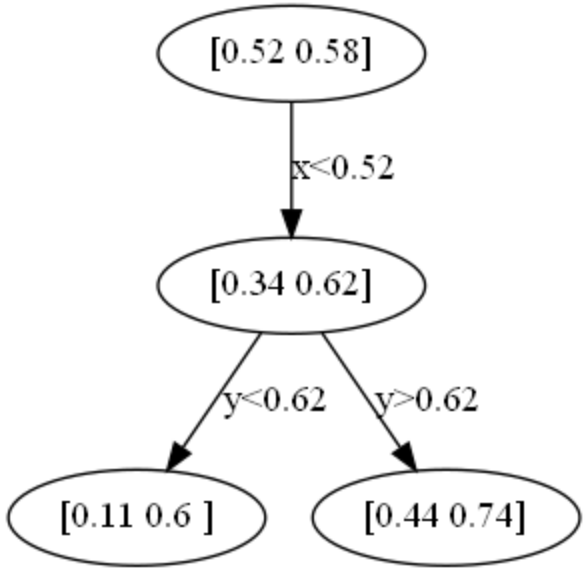

Data Points
1
0.8
0.6
0.4
0.2
0
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9

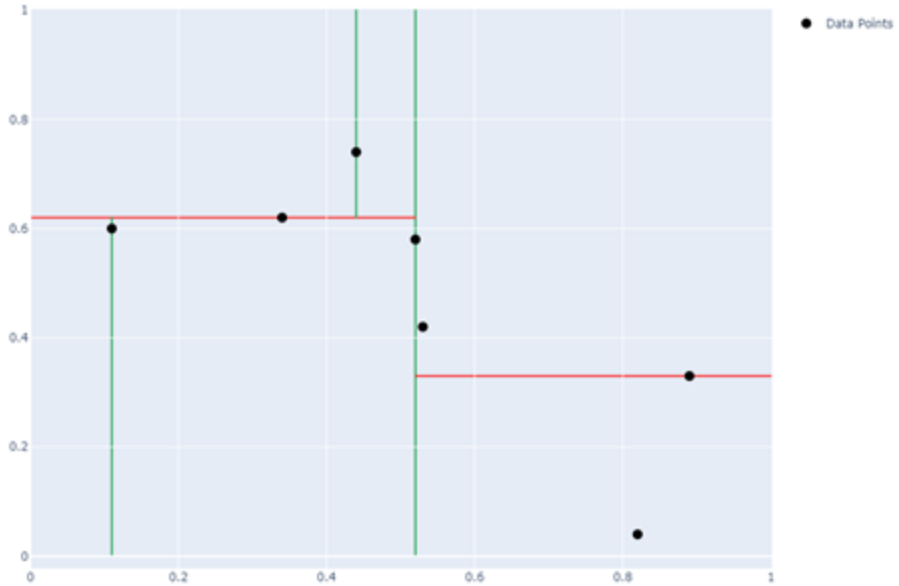

[0.52 0.58]
x<0.52
[0.34 0.62]

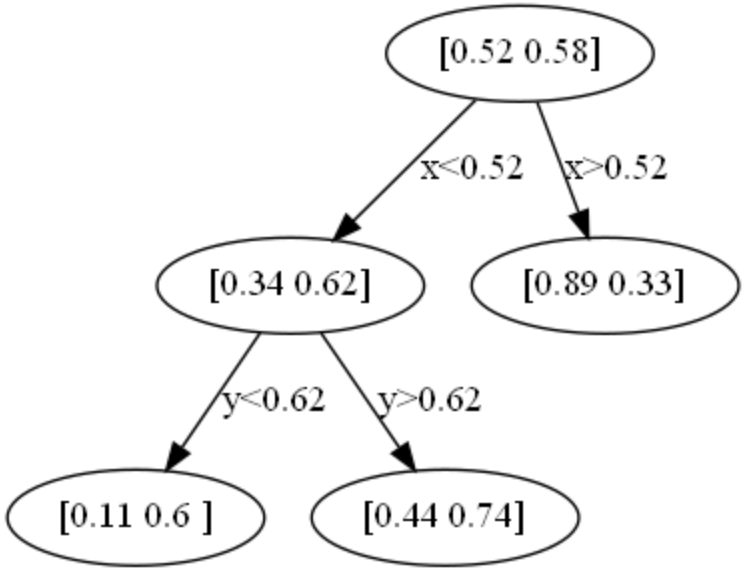

Data Points
1
0.8
0.6
0.4
0.2
0
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9

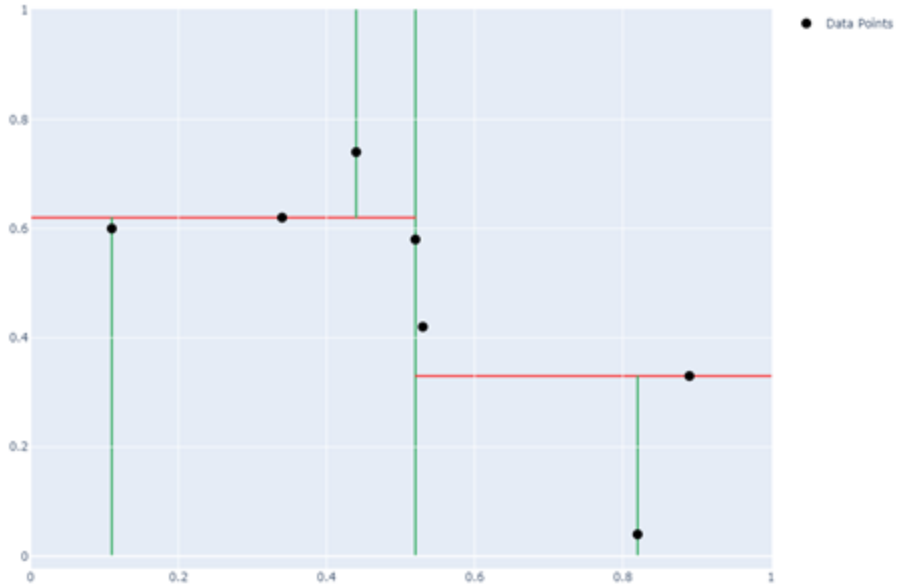

[0.52 0.58]
x<0.52
[0.34 0.62]
y<0.62
[0.11 0.6]

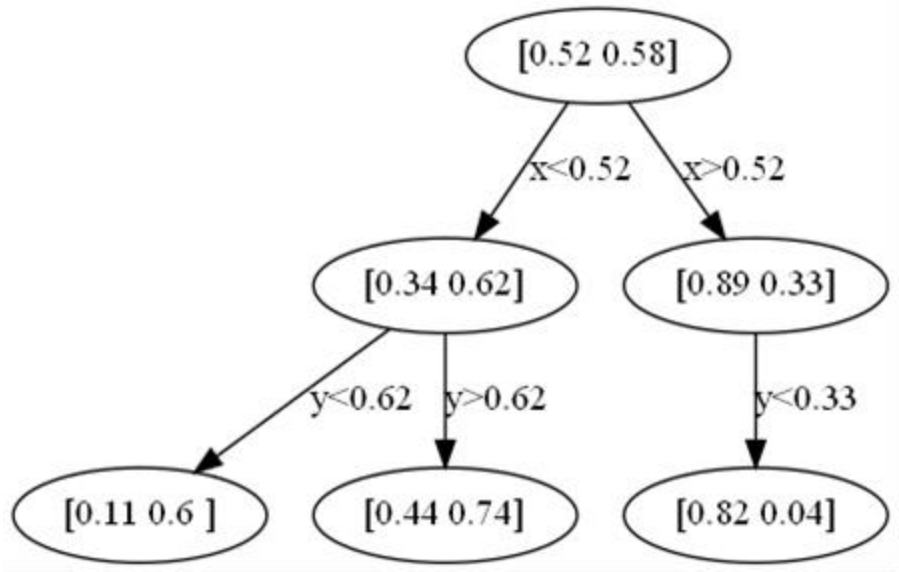

Data Points
1
0.8
0.6
0.4
0.2
0
0
0.1
0.2
0.3
0.4
0.5
0.6
0.7
0.8
0.9

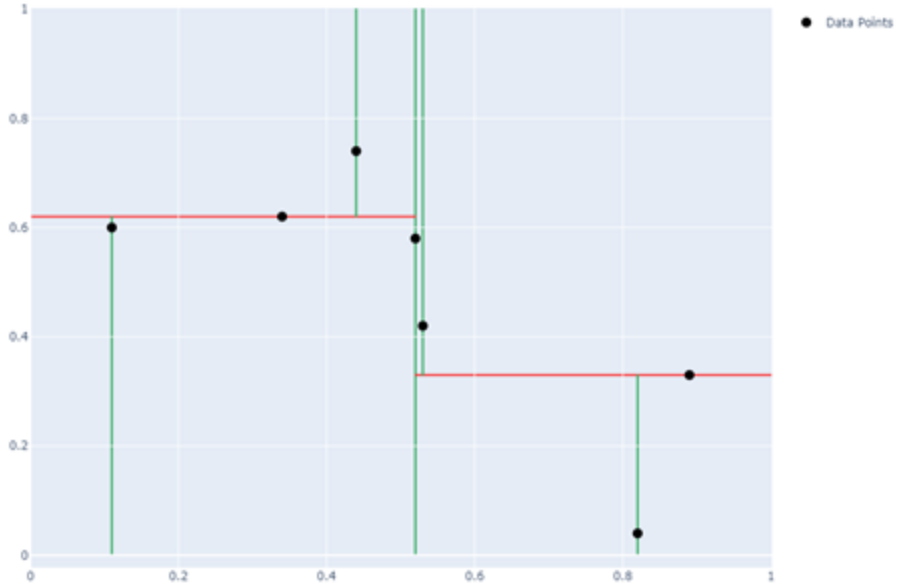

[0.52 0.58]
x<0.52
[0.34 0.62]
y<0.62
y>0.62
[0.11 0.6]
[0.44 0.74]

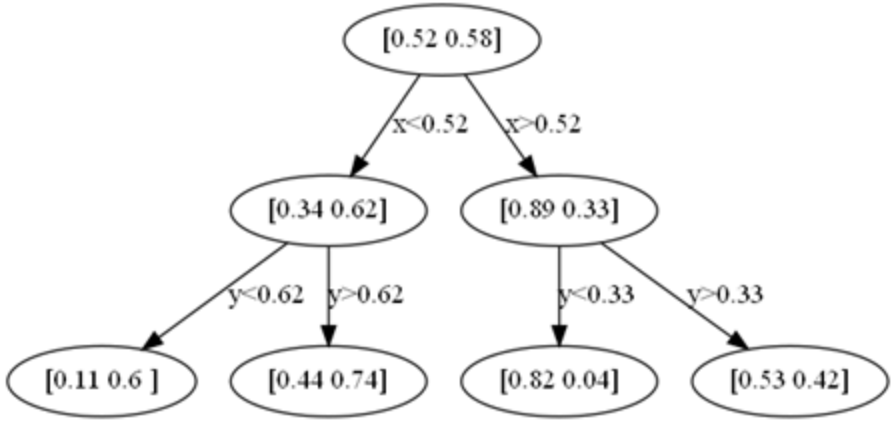

Data Points
0
0.2
0.4
0.6
0.8
1


[0.52 0.58]
x<0.52
x>0.52
[0.34 0.62]
[0.89 0.33]
y<0.62
y>0.62
[0.11 0.6]
[0.44 0.74]


Data Points
1
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1


[0.52 0.58]
x<0.52
x>0.52
[0.34 0.62]
[0.89 0.33]
y<0.62
y>0.62
y<0.33
[0.11 0.6]
[0.44 0.74]
[0.82 0.04]


Data Points
1
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1

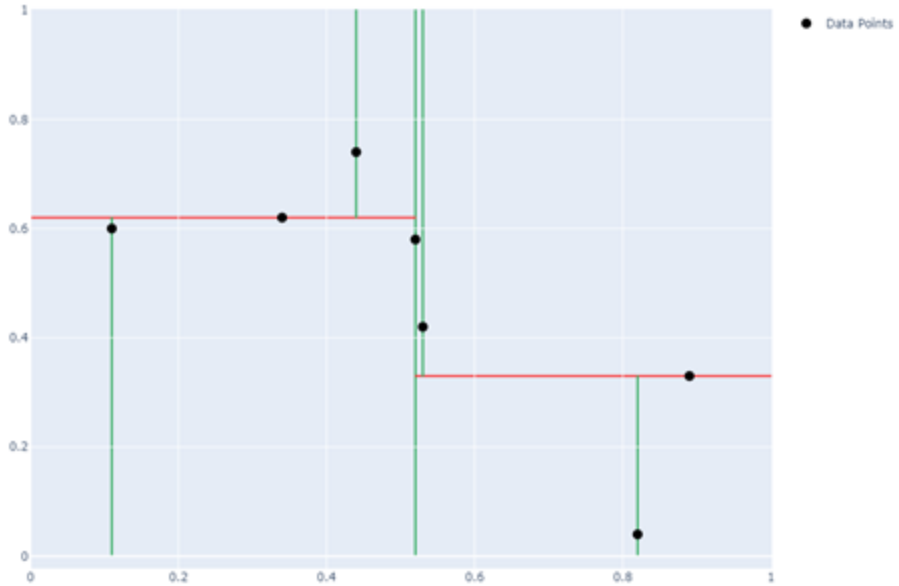

[0.52 0.58]
x<0.52
x>0.52
[0.34 0.62]
[0.89 0.33]
y<0.62
y>0.62
y<0.33
y>0.33
[0.11 0.6]
[0.44 0.74]
[0.82 0.04]
[0.53 0.42]

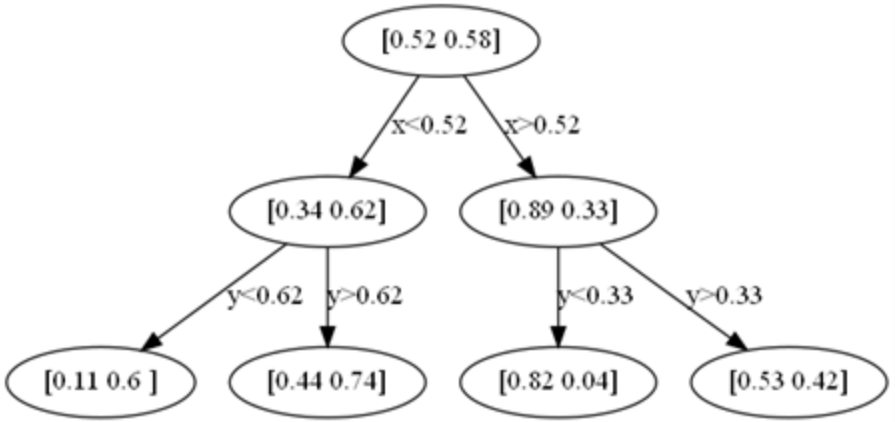
Insertion

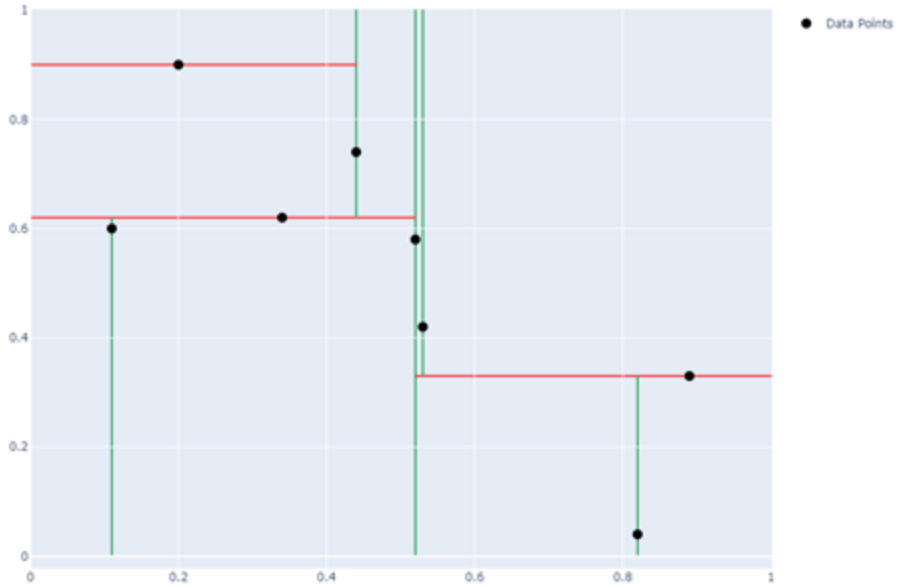
Insertion

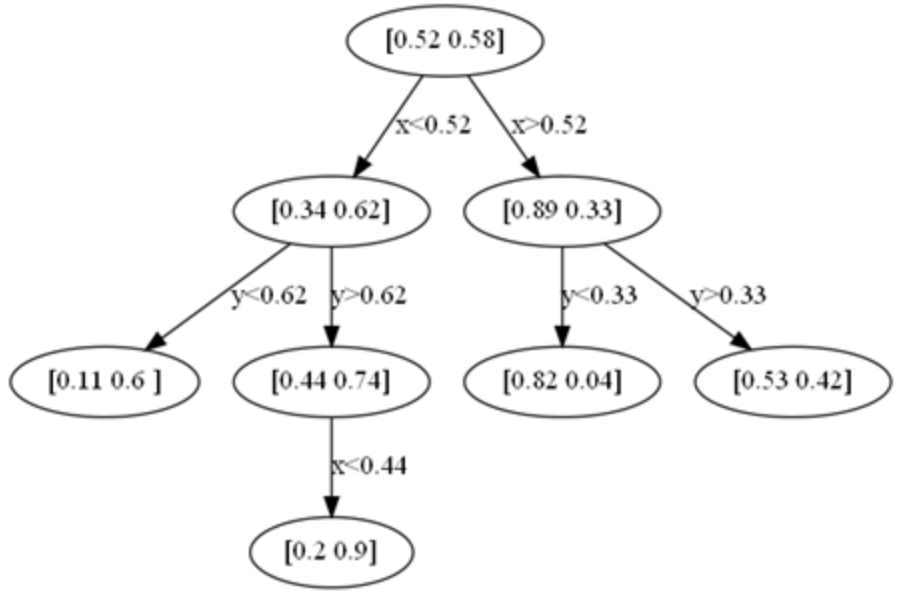
1. Start traversing the tree from the root
2. Compare point to insert with current node
3. Choose left or right subtree depending on comparison
4. Traverse until leaf is reached
5. Insert node below the leaf depending on its relative position

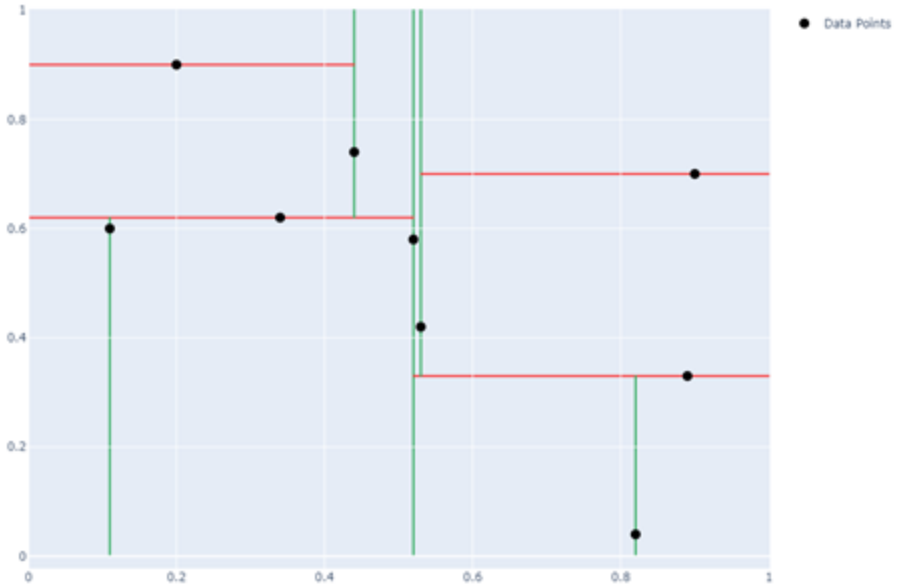
- Warning:
  - Tree may become unbalanced during insertions (NN-Search performance suffers)
  - Rebalancing steps possible

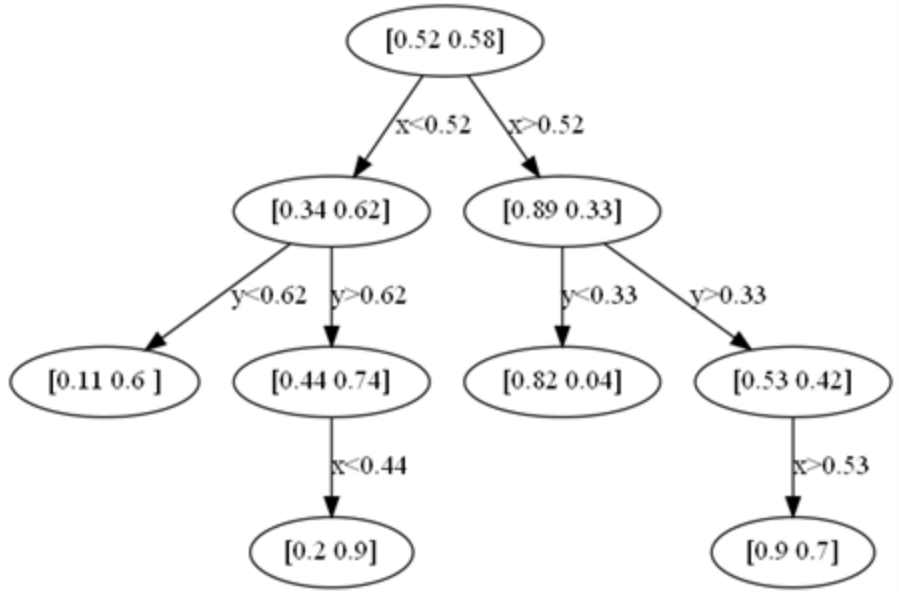

Data Points
1
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1

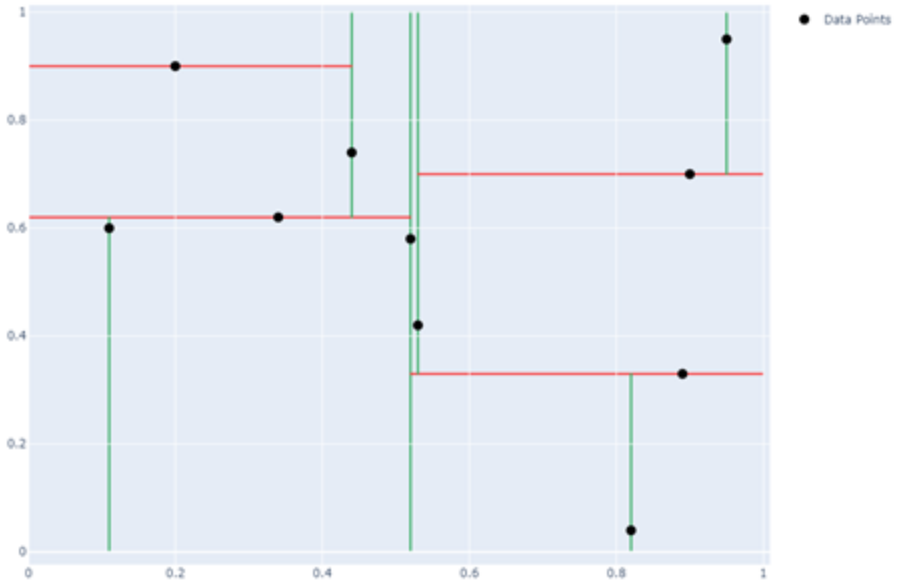

[0.52 0.58]
x<0.52
x>0.52
[0.34 0.62]
[0.89 0.33]
y<0.62
y>0.62
y<0.33
y>0.33
[0.11 0.6]
[0.44 0.74]
[0.82 0.04]
[0.53 0.42]

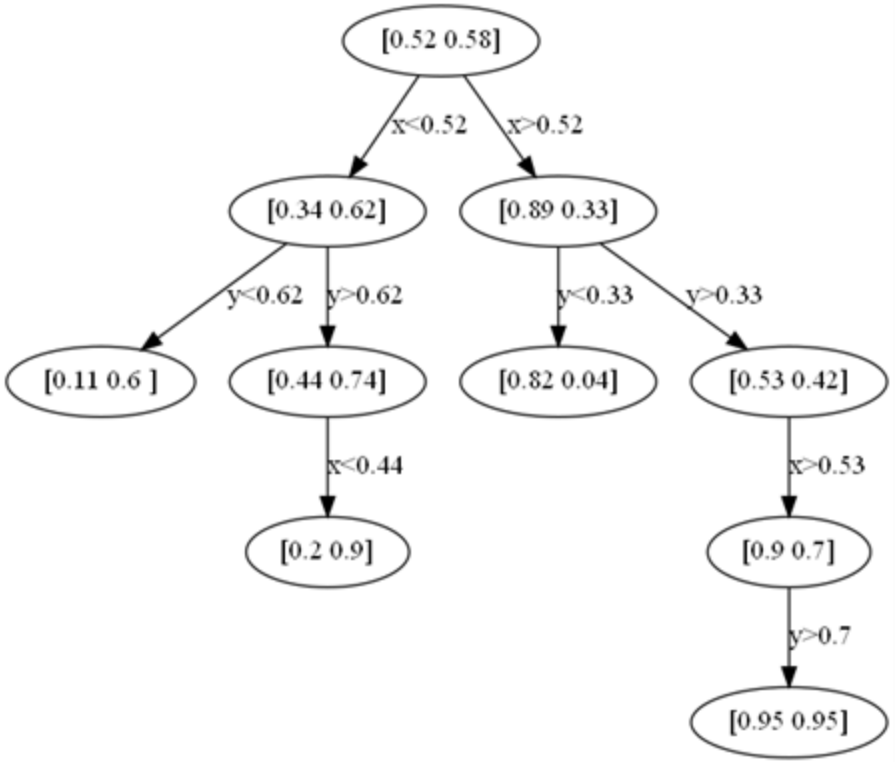

Data Points
0
0.2
0.4
0.6
0.8
1


[0.52 0.58]
x<0.52
x>0.52
[0.34 0.62]
[0.89 0.33]
y<0.62
y>0.62
y<0.33
y>0.33
[0.11 0.6]
[0.44 0.74]
[0.82 0.04]
[0.53 0.42]
x<0.44
[0.2 0.9]


Data Points
0
0.2
0.4
0.6
0.8
1


[0.52 0.58]
x<0.52
x>0.52
[0.34 0.62]
[0.89 0.33]
y<0.62
y>0.62
y<0.33
y>0.33
[0.11 0.6]
[0.44 0.74]
[0.82 0.04]
[0.53 0.42]
x<0.44
x>0.53
[0.2 0.9]
[0.9 0.7]


Data Points
0
0.2
0.4
0.6
0.8
1

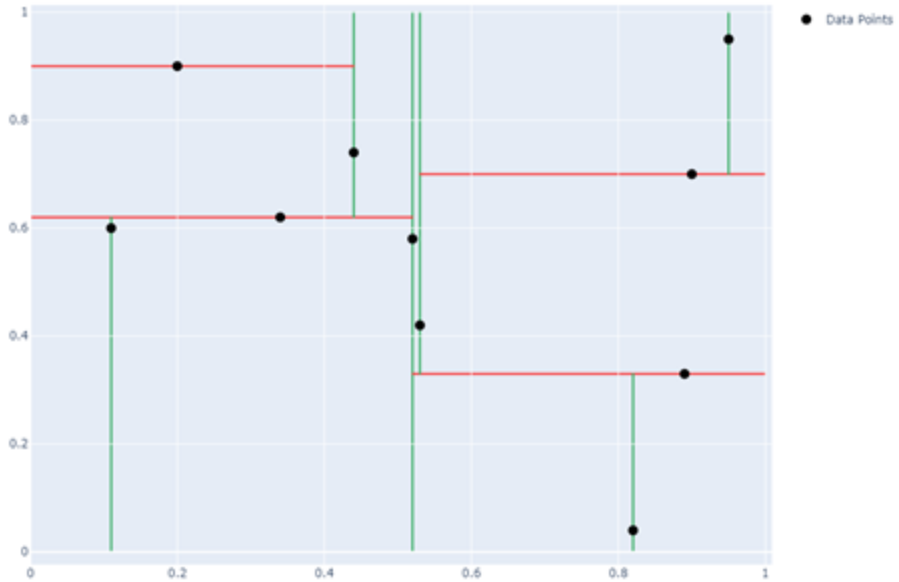

[0.52 0.58]
x<0.52
x>0.52
[0.34 0.62]
[0.89 0.33]
y<0.62
y>0.62
y<0.33
y>0.33
[0.11 0.6]
[0.44 0.74]
[0.82 0.04]
[0.53 0.42]
x<0.44
x>0.53
[0.2 0.9]
[0.9 0.7]
y>0.7
[0.95 0.95]

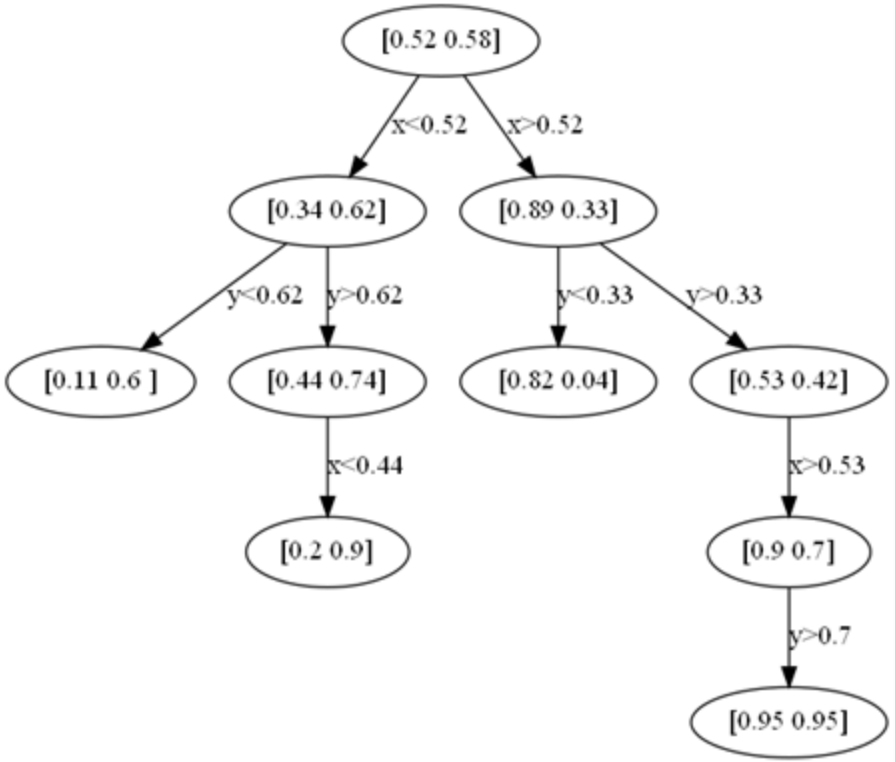
Deletion

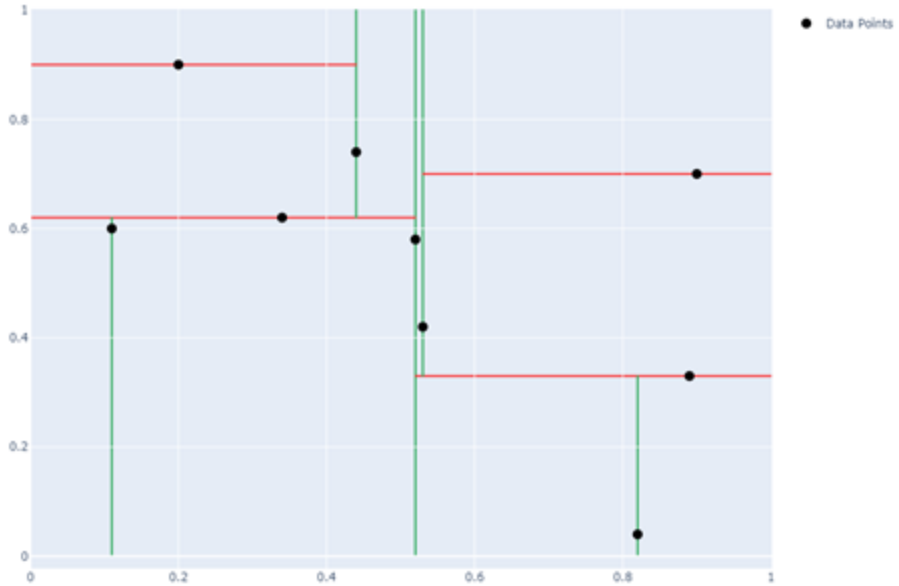
Deletion

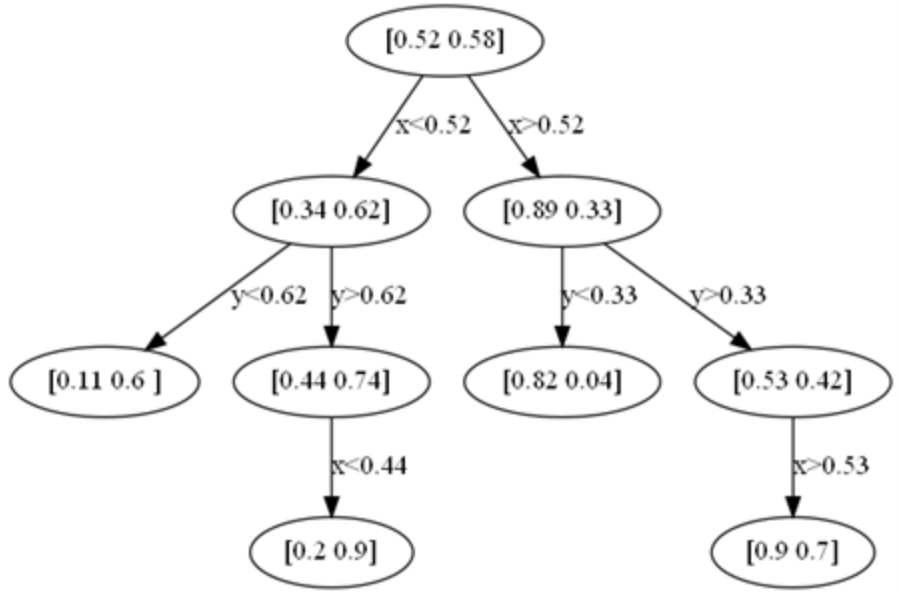
1. Start traversing the tree from the root
2. if current node == target node
   a. find “in-order successor” in right subtree
   b. replace current node with “in-order successor”
   c. recursively start deletion from “in-order successor”
3. else
   a. if current.left.contains(target): repeat 1. from current.left
   b. else: repeat 1. from current.right

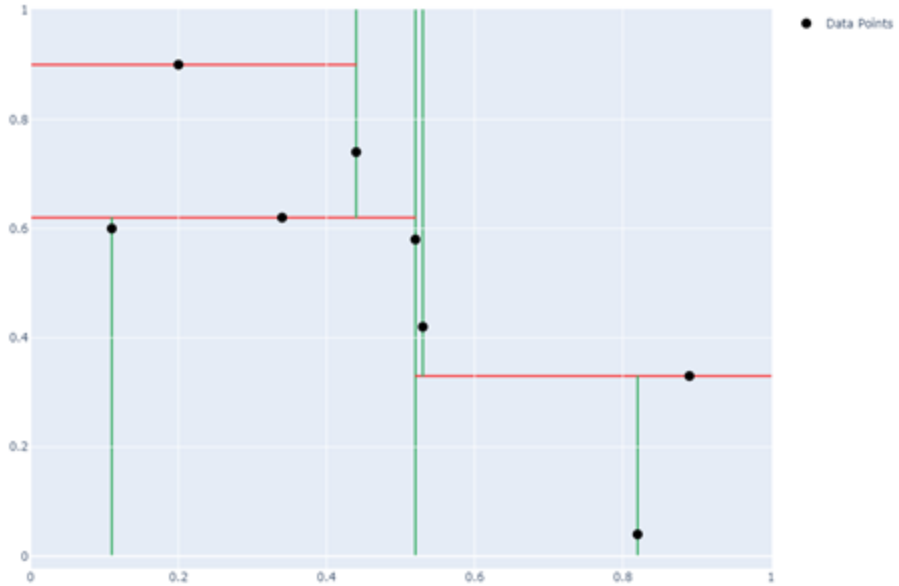
- Warning:
  - Tree may become unbalanced during insertions (NN-Search performance suffers)
  - Rebalancing steps possible

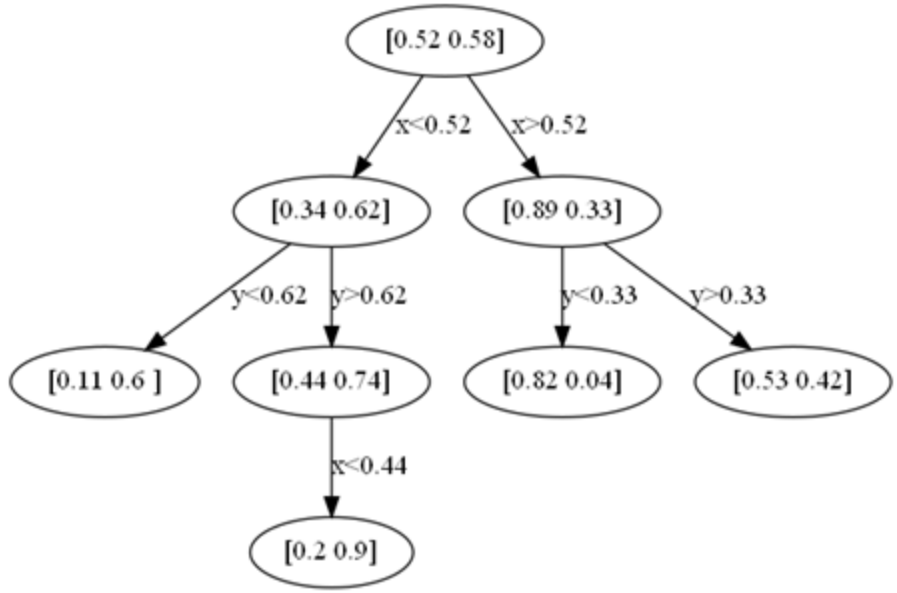

Data Points
0
0.2
0.4
0.6
0.8
1

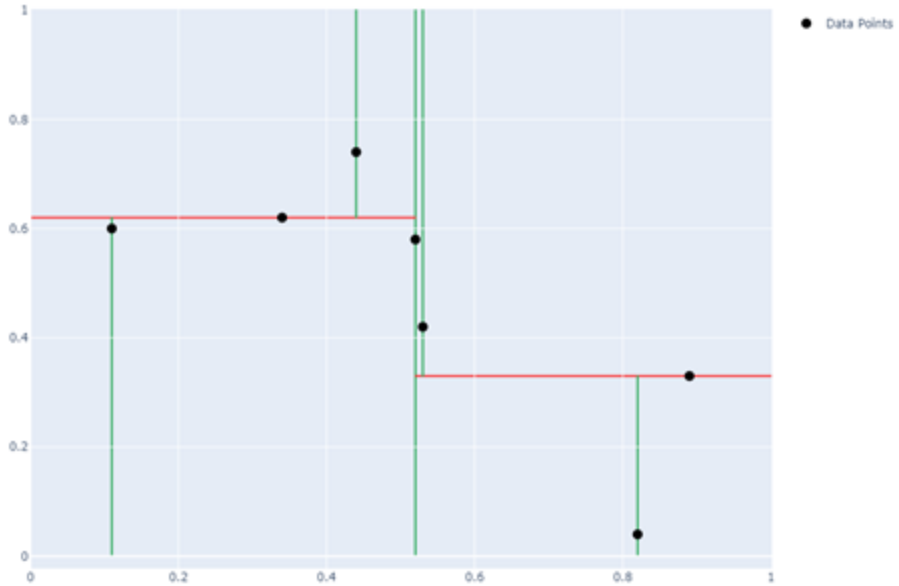

[0.52 0.58]
x<0.52
x>0.52
[0.34 0.62]
[0.89 0.33]
y<0.62
y>0.62
y<0.33
y>0.33
[0.11 0.6]
[0.44 0.74]
[0.82 0.04]
[0.53 0.42]
x<0.44
x>0.53
[0.2 0.9]
[0.9 0.7]
y>0.7
[0.95 0.95]

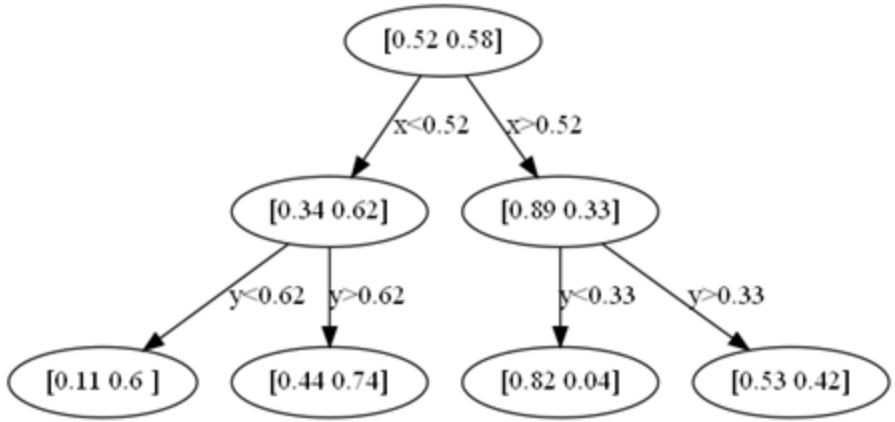

Data Points
0
0.2
0.4
0.6
0.8
1

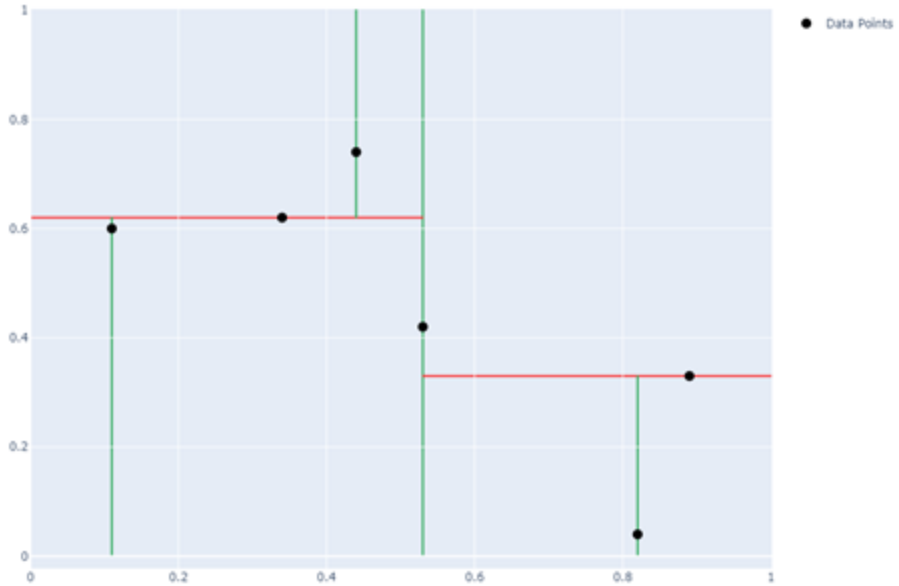

[0.52 0.58]
x<0.52
x>0.52
[0.34 0.62]
[0.89 0.33]
y<0.62
y>0.62
y<0.33
y>0.33
[0.11 0.6]
[0.44 0.74]
[0.82 0.04]
[0.53 0.42]
x<0.44
x>0.53
[0.2 0.9]
[0.9 0.7]

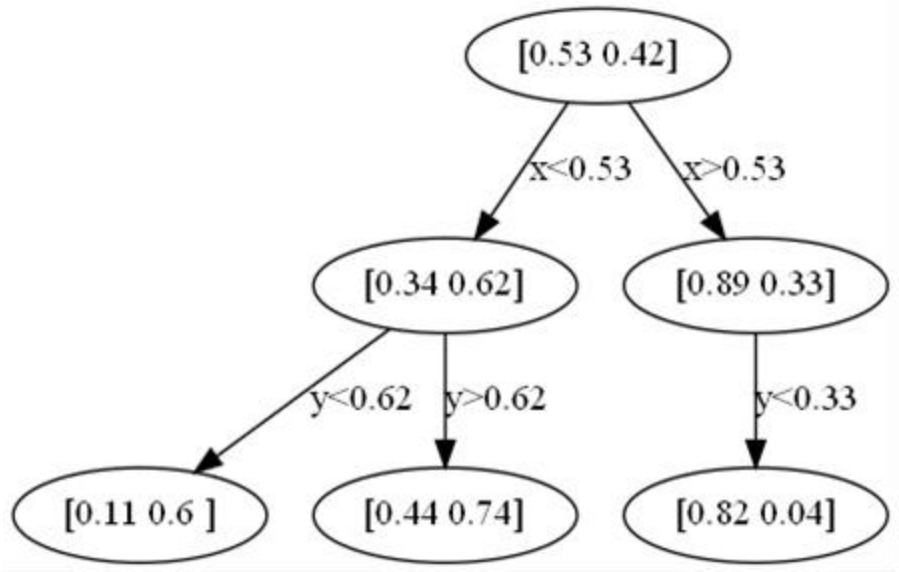

Data Points
1
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1


[0.52 0.58]
x<0.52
x>0.52
[0.34 0.62]
[0.89 0.33]
y<0.62
y>0.62
y<0.33
y>0.33
[0.11 0.6]
[0.44 0.74]
[0.82 0.04]
[0.53 0.42]
x<0.44
[0.2 0.9]


Data Points
1
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1


[0.52 0.58]
x<0.52
x>0.52
[0.34 0.62]
[0.89 0.33]
y<0.62
y>0.62
y<0.33
y>0.33
[0.11 0.6]
[0.44 0.74]
[0.82 0.04]
[0.53 0.42]

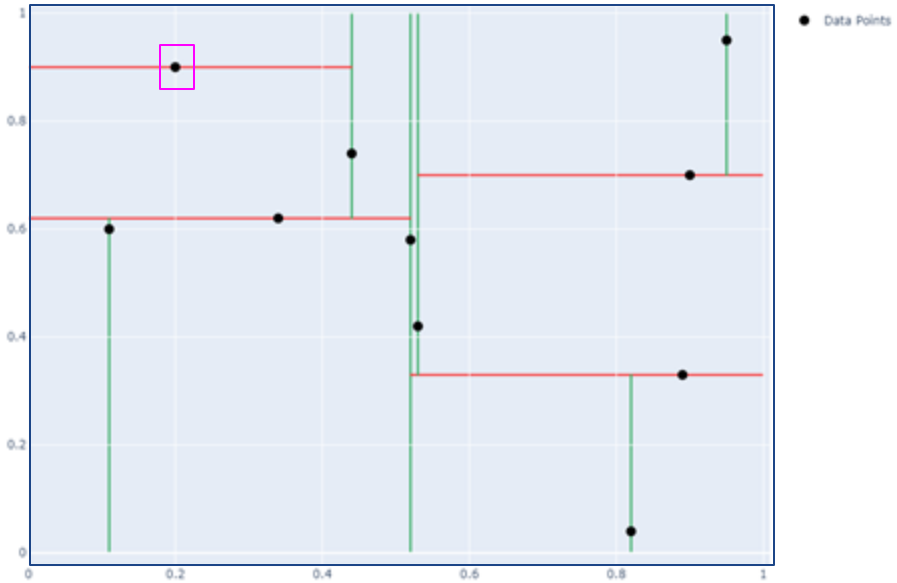

Data Points
1
0.8
0.6
0.4
0.2
0
0
0.2
0.4
0.6
0.8
1

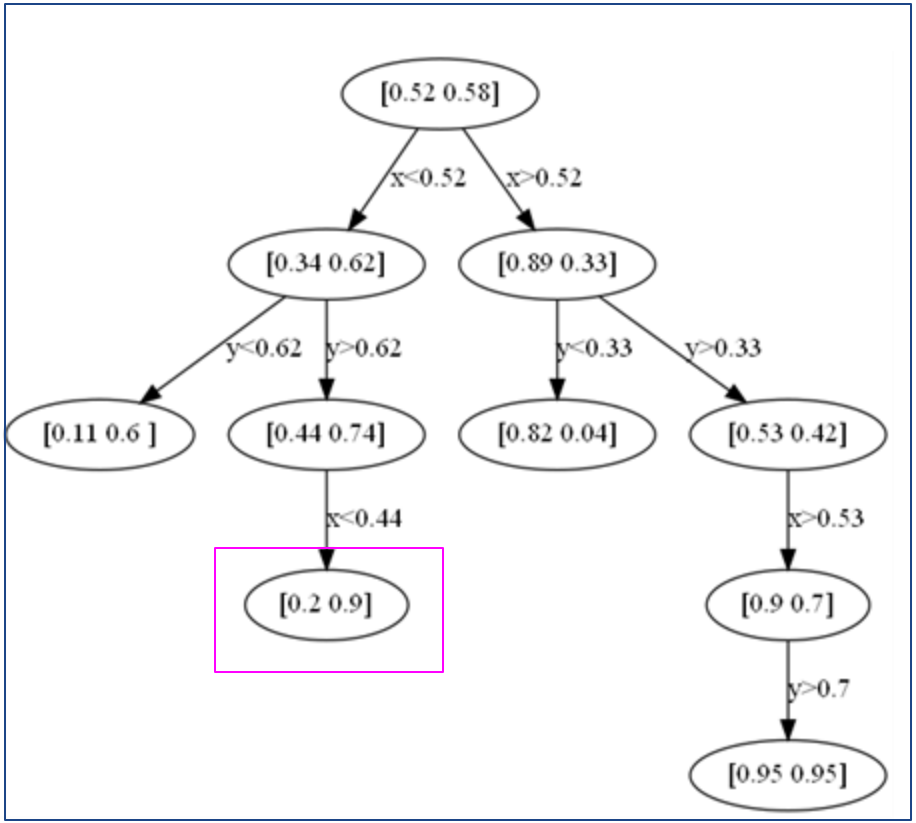

[0.53 0.42]
x<0.53
x>0.53
[0.34 0.62]
[0.89 0.33]
y<0.62
y>0.62
y<0.33
[0.11 0.6]
[0.44 0.74]
[0.82 0.04]

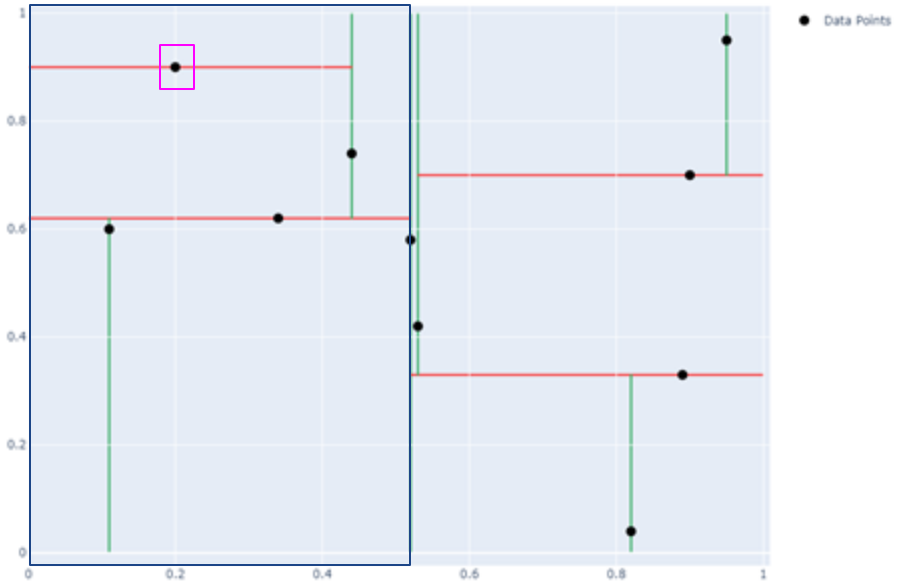
Query / Lookup

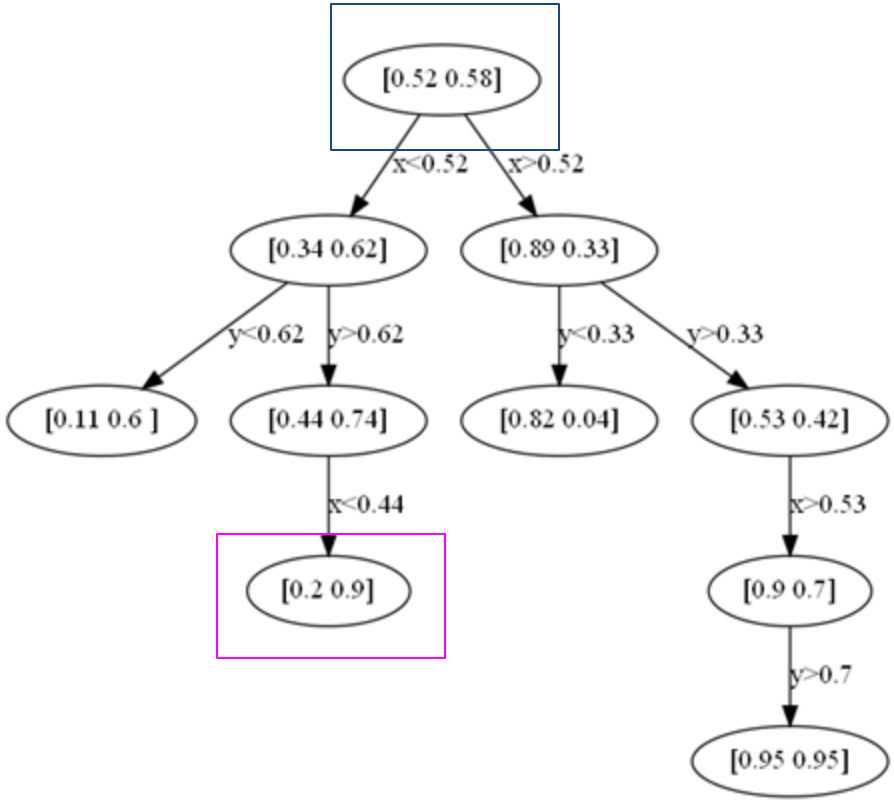
Query

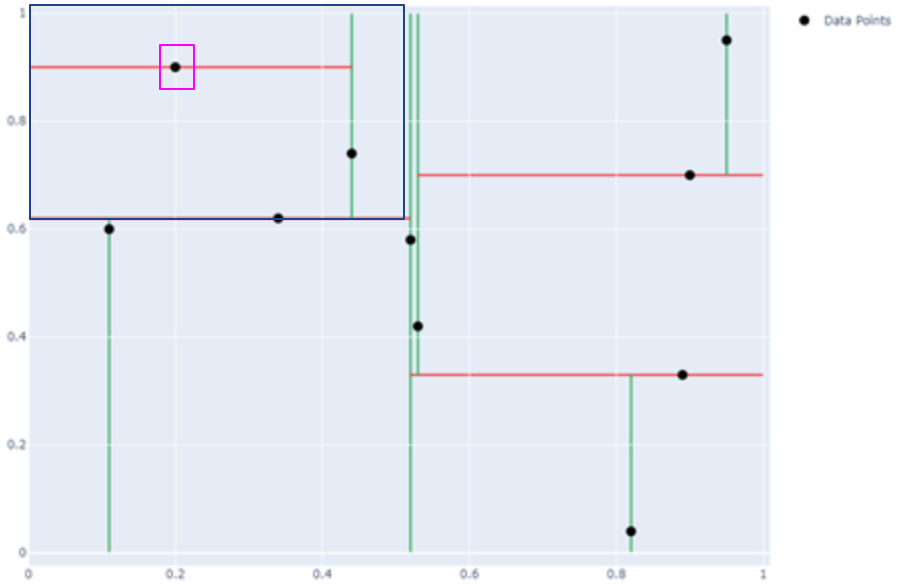
1. Start at root
2. Traverse tree down till search element is found or a not matching leaf is reached
3. At every node compare value to given condition
4. Traverse the subtree depending where the condition is true

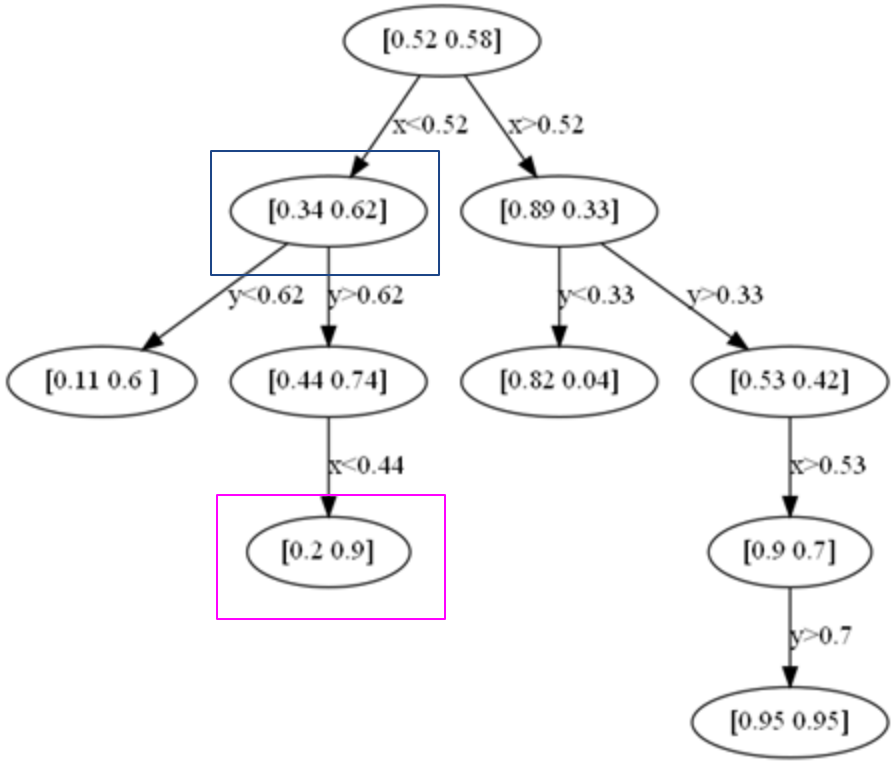

Data Points
0
0.2
0.4
0.6
0.8
1

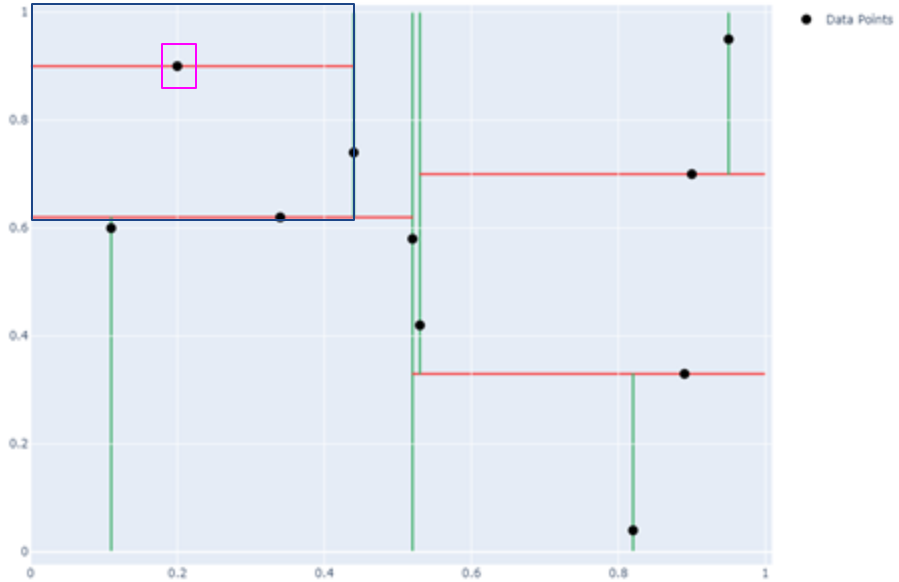

[0.52 0.58]
x<0.52
x>0.52
[0.34 0.62]
[0.89 0.33]
y<0.62
y>0.62
y<0.33
y>0.33
[0.11 0.6]
[0.44 0.74]
[0.82 0.04]
[0.53 0.42]
x<0.44
x>0.53
[0.2 0.9]
[0.9 0.7]
y>0.7
[0.95 0.95]

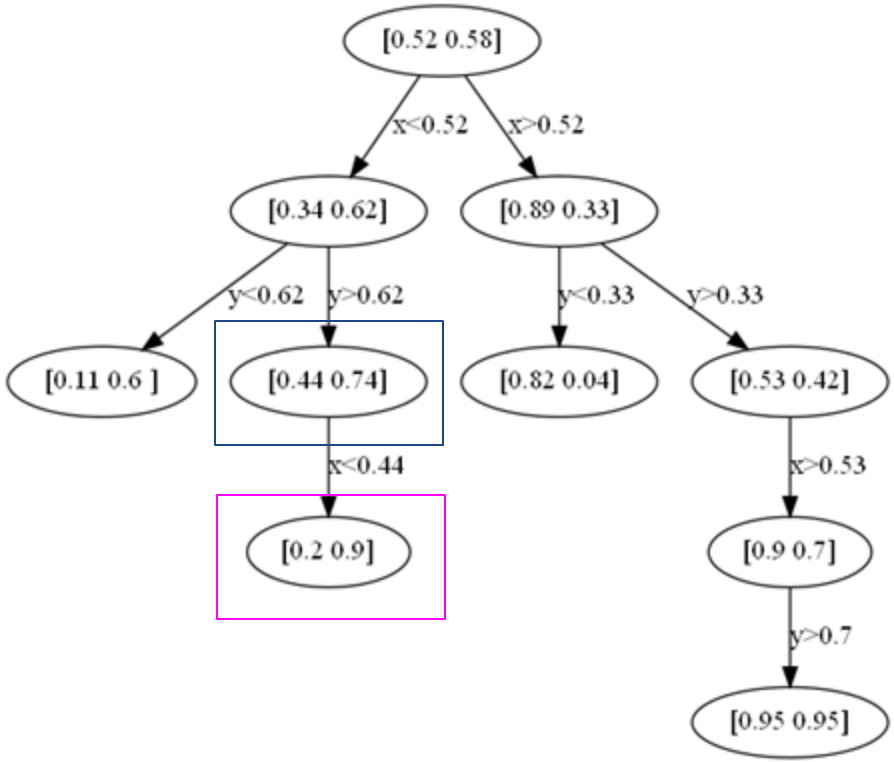

Data Points
0
0.2
0.4
0.6
0.8
1

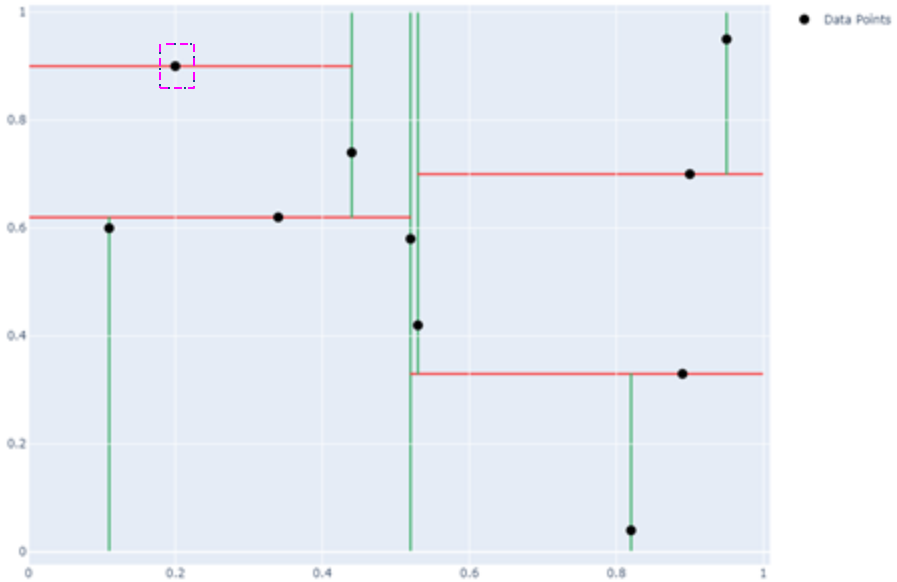

[0.52 0.58]
x<0.52
x>0.52
[0.34 0.62]
[0.89 0.33]
y<0.62
y>0.62
y<0.33
y>0.33
[0.11 0.6]
[0.44 0.74]
[0.82 0.04]
[0.53 0.42]
x<0.44
x>0.53
[0.2 0.9]
[0.9 0.7]
y>0.7
[0.95 0.95]

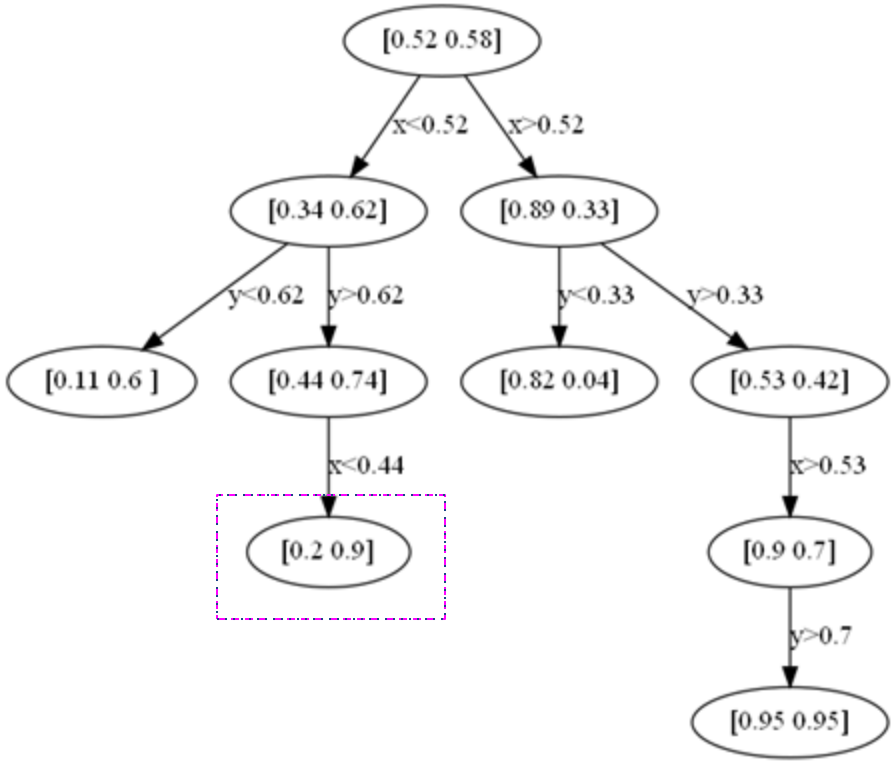

Data Points
0
0.2
0.4
0.6
0.8
1


[0.52 0.58]
x<0.52
x>0.52
[0.34 0.62]
[0.89 0.33]
y<0.62
y>0.62
y<0.33
y>0.33
[0.11 0.6]
[0.44 0.74]
[0.82 0.04]
[0.53 0.42]
x<0.44
x>0.53
[0.2 0.9]
[0.9 0.7]
y>0.7
[0.95 0.95]


Data Points
0
0.2
0.4
0.6
0.8
1


[0.52 0.58]
x<0.52
x>0.52
[0.34 0.62]
[0.89 0.33]
y<0.62
y>0.62
y<0.33
y>0.33
[0.11 0.6]
[0.44 0.74]
[0.82 0.04]
[0.53 0.42]
x<0.44
x>0.53
[0.2 0.9]
[0.9 0.7]
y>0.7
[0.95 0.95]


Data Points
0
0.2
0.4
0.6
0.8
1


[0.52 0.58]
x<0.52
x>0.52
[0.34 0.62]
[0.89 0.33]
y<0.62
y>0.62
y<0.33
y>0.33
[0.11 0.6]
[0.44 0.74]
[0.82 0.04]
[0.53 0.42]
x<0.44
x>0.53
[0.2 0.9]
[0.9 0.7]
y>0.7
[0.95 0.95]

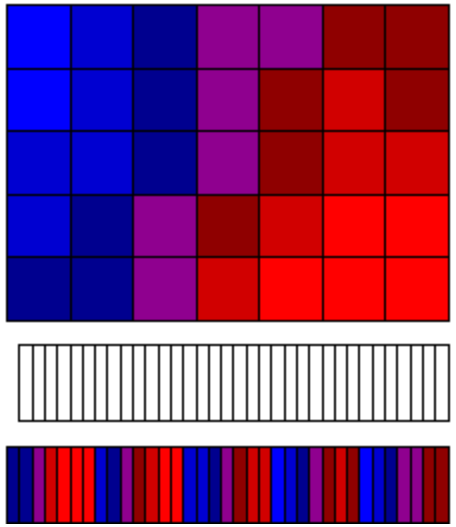
Nearest Neighbor Search

- start at root
- move recursively down to leave
- hypersphere around search point (radius = nearest distance)
  - crossed hyperplane to other branch -> maybe nearer points on the other side
  - if so search point moves down on the other side of the plane
  - if not other side is eliminated, search point moves up
- ends at root

Variations

Variations and Related Versions

- Adaptive kd-Tree
  - successive levels split along different dimensions
- Implicit kd-Tree
  - defined implicitly above a rectilinear grid
- Min/Max kd-Tree
  - min/max values of each inner node are equal to min/max values of a child note -> save memory
- Relaxed kd-Tree
  - leaves of node do not have to have the same split axis
  - this relaxation allows insertions at arbitrary dimensions without reordering



Implicit kd-tree example

https://upload.wikimedia.org/wikipedia/commons/b/b7/Implicitmaxkdtree.gif
Last accessed 25.5.20