PES UNIVERSITY

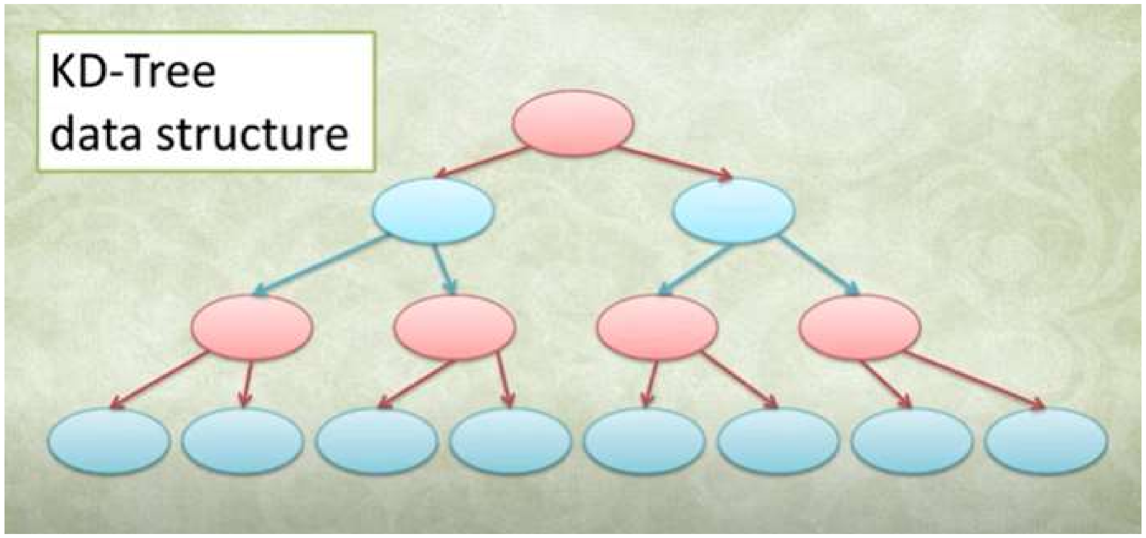
PES UNIVERSITY RR CAMPUS
ADVANCED ALGORITHMS
UE20CS311-PROJECT

NAME : S M SUTHARSAN RAJ
SRN : PES1UG20CS362
SECTION : F
TOPIC : KD TREE FOR KNN Search

PROBLEM STATEMENT

To Find Nearest Neighbour of a point from a given set of points with the help of a K-D Tree.

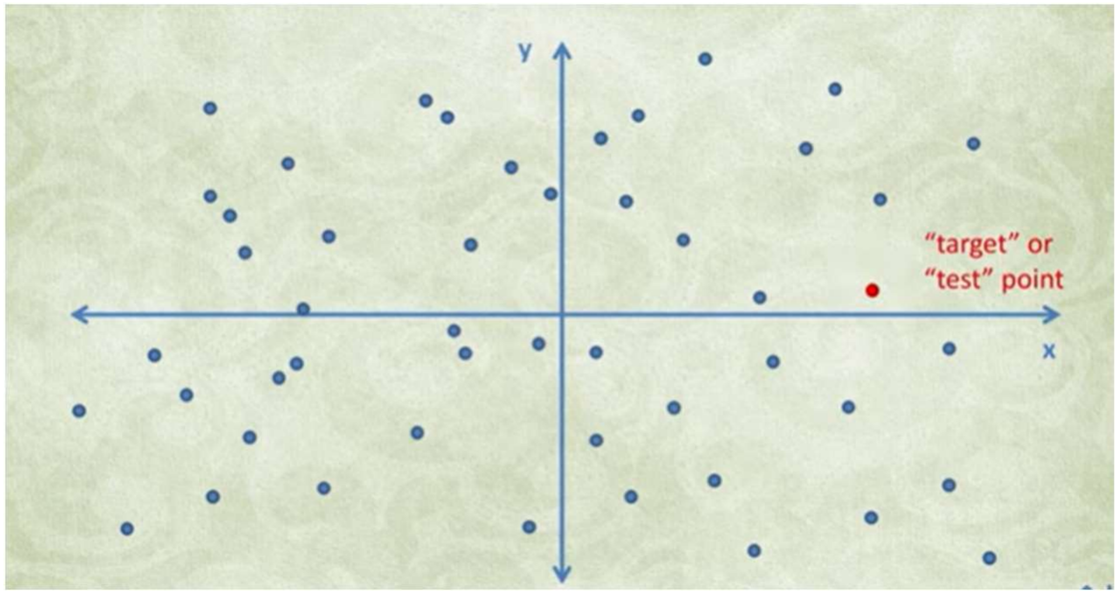
We also implement KNN for a given set of query points to show the multiple results.



MOTIVATION : PROBLEM OF NAÏVE APPROACH

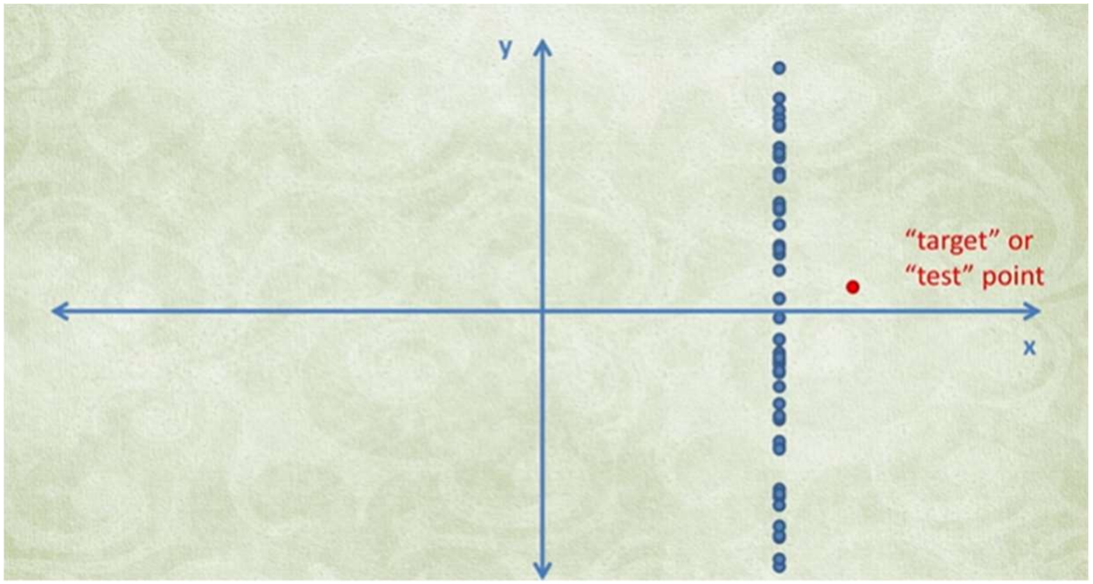
Though the Brute Force Approach also finds the KNN for a given query point,
By using an array or a naïve BST, we still have complexity issue.
Brute Force/Naïve kNN algorithm calculates the squared distances from each query feature vector to each reference feature vector in the training data set. Then, for each query feature vector it selects k objects from the training set that are closest to that query feature vector.

TIME COMPLEXITY : O(k*n)



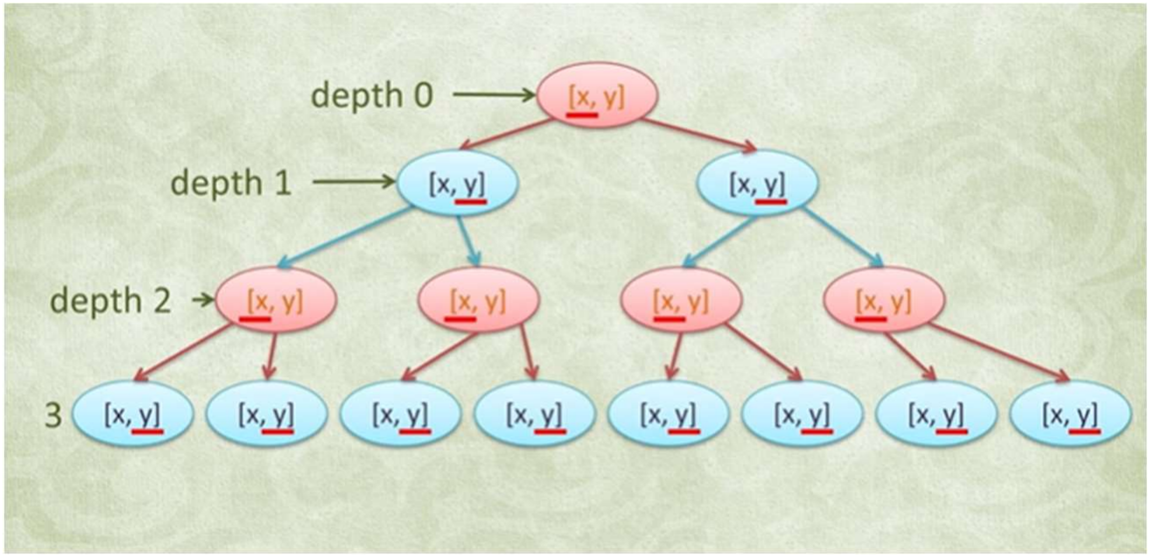
MOTIVATION : PROBLEM OF NAÏVE APPROACH

Naïve Approach 2 :
Points with same x values gets skewed up when we use naïve KNN approach by using an array or simple BST by taking only x values into consideration.



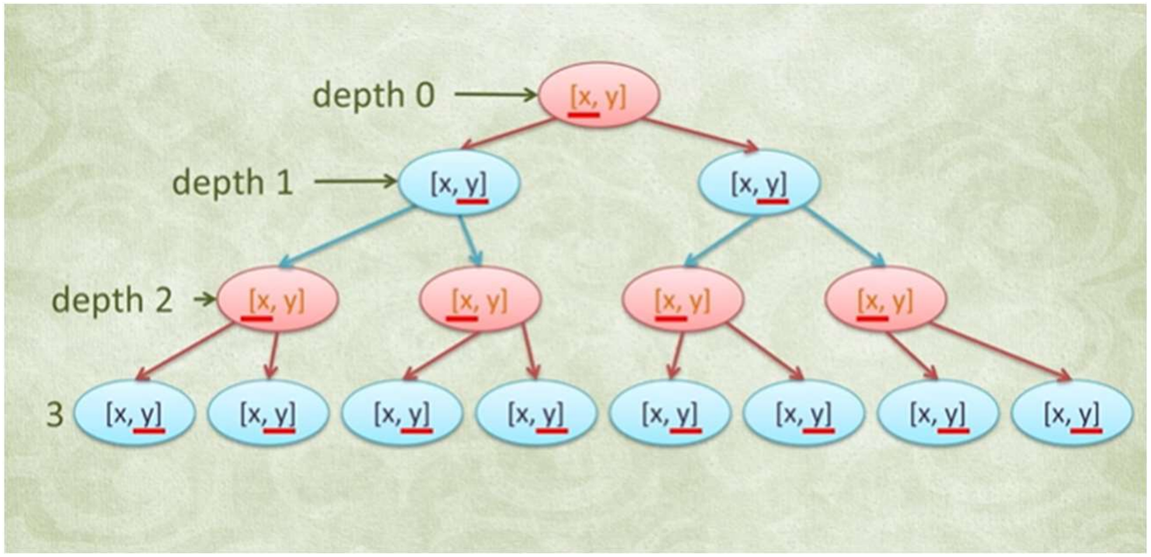
RELATED WORK

•ALGLIB has C# and C++ implementations of *k*-d tree based nearest neighbor and approximate nearest neighbor algorithms
•CGAL the Computational Algorithms Library, has an implementations of *k*-d tree based nearest neighbor, approximate nearest neighbor as well as range search algorithms.
•SciPy, a Python library for scientific computing, contains implementations of *k*-d tree based nearest neighbor lookup algorithms.
•scikit-learn, a Python library for machine learning, contains implementations of *k*-d trees to back nearest neighbor and radius neighbors searches.
•*k*-d trees are a useful data structure for several applications, such as searches involving a multidimensional search key (e.g. range searches and nearest neighbor searches) and creating point clouds. *k*-d trees are a special case of binary space partitioning trees.



APPROACH

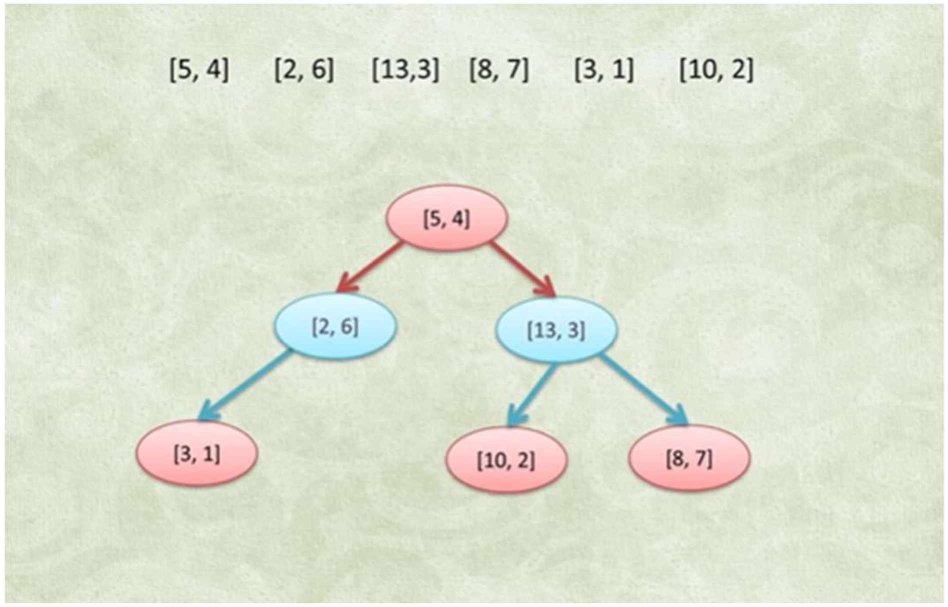
Let a point set P and a corresponding k-d tree built. A neighbourhood query for any point p ∈ Rd is then performed by traversing the tree to the leaf representing the box which contains the query point. From there, the query goes back to the root, investigating subtrees along the path where the splitting hyperplane is closer to p than the currently found nearest neighbours. When reaching a node, all elements in it are investigated as to whether they are closer to p than the current closest points found. The algorithm is fast, as it can be expected that several subtrees do not have to be investigated.

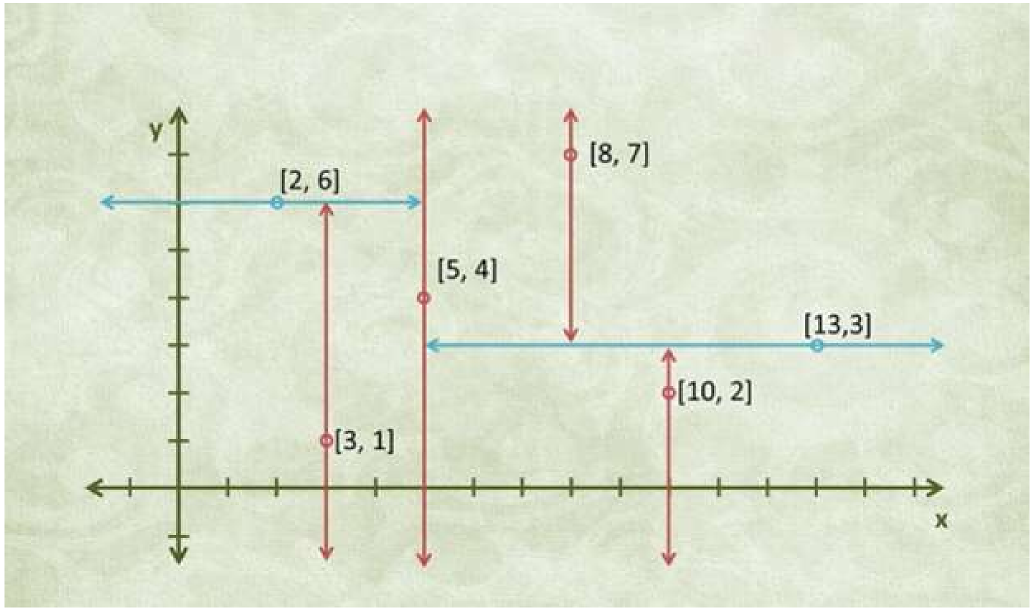


APPROACH

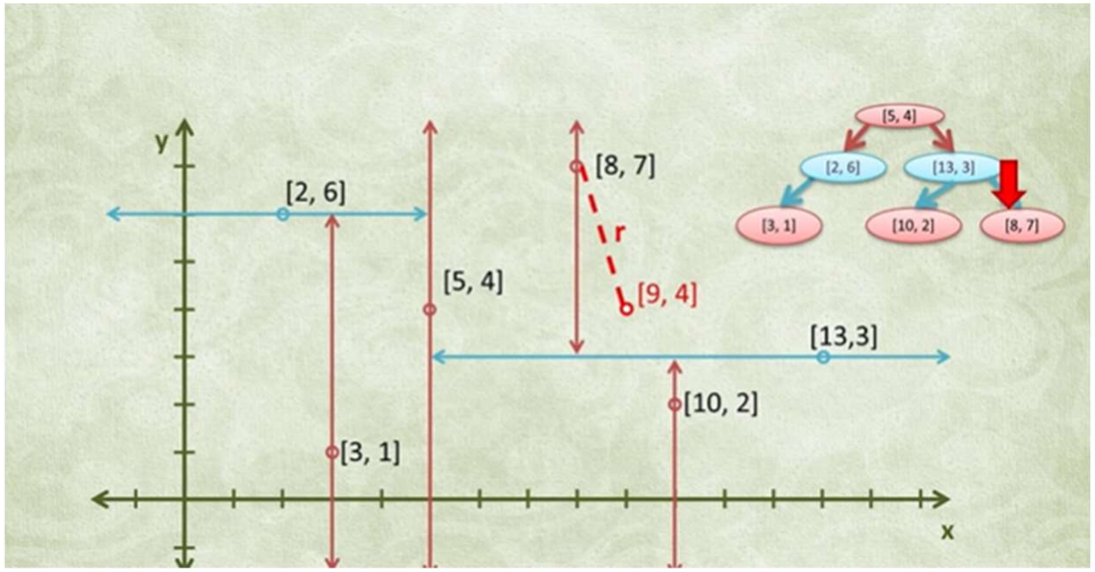
Building of K-D tree
At each depth x and y coordinates are alternatingly compared.

Each point's coordinate split sections by its respective x and y-axis





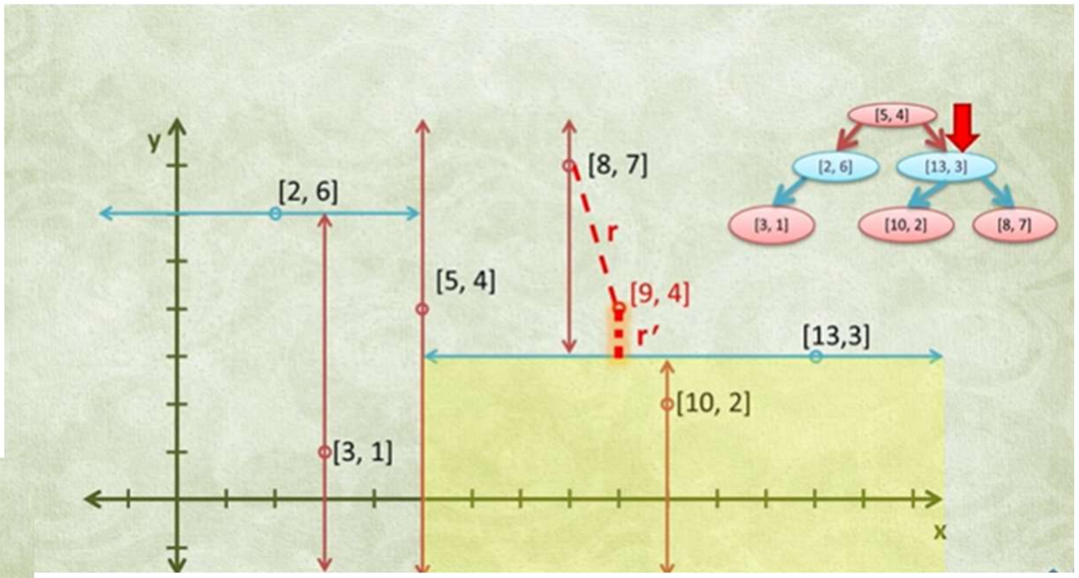
APPROACH

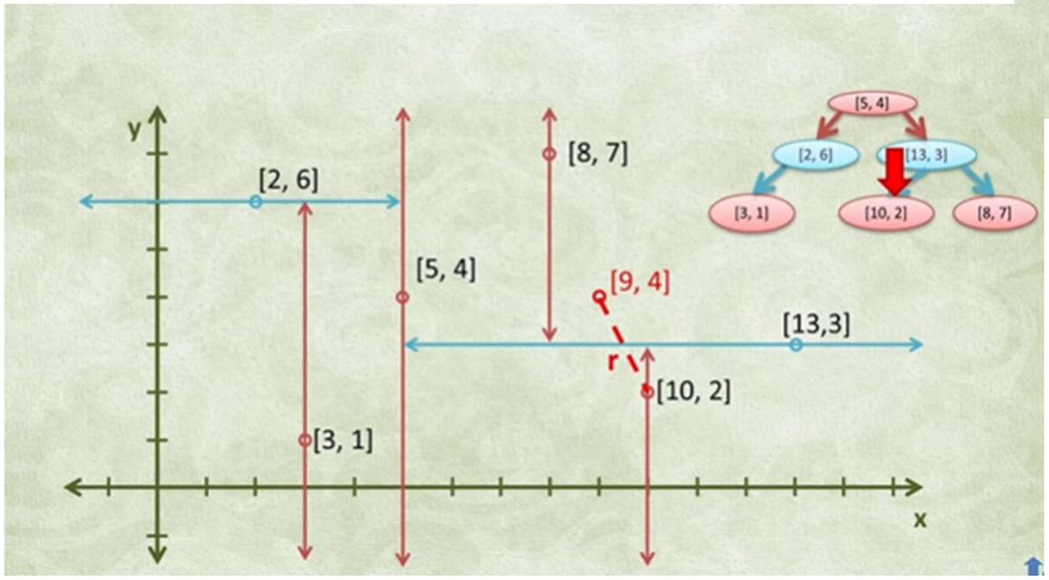


- Just by normal traversal we find our KNN for [9,4] to be [8,7] but still is not the best solution.
- There is still possibility of a much closer point.
- Think r to be the shortest distance.
- The distance r can be at any angle. We may not know.
- But this is not a problem!!!

APPROACH - CONTRIBUTION

The Solution to this :-
If we have a perpendicular line directly down to the section **not** visiting while recursive call is shorter then there is higher possibility that section may have the closer point, that is the KNN of our query point [9,4]



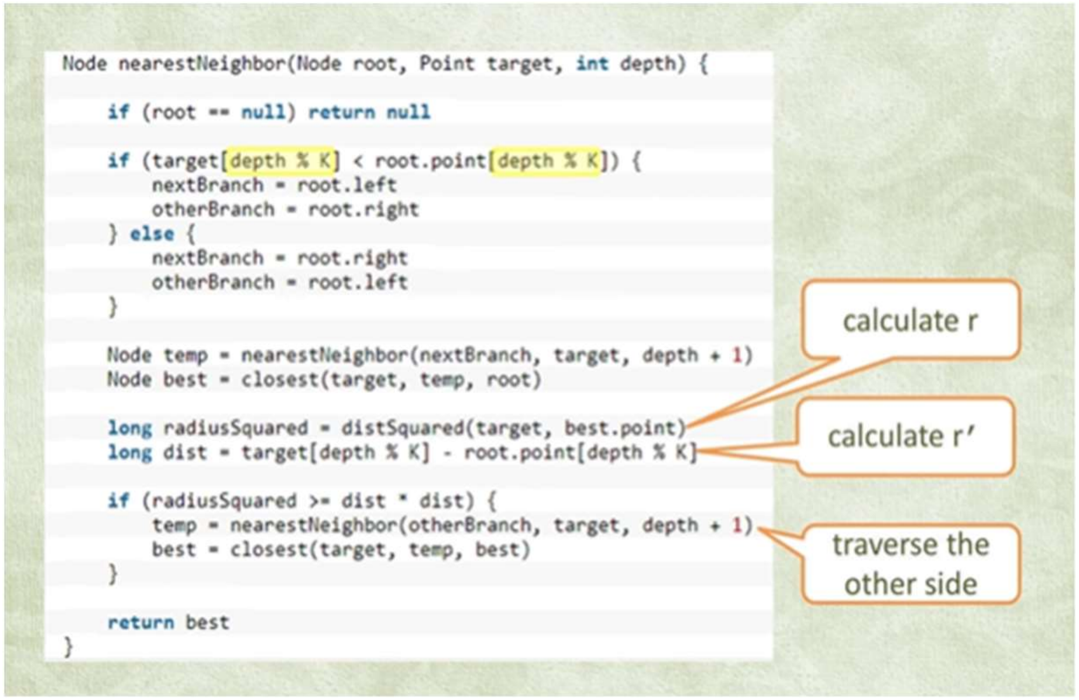


Hence, when we recurse back, we get the sectional distance r' and if r' < r, then r' is the best distance. Hence KNN is [10,2] !!!

Remember :: We don't visit all the nodes of tree.
If height of tree is h, then we atmost visit 2h nodes, which means, it is logarithmic

ALGORITHM AND COMPLEXITY

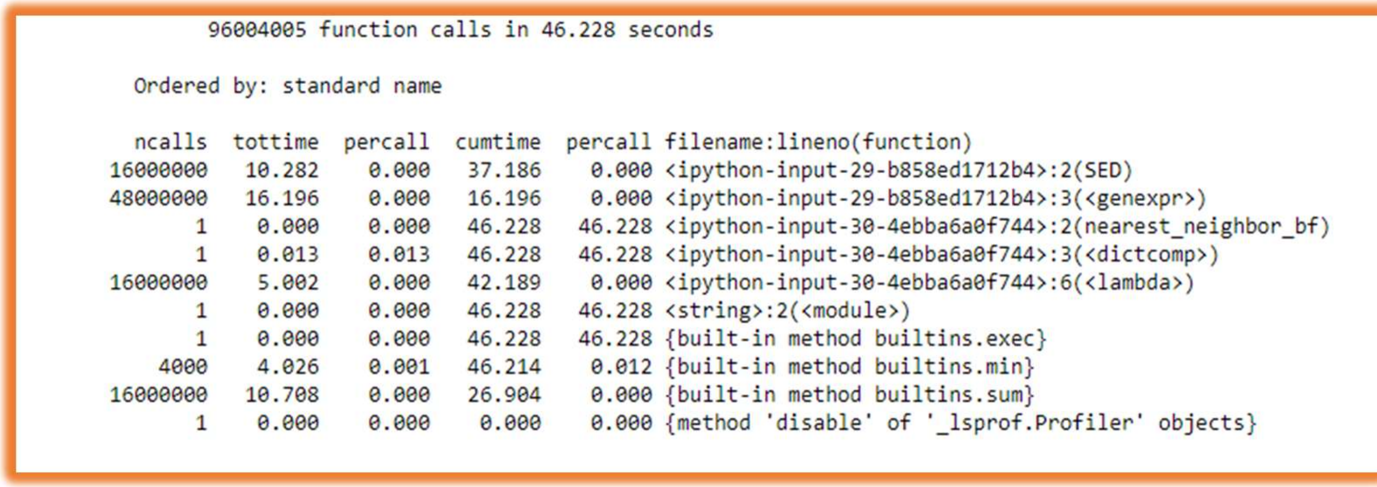
As for the prediction phase, the k-d tree structure naturally supports "k nearest point neighbors query" operation, which is exactly what we need for kNN. The simple approach is to just query k times, removing the point found each time — since query takes O(log(n)), it is O(k * log(n)) in total. But since the k-d tree already cuts space during construction, after a single query we approximately know where to look — we can just search the "surroundings" around that point. Therefore, practical implementations of k-d tree support querying for whole k neighbors at one time and with complexity O(sqrt(n) + k), which is much better for larger dimensionalities, which are very common in machine learning.



```
Node nearestNeighbor(Node root, Point target, int depth) {

    if (root == null) return null

    if (target[depth % K] < root.point[depth % K]) {
        nextBranch = root.left
        otherBranch = root.right
    } else {
        nextBranch = root.right
        otherBranch = root.left
    }

    Node temp = nearestNeighbor(nextBranch, target, depth + 1)
    Node best = closest(target, temp, root)

    long radiusSquared = distSquared(target, best.point)
    long dist = target[depth % K] - root.point[depth % K]

    if (radiusSquared >= dist * dist) {
        temp = nearestNeighbor(otherBranch, target, depth + 1)
        best = closest(target, temp, best)
    }

    return best
}
```

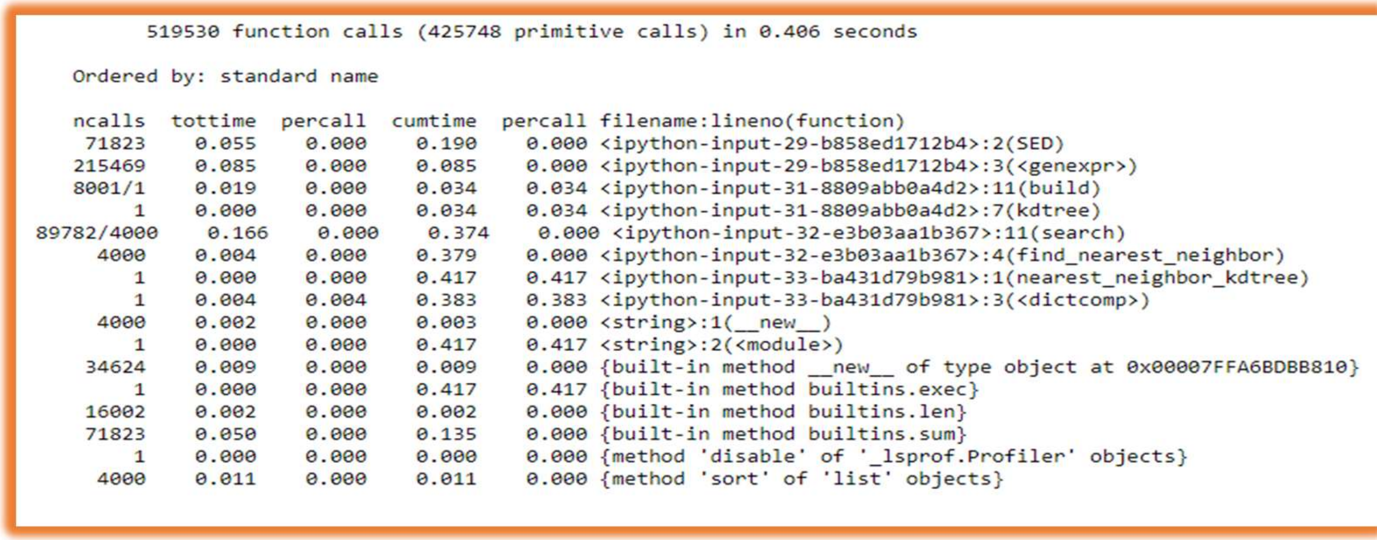
ANALYSING RESULTS

BRUTE FORCE METHOD/NAÏVE KNN for 4000 random points and selected random query points



```
         96004005 function calls in 46.228 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
 16000000   10.282    0.000   37.186    0.000 <ipython-input-29-b858ed1712b4>:2(SED)
 48000000   16.196    0.000   16.196    0.000 <ipython-input-29-b858ed1712b4>:3(<genexpr>)
        1    0.000    0.000   46.228   46.228 <ipython-input-30-4ebba6a0f744>:2(nearest_neighbor_bf)
        1    0.013    0.013   46.228   46.228 <ipython-input-30-4ebba6a0f744>:3(<dictcomp>)
 16000000    5.002    0.000   42.189    0.000 <ipython-input-30-4ebba6a0f744>:6(<lambda>)
        1    0.000    0.000   46.228   46.228 <string>:2(<module>)
        1    0.000    0.000   46.228   46.228 {built-in method builtins.exec}
     4000    4.026    0.001   46.214    0.012 {built-in method builtins.min}
 16000000   10.708    0.000   26.904    0.000 {built-in method builtins.sum}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
```

KD TREE METHOD KNN for 4000 random points and selected random query points

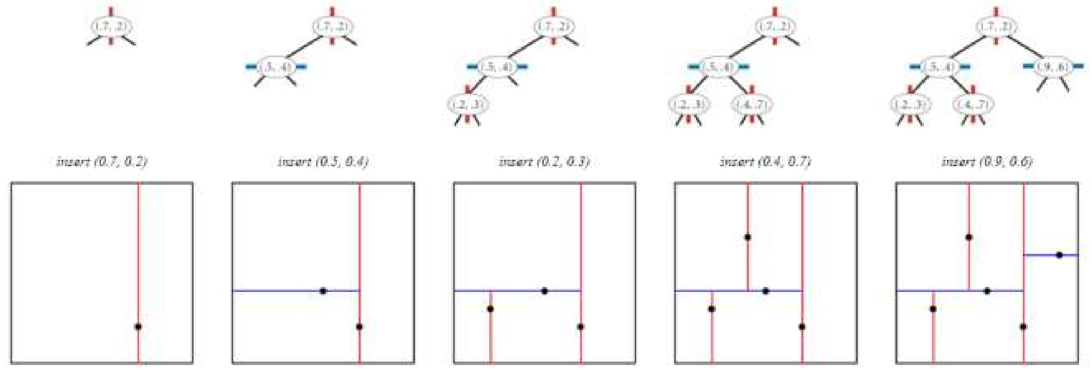


```
         519530 function calls (425748 primitive calls) in 0.406 seconds

   Ordered by: standard name

   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
    71823    0.055    0.000    0.190    0.000 <ipython-input-29-b858ed1712b4>:2(SED)
   215469    0.085    0.000    0.085    0.000 <ipython-input-29-b858ed1712b4>:3(<genexpr>)
   8001/1    0.019    0.000    0.034    0.034 <ipython-input-31-8809abb0a4d2>:11(build)
        1    0.000    0.000    0.034    0.034 <ipython-input-31-8809abb0a4d2>:7(kdtree)
89782/4000    0.166    0.000    0.374    0.000 <ipython-input-32-e3b03aa1b367>:11(search)
     4000    0.004    0.000    0.379    0.000 <ipython-input-32-e3b03aa1b367>:4(find_nearest_neighbor)
        1    0.000    0.000    0.417    0.417 <ipython-input-33-ba431d79b981>:1(nearest_neighbor_kdtree)
        1    0.004    0.004    0.383    0.383 <ipython-input-33-ba431d79b981>:3(<dictcomp>)
     4000    0.002    0.000    0.003    0.000 <string>:1(__new__)
        1    0.000    0.000    0.417    0.417 <string>:2(<module>)
    34624    0.009    0.000    0.009    0.000 {built-in method __new__ of type object at 0x00007FFA6BDBB810}
        1    0.000    0.000    0.417    0.417 {built-in method builtins.exec}
    16002    0.002    0.000    0.002    0.000 {built-in method builtins.len}
    71823    0.050    0.000    0.135    0.000 {built-in method builtins.sum}
        1    0.000    0.000    0.000    0.000 {method 'disable' of '_lsprof.Profiler' objects}
     4000    0.011    0.000    0.011    0.000 {method 'sort' of 'list' objects}
```

SCOPE AND DESIGN

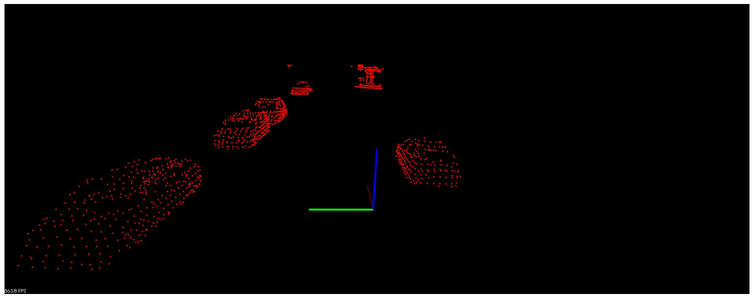
1. KNN can be used for Recommendation Systems. Although in the real world, more sophisticated algorithms are used for the recommendation system. KNN is not suitable for high dimensional data, but KNN is an excellent baseline approach for the systems. Many companies make a personalized recommendation for its consumers, such as Netflix, Amazon, YouTube, and many more.
2. KNN can search for semantically similar documents. Each document is considered as a vector. If documents are close to each other, that means the documents contain identical topics.
3. KNN can be effectively used in detecting outliers. One such example is Credit Card fraud detection.

K Dimensional tree (or k-d tree) is a tree data structure that is used **to represent points in a k-dimensional space**. It is used for various applications like nearest point (in k-dimensional space), efficient storage of spatial data, range search etc.

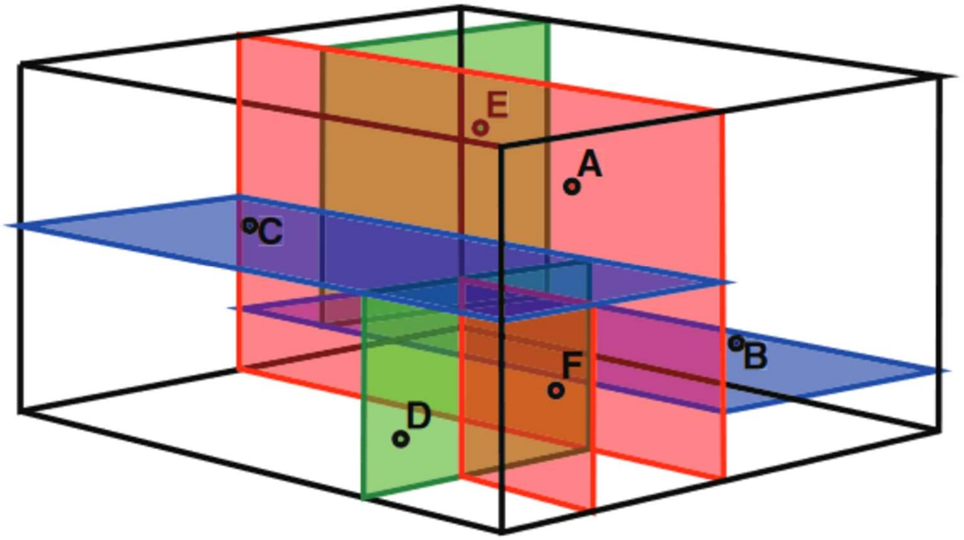


Taken from the Coursera Course Algorithms-I by Princeton University

DEMO, CONCLUSION AND FUTURE WORK



Return of the nearest neighbour and comparison with the naïve algorithm by showing the wall clock time too. Successful insertion and search are also displayed.
We also see a drastic improvements in performance of KD tree when compared to Naïve method.
Hence, KD is optimal for KNN by this approach.



FUTURE WORKS :-

The use of KD tree in

1. Efficient storage of spatial data, and
2. Database queries involving a multidimensional search key
3. Working with 3D or Higher Dimension search.

Will be worked upon.

- S M SUTHARSAN RAJ
- PES1UG20CS362