# APPLICATION OF APPROXIMATE NEAREST NEIGHBOR SEARCH ALGORITHMS: KD TREE

**Enrollment No. -** 101227

**Name of Student -** Deepti Garg

**Name of Supervisors –** Mr. Suman Saha

**MAY 2014**

Submitted in partial fulfillment of the Degree of

Bachelor of Technology

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY, WAKNAGHAT

# TABLE OF CONTENTS

# <u>CERTIFICATE</u>

This is to certify that the work titled " **APPROXIMATE NEAREST NEIGHBOR SEARCH ALGORITHM: KD TREE**" submitted by " **Deepti Garg** " in partial fulfillment for the award of degree of Bachelor Technology, of Jaypee University Of Information Technology, Waknaghat has been carried out under my supervision. This work has not been submitted partially or wholly to any other University or institute for the award of this or any other degree or diploma.

Signature of Supervisor:… Suman Saha

Name of Supervisor: Suman Saha

Designation: Assistant Professor (Grade II)

Date: 15th May 2014

# ACKNOWLEDGEMENT

Every project big or small is successful largely due to the effort of a number of wonderful people who have always given their valuable advice or lent a helping hand. I sincerely appreciate the inspiration, support and guidance of all those people who have been instrumental in making this project a success.

I, **Deepti Garg**, the student of **Jaypee University Of Information Technology**, feel deeply honored in expressing thanks to **Mr. Suman Saha** for making the resources available at right times and providing right insights leading to the successful completion of my project. I feel motivated and encouraged every time. For his coherent guidance throughout the semester, I feel fortunate to be taught by him, who gave us his unwavering support.

Also, I would like to thank my colleagues in developing the project and people who have willingly helped me out with their abilities.

DEEPTI GARG

15th May 2014

# SUMMARY

As we all know, that the amount of data generated by the businesses, government organizations etc. is increasing day by day. As the data is increasing, the need for storage of the data and to label the data in such a manner so that we can find it efficiently and effectively whenever it is required, is also increasing in a large manner and this is also known as Data Proliferation.

The amount of data is in millions or trillions or even more, so we can imagine that it is not so easy to label and access that data. It will take a lot of time to find out the exact position of the data so, in order to access the data in lesser amount of time we use approximate nearest neighbor algorithms instead of the exact neighbor algorithm.

This report contains the study, implementation and application of approximate nearest neighbor algorithm and will show that by considering the range or approximation in our results how the time and search complexity of finding out the data is reduced. Because of using the approximation in our results the time complexity is very much improved.

So, this report will make you understand the need of approximate nearest neighbor algorithms and graphically represents the results, the accuracy and time complexity and will also show you the comparisons between two different algorithms.

Name: Deepti Garg
Date: 15th May 2014

Name: Mr. Suman Saha
Date: 15th May 2014

# LIST OF FIGURES

# Chapter 1: INTRODUCTION

## 1.1 Introduction

Over the last decade, a very large amount of information has become available. From collections of camera pictures, to biological information, and to network traffic data, modern and cheap storage technologies have made it possible to gather such heavy datasets. But can we effectively use all this data or information? The ever increasing sizes of the datasets make it necessary to design new algorithms which must be capable of shifting through this data with extreme efficiency.

**Figure 1.1**

The challenges include capture, storage, search, sharing, transfer, analysis, and visualization. The trend of large amount of data sets is due to the additional information derivable from analysis of each and every large set of related information. As of 2012, limits on the ever increasing size of data sets that are feasible to process in efficient amount of time were on the order of terabytes of information. Scientists are regularly trying to find out the limitations due to large data sets in many areas, including meteorology, complex physics simulations, genomics and biological and environmental research. The limitations also affect Internet search, finance and business informatics. Data sets increase in size because they are increasingly being gathered or stored by universal information-sensing mobile devices, aerial sensory technologies (remote sensing), cameras, microphones, software records, radio-frequency identification readers, and wireless sensor networks. The world's technological per-capita capacity to store information has roughly doubled every 50 months since the 1970s; as of 2010, every day 2.6 terabytes ($2.6 \times 10^{18}$) of data were created. The challenge for big enterprises is determining who should take up these big data initiatives that can take stand for the entire organization.

Big data is difficult to work with using most relational database management systems and desktop statistics and visualization packages, requiring instead "big-big parallel software running on hundreds, or even thousands of servers".

A fundamental computational primitive for dealing with a very large dataset is the Nearest Neighbor (NN) problem. The nearest-neighbor (NN) problem occurs under different names, like the best match or the post office problem. The problem is of great importance to several areas of computer science, including pattern recognition, searching in multimedia data, compression techniques, computational statistics, and data mining.

In the NN problem, the goal is to process a set of data points, so that later, given a query point, one can find efficiently the data point most similar to the given query point. To represent the points and the similarity measures, one often uses geometric notions. For example, the image below may be modeled by a high-dimensional data point, with one coordinate per pixel, whereas the similarity measure may be the standard Euclidean distance between the resulting data points.

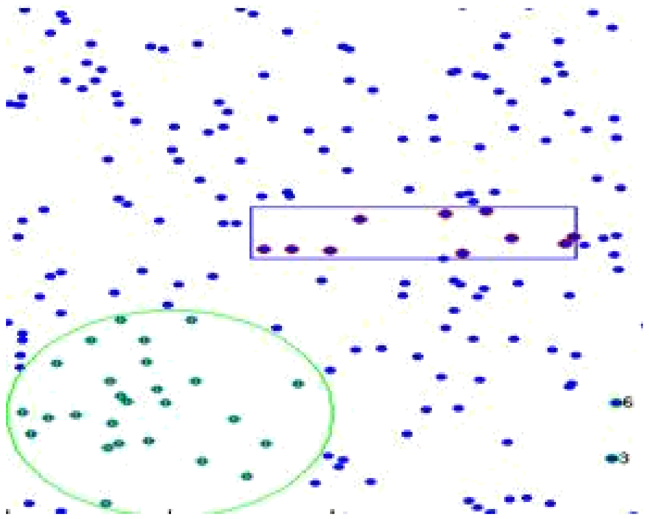


**Figure 1.2**

For many of today's applications, large amount of information is available. This makes nearest-neighbor approaches particularly important, but on another hand it increases the concern regarding the computational complexity of NN search. Thus, it is important to design algorithms for nearest-neighbor search, as well as for the related classification, regression, and retrieval tasks, which remain efficient even as the number of data points or the dimensionality of the data points increases randomly. This is a research area on the boundary of a number of disciplines: computational geometry, algorithmic theory, and the application fields such as machine learning.

## 1.2 Genesis Of Problem

Data proliferation is the main origin of Approximate Nearest Neighbor Algorithm. As the amount of data is increasing day by day, there is a need to properly collect and store this data and label the data in such a manner so that we can later on find out it effectively and efficiently.

**Figure 1.3**

We need such algorithms which help us to find out the required data from such a large amount of data and within the least possible amount of time. That's why there is a need of Approximate nearest neighbor algorithms. With very less compromise in accuracy, we can find the data in very less amount of time.

## 1.3 Problem Statement

The nearest neighbor problem involves a large amount of data points. This makes nearest-neighbor approaches particularly important, but on another hand it increases the concern regarding the computational complexity of NN search. It has become an area of research and people have been trying to reduce the complexity.

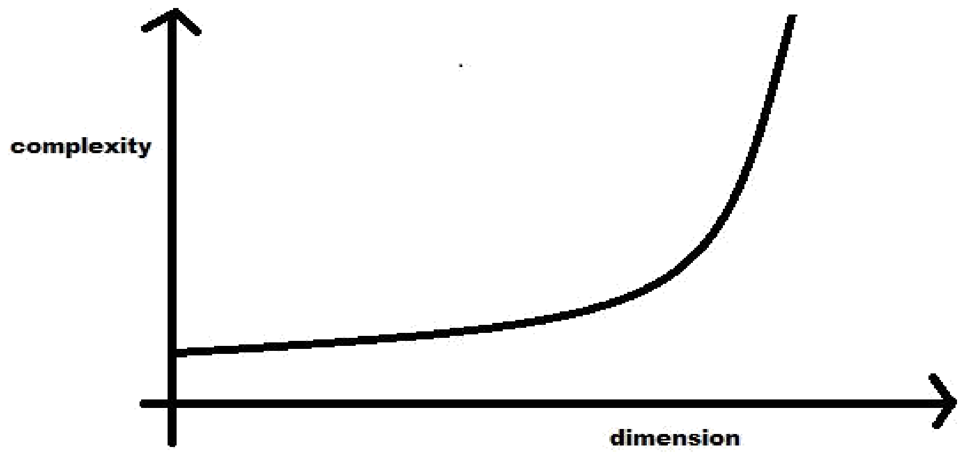


**Figure 1.4**

A basic algorithm for this problem is as follows: given a query point q, compute the distance from the given query point q to each point in the given data set P, and report the point with the minimum distance. This linear scan approach has query time of O(dn). This is tolerable for small data sets, but is too inefficient for large ones. The nearest-neighbor problem has been extensively investigated in the field of computational geometry. As a result of this research effort, many efficient solutions have been found out for the case where the points lie in a space of constant dimension. For example, if the points lie in the plane, the nearest-neighbor problem can be solved with O(log n) time per query, using only O(n) storage . Similar results can be obtained for other problems as well. Unfortunately, as the dimension increases, the algorithms become less and less efficient. More specifically, their space or time requirements grow exponentially in the dimension. In particular, the nearest-neighbor problem has a solution with O(d O(1) log n) query time, but using roughly nO(d) space.

The lack of success in removing the exponential dependence on the dimension led many researchers to guess that no efficient solution exist for this problem when the dimension is sufficiently large. At the same time, it raised the question: Is it possible to remove the exponential dependence on d.

## 1.4 Objective

The main goal is to remove the exponential dependence of the algorithms into polynomial time so that one can efficiently find out the required information in minimum amount of time. This led us to “Approximate Nearest Neighbor” problem.

The approximate nearest neighbor problem was first discovered for low dimensional version of the problem. This algorithm provides large speedups with only very less loss in accuracy. During recent years, several researchers have shown that indeed in many cases approximation enables reduction of the dependence on dimension from exponential to polynomial. In addition, there are many approximate nearest-neighbor algorithms that are more efficient than the exact ones, even though their query time and/or space usage is still exponential in the dimension.

The approximate nearest neighbor algorithm can be best implemented using data structures like basic k-nearest neighbor, KD trees and locality sensitive hashing.

## 1.5 Approaches followed

This report includes the three approximate nearest neighbor algorithms, the study, implementation and the results and applications related to these three algorithms :

- ✓ Basic KNN Algorithm
- ✓ KNN using KD Tree Algorithm
- ✓ KNN using LSH

# Chapter 2 : LITERATURE REVIEW

## 2.1 Approximate Nearest Neighbor Algorithm by Gregory Shakhnarovich, Piotr Indyk and Trevor Darrell

### 2.1.1 The Nearest Neighbor Search Problem:

The exact nearest neighbor search problem in Euclidean space is defined as follows :

Given a set P of points in a d dimensional space $R^d$, construct a data structure which given any query point q finds the point in P with the smallest distance to q.

The problem is not fully specified without defining the distance between an arbitrary pair of point p and q. The distance between p and q is defined as $|p – q|_s$

where, $||x|| = (\sum_{i=1}^{d} |x|\hat{}s)\hat{}(1/s)$

### 2.1.2 Approximate Nearest Neighbor:

Definition of c-Approximate Nearest Neighbor:

Given a data set P of points in a d dimensional space $R^d$, construct a data structure, on giving any query point q, reports any point within distance at most c times the distance from q to p, where p is the point in P closest to q.

#### 2.1.2.1 Implementation of Approximate Nearest Neighbor Algorithms :

- ✓ **KNN using KD Tree :**

The KD-tree is a data structure invented by Jon Bentley in 1975. Despite of its fairly old age, KD-tree and its variants remain probably the most popular data structures used for searching in multidimensional spaces, at least in main memory.
Given a set of *n* points in a *d*-dimensional space, the KD-tree is constructed recursively as follows. First, one finds a median of the values of the $i^{th}$ coordinates of the points (initially, i=1).

That is, a value M is computed, so that at least 50% of the points have their $i^{th}$ coordinate greater-or-equal to M, while at least 50% of the points have their coordinate smaller than or equal to M. The value of x is stored, and the set *P* is partitioned into *PL* and *PR*, where *PL* contains only the points with their $i^{th}$ coordinate smaller than or equal to M, and $|PR| = |PL| \pm 1$. The process is then repeated recursively on both *PL* and *PR*, with i replaced by i+1 (or 1, if *i*=*d*). When the set of points at a node has size 1, the recursion stops. The resulting data structure is a binary tree with n leaves, and depth [log n]. In particular, for *d* = 1, we get a (standard) balanced binary search tree. Since a median of *n* coordinates can be found in *O*(*n*) time, the whole data structure can be constructed in time *O*(*n* log *n*). For the problem of finding the nearest neighbor in P of a given query *q*, several methods exist.

Search starts recursively from the root of the tree. At any point of time, the algorithm maintains the distance R to the point closest to query point q encountered so far, initially, $R = \infty$. At a leaf node (containing, say, point p) the algorithm checks if $\|q-p'\|<R$. If so, R is set to $\|q-p'\|$, and p' is stored as the closest point candidate. In an internal node, the algorithm proceeds as follows. Let M, be the median value stored at the node, computed with respect to the $i^{th}$ coordinates. The algorithm checks if, the $i^{th}$ coordinate of q is smaller than or equal to M. If so, the algorithm recurses on the left node; otherwise it recurses on the right node. After returning from the recursion, the algorithm checks whether a ball of radius R around q contains any point in d whose $i^{th}$ coordinate is on the opposite side of M with respect to q. If this is the case, the algorithm recurses on the yet-unexplored child of the current node. Otherwise, the recursive call is terminated. At the end, the algorithm reports the final closest-point candidate.

- ✓ **KNN using Locality Sensitive Hashing**

LSH is randomized. The randomness is typically used in the construction of the data structure. Moreover, these algorithms often solve a near-neighbor problem, as opposed to the nearest-neighbor problem. The former can be viewed as a decision version of the latter. Formally, the problem definitions are as follows. A point p is an R-near neighbor of q if the distance from p to q is at most R.

Randomized *c*-approximate near-neighbor:
Given a set P of points in a d-dimensional space $R^d$, and parameters $R > 0$, $\delta > 0$, construct a data structure which, given any query point q, does the following with probability $1 - \delta$: if there is an R-near neighbor of q in P, it reports a cR-near neighbor of q in P.

Randomized near-neighbor reporting:
Given a set P of points in a d-dimensional space $R^d$, and parameters $R > 0$, $\delta > 0$, construct a data structure which, given any query point q, reports each R-near neighbor of q in P with probability $1 - \delta$.

Among all the algorithms discussed here, the LSH is probably the one that has received the most attention. Its main idea is to hash the data points using several hash functions so as to ensure that for each function, the probability of collision is much higher for points which are close to each other than for those which are far apart. Then, one can determine near neighbors by hashing the query point and retrieving elements stored in buckets containing that point.

The LSH algorithm can be used to solve either the approximate or the exact near-neighbor problem. It relies on the existence of LSH functions, defined in the following manner.

Consider a family H of hash functions mapping $R^d$ to some universe U.

Definition of Locality Sensitive Hashing:
A family H is called (r, cr, P1, P2) – sensitive if for any p, q € $R^d$

- If $||p-q|| < R$ then, $P_H [ h(q) = h(p) ] \geq P_1$
- If $||p-q|| > R$ then, $P_H [ h(q) = h(p) ] \leq P_2$

In order for a LSH family to be useful, it has to satisfy P1 > P2.

The nearest-neighbor approach consists of finding one or more examples most similar to the query, and using the labels of those examples to produce the desired estimate of the query's label. This broad description leaves two important questions:

1. What are the criteria of similarity?

The answer depends on the underlying distance measure as well as on the selection criteria, e.g., the k examples closest to the query for a fixed k, or the examples closer than a fixed threshold r.

2. How are the labels from the neighbors to be combined?

The simplest way is to take the majority vote, in classification setup, or the average, in regression setup. However, more sophisticated methods, such as locally weighted regression, have been shown to produce sometimes dramatically better results.

## 2.2 An Optimal Algorithm for Approximate Nearest Neighbor Searching in Fixed Dimensions by Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman and Angela Y. WU

Nearest Neighbor Searching is the following problem:

We are given a set s of n data points in a metric space, X, and the task is to preprocess these points so that, given any query point q € X, the data point nearest to q can be reported quickly. This is also called the closest-point problem and the post office problem Nearest neighbor searching is an important problem in a variety of applications, including knowledge discovery and data mining, pattern recognition and classification, machine learning, data compression, multimedia databases, document retrieval, and statistics.

High-dimensional nearest neighbor problems arise naturally when complex objects are represented by vectors of d numeric features. Throughout we will assume the metric space X is real d-dimensional space $R^d$. We also assume distances are measured using any Minkowski $L_m$ distance metric. For any integer $m \geq 1$, the $L_m$-distance between points $p = (p_1, p_2, p_3 \ldots\ldots p_d)$ and $q = (q_1, q_2, \ldots\ldots.. q_d)$ in $R^d$ is defined to be the m$^{th}$ root of $\sum_{1 \leq i \leq d} |p_i - q_i|^m$ .

Consider a set S of data points in Rd and a query point q € $R^d$. Given € > 0, we say that a point p € S is a (1 + €)-approximate nearest neighbor of q if

$$\text{Dist}(p, q) \leq (1 + €)\, \text{dist}(p^*, q)$$

Where $p^*$ is the true nearest neighbor to q. In other words, p is within relative error € of the true nearest neighbor. More generally, for $1 \leq k \leq n$, a kth (1 + € )- approximate nearest neighbor of q is a data point whose relative error from the true kth nearest neighbor of q is €.

Overview of an approximate nearest neighbor algorithm:

Given the query point q, we begin by locating the leaf cell containing the query point in O(log n) time by a simple descent through the tree. Next, we begin enumerating the leaf cells in increasing order of distance from the query point. We call this priority search. When a cell is visited, the distance from q to the point associated with this cell is computed. We keep track of the closest point seen so far. For example, Figure below shows the cells of such a subdivision. Each cell has been numbered according to its distance from the query point.

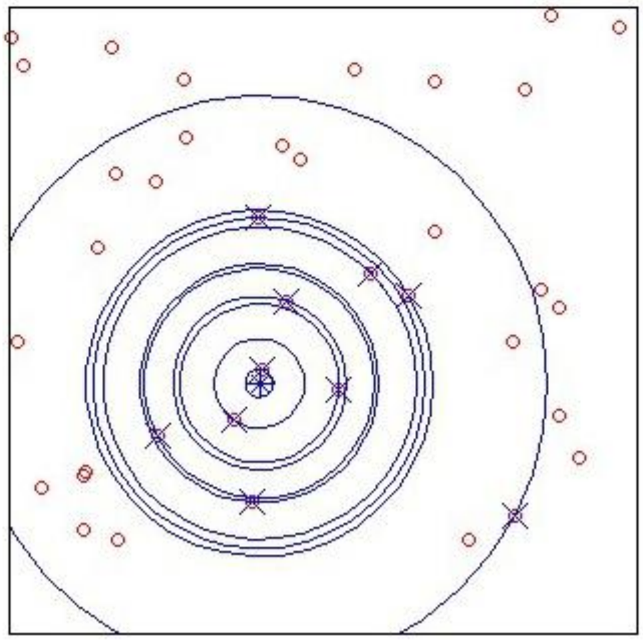


**Figure 2.1**

Let p denote the closest point seen so far. As soon as the distance from q to the current leaf cell exceeds dist(q, p)/(1 + € ) , it follows that the search can be terminated, and p can be reported as an approximate nearest neighbor to q. The reason is that any point located in a subsequently visited cell cannot be close enough to *q* to violate *p*'s claim to be an approximate nearest neighbor. By using an auxiliary heap, priority search can be performed in time O(d log n) times the number of leaf cells that are visited.

It is an easy matter to extend this algorithm to enumerate data points in "approximately" increasing distance from the query point. In particular, we will show that a simple generalization to this search strategy allows us to enumerate a sequence of $k$ approximate nearest neighbors of $q$ in additional $O(kd \log n)$ time.

### 2.2.1 Approximate Nearest Neighbor Queries :

Let $q$ be the query point in $R^{d.}$ Recall that the output of our algorithm is a data point $p$ whose distance from $q$ is at most a factor of (1 + € ) greater than the true nearest neighbor distance. Next, we enumerate the leaf cells of the subdivision in increasing order of distance from $q$.
As each cell is visited, we process it by computing the distance from $q$ to these data points and maintaining the closest point encountered so far. Let $p$ denote this point. The search terminates if the distance $r$ from the current cell to $q$ exceeds $dist(q, p)$/(1 + € ). The reason is that no subsequent point to be encountered can be closer to $q$ than $dist(q, p)$/(1 + €), and hence $p$ is a (1 + € )-approximate nearest neighbor.
Now, let us have a look on how to generalize the approximate nearest neighbor procedure to the problem of computing approximations to the k nearest neighbors of a query point. Recall that a point p is a (1 + €)-approximate jth nearest neighbor to a point q if its distance from q is a factor of at most (1 + €) times the distance to q's true jth nearest neighbor. An answer to the approximate k-nearest neighbors query is a sequence of k distinct data points p1, p2, . . . , pk, such that pj is a (1 + €)-approximation to the jth nearest neighbor of q, for $1 \leq j \leq k$.
The algorithm is a simple generalization of the single nearest neighbor algorithm. We locate the leaf cell containing the query point, and then enumerate cells in increasing order of distance from $q$. We maintain the $k$ closest data points to $q$ encountered in the search, say, by storing them in the balanced binary search tree sorted by distance. Let $r_k$ denote the distance to the $k$th closest point so far ($r_k = \infty$ if fewer than $k$ distinct points have been seen so far). The search terminates as soon as the distance from the current cell to $q$ exceeds $r_k$/(1 + €).
The reason is that no subsequently visited data point can be closer to $q$ than $r_k$/(1 1 e), and hence the data point at distance $r_k$ is an approximate $k$th nearest neighbor. There are at least $k$ -1 data points that are closer to the query point. It is easy to verify that the sorted sequence of $k$ data points seen so far is a solution to the approximate $k$-nearest neighbors query.

## 2.3 An introductory tutorial on KD Trees by Andrew W. Moore, Carnegie Mellon University

### 2.3.1 Introduction to KD Trees:

A KD-tree is a data structure for storing a finite set of points from a k-dimensional space. It was examined in detail by J. Bentley [Bentley, 1980]. Recently, S. Omohundro has recommended it in a survey of possible techniques to increase the speed of neural network learning [Omohundro, 1987].

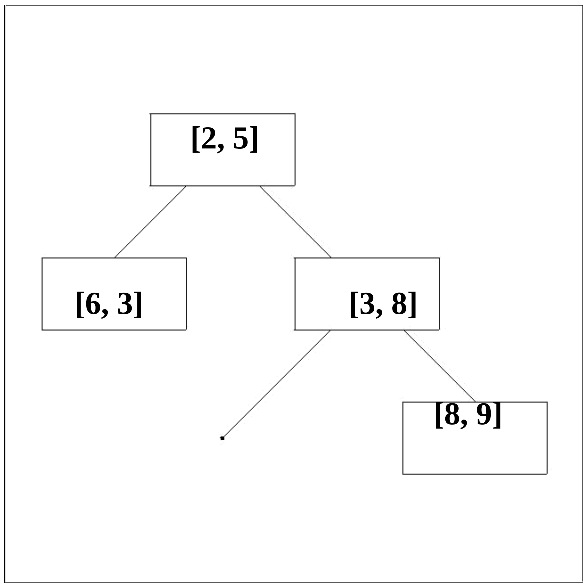


**Figure 2.2**

The splitting planes are not indicated. The [2, 5] node splits along the y = 5 plane and the [3, 8] node splits along the x = 3 plane.

### 2.3.2 Construction of KD Tree :

A KD-tree can be constructed by the algorithm in the table below. The pivot-choosing procedure of Step 2 inspects the set and chooses a \good" domain vector from this set to use as the tree's root. Whichever exemplar is chosen as root will not affect the correctness of the KD-tree, though the tree's maximum depth and the shape of the hyper regions will be affected.

Algorithm: Constructing a d-tree

Input: set of type exemplar-set
Output: kd of type kd tree

Code:

```
1: if P contains only one point
2: then return a leaf storing this point
3: else if depth is even
4: then Split P into two subsets with a vertical line l by taking the first element.
5: Let P1 be the elements smaller than the starting element and P2 be the elements having greater value than that first element.
6: $v_{left}$ BUILDKDTREE(P1, depth +1).
7: $v_{right}$ BUILDKDTREE(P2, depth +1).
8: Create a node v storing l, make $v_{left}$ the left child of v, and make $v_{right}$ the right child of v.
```

### 2.3.3 Nearest Neighbor Searching :

A first approximation is initially found at the leaf node which contains the target point. In Figure 2.3 the target point is marked X and the leaf node of the region containing the target is colored black. As is exemplified in this case, this first approximation is not necessarily the nearest neighbor, but at least we know any potential nearer neighbor must lie closer, and so must lie within the circle having center on X and passing through the leaf node. We now back up to the parent of the current node. In Figure 2.4 this parent is the black node. We compute whether it is possible for a closer solution to that so far found to exist in this parent's other child. Here it is not possible, because the circle does not intersect with the (shaded) space occupied by the parent's other child. If no closer neighbor can exist in the other child, the algorithm can immediately move up a further level, else it must recursively explore the other child. In this example, the next parent which is checked will need to be explored, because the area it covers (i.e. everywhere north of the central horizontal line) does intersect with the best circle so far.

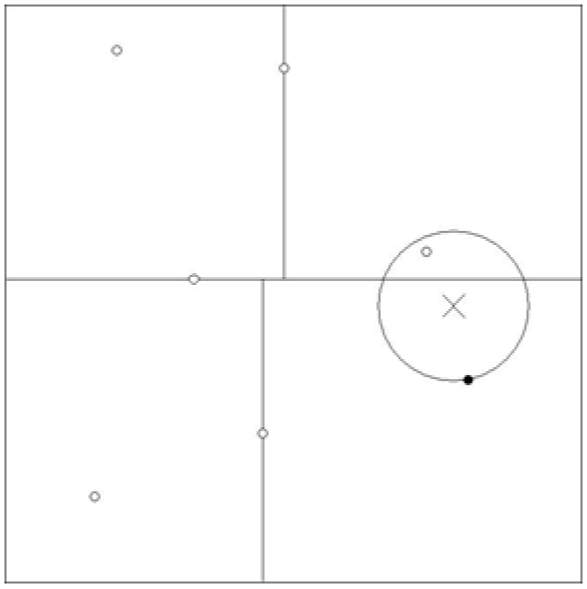


**Figure 2.3**

The black dot is the dot which owns the leaf node containing the target (the cross). Any nearer neighbor must lie inside this circle.

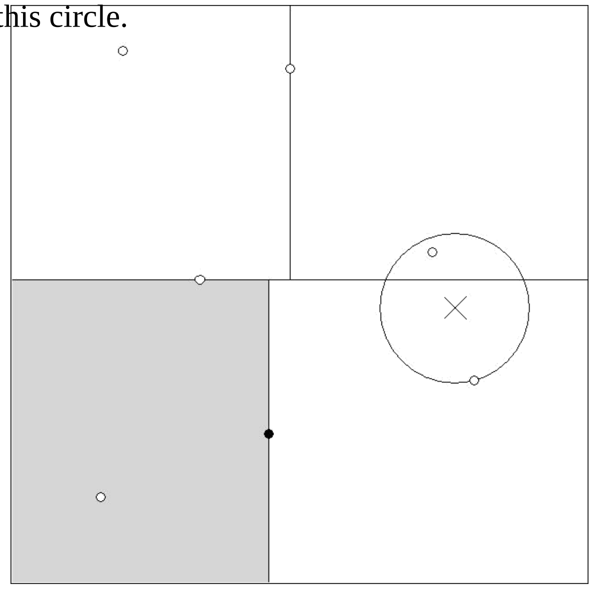


**Figure 2.4**

The black dot is the parent of the closest found so far. In this case the black dots other child (shaded grey) need not be searched.

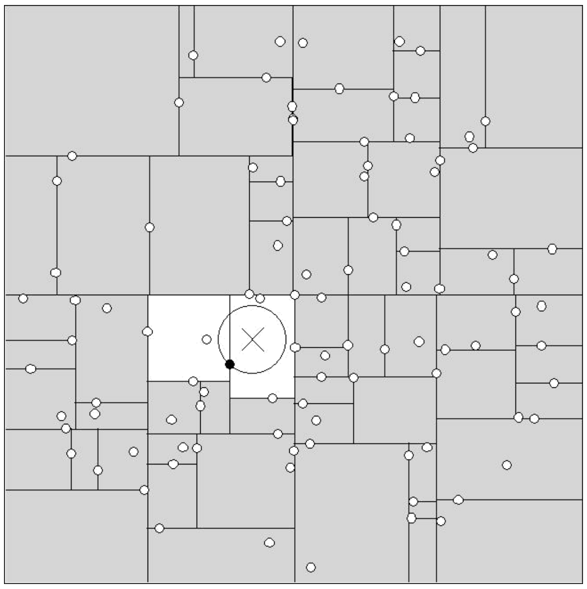


**Figure 2.5**

Generally during a nearest neighbor search only a few leaf nodes need to be inspected.

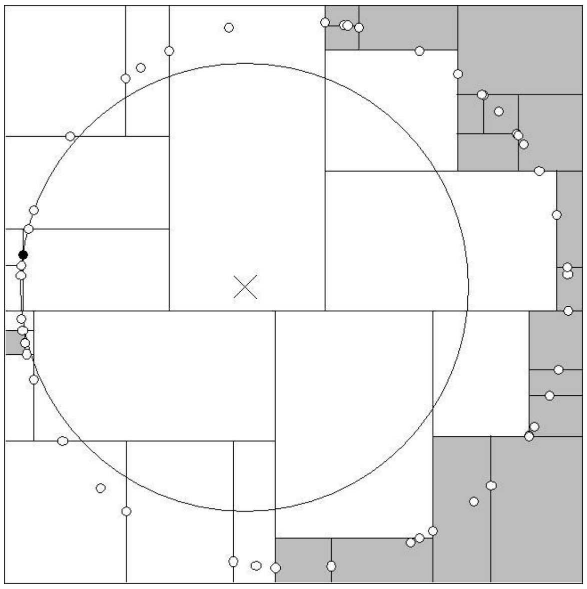


**Figure 2.6**

A bad distribution forces almost all the nodes to be inspected.

## 2.4 Range Searching using KD Tree by Hemant M. Kakde

At first sight it seems that database has little to do with geometry. The queries about the data in database can be interpreted geometrically. In this case the records in the database are transformed into points in multidimensional space and the queries about records are transformed into the queries over this set of points. Every point in the space will have some information of person associated with it.
Consider the example of the database of personal administration where the general information of every employee is stored. Consider an example of query where we want to report all employees born between 1950 and 1955, who earns between Rs.3000 and Rs.4000 per month. The query will report all the points that whose frost co-ordinate lies between 1950 and 1955, and second co-ordinate lies between 3000 and 4000. In general if we are interested in answering queries on d - fields of the records in our database, we transform the records to points in d-dimensional space. Such a query is called rectangular range query, or an orthogonal range query. In Range Search problems, the collection of points in space and the query is some standard geometric shape translatable in space. Range search consist either of retrieving (report problems) or of counting ( count problems ) all points contained within the query domain.

Let us consider the set of points $P=p_1,p_2......p_n$. We have to search the range [x, x'] and we have to report which points lies in that range. To solve the problem of range searching we use the data structure known as balanced binary search tree T. The leaves of the T store the points of P and the internal nodes of T will store the splitting values to guide the search. Let $x_v$ denote the value stored at each split node v. The left sub tree of the node v contains all points smaller than or equal to $x_v$, and the right sub tree contains all the points strictly greater than $x_v$.

Let we search with x and x' in T. μ and μ' be the two leaves where the searches end respectively. Then the points in the interval [x: x'] are stored in the leaves in between μ and μ' including μ and μ'. To find the leaves between μ and μ', we select the tree rooted at nodes v in between the two search paths whose parent are on the search path. To find these nodes we first find the node $v_{split}$ where the paths to x and x' splits.

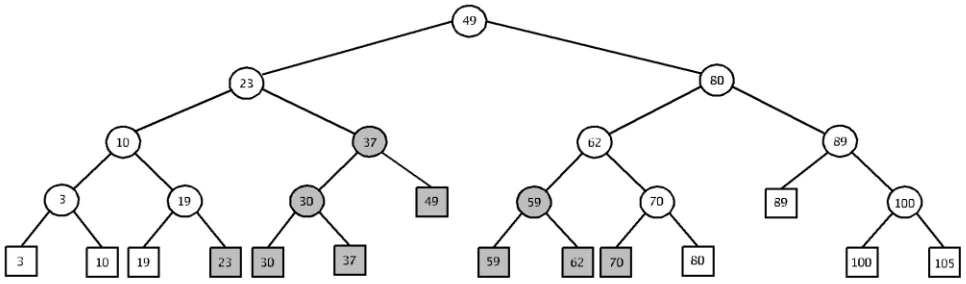


**Figure 2.7**

### 2.3.1 Algorithm:

To find the split node let us consider lc(v) and rc(v) denote the left and right child, respectively, of the node v.

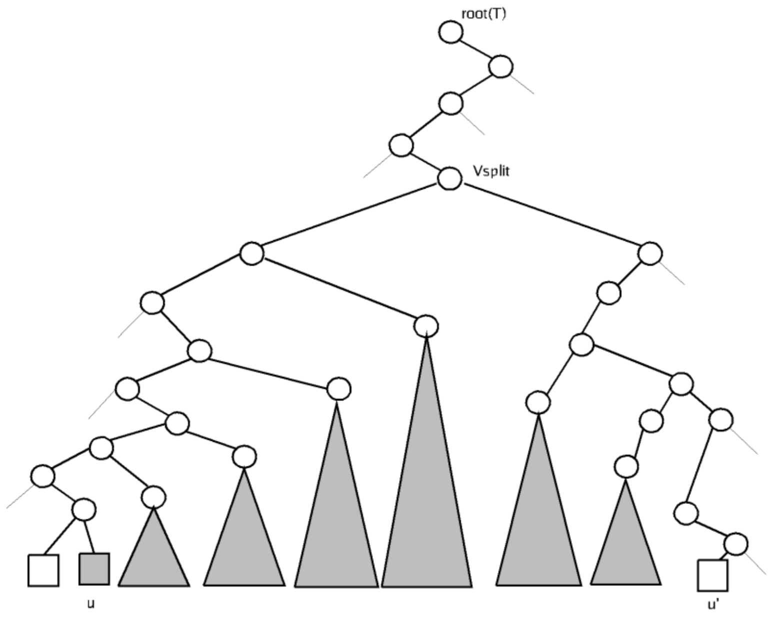


**Figure 2.8**

Procedure name FINDSPLITNODE

Input: A Tree T and two values x and x' with xx'.

Output: The node v where the paths to x and x' split, or the leaf where both path ends.

1: v root(T).

2: while v is not a leaf and ( x' ≤ xv or x > xv).

3: do if x' ≤ xv

```
4: then v ← lc(v)
5: else v ← rc(v)
6: return v
```

Starting from $v_{split}$ we then follow the search path of x. At each node where the path goes left, we report all the leaves in the right sub tree, because this sub tree is in between the two search paths. Similarly, we follow the path of x' and we report the leaves in the left sub tree of the node where the path goes right. Finally we check the points stored at the leaves whether they lies in the range [ x , x' ] or not.

```
Procedure name 1D_RANGE_QUERY(T,[X:X'])
Input: A binary search tree T and a range [x, x'].
Output: All points stored in T that lie in the range.
1: $v_{split}$  FIND_SPLIT_NODE(T, x, x').
2: if $v_{split}$ is a leaf.
3: then check if the point stored at $v_{split}$ must be reported.
4: else (* Follow the path to x and report the points in sub tree right of the path*).
5: v ← lc( $v_{split}$ )
6: while v is not a leaf.
7: do if $x' \leq xv$
8: then REPORT_SUB_TREE(rc(v))
9: v lc(v)
10: else v rc(v)
11: check if the point stored at the leaf v must be reported.
12: Similarly, follow the path to x', reports the points sub tree left of the path, and check if the
point stored at the leaf where the path ends must be reported.
```

### 2.3.2 Complexity:

The data structure binary search tree uses O(n) storage and it can be built in O(n logn) time. In worst case all the points could be in query range, so the query time will be O(n). The query time of O(n) cannot be avoided when we have to report all the points. Therefore we shall give more refined analysis of query time. The refined analysis takes not only n, the number of points in the set P, into account but also k, the number of reported points.

As we know that the REPORTSUBTREE is linear in the number of reported points, then the total time spent in all such calls is O(k). The remaining nodes that are visited are the nodes of the search path of x and x'. Because T is balanced, these paths have a length O(log n). The time we spent at each node is O(1), so the total time spent in these nodes is O(log n), which gives a query time of O(logn + k).

### 2.3.3 Example of KD Tree:

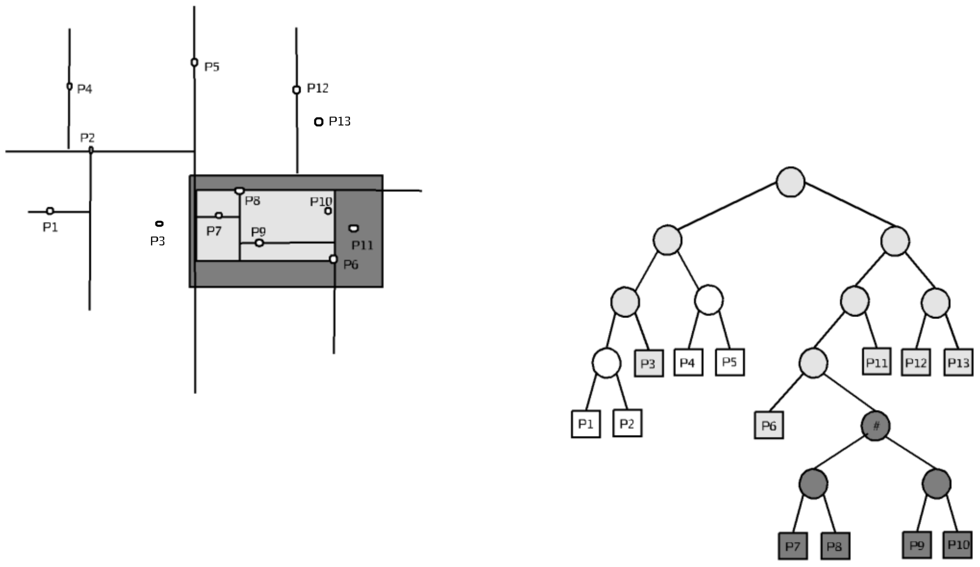


Fig : A Query on kd-tree.

**Figure 2.9**

# Chapter 3: IMPLEMENTATION AND STUDY

## 3.1 K-Nearest Neighbor Algorithm:

KNN is a non parametric algorithm. That is a quite concise statement, when you say a technique is non-parametric, it means that it does not make any assumptions on the underlying data distribution. This is quite useful, as in the real scenario, most of the practical data does not obey the typical theoretical assumptions made. Non parametric algorithms like KNN come to the rescue here.

It is also a lazy algorithm. What this means is that it does not use the training data points to do any generalization. In other words, there is no explicit training phase or it is very minimal. This means the training phase is pretty fast. Lack of generalization means that this, algorithm keep all the training data. More exactly, all the training data is needed during the testing phase. (Well this is an exaggeration, but not far from truth). Most of the lazy algorithms – especially KNN makes decision based on the entire training data set (in the best case a subset of them).

The dilemma is pretty obvious here – There is a non-existent or minimal training phase but a costly testing phase. The cost is in terms of both time and memory. More time might be needed as in the worst case, all data points might take point in decision. More memory is needed as we need to store all training data.

### 3.1.1 Assumptions in KNN:

Before using KNN, let us revisit some of the assumptions in KNN.
KNN assumes that the data is in a feature space. More exactly, the data points are in a metric space. The data can be scalars or possibly even multidimensional vectors. Since the points are in feature space, they have a notion of distance – This need not necessarily be Euclidean distance although it is the one commonly used.

Each of the training data comprises of a set of vectors and class label associated with each vector. In the simplest case, it will be either + or – (for positive or negative classes). But KNN,

can work equally well with arbitrary number of classes.
We are also given a single number “k”. This number k decides how many neighbors (where neighbors are defined based on the distance metric) influence the classification. This is usually an odd number if the number of classes is 2. If k=1, then the algorithm is simply called the nearest neighbor algorithm.

### 3.1.2 KNN for Classification:

Let us see how to use KNN for classification. In this case, we are given some data points for training and also a new unlabeled data for testing. Our aim is to find the class label for the new point. The algorithm has different behavior based on k.

**Case 1: k=1 or Nearest Neighbor Rule:**

This is the simplest scenario. Let x be the point to be labeled. Find the point closest to x. Let it be y. Now nearest neighbor rule asks to assign the label of y to x. This seems too simplistic and sometimes even counter intuitive. If you feel that this procedure will result a huge error, you are right – but there is a catch. This reasoning holds only when the number of data points is not very large.

If the number of data points is very large, then there is a pretty high chance that labels of x and y are same. An example might help – Let us say you have a (potentially) biased coin. You toss it for 1 million times and you got head 900,000 times. Then, most likely your next call will be head. We can use a similar argument here.

Let us try an informal argument here – Assume all points are in a d dimensional plane. The number of points is quite large. This means that the density of the plane at any point is fairly high. In other words, within any space there is adequate number of points. Consider a point x in the space which also has a lot of neighbors. Now let y be the nearest neighbor. If x and y are nearly close, then there is a probability that x and y belong to same class.

**Case 2: K=k or k-Nearest Neighbor Rule:**

This is simply the extension of 1NN. Basically what we do is that, we try to find the k nearest neighbor and do a majority voting. Typically k is odd when the number of classes is 2. Let us say k = 5 and there are 5 instances of C1 and 2 instances of C2. In this case, KNN says that new point has to be labeled as C1 as it forms the majority. We follow a similar argument when there are multiple classes.

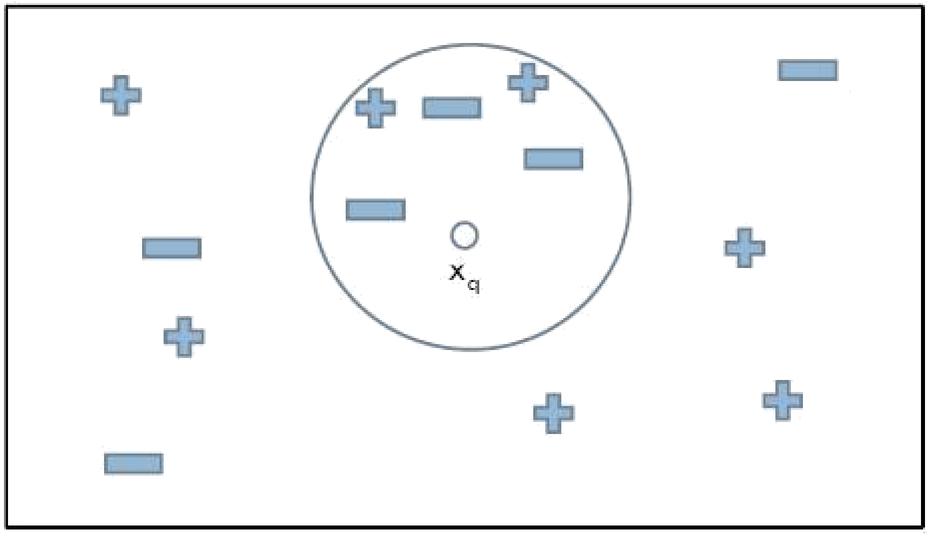


**Figure 3.1**

A very common thing to do is weighted KNN where each point has a weight which is typically calculated using its distance. For example, under inverse distance weighting, each point has a weight equal to the inverse of its distance to the point to be classified. This means that neighboring points have a higher chance than the farther points.

It is quite obvious that the accuracy might increase when you increase k but the computation cost also increases.

### 3.1.3 Curse of dimensionality:

Distance usually relates to all the attributes and assumes all of them have the same effects on distance. The similarity metrics do not consider the relation of attributes which result in inaccurate distance and then impact on classification precision. Wrong classification due to

presence of many irrelevant attributes is often termed as the curse of dimensionality.
For example: Each instance is described by 20 attributes out of which only 2 are relevant in determining the classification of the target function. In this case, instances that have identical values for the 2 relevant attributes may nevertheless be distant from one another in the 20 dimensional instance spaces.

### 3.1.4 Source Code:

```
#include <stdio.h>

#include <stdlib.h>

#include <math.h>

#include<string.h>

int m=0,f=0,n=0,s;

float dist[300][2],arrm[100][4],arrf[100][4],arrn[100][4];

int caldistance(float x[4])      /* calculates the distance of query point from the points in training
data*/

{int i;

int l=0;

for(i=0;i<m;i++)

{

dist[l][0]=sqrt(sqrt(pow((x[0]-arrm[i][0]),2) + pow((x[1]-arrm[i][1]),2) + pow((x[2]-
arrm[i][2]),2) + pow((x[3]-arrm[i][3]),2)));

dist[l][1]=0.00;

l++;

}
```

```
for(i=0;i<f;i++)

{

dist[l][0]=sqrt(sqrt(pow((x[0]-arrf[i][0]),2) + pow((x[1]-arrf[i][1]),2)+ pow((x[2]-arrf[i][2]),2) +
pow((x[3]-arrf[i][3]),2)));

dist[l][1]=2.00;

l++;

}

for(i=0;i<n;i++)

{

dist[l][0]=sqrt(sqrt(pow((x[0]-arrn[i][0]),2) + pow((x[1]-arrn[i][1]),2) + pow((x[2]-arrn[i][2]),2)
+ pow((x[3]-arrn[i][3]),2)));

dist[l][1]=1.00;

l++;

}

return l;

}

void sort(int l)  /* sorts the distances present in the dist[][] array */

{

int y,z,temp,temp1;

for(y=0;y<l;y++)

{

for(z=0;z<l-1;z++)
```

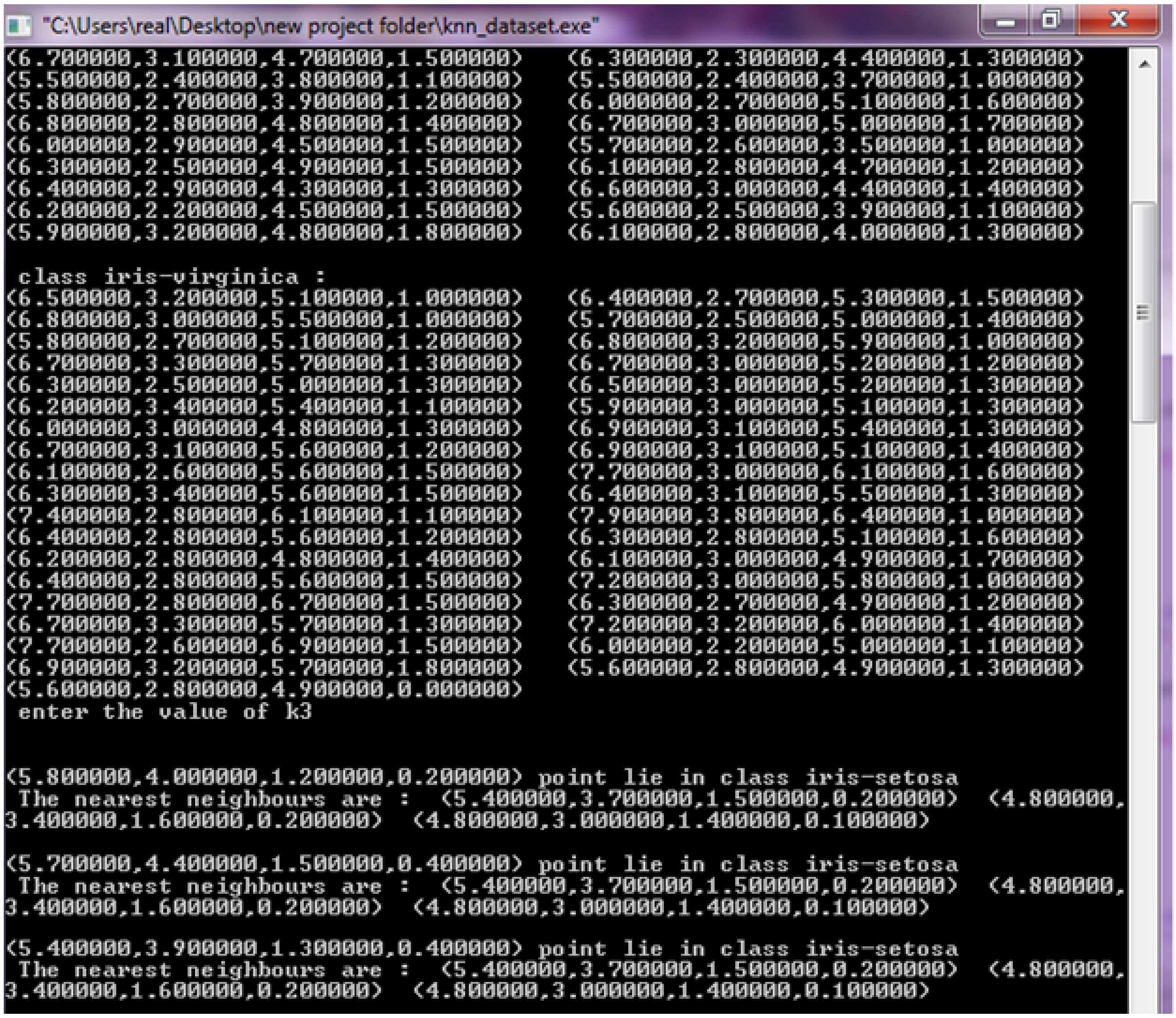
```
{
if(dist[z][0]>=dist[z+1][0])
{ temp=dist[z][0];
temp1=dist[z][1];
dist[z][0]=dist[z+1][0];
dist[z][1]=dist[z+1][1];
dist[z+1][0]=temp;
dist[z+1][1]=temp1;
}}
}}
void neigh(float x[4],int k)  /* allot the class to the query point and find its nearest neighbors*/
{
int i;
if((dist[0][1]==0.00 && dist[1][1]==0.00 && dist[2][1]==0.00) || (dist[0][1]==0.00 &&
dist[1][1]==0.00) || (dist[0][1]==0.00 && dist[2][1]==0.00) || (dist[1][1]==0.00 &&
dist[2][1]==0.00))
{
arrm[++m][0]=x[0];
arrm[m][1]=x[1];
arrm[m][2]=x[2];
arrm[m][3]=x[3];
printf(" \n(%f,%f,%f,%f) point lie in class iris-setosa",x[0],x[1],x[2],x[3]);
printf("\n The nearest neighbours are : ");

for(i=0;i<k;i++)

printf(" (%f,%f,%f,%f) ",arrm[i][0],arrm[i][1],arrm[i][2],arrm[i][3]);

}

else if((dist[0][1]==1.00 && dist[1][1]==1.00 && dist[2][1]==1.00) || (dist[1][1]==1.00 &&
dist[2][1]==1.00)|| (dist[0][1]==1.00 && dist[2][1]==1.00)|| (dist[0][1]==1.00 &&
dist[1][1]==1.00) )

{

arrn[++n][0]=x[0];

arrn[n][1]=x[1];

arrn[n][2]=x[2];

arrn[n][3]=x[3];

printf(" (%f,%f,%f,%f) point lie in class iris-versicolor",x[0],x[1],x[2],x[3]);

printf("\nThe nearest neighbours are : ");

for(i=0;i<k;i++)

printf("(%f,%f,%f,%f) ",arrn[i][0],arrn[i][1],arrn[i][2],arrn[i][3]);}

else if((dist[0][1]==2.00 && dist[1][1]==2.00 && dist[2][1]==2.00) || (dist[0][1]==2.00 &&
dist[2][1]==2.00) || (dist[0][1]==2.00 && dist[1][1]==2.00) || (dist[1][1]==2.00 &&
dist[2][1]==2.00) )

{

arrf[++f][0]=x[0];

arrf[f][1]=x[1];
```

```
arrf[f][2]=x[2];

arrf[f][3]=x[3];

printf(" \n(%f,%f,%f,%f) point lie in class iris-virginica",x[0],x[1],x[2],x[3]);

printf(" \nThe nearest neighbours are : ");

for(i=0;i<k;i++)

printf(" (%f,%f,%f,%f) ",arrf[i][0],arrf[i][1],arrf[i][2],arrf[i][3]);

}}

int main()

{   FILE *fp,*fp1;

    int i,j,k;

    float val[4],p,q,r,o;

    char str[15],c,str1[15],str2[15],str3[15];

    fp = fopen("sirdata.txt","r");

    fp1=fopen("knn_sir_data.txt","r");

    strcpy(str1,"Irissetosa");

    strcpy(str2,"Irisversicolor");

    strcpy(str3,"Irisvirginica");

    if(fp==NULL)

    {       printf("error");

        return(-1);}

    else
```

```
{     while(!feof(fp))
{ fscanf(fp, "%f" ,&o);
    fscanf(fp, "%f" ,&p);
    fscanf(fp,"%f",&q);
    fscanf(fp,"%f",&r);
    fscanf(fp,"%s",&str);
    if(strcmp(str,str1)==0)
    { arrm[m][0]=o;
    arrm[m][1]=p;
    arrm[m][2]=q;
    arrm[m][3]=r;
    m++;}
    else if(strcmp(str,str2)==0)
    {arrn[n][0]=o;
    arrn[n][1]=p;
    arrn[n][2]=q;
    arrn[n][3]=r;
    n++;        }
    else if(strcmp(str,str3)==0)
    { arrf[f][0]=o;
    arrf[f][1]=p;
    arrf[f][2]=q;

    arrf[f][3]=r;

    f++;}}}

printf("\n class iris-setosa :\n");

for(i=0;i<m;i++)

        printf("(%f,%f,%f,%f)   ",arrm[i][0],arrm[i][1],arrm[i][2],arrm[i][3]);

printf("\n class iris-vesicolor :\n");

for(i=0;i<n;i++)

        printf("(%f,%f,%f,%f)   ",arrn[i][0],arrn[i][1],arrn[i][2],arrn[i][3]);

    printf("\n class iris-virginica :\n");

for(i=0;i<f;i++)

        printf("(%f,%f,%f,%f)   ",arrf[i][0],arrf[i][1],arrf[i][2],arrn[i][3]);

printf("\n enter the value of k");

scanf("%d",&k);

while(!feof(fp1))

{ fscanf(fp1,"%f",&val[0]);

fscanf(fp1,"%f",&val[1]);

fscanf(fp1,"%f",&val[2]);

fscanf(fp1,"%f",&val[3]);

s=caldistance(val);

sort(s);
```

```
neigh(val,k);}

fclose(fp);

fclose(fp1);  return 0;}
```

### 3.1.4.1 Output :



"C:\Users\real\Desktop\new project folder\knn_dataset.exe"

```
(6.700000,3.100000,4.700000,1.500000)   (6.300000,2.300000,4.400000,1.300000)
(5.500000,2.400000,3.800000,1.100000)   (5.500000,2.400000,3.700000,1.000000)
(5.800000,2.700000,3.900000,1.200000)   (6.000000,2.700000,5.100000,1.600000)
(6.800000,2.800000,4.800000,1.400000)   (6.700000,3.000000,5.000000,1.700000)
(6.000000,2.900000,4.500000,1.500000)   (5.700000,2.600000,3.500000,1.000000)
(6.300000,2.500000,4.900000,1.500000)   (6.100000,2.800000,4.700000,1.200000)
(6.400000,2.900000,4.300000,1.300000)   (6.600000,3.000000,4.400000,1.400000)
(6.200000,2.200000,4.500000,1.500000)   (5.600000,2.500000,3.900000,1.100000)
(5.900000,3.200000,4.800000,1.800000)   (6.100000,2.800000,4.000000,1.300000)

 class iris-virginica :
(6.500000,3.200000,5.100000,1.000000)   (6.400000,2.700000,5.300000,1.500000)
(6.800000,3.000000,5.500000,1.000000)   (5.700000,2.500000,5.000000,1.400000)
(5.800000,2.700000,5.100000,1.200000)   (6.800000,3.200000,5.900000,1.000000)
(6.700000,3.300000,5.700000,1.300000)   (6.700000,3.000000,5.200000,1.200000)
(6.300000,2.500000,5.000000,1.300000)   (6.500000,3.000000,5.200000,1.300000)
(6.200000,3.400000,5.400000,1.100000)   (5.900000,3.000000,5.100000,1.300000)
(6.000000,3.000000,4.800000,1.300000)   (6.900000,3.100000,5.400000,1.300000)
(6.700000,3.100000,5.600000,1.200000)   (6.900000,3.100000,5.100000,1.400000)
(6.100000,2.600000,5.600000,1.500000)   (7.700000,3.000000,6.100000,1.600000)
(6.300000,3.400000,5.600000,1.500000)   (6.400000,3.100000,5.500000,1.300000)
(7.400000,2.800000,6.100000,1.100000)   (7.900000,3.800000,6.400000,1.000000)
(6.400000,2.800000,5.600000,1.200000)   (6.300000,2.800000,5.100000,1.600000)
(6.200000,2.800000,4.800000,1.400000)   (6.100000,3.000000,4.900000,1.700000)
(6.400000,2.800000,5.600000,1.500000)   (7.200000,3.000000,5.800000,1.000000)
(7.700000,2.800000,6.700000,1.500000)   (6.300000,2.700000,4.900000,1.200000)
(6.700000,3.300000,5.700000,1.300000)   (7.200000,3.200000,6.000000,1.400000)
(7.700000,2.600000,6.900000,1.500000)   (6.000000,2.200000,5.000000,1.100000)
(6.900000,3.200000,5.700000,1.800000)   (5.600000,2.800000,4.900000,1.300000)
(5.600000,2.800000,4.900000,0.000000)
 enter the value of k3


(5.800000,4.000000,1.200000,0.200000) point lie in class iris-setosa
 The nearest neighbours are :  (5.400000,3.700000,1.500000,0.200000)  (4.800000,
3.400000,1.600000,0.200000)  (4.800000,3.000000,1.400000,0.100000)

(5.700000,4.400000,1.500000,0.400000) point lie in class iris-setosa
 The nearest neighbours are :  (5.400000,3.700000,1.500000,0.200000)  (4.800000,
3.400000,1.600000,0.200000)  (4.800000,3.000000,1.400000,0.100000)

(5.400000,3.900000,1.300000,0.400000) point lie in class iris-setosa
 The nearest neighbours are :  (5.400000,3.700000,1.500000,0.200000)  (4.800000,
3.400000,1.600000,0.200000)  (4.800000,3.000000,1.400000,0.100000)
```

**Figure 3.2**

## 3.2 KNN using KD Trees:

KD-trees are data structures which are used to store points in k-dimensional space. As it follows from its name, KD-tree is a tree. Tree leaves store the data points of the datasets (one or several points in each leaf). Each point is stored in one and only one leaf, each leaf stores at least one data point. Tree nodes correspond to splits of the space (axis-oriented splits are used in most implementations). Each split divides space and dataset into two distinct parts. Subsequent splits from the root node to one of the leaves remove parts of the dataset (and space) until only small part of the dataset (and space) is left.

KD trees allow to efficiently perform searches like "all points at distance lower than R from X" or "k nearest neighbors of X". When processing such query, we find a leaf which corresponds to X. Then we process points which are stored in that leaf, and then we start to scan nearby leaves. At some point we may notice that distance from X to the leaf is higher than the worst point found so far. It is time to stop search, because next leafs won't improve search results. Such algorithm is good for searches in low-dimensional spaces. However, its efficiency decreases as dimensionality grows, and in high-dimensional spaces KD trees give no performance over basic O(N) linear search.

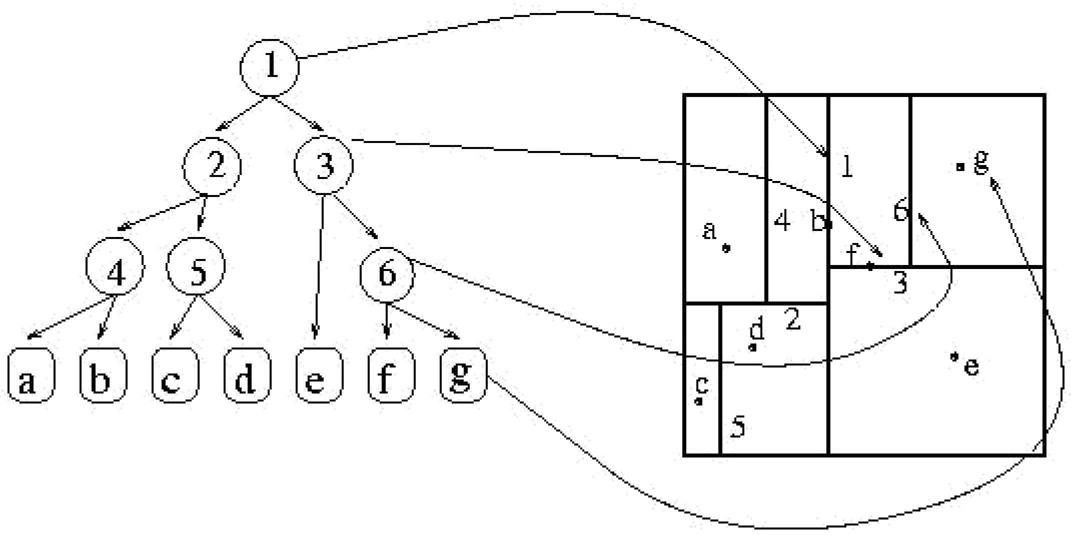


**Figure 3.3: an example of a kd tree**

### 4.2.1 Algorithm for creating a KD tree:

```
Procedure name BUILDKDTREE (P, depth)

Input: A set of points P and the current depth

Output: The root of the kd-tree storing P.

 1: if P contains only one point

2: then return a leaf storing this point

3: else if depth is even

4: then Split P into two subsets with a vertical line l by taking the first element.

5: Let P1 be the elements smaller than the starting element and P2 be the elements having greater
value than that first element.

6: vleft BUILDKDTREE(P1, depth +1).

7: vright BUILDKDTREE(P2, depth +1).

8:Create a node v storing l, make vleft the left child of v, and make vright the right child of v.
```

### 4.2.2 Algorithm for implementation of K Nearest Neighbor using KD Tree:

```
Procedure name SEARCHKDTREE (v, R)

Input: The root of ( a sub tree of ) a kd-tree, and a range R.

Output: All points at leaves below v that lies in range.

1: if v is a leaf

2: then Report the stored at v if it lies in R

3: else if region(lv(c)) is fully contained in R
```

```
4: then REPORTSUBTREE(lc(v))
5: else if region(lc(v)) intersects R
6: then SEARCHKDTREE(lc(v),R)
7: if region(rv(c)) is fully contained in R
8: then REPORTSUBTREE(rc(v))
9: else if region(lc(v)) intersects R
10: then SEARCHKDTREE(lc(v),R)
```

**3.2.3 Source Code :**

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
float inor[150][4];
int i,j,flag;
typedef struct KdNode  /* structure of the nodes present in the tree */
{  float Data[4];
   struct KdNode *Left;
   struct KdNode *Right; };
int a=0;
struct KdNode *root = NULL;
/* create node function is allocating the memory to each newly created node */
```

```
struct KdNode* createNode(float val[4]) {
        struct KdNode *newNode;
        newNode = (struct KdNode *) malloc(sizeof (struct KdNode));
        newNode->Data[0] = val[0];
        newNode->Data[1] = val[1];
        newNode->Data[2] = val[2];
        newNode->Data[3] = val[3];
        newNode->Left = NULL;
        newNode->Right = NULL;
        return(newNode);
  }
/* inserts new node in the tree*/
void insertion(struct KdNode **node, float Val[4])
 { int count=0;
 if (*node == NULL) {
        *node = createNode(Val);
}
else{
        up: if (Val[count] <= (*node)->Data[count]) {
                if((*node)->Left == NULL)
                {    (*node)->Left= createNode(Val); }
        else
        {  count++;
            (*node)= (*node)->Left;
            if(count>=4)
                count=0;
            goto up; }}
            else if(Val[count] > (*node)->Data[count])
            { if((*node)->Right == NULL)
                { (*node)->Right= createNode(Val); }
        else
        {    count++;
            *node = (*node)->Right;
            if(count>=4)
                count=0;
            goto up;}}}}
void inorder(struct KdNode *root) /* prints the inorder traversal of the tree created*/
{
   struct KdNode *temp;
   temp=root;
   if(temp!=NULL)
    {
      inorder(temp->Left);

    inor[a][0]=temp->Data[0];

    inor[a][1]=temp->Data[1];

    inor[a][2]=temp->Data[2];

    inor[a][3]=temp->Data[3];

    a++;

    inorder((temp->Right));

    }}

struct KdNode *temp1= NULL;

struct KdNode *temp = NULL;

void search(float Item[4], int k, struct KdNode *root) /* search whether the node is already
present in the tree if not, then insert it and find nearest neighbors*/

{ flag=1;

 while(temp!=NULL)

    { if((temp->Data[0]==Item[0] )&& (temp->Data[1]==Item[1]) && (temp-
>Data[2]==Item[2]) && (temp->Data[3]==Item[3]))

  {

    here: flag=0;

    for(i=0;i<a;)

      {   if(Item[0]==inor[i][0]  &&  Item[1]==inor[i][1] && Item[2]==inor[i][2] &&
Item[3]==inor[i][3])

               { goto here2;   }
```

```
          else
            i++;  }
        here2: printf("\nThe nearest neighbours for the entered element are: ");
               for(j=i-1;j>=(i-k/2);j--)
                   printf("(%f,%f,%f,%f)  ", inor[j][0],inor[j][1],inor[j][2],inor[j][3]);
               for(j=i+1;j<=(i+1+k/2);j++)
                   printf("(%f,%f,%f,%f)  ", inor[j][0],inor[j][1],inor[j][2],inor[j][3]);
          goto end1; }
  if(flag==1)
  {      search(Item,k,temp->Left);
      search(Item,k,temp->Right);
  }
  else
    break; }
 if(flag==1)
 {  insertion(&root,Item);
    a=0;
    inorder(root);
            goto here; }
 end1: printf("\n Searching done"); }
int main( )
```

```
{
   FILE *fp,*fp1;
   struct KdNode *T;
   float It[4],val[4];
   int j=0,k;
   char str[11],c;
   double time_taken;
   clock_t t;
   T = NULL;
   fp=fopen("sir_data1.txt","r");
  fp1=fopen("knn_sir_data.txt","r");
   if(fp==NULL)
   {
 printf("error");
   return(-1);
 }
   else
   {
     while(!feof(fp))
     {
      fscanf(fp,"%f",&It[0]);
```

```
        fscanf(fp,"%f",&It[1]);

        fscanf(fp,"%f",&It[2]);

        fscanf(fp,"%f",&It[3]);

        fscanf(fp,"%s",&str);

        insertion(&T,It);

    }

}

inorder(T);

printf("\n Enter the no of nearest neighbours: ");

scanf("%d",&k);

while(!feof(fp1))

    {

    fscanf(fp1,"%f",&val[0]);

    fscanf(fp1,"%f",&val[1]);

    fscanf(fp1,"%f",&val[2]);

    fscanf(fp1,"%f",&val[3]);

    printf(" %f,%f,%f,%f\n ",val[0],val[1],val[2],val[3]);

    t=clock();

    search(val,k,T);}

t= clock()-t;

time_taken=(double)t/CLOCKS_PER_SEC;
```

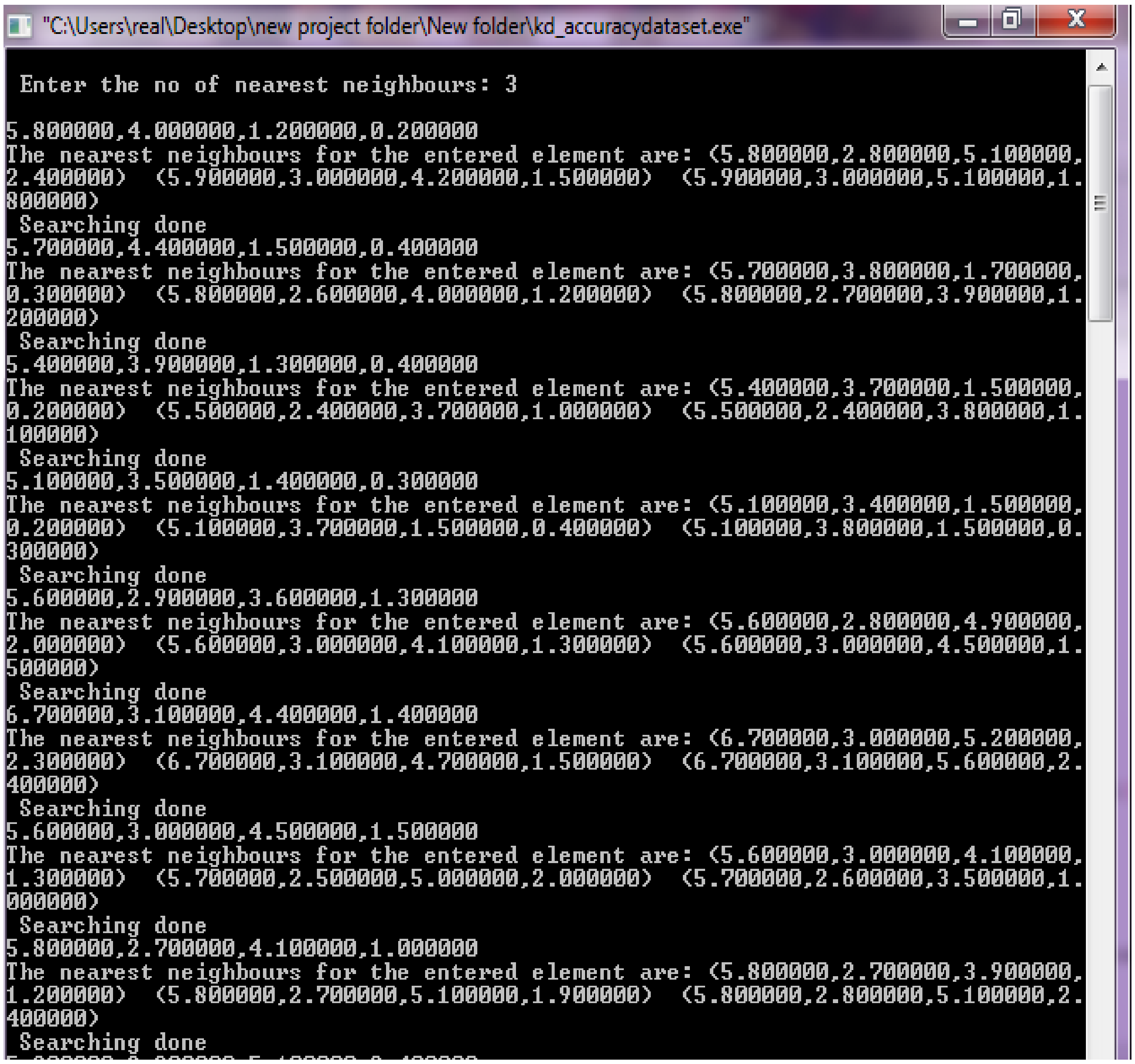
printf("Time taken to search neighbours: %lf",time_taken);

fclose(fp);

fclose(fp1);

    return 0;}

### 3.2.3.1 Output :



```
"C:\Users\real\Desktop\new project folder\New folder\kd_accuracydataset.exe"

 Enter the no of nearest neighbours: 3

5.800000,4.000000,1.200000,0.200000
The nearest neighbours for the entered element are: (5.800000,2.800000,5.100000,
2.400000)  (5.900000,3.000000,4.200000,1.500000)  (5.900000,3.000000,5.100000,1.
800000)
 Searching done
5.700000,4.400000,1.500000,0.400000
The nearest neighbours for the entered element are: (5.700000,3.800000,1.700000,
0.300000)  (5.800000,2.600000,4.000000,1.200000)  (5.800000,2.700000,3.900000,1.
200000)
 Searching done
5.400000,3.900000,1.300000,0.400000
The nearest neighbours for the entered element are: (5.400000,3.700000,1.500000,
0.200000)  (5.500000,2.400000,3.700000,1.000000)  (5.500000,2.400000,3.800000,1.
100000)
 Searching done
5.100000,3.500000,1.400000,0.300000
The nearest neighbours for the entered element are: (5.100000,3.400000,1.500000,
0.200000)  (5.100000,3.700000,1.500000,0.400000)  (5.100000,3.800000,1.500000,0.
300000)
 Searching done
5.600000,2.900000,3.600000,1.300000
The nearest neighbours for the entered element are: (5.600000,2.800000,4.900000,
2.000000)  (5.600000,3.000000,4.100000,1.300000)  (5.600000,3.000000,4.500000,1.
500000)
 Searching done
6.700000,3.100000,4.400000,1.400000
The nearest neighbours for the entered element are: (6.700000,3.000000,5.200000,
2.300000)  (6.700000,3.100000,4.700000,1.500000)  (6.700000,3.100000,5.600000,2.
400000)
 Searching done
5.600000,3.000000,4.500000,1.500000
The nearest neighbours for the entered element are: (5.600000,3.000000,4.100000,
1.300000)  (5.700000,2.500000,5.000000,2.000000)  (5.700000,2.600000,3.500000,1.
000000)
 Searching done
5.800000,2.700000,4.100000,1.000000
The nearest neighbours for the entered element are: (5.800000,2.700000,3.900000,
1.200000)  (5.800000,2.700000,5.100000,1.900000)  (5.800000,2.800000,5.100000,2.
400000)
 Searching done
```

**Figure 3.4**

## 3.3 KNN using Locality Sensitive Hashing:

The LSH algorithm relies on the existence of locality-sensitive hash functions. Let F be a family of hash functions mapping to some universe U. For any two points x and y, consider a process in which we choose a function f from F uniformly at random, and analyze the probability that f(x) = f(y). The family F is called locality sensitive (with proper parameters) if it satisfies the following condition:

**Definition:**

A family is called (R, cR, X1, X2)-sensitive if for any two points x, y € D.

- if $\|x - y\| \leq R$ then PrF [f(y) = f(x)] ≥ X1,
- if $\|x - y\| \geq cR$ then PrF [f(y) = f(x)] ≤ X2.

In order for a locality-sensitive hash (LSH) family to be useful, it has to satisfy X1 > X2.

One of the main applications of LSH is to provide a method for efficient approximate nearest neighbor search algorithms. Consider an LSH family F. The algorithm has two main parameters: the width parameter K and the number of hash tables L.

In the first step, we define a new family G of hash functions g, where each function g is obtained by concatenating k functions $h_1 \ldots h_k$ from F, i.e. $g(p)=[h_1(p),\ldots,h_k(p)]$. In other words, a random hash function g is obtained by concatenating k randomly chosen hash functions from F. The algorithm then constructs L hash tables, each corresponding to a different randomly chosen hash function g.

In the preprocessing step we hash all n points from the data set S into each of the L hash tables. Given that the resulting hash tables have only n non-zero entries, one can reduce the amount of memory used per each hash table to O(n) using standard hash functions.

Given a query point y, the algorithm iterates over the L hash functions g. For each g considered, it retrieves the data points that are hashed into the same bucket as y. The process is stopped as soon as a point within distance $cR$ from y is found.

**3.3.2 Source Code :**

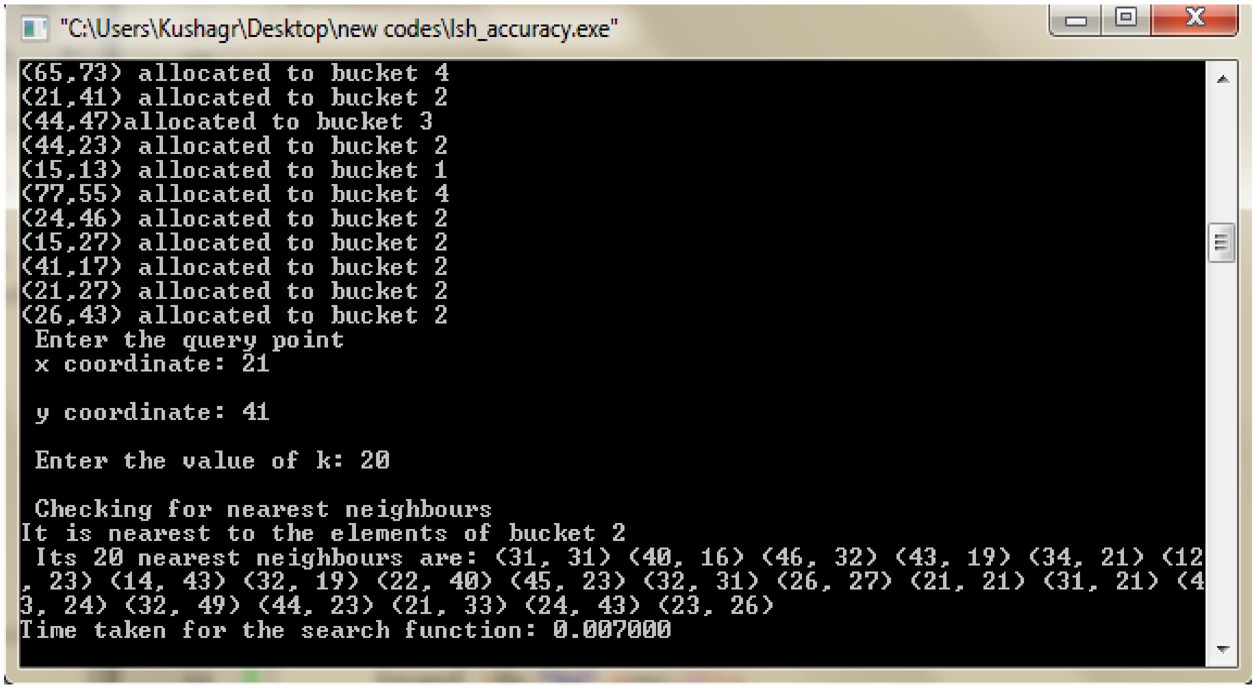
```
#include<stdio.h>
#include<math.h>
#include<time.h>
void read_hash();
void hash2( );
int k,l, m;
float dist1,dist2,dist3;
float b1[100][4],b2[100][4],b3[100][4];
float near_dist,far_dist;
float val[4];
void main()
{
float query_pt[3],radius;
clock_t t;
k=0;
l=0;m=0;
printf("\n Enter the radius: ");
scanf("%f",&radius);
near_dist=(float)radius/2;
b1[0][0]=5.0;
b1[0][1]=3.0;
b1[0][2]=1.5;
b1[0][3]=0.2;
b2[0][0]=6.0;
b2[0][1]=3.0;
b2[0][2]=4.0;
b2[0][3]=1.5;
b3[0][0]=7.0;
b3[0][1]=2.8;
b3[0][2]=5.3;
```

```
b3[0][3]=2.0;
printf("\nReading random points to be filled in buckets");
read_hash();
hash2();
getch();
}
void read_hash()  /* reads the training data and distributes it in various buckets */
{
int i=0;
float val[4];
char ch;
FILE *fp;
fp=fopen("testing_20.txt","r");
if(fp==NULL)
   printf("File does not exist");
else{
      while (!feof (fp))
   {
     fscanf (fp, "%f", &val[0]);
     fscanf (fp, "%f", &val[1]);
     fscanf (fp, "%f", &val[2]);
     fscanf (fp, "%f", &val[3]);
  dist1=sqrt(pow((b1[0][0]-val[0]),2)+pow((b1[0][1]-val[1]),2)+pow((b1[0][2]-
val[2]),2)+pow((b1[0][3]-val[3]),2));
  dist2=sqrt(pow((b2[0][0]-val[0]),2)+pow((b2[0][1]-val[1]),2)+pow((b2[0][2]-
val[2]),2)+pow((b2[0][3]-val[3]),2));
  dist3=sqrt(pow((b3[0][0]-val[0]),2)+pow((b3[0][1]-val[1]),2)+pow((b3[0][2]-
val[2]),2)+pow((b3[0][3]-val[3]),2));
  if(dist1<dist2 && dist1<dist3 && dist1<near_dist)
   {
   b1[++k][0]=val[0];
```

```
    b1[k][1]=val[1];
    b1[k][2]=val[2];
    b1[k][3]=val[3];
    printf("\n(%1f,%1f,%1f,%1f) allocated to bucket 1\n",val[0],val[1],val[2],val[3]);
   }
  else if(dist2<dist1 && dist2<dist3  && dist2<near_dist)
   {
   b2[++l][0]=val[0];
   b2[l][1]=val[1];
   b2[l][2]=val[2];
   b2[l][3]=val[3];
   printf("\n(%1f,%1f,%1f,%1f) allocated to bucket 2\n",val[0],val[1],val[2],val[3]);
   }
  else if(dist3<dist1 && dist3<dist2 && dist3<near_dist )
   {
   b3[++m][0]=val[0];
   b3[m][1]=val[1];
   b3[m][2]=val[2];
   b3[m][3]=val[3];
   printf("\n(%1f,%1f,%1f,%1f)allocated to bucket 3\n",val[0],val[1],val[2],val[3]);
   }
  }
 }
fclose(fp);
}

void hash2()  /* allocate the class to the query point and find its nearest neighbors*/
{
int i,s;
float val[3];
FILE *ft;
```

```
double time_taken;
clock_t t;
ft=fopen("testing_30.txt","r");
s=0;
while(!feof(ft))
{
    fscanf(ft,"%f",&val[0]);
    fscanf(ft,"%f",&val[1]);
    fscanf(ft,"%f",&val[2]);
    fscanf(ft,"%f",&val[3]);
  dist1=sqrt(pow((b1[0][0]-val[0]),2)+pow((b1[0][1]-val[1]),2)+pow((b1[0][2]-
val[2]),2)+pow((b1[0][3]-val[3]),2));
  dist2=sqrt(pow((b2[0][0]-val[0]),2)+pow((b2[0][1]-val[1]),2)+pow((b2[0][2]-
val[2]),2)+pow((b2[0][3]-val[3]),2));
  dist3=sqrt(pow((b3[0][0]-val[0]),2)+pow((b3[0][1]-val[1]),2)+pow((b3[0][2]-
val[2]),2)+pow((b3[0][3]-val[3]),2));

t=clock();
check: printf("\n%d. ",++s);
   if(dist1<dist2 && dist1<dist3 && dist1<=near_dist)
   {
   b1[++k][0]=val[0];
   b1[k][1]=val[1];
   b1[k][2]=val[2];
   b1[k][3]=val[3];
   printf("It is nearest to the elements of bucket 1");
    }
   else if(dist2<dist1 && dist2<dist3 && dist2<=near_dist)
   {
   b2[++l][0]=val[0];
   b2[l][1]=val[1];
   b2[l][2]=val[2];
   b2[l][3]=val[3];
   printf("It is nearest to the elements of bucket 2");
   }
   else if(dist3<dist1 && dist3<dist2 && dist3<=near_dist)
  {
    b3[++m][0]=val[0];
    b3[m][1]=val[1];
    b3[m][2]=val[2];
    b3[m][3]=val[3];
    printf("It is nearest to the elements of bucket 3");
    }
   else
   {
   dist1=sqrt(pow((b1[5][0]-val[0]),2)+pow((b1[5][1]-val[1]),2)+pow((b1[5][2]-
val[2]),2)+pow((b1[5][3]-val[3]),2));
   dist2=sqrt(pow((b2[5][0]-val[0]),2)+pow((b2[5][1]-val[1]),2)+pow((b2[5][2]-
val[2]),2)+pow((b2[5][3]-val[3]),2));
   dist3=sqrt(pow((b3[5][0]-val[0]),2)+pow((b3[5][1]-val[1]),2)+pow((b3[5][2]-
val[2]),2)+pow((b3[5][3]-val[3]),2));
   goto check;
   }
}
   t=clock()-t;
time_taken=(double)(t)/CLOCKS_PER_SEC;
printf("\nTime taken for the search function: %lf",time_taken);
//printf("\n\n\n\nFinding the actual nearest neighbours of the query point");
//actual_nearest_neighbours(val,s);
}
```

**OUTPUT:**



"C:\Users\Kushagr\Desktop\new codes\lsh_accuracy.exe"

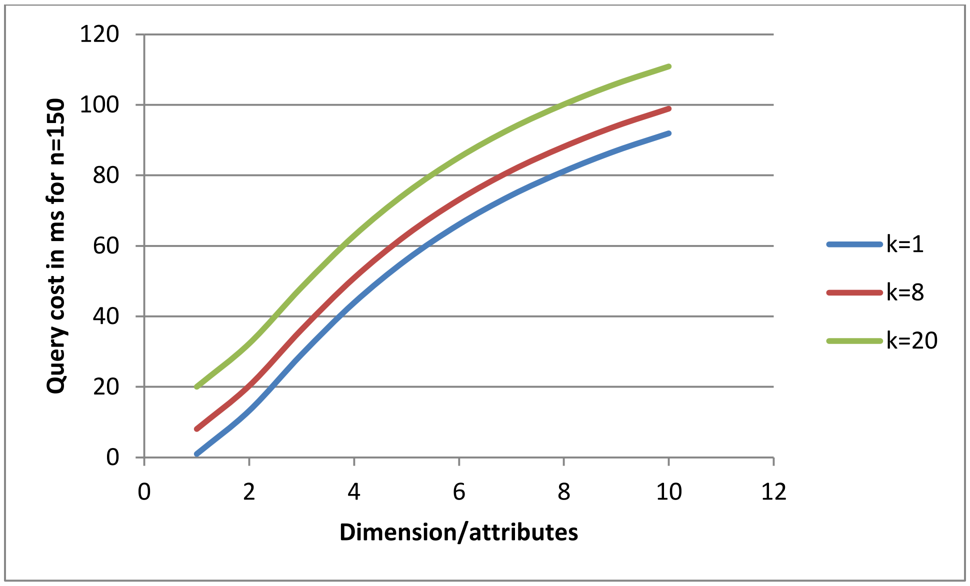
```
(65,73) allocated to bucket 4
(21,41) allocated to bucket 2
(44,47)allocated to bucket 3
(44,23) allocated to bucket 2
(15,13) allocated to bucket 1
(77,55) allocated to bucket 4
(24,46) allocated to bucket 2
(15,27) allocated to bucket 2
(41,17) allocated to bucket 2
(21,27) allocated to bucket 2
(26,43) allocated to bucket 2
 Enter the query point
 x coordinate: 21

 y coordinate: 41

 Enter the value of k: 20

 Checking for nearest neighbours
It is nearest to the elements of bucket 2
 Its 20 nearest neighbours are: (31, 31) (40, 16) (46, 32) (43, 19) (34, 21) (12
, 23) (14, 43) (32, 19) (22, 40) (45, 23) (32, 31) (26, 27) (21, 21) (31, 21) (4
3, 24) (32, 49) (44, 23) (21, 33) (24, 43) (23, 26)
Time taken for the search function: 0.007000
```

**Figure 3.5**

# Chapter 4: RESULTS AND GRAPHS

This chapter includes the comparison between two algorithms of approximate nearest neighbor, one is K-nearest neighbor and another one is K-nearest neighbor using Kd, on the basis of time taken, accuracy and dependence on dimension.

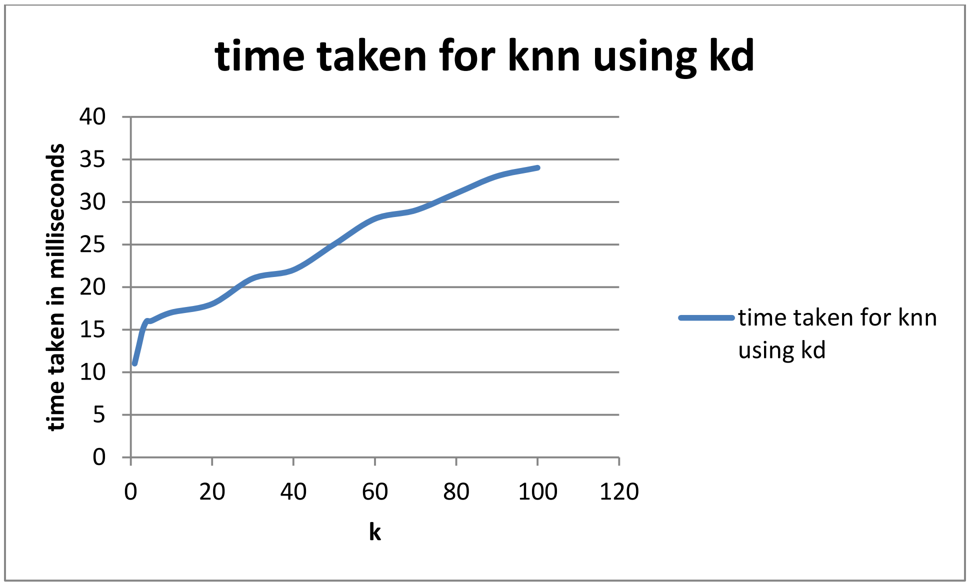
## Query cost VS Dimension for KNN using KD:



The general query cost or time taken by Kd for finding out the neighbors of a given query point is being represented by O(n^(1-1/d)+k) where, n is the number of testing points, d is the dimension or the number of attributes and k is the number of nearest neighbors we want to find out. On putting various values of k and d and keeping n=150, the above graph is plotted. Suppose, d=2 and k=10 then, it is clear that the time taken will be polynomial in nature. This graph is showing that the time taken by Kd is polynomial time.

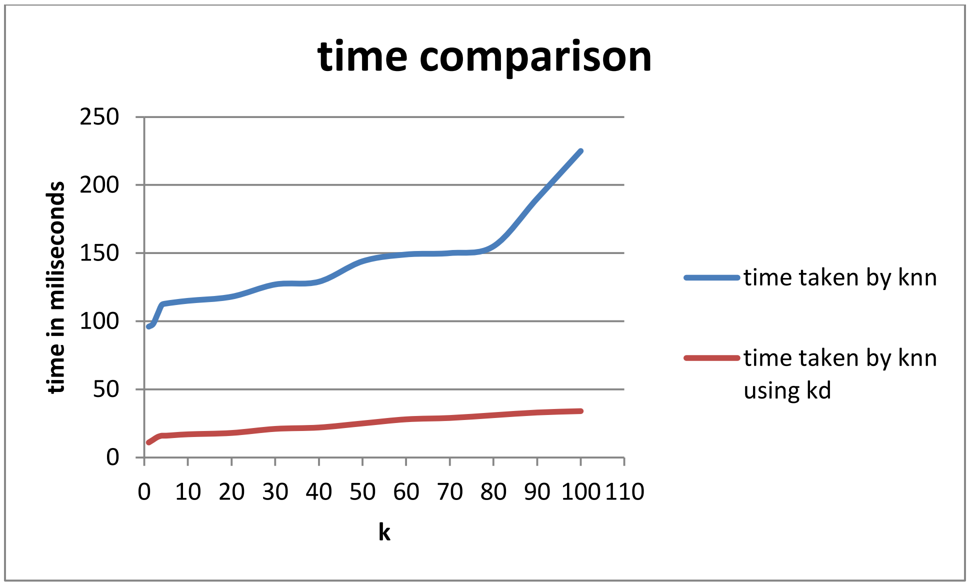
## 4.1 Iris Dataset:

### 4.1.1 Time taken by KNN using Kd:



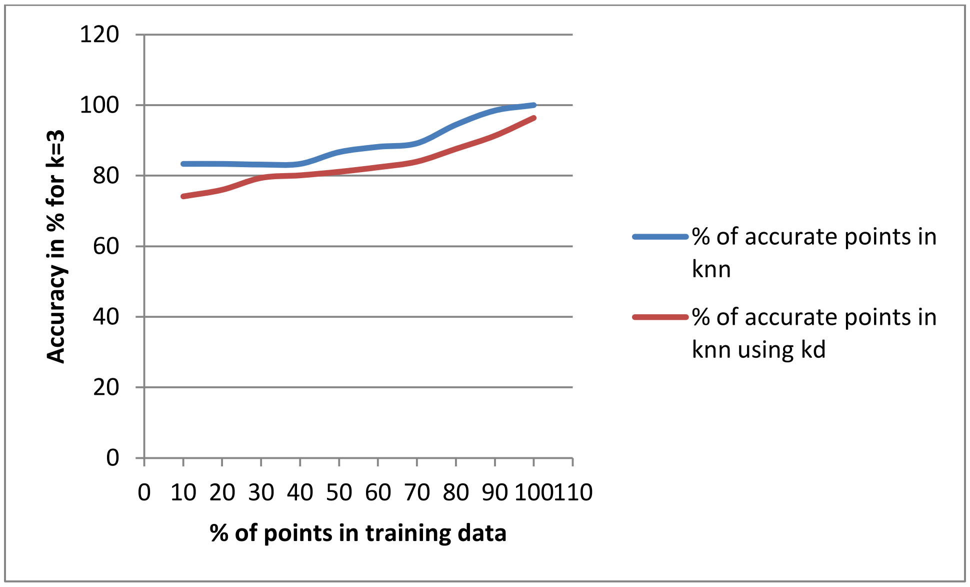
Above graph is showing the time taken by kd with different values of k, no. of attributes are four and the number of data points are twenty. By the graph we can see that it is polynomial in nature.

### 4.1.2 Time comparison:



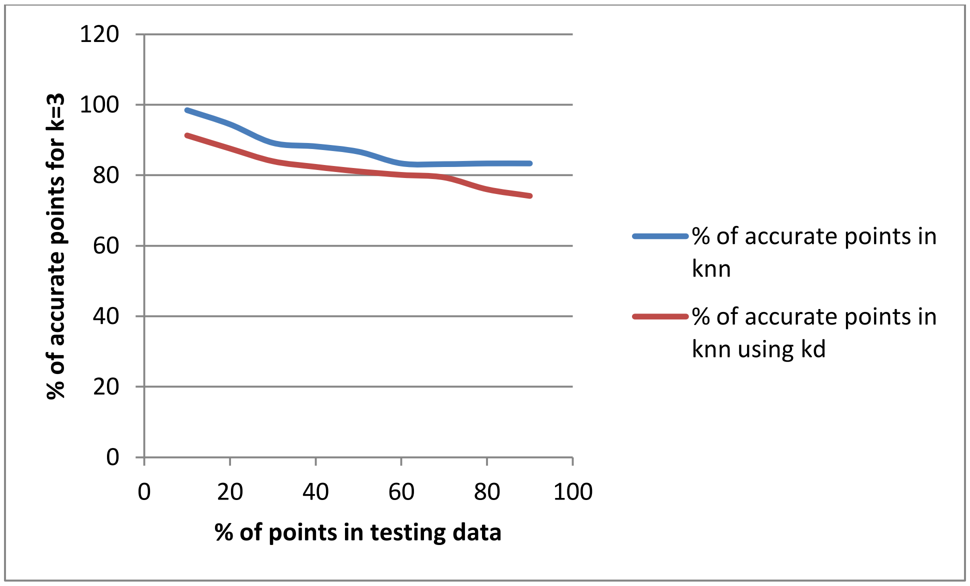
This graph is showing the time comparison between the KNN and KNN using KD, as we can see that time taken by KNN is quite large in comparison to the time taken by KD.

### 4.1.3 Accuracy Graph with variation in training data:



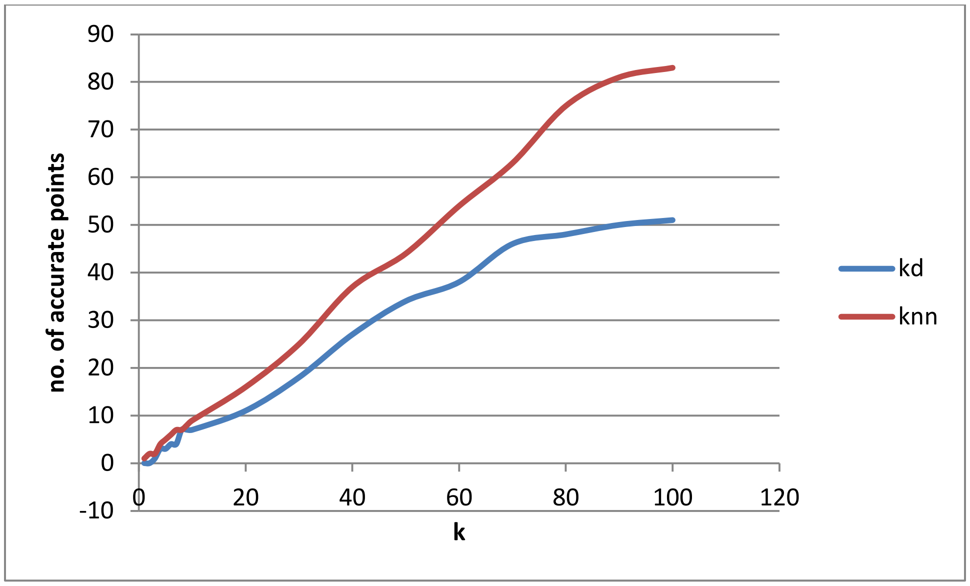
Above graph is plotted by varying the size of training data and as the size of training data is increasing obviously the accuracy will increase because the number of points in testing data will decrease and the training data is increasing.

### 4.1.4 Accuracy graph with variation in training data:



This graph is showing the variation of accuracy with the variation in the size of the testing data and as the testing data is increasing the amount of training data is decreasing and hence it is quite obvious that the accuracy will decrease with increase in the size of testing data.

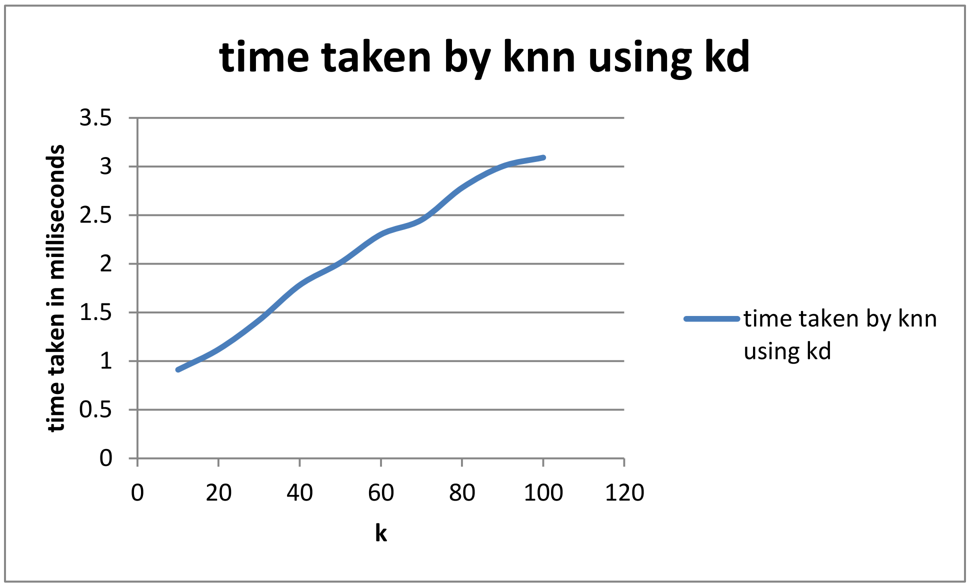
### 4.1.5 Accuracy graph with variation in k:



This graph is plotted by keeping the amount of training and testing data as constant and varying the value of k i.e. the number of nearest neighbors to be found out.

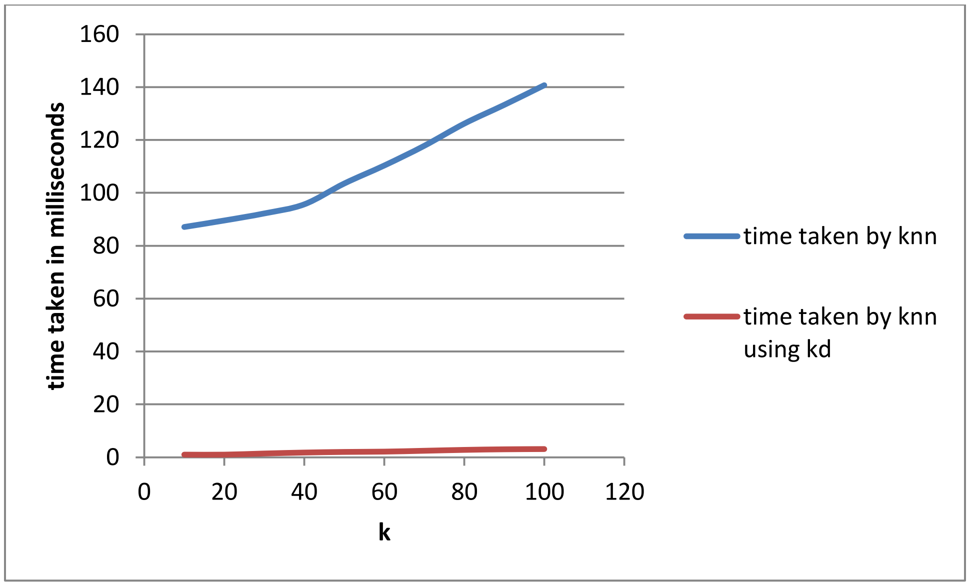
## 4.2 Letter Dataset:

### 4.2.1 Time taken by KNN using Kd:



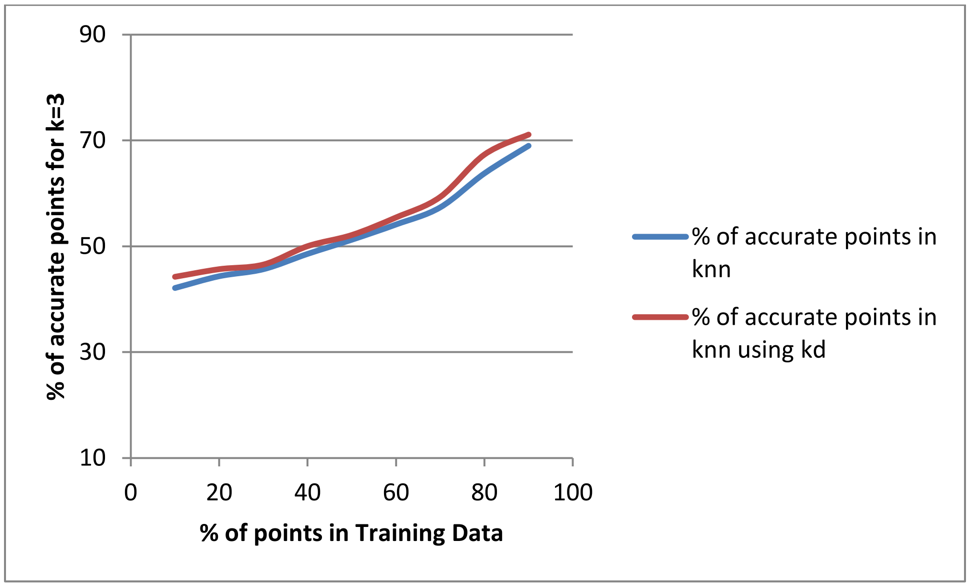
Above graph is showing the time taken by KD with different values of k, no. of attributes are sixteen and the number of data points are fifteen. By the graph we can see that it is polynomial in nature.

### 4.2.2 Time comparison:



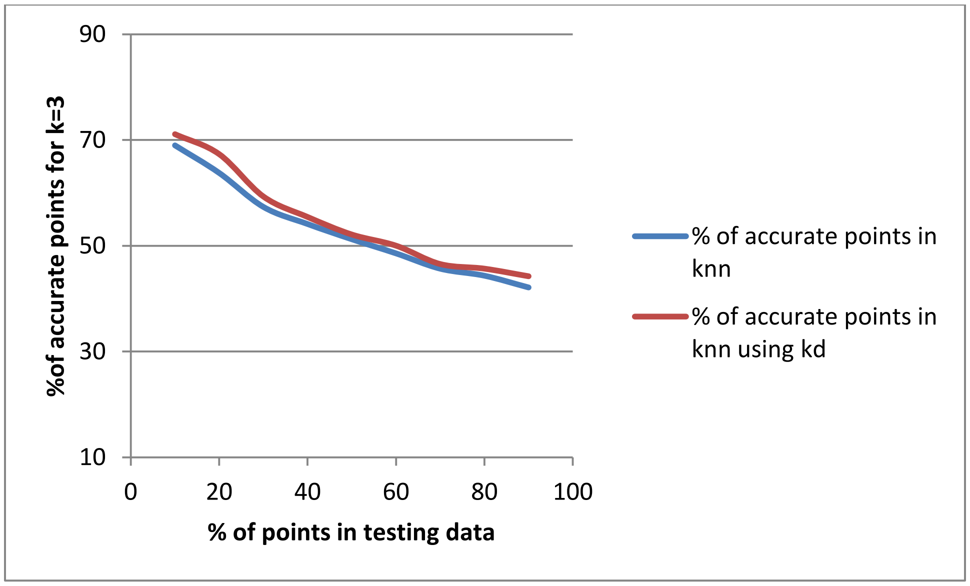
This graph is showing the time comparison between the KNN and KNN using KD, as we can see that time taken by KNN is quite large in comparison to the time taken by KD.

### 4.1.3 Accuracy Graph with variation in training data:



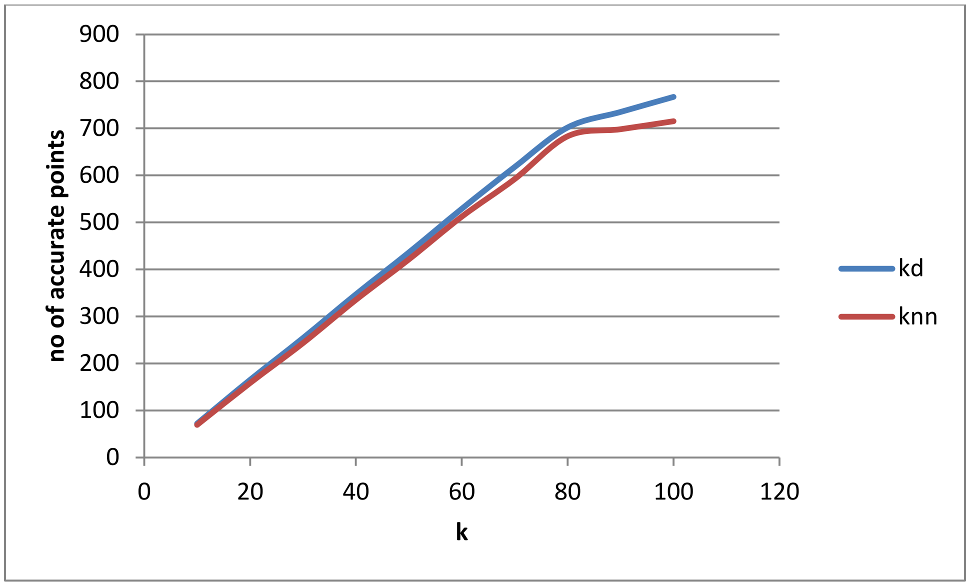
Above graph is plotted by varying the size of training data and as the size of training data is increasing obviously the accuracy will increase because the number of points in testing data will decrease and the training data is increasing.

### 4.1.4 Accuracy graph with variation in training data:



This graph is showing the variation of accuracy with the variation in the size of the testing data and as the testing data is increasing the amount of training data is decreasing and hence it is quite obvious that the accuracy will decrease with increase in the size of testing data.

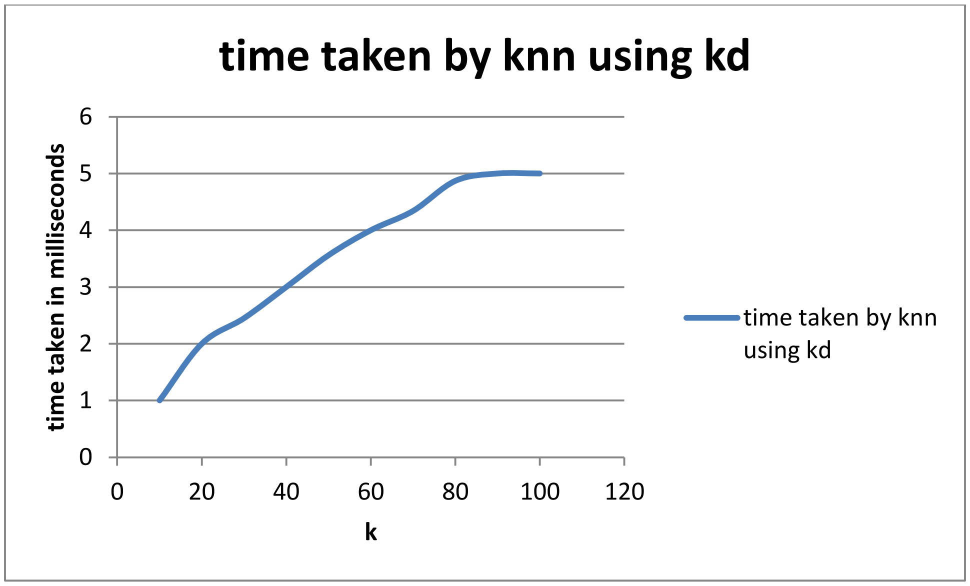
### 4.1.5 Accuracy graph with variation in k:



This graph is plotted by keeping the amount of training and testing data as constant and varying the value of k i.e. the number of nearest neighbors to be found out and the no. of testing points are ten.

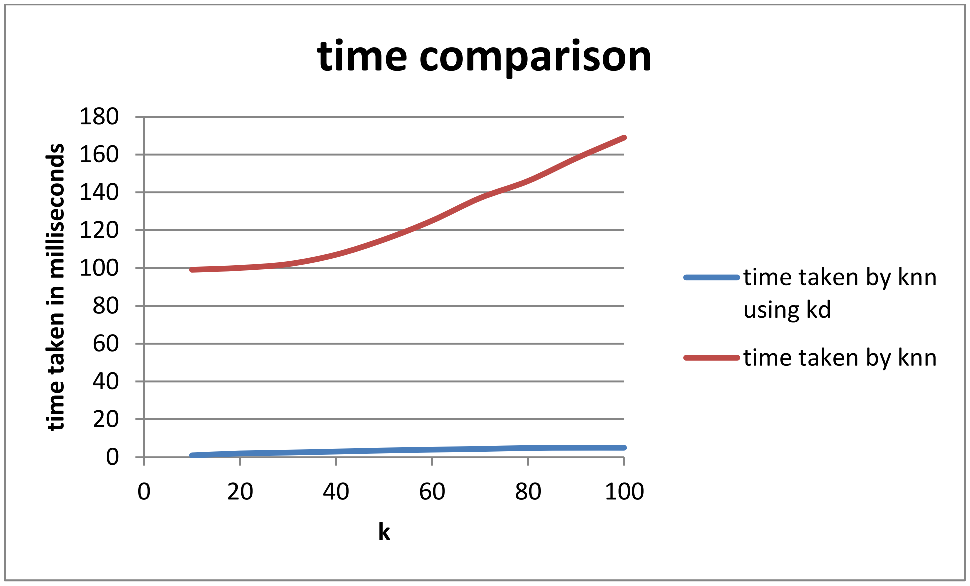
## 4.3 Chess Dataset:

### 4.1.1 Time taken by KNN using Kd:



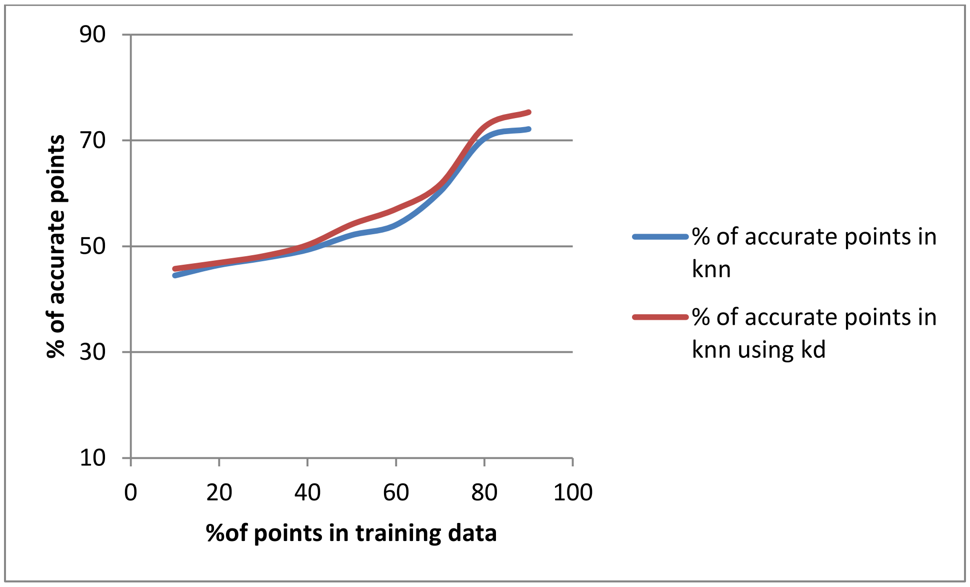
Above graph is showing the time taken by kd with different values of k, no. of attributes are six and the number of data points are fifteen. By the graph we can see that it is polynomial in nature.

### 4.1.2 Time comparison:



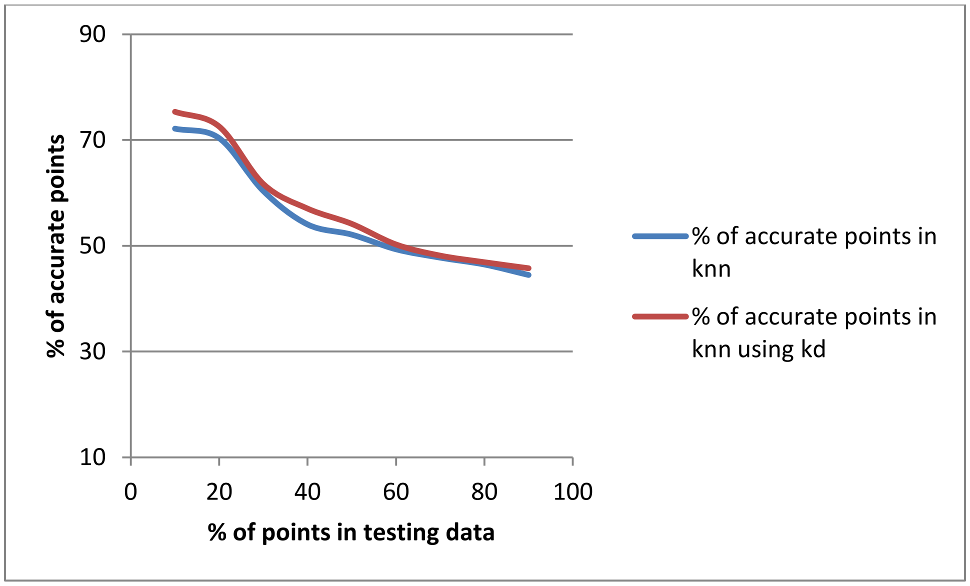
This graph is showing the time comparison between the KNN and KNN using KD, as we can see that time taken by KNN is quite large in comparison to the time taken by KD.

### 4.1.3 Accuracy Graph with variation in training data:



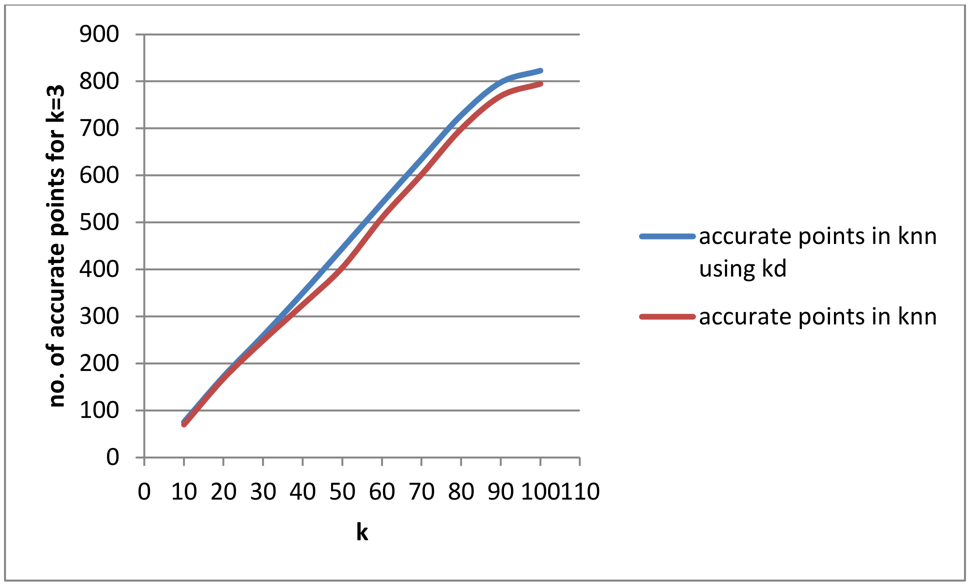
Above graph is plotted by varying the size of training data and as the size of training data is increasing obviously the accuracy will increase because the number of points in testing data will decrease and the training data is increasing.

### 4.1.4 Accuracy graph with variation in training data:



This graph is showing the variation of accuracy with the variation in the size of the testing data and as the testing data is increasing the amount of training data is decreasing and hence it is quite obvious that the accuracy will decrease with increase in the size of testing data.

### 4.1.5 Accuracy graph with variation in k:



This graph is plotted by keeping the amount of training and testing data as constant and varying the value of k i.e. the number of nearest neighbors to be found out and no. of testing points are ten.

# Chapter 5: CONCLUSION

Chapter 4 is showing all the results, graphs and comparisons between the two algorithms i.e. KNN and KNN using KD.

As we have seen that for all the three datasets the time taken by KNN is very large in comparison to the time taken by KNN using KD. This shows that KNN using KD is very much efficient in the case of time taken.

Iris data set is a small dataset in which we have seen that the KNN is giving slightly better performance than KNN using KD. But, for bigger and quite random datasets like Letter and the Chess Dataset, the KNN using KD is giving better performance or accuracy than actual KNN. And, the time taken by KNN using KD is also very-very less in comparison with the time taken by KNN.

So, from the above results we can conclude that for bigger and random data sets the approximate nearest neighbor algorithms can give much better results.

Like in iris data set, with slight compromise in the performance or accuracy, the time performance is very high i.e. in very less amount of time we can find out the solution for given query point.

Generally, the approximate nearest neighbors are little less accurate than the exact ones but in some cases like in letter and chess datasets, where the size of datasets is very-very large and the data is quite randomly allocated to different classes, the approximate nearest neighbor algorithms can give better results in case of both time and accuracy.

So, for large volume of somewhat random datasets, it is better to use the approximation algorithms so that we can find the required results accurately and in efficient amount of time.

# Chapter 6: REFERENCES

- ✓ Introduction to approximate nearest neighbor algorithms by Gregory Shakhnarovich, Piotr Indyk, and Trevor Darrell.
- ✓ Approximate nearest neighbor queries in fixed dimension by Sunil Arya and David M. Mount.
- ✓ An optimal algorithm for Approximate nearest neighbor searching in fixed dimensions by Sunil Arya, David M. Mount, Nathan S. Nethanyahu, Ruth Sh Silverman and Angela Y. WU.
- ✓ Are you using the right approximate nearest neighbor algorithm by Stephen O' Hara and Bruce A. Draper, Coloroda State University.
- ✓ An introductory tutorial on KD Tree by Andrew W. Moore, Carnegie Mellon University.
- ✓ Range Searching using KD Tree by Hemant M. Kakde.
- ✓ https://archive.ics.uci.edu/ml/datasets.html
- ✓ Fast approximate nearest neighbor with automatic algorithm algorithm configuration by Marius Mauja, David G. Lowe.
- ✓ Real Time registration – Based Tracking via approximate nearest neighbor search by Travis Dick,  Camilo Perez, Azad Shademan, Martin Jagersand, University of Alberta.
- ✓ CS 9633 Machine learning, K-nearest neighbor algorithms by Tom Mitchell
- ✓ Near-optimal hashing algorithms for Approximate Nearest Neighbor in high dimension by Alexandr Andoni and Piotr Indyk.
- ✓ S. Arya and D. Mount. ANN: Library for approximate nearest neighbor searching. http://www.cs.umd.edu/~mount/ANN
- ✓ Nearest Neighbor Search: the Old, the New, and the Impossible by Alexandr Andoni.
- ✓ S. M. Omohundro. KD tree construction algorithms. Technical Report TR-89-063, International Computer Science Institute, Berkeley, CA, December 1989.