

**APPLICATION OF APPROXIMATE NEAREST
NEIGHBOUR SEARCH ALGORITHM: LOCALITY
SENSITIVE HASHING**

Enrolment no.: 101226

Name of student: Shalagha Mahajan

Name of Supervisor: Mr. Suman Saha



May 2014

Submitted in partial fulfillment of the Degree of Bachelor of
Technology

DEPARTMENT OF COMPUTER SCIENCE

JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY,

WAKNAGHAT

TABLE OF CONTENTS

Chapter No.	Topics	Page No.
	Certificate from the Supervisor	II
	Acknowledgement	III
	Summary	IV
	List of Figures	V
Chapter-1	Introduction	1-6
Chapter-2	Literature Review	7-19
Chapter-3	Algorithms and Codes	20-39
Chapter-4	Graphs and Results	40-50
Chapter-5	Conclusion	51
Chapter-6	References	52

CERTIFICATE

This is to certify that the work titled **APPLICATIONS OF APPROXIMATE NEAREST NEIGHBOUR SEARCH ALGORITHMS: Locality Sensitive Hashing** submitted by **Shalagha Mahajan** in partial fulfillment for the award of degree of Bachelor of 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: Mr. Suman Saha

Designation: Assistant Professor

Date: 15th May, 2014

ACKNOWLEDGEMENT

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

I, Shalagha Mahajan, the student of **Jaypee University of Information Technology(CSE)** am extremely grateful to institute for the confidence bestowed in me and supporting me through my project entitled **Applications of Approximate Nearest Neighbours Search Algorithms** .

At this point in time I feel deeply honored in expressing my sincere thanks to **Mr. Suman Saha** for making the resources available at right time and providing valuable insights leading to the successful completion of my project.

I would also like to thank all the faculty members of **JUIT** for their critical advice and guidance without which this project would not have been possible.

Also, I place a deep sense of gratitude to my family members and my friends who have been constant source of inspiration during the preparation of this project work.

Name of Student : Shalagha Mahajan

Date : 15th May, 2014

SUMMARY

In computer science, approximation algorithms are algorithms used to find approximate solutions to optimization problems. Exact nearest neighbour search algorithms return accurate results but with the dimensions the complexity, both time and space increase exponentially.

Thus striking a tradeoff between complexity and accuracy, we shift to approximate nearest neighbour search algorithms wherein we cut down on the complexity and in turn compromise on the accuracy. In this year long project, various approximate nearest neighbour search algorithms were explored, the prominent ones being implemented as well.

Thereafter, the algorithms so implemented were compared with each other on various grounds like time complexity and dependence on dimension by running the algorithms for various datasets.

Name: Shalagha Mahajan

Date: 15th May, 2014

Name: Mr. Suman Saha

Date: 15th May, 2014

LIST OF FIGURES

Figure no.	Label	Page no.
1.1.a	Abundance of Data	1
1.2.a	Parallelization of Data	3
1.2.b	Nearest Neighbour Search	4
1.3.a	Complexity of NN Algorithm	5
2.1.a	KD Trees	7
2.1.b	Locality Sensitive Hashing	8
2.2 a	Implementing LSH	10
2.3 a	KNN example	11
2.4 a	Implementation of kD trees	14
2.5 a	Projection method in place of LSH	15

CHAPTER 1: INTRODUCTION

1.1 Abstract:

Over the last few decades, immense amount of data has become available. From collections of photos and genetic data, to network traffic statistics, modern technologies and cheap storage have made it possible to accumulate huge datasets. The ever growing sizes of the datasets make it imperative, very important to design new algorithms capable of parsing through this data with extreme efficiency.



Figure 1.1.a: Abundance of Data

The challenges faced due to this plethora of data include capture, curation, storage, search, sharing, transfer, analysis, and visualization. The trend towards larger data sets is due to the additional information derivable from analysis of a single large set of related data. As of now, limits on the size of data sets that are feasible to process in a reasonable amount of time are on the order of terabytes of data.

Scientists regularly encounter a variety of issues due to large size of data sets in many areas, including meteorology, complex physics simulations, and biological and environmental

research. The limitations also have an effect on Internet search, finance and business informatics. Data sets grow in size in part because they are increasingly being accumulated by ubiquitous information-sensing mobile devices, aerial sensory technologies (remote sensing), software logs, cameras, microphones, radio-frequency identification readers, and wireless sensor networks. The world's technological capacity to store information has roughly doubled every 40 months since the 1980s; as of 2012, every day 2.5 terabytes (2.5×10^{18}) of data is being created. The challenge for large enterprises is that of determining who should own big data initiatives that straddle the entire organization.

Big data is difficult to work with because it requires using most relational database management systems and desktop statistics and visualization packages, easily handled if uses instead "massively parallel software running on tens, hundreds, or even thousands of servers".

1.2 Algorithm Engineering for searching in Large Data Sets:

1.2.1 Parallelization:

Large sets of data provide a challenge in time consumption while solving the cluster identification problem and this becomes the reason why a parallel algorithm is so needed for identifying dense clusters in a noisy background. This algorithm works on a graph representation of the data set to be analyzed. It identifies clusters by the identification of densely intra connected sub graphs. We employ a minimum spanning tree (MST) representation of the graph and solve the cluster identification problem using this representation. The computational bottleneck of this algorithm of parallelization is the construction of an MST of a graph, for which a parallel algorithm is employed. A complex strategy for the parallel MST construction algorithm is to first partition the graph, then construct MSTs for the partitioned sub graphs and auxiliary bipartite graphs based on the sub graphs, and finally merge these MSTs to derive an MST of the original graph. The computational results indicate that when running on 150 to 200 CPUs, a parallel algorithm can solve a cluster identification problem on a data set with 1,000,000 data points almost 100 times faster than on single CPU, which is excellent performance, indicating that this program is capable of handling very large data clustering problems in an efficient manner.

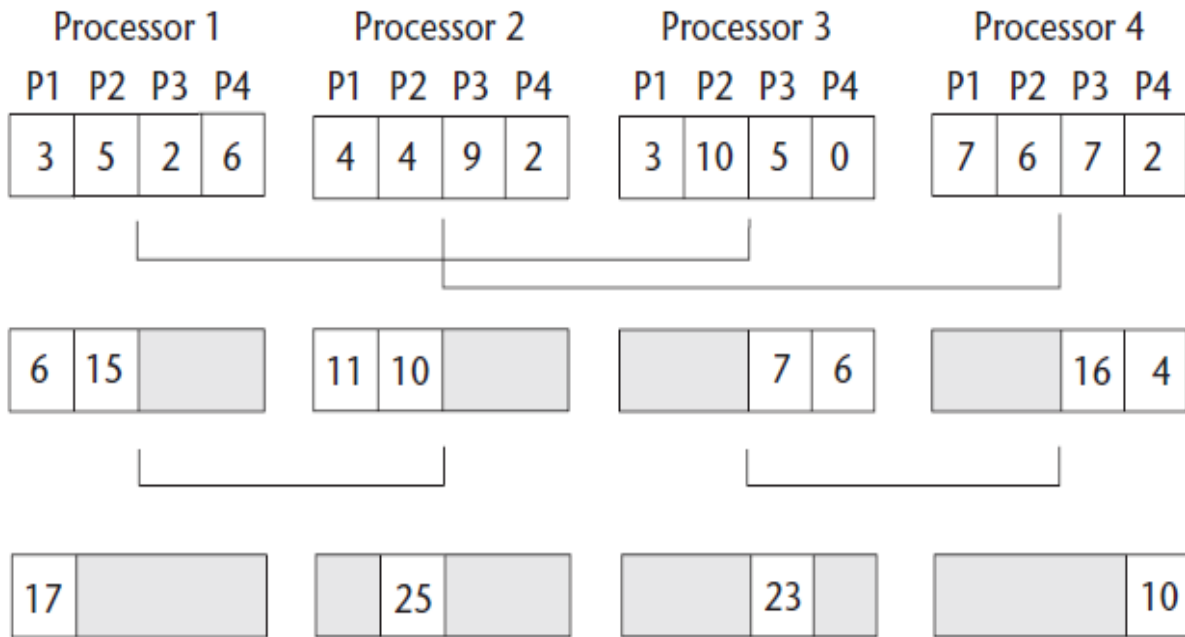


Figure 1.2.a :Parallelization of Data

The key effort in designing the parallel algorithm goes into minimizing the communication efforts among processors.

1.2.2 Nearest Neighbour Searches:

A fundamental computational primitive for dealing with massive datasets is the Nearest Neighbour (NN) problem. The nearest-neighbour (NN) problem comes under many names, including the best match or the post office problem. This problem is of significant importance to several areas of computer science, including pattern recognition, searching in multimedia data, vector compression, computational statistics, and data mining.

In the NN problem, the goal is to preprocess a set of data called training data, so that later, given a query object, one can find efficiently the data item most similar to the query. To represent the

objects and the similarity measures, one often uses geometric notions or Euclidian measures. For example, a black-and-white image may be modeled by a high-dimensional vector, with one coordinate per pixel, whereas the similarity measure may be the standard Euclidean distance between the resulting vectors.

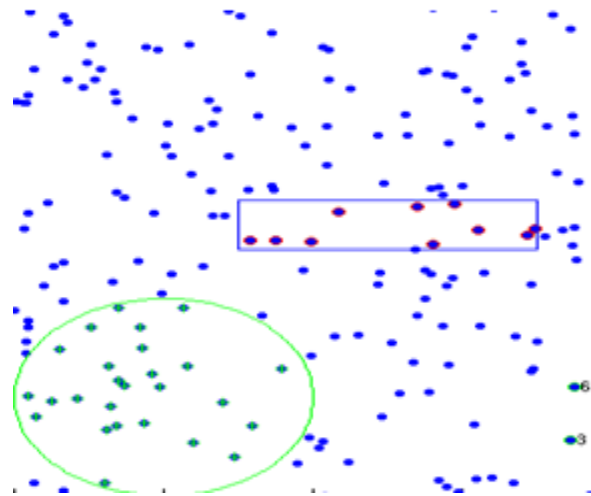


Figure 1.2 b: Nearest neighbour search

For most of today's applications, large amounts of data are available. This makes nearest-neighbour approaches particularly attractive, but it also increases the concern regarding the computational complexity of NN search. Thus it is important to design algorithms not only for nearest-neighbour search, but also for the related classification, regression, and retrieval tasks, which remain efficient even as the number of points or the dimensionality of the data grows large. 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, it covers a wide range.

Hence, in our project we define the exact and approximate nearest-neighbour search problems, and briefly survey a number of popular data structures and algorithms developed for these problems.

1.3 Problem Statement and motivation:

The nearest neighbour problem involves a large amount of data. This makes nearest-neighbour approaches particularly appealing, but on the other 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.

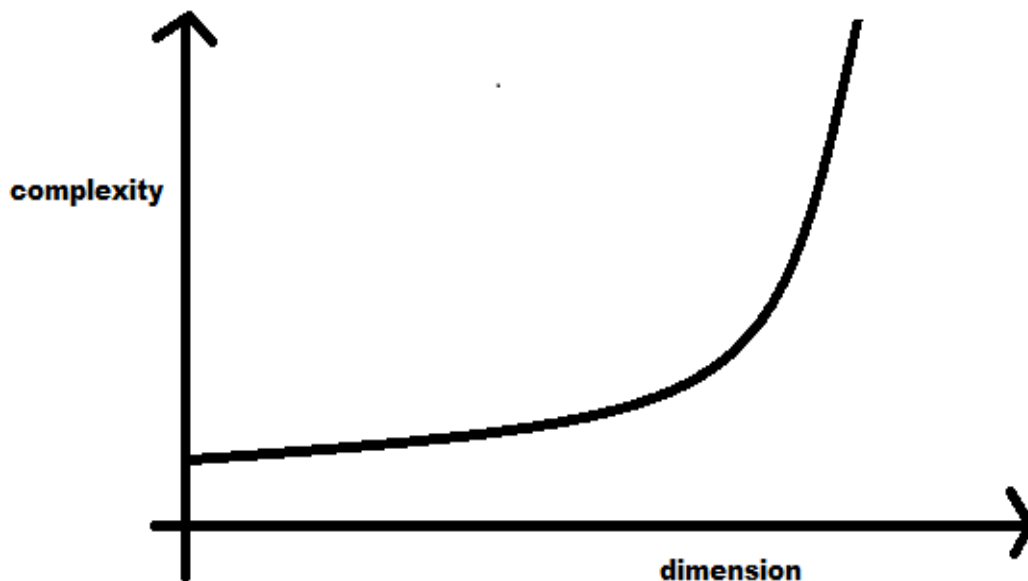


Figure 1.3 a: complexity of NN algorithm v/s the dimension

A basic algorithm for this problem is as follows: given a query q , compute the distance from q to each point in P , and report the point with the minimum distance. This is the *linear scan* approach which has query time of $\Theta(dn)$. This is tolerable for small data sets, but it is too inefficient for large ones. The “holy grail” of the research in this area is to design an algorithm for this problem that achieves *sublinear* (or even *logarithmic*) query time and cuts down on the costs. The nearest-neighbour problem is being extensively investigated in the field of computational geometry. As a result of this effort, many efficient solutions have been discovered for the case when the points lie in a *constant* dimensional space. For example, if the points lie in the plane, the nearest-neighbour problem can be solved with $O(\log n)$ time per query, using only $O(n)$

storage, which again is commendable . Similar outcomes can be obtained for other problems as well. Unfortunately, as the dimension grows, the algorithms become less and lesser efficient. More specifically, their space or time requirements grow *exponentially* with the dimension. In particular, the nearest-neighbour problem has a solution with $O(dO(1) \log n)$ query time, using $nO(d)$ space.

The lack of success in removing the exponential dependence on the dimension, the curse of dimensionality, led many researchers to the conjecture that no efficient solutions exist for this problem when the dimension is sufficiently large. At the same time, it raised the question as to Is it possible to remove the exponential dependence on d , if we allow the answers to be *approximate*?

The answer to this very same question led us to the approximate nearest neighbour problem.

The approximate nearest neighbour problem was first discovered for low dimensional version of the Nearest Neighbour search problem. This algorithm provides large speedups with only minor loss in accuracy, which of course fetches it a thumbs up. During recent years, several researchers have shown that indeed in many cases approximation enables reduction of the dependence on dimension from exponential to polynomial with minimal loss in accuracy. In addition, there are many approximate nearest-neighbour algorithms that are more efficient than the exact ones, even though their query time and/or space usage is still exponential in the dimension, example being K Nearest Neighbours Algorithm.

The approximate nearest neighbour algorithm can be best implemented using data structures like knn, kD trees and locality sensitive hashing.

CHAPTER 2: LITERATURE REVIEW

2.1 Research Paper: Approximation Algorithms for Nearest Neighbour Search by Gregory Shakhnarovich, Piotr Indyk, and Trevor Darrell

Definition 1.1 (Nearest neighbour) 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 .

Definition 1.2 (c -Approximate nearest neighbour) Given a set P of points in a d -dimensional space $R(d)$, construct a data structure which given 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 .

Kd-Trees

The kd-tree is a data structure invented by Jon Bentley in 1975. Despite the fact that it is fairly old, kd-tree and its variants remain probably the most popular data structures used for searching in multidimensional spaces, at least in main memory.

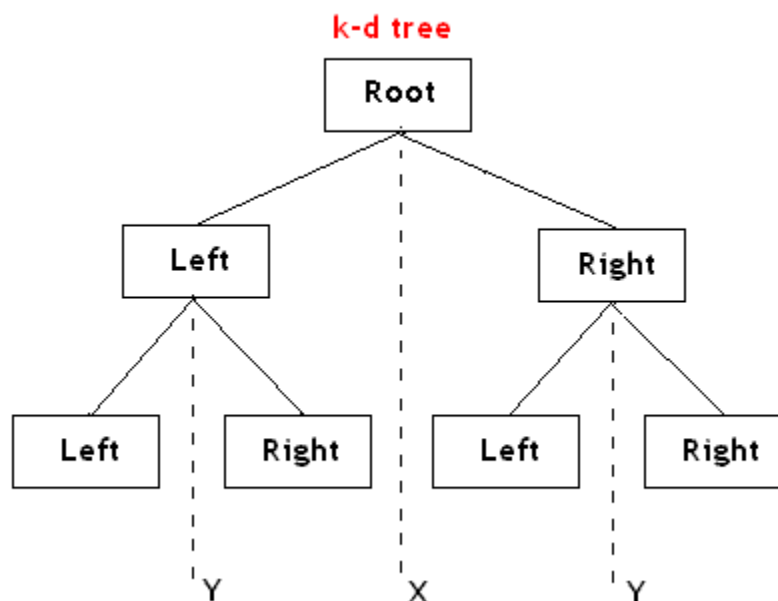


Figure 2.1. a:KD Trees

Given a set of n points in a d -dimensional space, the kd-tree is constructed recursively. Firstly, 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 i th coordinate smaller than or equal to M . The value of x is stored, and the set P is partitioned into $P(left)$ and $P(right)$, where $P(left)$ contains only the points with their i th coordinate smaller than or equal to M , and $|P(right)| = |P(left)| \pm 1$. The process is then repeated recursively on both $P(left)$ and $P(right)$, with i replaced by $i + 1$ (or 1, if $i = d$). When the set of points at a node has reached the size 1, the recursion stops.

Locality-Sensitive Hashing (LSH):

LSH is a *randomized algorithm*. A point p is an R -near neighbour of q if the distance from p to q is at most R .

Definition (Locality-sensitive hashing) A family H is called $(r, cr, P1, P2)$ -sensitive if for any $p, q \in \mathbb{R}^d$

- if $\|p - q\| \leq R$ then $\Pr[h(q) = h(p)] \geq P1$,
- if $\|p - q\| \geq cr$ then $\Pr[h(q) = h(p)] \leq P2$.

In order for a LSH family to be useful, it has to satisfy $P1 > P2$. Given a family H of hash functions with parameters $(r, cr, P1, P2)$ as in the above definition, we can amplify the gap between the “high” probability $P1$ and the “low” probability $P2$ by concatenating several hash functions.

During the pre processing of the available elements, we store each $p \in P$ (input point set) in the bucket $g_j(p)$, for $j = 1, \dots, L$.

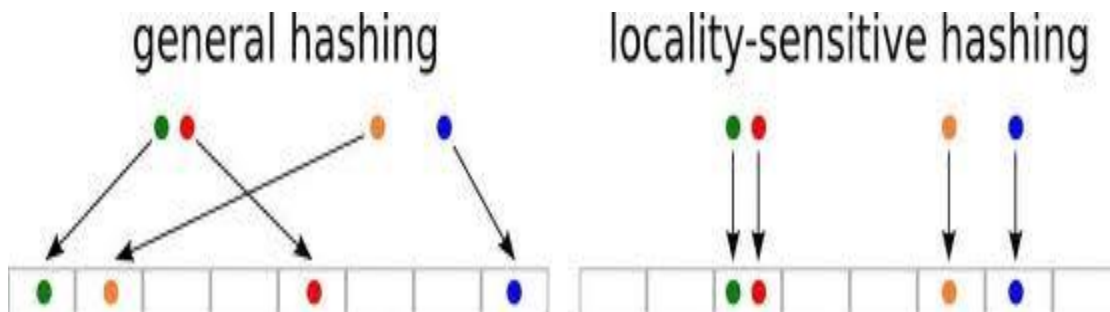


Figure 2.1 b: Locality Sensitive Hashing

The unifying theme of the sections in the paper is exploring the ways to make the nearest-neighbour approach practicable in machine learning application where the dimensionality of the data, and the size of the data sets, make the naive methods for nearest-neighbour search prohibitively expensive. Also it has been noticed that in nearest-neighbour classification, what matters for the decision is the distribution of labels of the query's neighbours, rather than the neighbours themselves. They use this observation to develop new algorithms, using data structures like Kd trees and Locality Sensitive Hash tables, that afford a significant speedup while maintaining the accuracy of the exact k -nearest-neighbour classification.

2.2 . Approximate Nearest Neighbour Problem in High Dimensions by Alexandr Andoni

The nearest neighbour problem is defined as : given a set S of n points in R dimensional space, construct a data structure that, given any $q \in R^m$, quickly finds the point $p \in S$ that has the smallest distance to q . This problem is the central problem in computational geometry. Since the exact problem is surprisingly difficult, recent research has focused on designing efficient approximation algorithms. Furthermore, the approximate nearest neighbour is reducible to the approximate R -near neighbour, and, therefore, primarily concentrate our attention on the problem.

In the approximate R -near neighbour problem, the data structure needs to report a point within distance $c \cdot R$ from q for some constant $c > 1$, but only if there exists a point at distance R from q .

The approximate near and nearest neighbour problems are being studied for a long time. The approximate nearest neighbour algorithms were first discovered for the “low-dimensional” version of the problem, where m is constant . Only after a few results were obtained for the “high-dimensional” case, where m is a parameter, the approximate nearest neighbour came to limelight. In particular, the Locality-Sensitive Hashing (LSH) algorithm solves the (R, c) -near neighbour problem using $2 \cdot O(mn^{1+1/c})$ pre processing time, $O(mn^{1+1/c})$ space and $O(mn^{1/c})$ query time. By using the dimensionality reduction, the query time and the pre processing time can be reduced. *The LSH algorithm has been successfully used in several applied scenarios, including computational biology.*

Locality-Sensitive Hashing:

Definition:

The (R, c) -near neighbour problem is defined as follows. Given a set S of n points in the metric space (Σ^d, D) , construct a data structure that, for a query point $q \in \Sigma^d$, outputs a point v such that $D(v, q) \leq cR$ if there exists a point v^* such that $D(v^*, q) \leq R$.

Generic locality-sensitive hashing scheme

A family $H = \{h : \Sigma^d \rightarrow U\}$ is called (r_1, r_2, p_1, p_2) -sensitive, if for any $q \in S$:

- If $v \in B(q, r_1)$, then $\Pr[h(q) = h(v)] \geq p_1$;
- If v does not belong to $B(q, r_2)$, then $\Pr[h(q) = h(v)] \leq p_2$.

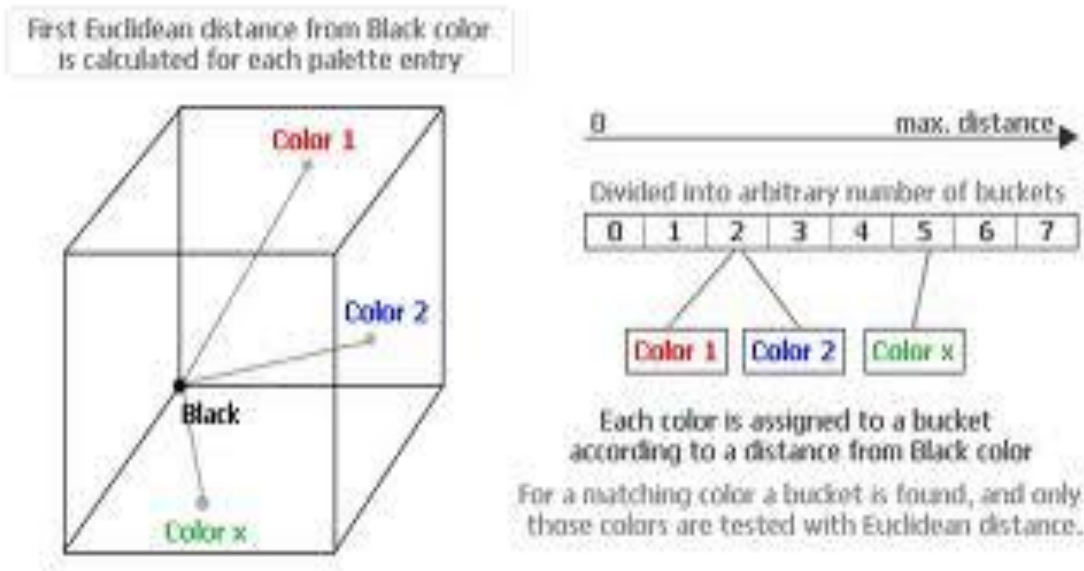


Figure 2.2 a: Implementing LSH

As desired, $r_1 < r_2$ and $p_1 > p_2$; that is, if the query point q is close to v , then q and v should likely fall in the same bucket. Similarly, if q is far from v , then q and v should be less likely to fall in the same bucket. In particular, we choose $r_1 = R$ and $r_2 = cR$.

Since the time for calculating the distance between the query point and the available training data is saved as hash functions suitably allot buckets to the query points, performance in terms of time for LSH outshines any other algorithm's performance.

2.3 k-Nearest Neighbours: Instance Based Learning by Tom Mitchell, Mississippi State University

K Nearest Neighbours Algorithm is an Instance Based Learning Algorithm which along with algorithms like neural networks, Bayesian Classifiers, etc learns and performs:

- Learning: store all the data instances
- Performance:
 - when a new query instance is encountered
 - » retrieve a similar set of related instances from memory
 - » use to classify the new query

KNN algorithm has several roles to play as a:

Training algorithm:

For each training example $\langle x, f(x) \rangle$, add the example
to the list *training_examples*

Classification algorithm:

Given a query instance x_q to be classified

Let $x_1 \dots x_k$ be the k training examples nearest to x_q

Return

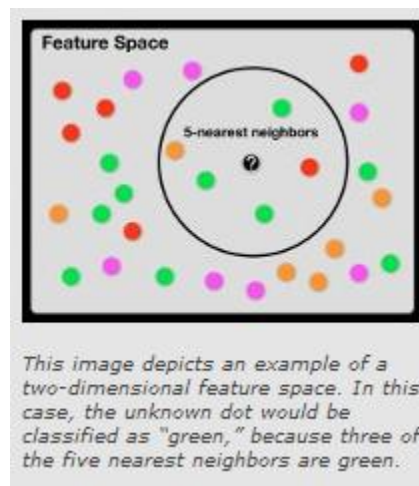


Figure 2.3 a: KNN example

KNN for classification:

KNN is specifically used for classification. In this case, we are given some data points for training and also a new unlabelled 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 neighbour rule

This is the simplest scenario. Let x be the point to be labeled and classified. We need to 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 not just too simplistic but sometimes even counter intuitive. If one feels that this procedure will result a huge error , one could be right.

If the number of data points is very large, then there is a very high chance that label of x and y would be the same. An example – Let us say we have a (potentially) biased coin i.e. when we toss it for 1 million time and you have got head 900,000 times. Then most likely your next call will be head. A similar argument can be used here.

Like the case above, assume all points are in a D dimensional plane . The number of points is reasonably large. This means that the density of the plane at any point is fairly high. Within any subspace there is adequate number of points. Consider a point x in the subspace which also has a lot of neighbors. Now let y be the nearest neighbor. If x and y are sufficiently close, then we can assume that probability that x and y belong to same class is fairly same – Then by decision theory, x and y have the same class.

The book "Pattern Classification" by Duda and Hart has an discussion about the Nearest Neighbor rule. One of their striking results is to obtain a fairly tight error bound to the Nearest Neighbor rule. The bound is

$$P^* \leq P \leq P^* \left(2 - \frac{c}{c-1} P^* \right)$$

Where P^* is the Bayes error rate, c is the number of classes and P is the error rate of Nearest Neighbor. The result is indeed very striking because it says that if the number of points is fairly large then the error rate of Nearest Neighbor is less than twice the Bayes error rate.

Case 2: $k=K$ or k -nearest neighbour Rule

This is a straightforward extension of 1NN. Basically what one does is that he tries to find the k nearest neighbor and do a majority voting. Typically k is odd when the number of classes is 2. Let's say $k = 5$ and there are 3 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.

In nutshell, K Nearest Neighbour Algorithm is a lazy learning algorithm which first assumes all instances are points in n -dimensional space. Then it requires a distance measure to determine the proximity of instances. Thereon, it classifies an instance by finding its nearest neighbors and picking the most popular class among the neighbours.

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 space. Thus, there is no intelligent calculation, which causes the curse of dimensionality to persist.

2.4 . Fast Approximate Nearest Neighbours with Automatic Algorithm Configuration by Marius Muja, David G. Lowe:

The most computationally costly part of computer vision algorithms consists of searching for the closest matches to high-dimensional vectors. Examples of such problems include finding the best matches for local images in large datasets clustering local features into visual words using the

similarity algorithms. The nearest neighbor search problem is also of major importance in many other applications, including machine learning, document retrieval, data compression, bioinformatics, and data analysis.

We can define the nearest neighbor search problem as follows:

given a set of points $P = \{p_1, \dots, p_n\}$ in a vector space X , these points must be pre processed in such a way that given a new query point q in X , finding the points in P that are nearest to q can be performed efficiently. In this paper, they have assumed that X is an Euclidean vector space, which is appropriate for most problems in computer vision.

They have described potential extensions of their approach to general metric spaces, although this would come at some cost in efficiency or accuracy.

For high-dimensional spaces, there are no known algorithms for nearest neighbor search that are more efficient than simple linear search. As linear search is too costly, this has generated an interest in algorithms that perform approximate nearest neighbor search, in which non optimal neighbors are sometimes returned. Such approximate algorithms can be orders of magnitude faster than exact search, while still providing near optimal accuracy. There have been hundreds of papers published on algorithms for approximate nearest neighbor search, but there has been little systematic comparison to guide the choice among algorithms and set their internal parameters. One reason for this is that the relative performance of the algorithms varies widely based on properties of the datasets, such as dimensionality, correlations, clustering characteristics, and size.

FINDING FAST APPROXIMATE NEAREST NEIGHBORS

Comparison between many different algorithms for approximate nearest neighbor search on Datasets with a wide range of dimensionality were made. The accuracy of the approximation is measured in terms of precision, also defined as the percentage of query points for which the correct nearest neighbor is found. In the experiments, one algorithm obtained the best performance, depending on the dataset and desired precision. This algorithms uses the multiple randomized kd-trees.

The randomized kd-tree algorithm

The classical kd-tree algorithm is efficient in low dimensions, but in high dimensions the performance rapidly degrades, i.e. it too suffers the curse of dimensionality. To obtain a speedup over linear search it becomes necessary to settle for an approximate nearest-neighbor. This improves the speed of searching at the cost of the algorithm not always returning the exact nearest neighbors. The original kd-tree algorithm splits the data in half at each level of the tree on the dimension for which the data exhibits the greatest variance.

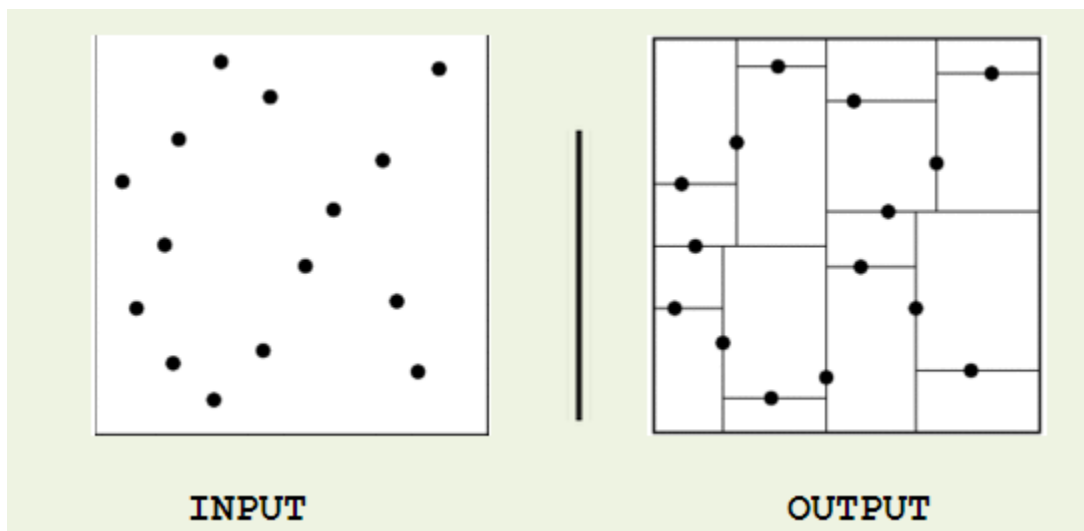


Figure 2.4 a: Implementation effect of KD tree

By comparison, the randomized trees are built by choosing the split dimension randomly from the first D dimensions on which data has the greatest variance. They used the fixed value $D = 5$ in their implementation, as this performs well across all our datasets and does not benefit significantly from tuning. When searching the trees, a single priority queue is maintained across all the randomized trees so that search can be ordered by increasing distance to each boundary. The degree of approximation is determined by examining a fixed number of leaf nodes, at which point the search is terminated and the best candidates returned. The user specifies only the desired search precision, which is used during training to select the number of leaf nodes that will be examined in order to achieve this precision.

At a high level, a kd-tree is a generalization of a binary search tree that stores points in k -dimensional space. That is, you could use a kd-tree to store a collection of points in the Cartesian plane, in three-dimensional space, etc. One could also use a kd-tree to store biometric data, for example, by representing the data as an ordered tuple, perhaps (height, weight, blood pressure, cholesterol).

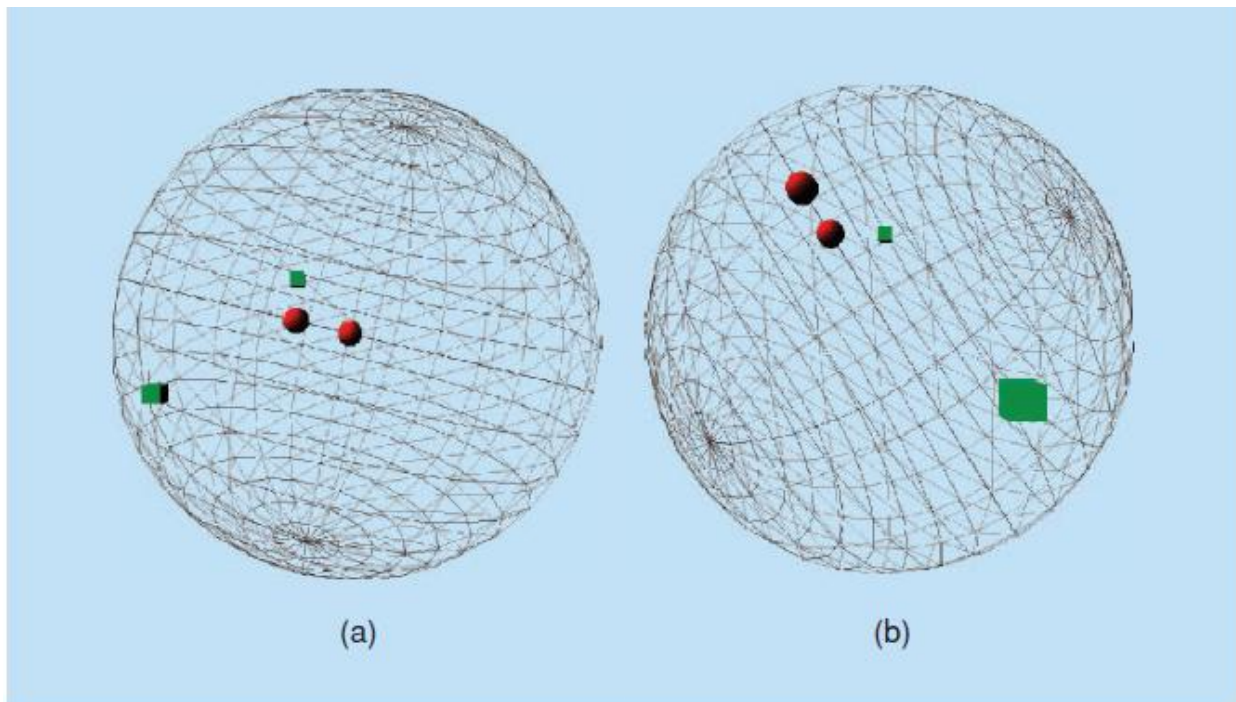
2.5 Research Paper: Locality-Sensitive Hashing for Finding Nearest Neighbor by Malcolm Slaney and Michael Casey

By building a hash table, i.e., a data structure that allows us to quickly map between a symbol and a value, when given a query we can calculate an arbitrary function of the symbol that maps the symbol into an integer that indexes a table. Thus a symbol with dozens of characters, and perhaps hundreds of bits of data, is mapped to a relatively small index into the table. A collision occurs when two points hash to the same value and there are special provisions to allow more than one symbol per hash value.

A well-designed hash table allows a symbol lookup in $O(1)$ time with $O(N)$ memory, where N is the number of entries in the table. Also a well designed hash function separates two symbols that are close together into different buckets. This makes a hash table a good means of finding *exact* matches. To find *approximate (near)* matches efficiently we use a locality sensitive hash. LSH is based on the simple idea that, if two points are close together, then after a “projection” operation these two points will remain close together, i.e in the same bucket. This idea can be easily understood using the examples shown in Figure 2.5.a. Two points that are close together on the sphere are also close together when the sphere is projected onto the two dimensional page. This is true no matter how we rotate the sphere. Two other points on the sphere that are far apart will, for some orientations, be close together on the page, but it is more likely that the points will remain far apart.

To further expand this basic idea, a random projection operation that maps a data point from a high-dimensional point to a low-dimensional subspace is started. First, one notes which points are close to our query points. Second, one creates projections from a number of different

directions and keeps track of the nearby points. A list of these found points and note the points that appear close to each other in more than one projection is kept. Part of the art of solving this problem is defining a projection, defining the notion of “nearby” so that we keep track of a manageable number of points, finding a good hash implementation, and analyzing the hash performance.



[FIG1] Two examples showing projections of two close (circles) and two distant (squares) points onto the printed page.

Figure 2.5 a: Projection Method instead of Hashing

APPLICATIONS

The LSH algorithm has been applied successfully to quickly find nearest neighbors in databases with very large size. Instead of finding exact matches as conventional hashes would, LSH takes into account the locality of the points so that nearby points remain nearby. Examples of such applications include finding duplicate pages on the Web, image retrieval, and music retrieval.

FINDING DUPLICATE PAGES ON THE WEB

The Web contains many duplicate pages, partly because content is duplicated across sites and partly because there is more than one URL that points to the same file on a disk. Yet search

engines should not return several copies of the same page. A solution to identify Web page duplicates makes use of shingles. Each shingle represents a portion of a Web page and is computed by forming a histogram of the words found within that portion of the page. One can test to see if a portion of the page is duplicated elsewhere on the Web by looking for other shingles with the same histogram. Given that there are billions of pages on the Web and any portion of any page might be a duplicate, there are an enormous number of shingles to test

AltaVista, the first large-scale Web search engine, used random selections (similarly to LSH) to test the similarity of pages. If the shingles of the new page match shingles from the database, then it is likely that the new page bears a strong resemblance to an existing page. An approximate solution to this problem is desired, especially when balanced with the computational savings of a solution like LSH.

RETRIEVING IMAGE AND MUSIC

LSH can be used as an image retrieval as an object recognition tool. One can compute a detailed metric for many different orientations and configurations of an object one wishes to recognize. Then, given a new image one simply checks the database to see if a pre computed object's metrics are close to our query. This database contains millions of poses and LSH allows us to quickly check if the query object is known.

In music retrieval typically we use conventional hashes and robust features to find musical matches. The features can be fingerprints, i.e., representations of the audio signal that are robust to common types of abuse that are performed to audio before it reaches our ears. Fingerprints can be computed, for instance, by noting the peaks in the spectrum (because they are robust to noise) and encoding their position in time and space. One then just has to query the database for the same fingerprint.

However, to find similar songs we cannot use fingerprints because these are different when a song is remixed for a new audience or when a different artist performs the same song. Instead, we can use several seconds of the song— a snippet—as a shingle. To determine if two songs are similar, we need to query the database and see if a large enough number of the query shingles are close to one song in the database. Although closeness depends on the feature vector, we know that long shingles provide specificity. This is particularly important because we can eliminate

duplicates to improve search results and to link recommendation data between similar songs. As discussed earlier, LSH proves useful to identify nearest neighbors quickly even when the database is very large.

Unlike conventional computer hashes that are designed to return *exact* matches in $O(1)$ time, an LSH algorithm uses dot products with random vectors to quickly find *nearest* neighbors. LSH provides a probabilistic guarantee that it will return the correct answer. In systems that have other sources of error (perhaps due to mislabelled data) one can reduce the LSH error below the error due to other sources, while significantly improving the computational performance. This makes LSH in particular, and randomized algorithms in general, important in today's world of Internet-sized databases.

CHAPTER 3: ALGORITHMS AND CODES

The algorithms were studied and the best implementation for the same was done. Code Blocks (Release 12.11, gcc 4.7.1, Windows/Unicode -32bit) has been used as the platform for developing all of the codes for the basic implementation to the codes required for analysis. Nevertheless, the codes are in C language and hence can be run in any other simulation environment, with compiler supporting C (example DOSBox or Turbo C).

3.1 K Nearest Neighbours Search:

The training phase for kNN consists of storing all known data items and their class labels. The testing phase for a new instance 'q', given a known set 'R' is as follows:

1. Compute the distance between 'q' and each instance in 'R'
2. Sort the distances in increasing numerical order and pick the first 'k' elements
3. Compute and return the most frequent class in the 'k' nearest neighbours, optionally weighting each instance's class by the inverse of its distance to 't'.

The main benefits of using kNN algorithm for classification are:

- Very simple implementation.
- Robust with regard to the search space; for instance, classes don't have to be linearly separable.
- Few parameters to tune: distance metric and k.

The main disadvantages of the algorithm are:

- Expensive testing of each instance, as we need to compute its distance to all known instances.

- Sensitiveness to noisy or irrelevant attributes, which can result in less meaningful distance numbers. Scaling and/or feature selection should be used in combination with kNN to mitigate this agenda.
- Sensitiveness to very unbalanced datasets, where most entities belong to one or a few classes, and infrequent classes are therefore often dominated in most neighborhoods.

CODE:

Main functions:

- (i) **Caldistance:** This function calculates the distance of the query point from all the points in the training data. The distances so calculated are stored into a 2 dimensional distance array, in which the first dimension holds the distance and the second dimension holds the class number of the point from which the distance of the query point is calculated.

```
int caldistance(float x[4])
{
    int i;
    int l=0;
    for(i=0;i<m;i++)
    {
        dist[l][0]=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;
        l++;
    }
    for(i=0;i<f;i++)
    {
        dist[l][0]=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]=1;
        l++;
    }
    for(i=0;i<n;i++)
    {
```

```

        dist[l][0]=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]=2;
        l++;
    }
    return l;
}

```

- (ii) Sort function: This function sorts the distances calculated in the function above.

```

void sort1(int l)
{
    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;
            }
        }
    }
}

```

- (iii) Neighbour search function: This function primarily considers the top n distances, where $n \geq$ number of attributes of the items in dataset, and checks for the second dimension of these top values. The class to which the maximum number of the sorted values belongs is then allotted to the query point.

```

        void neigh(float x[4],int k)
    {
        int i;

        if((dist[0][1]==0 && dist[1][1]==0 && dist[2][1]==0) || (dist[0][1]==0 && dist[1][1]==0)
|| (dist[0][1]==0 && dist[2][1]==0) || (dist[1][1]==0 && dist[2][1]==0))
        {
            arrm[+m][0]=x[0];
            arrm[m][1]=x[1];
            arrm[m][2]=x[2];
            arrm[m][3]=x[3];
            printf("(%f,%f,%f,%f) point lies in class m\n",x[0],x[1],x[2],x[3]);
            //printf("\n\nThe nearest neighbours are : ");
            for(i=0;i<k;i++)
                printf("\n(%f,%f,%f,%f) ",arrm[i][0],arrm[i][1],arrm[i][2],arrm[i][3]);
        }
        else if((dist[0][1]==1 && dist[1][1]==1 && dist[2][1]==1) || (dist[0][1]==1 &&
dist[2][1]==1) || (dist[0][1]==1 && dist[1][1]==1) || (dist[1][1]==1 && dist[2][1]==1) )
        {

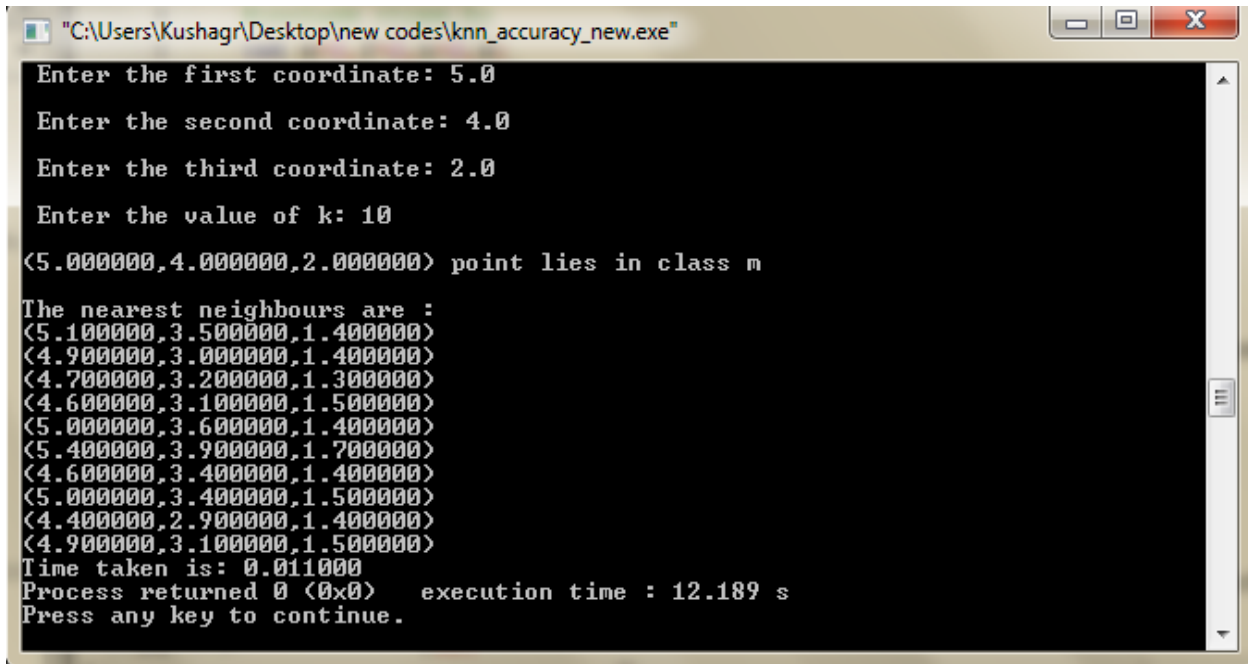
            arrf[+f][0]=x[0];
            arrf[f][1]=x[1];
            arrf[f][2]=x[2];
            arrf[f][3]=x[3];
            printf("(%f,%f,%f,%f) point lies in class f\n",x[0],x[1],x[2],x[3]);
            //printf("\n\nThe nearest neighbours are : ");
            for(i=0;i<k;i++)
                printf("\n(%f,%f,%f,%f) ",arrf[i][0],arrf[i][1],arrf[i][2],arrf[i][3]);
        }
        else if((dist[0][1]==2 && dist[1][1]==2 && dist[2][1]==2) || (dist[1][1]==2 &&
dist[2][1]==2) || (dist[0][1]==2 && dist[2][1]==2) || (dist[0][1]==2 && dist[1][1]==2) )
        {

            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 lies in class n\n",x[0],x[1],x[2],x[3]);
            //printf("\n\nThe nearest neighbours are : ");
            //for(i=0;i<k;i++)
                //printf("\n(%f,%f,%f,%f) ",arrn[i][0],arrn[i][1],arrn[i][2],arrn[i][3]);
        }
    }
}

```

- (iv) Main function: In the main function, training data is read and the query point is scanned in by the user. Thereafter the functions above are called in the same order as they are explained above.

OUTPUT SCREENSHOT:



```
"C:\Users\Kushagr\Desktop\new codes\knn_accuracy_new.exe"
Enter the first coordinate: 5.0
Enter the second coordinate: 4.0
Enter the third coordinate: 2.0
Enter the value of k: 10
<5.000000,4.000000,2.000000> point lies in class m
The nearest neighbours are :
<5.100000,3.500000,1.400000>
<4.900000,3.000000,1.400000>
<4.700000,3.200000,1.300000>
<4.600000,3.100000,1.500000>
<5.000000,3.600000,1.400000>
<5.400000,3.900000,1.700000>
<4.600000,3.400000,1.400000>
<5.000000,3.400000,1.500000>
<4.400000,2.900000,1.400000>
<4.900000,3.100000,1.500000>
Time taken is: 0.011000
Process returned 0 (0x0)   execution time : 12.189 s
Press any key to continue.
```

3.2 KNN using KD trees:

The nearest neighbour search (NN) algorithm aims at finding the point in the tree that is nearest to a given input point. This search can be done efficiently by using the tree properties to quickly eliminate large portions of the search space, just like the Binary Search Trees.

Searching for a nearest neighbour in a k -d tree proceeds as in the steps following:

1. Starting with the root node, the algorithm moves down the tree recursively, in the same way that it would if the search point were being inserted (i.e. it goes left or right depending on whether the point is less than or greater than the current node in the split dimension).
2. Once the algorithm reaches a leaf node, it saves that node point as the "current best"
3. The algorithm unwinds the recursion of the tree, performing the following steps at each node:
 1. If the current node is closer than the current best, then it becomes the current best.

2. The algorithm checks whether there could be any points on the other side of the splitting plane that are closer to the search point than the current best. In concept, this is done by intersecting the splitting hyperplane with a hypersphere around the search point that has a radius equal to the current nearest distance. Since the hyperplanes are all axis-aligned this is implemented as a simple comparison to see whether the difference between the splitting coordinate of the search point and current node is less than the distance (overall coordinates) from the search point to the current best.
4. When the algorithm finishes this process for the root node, then the search is complete.

KD-tree		
Type	Multidimensional BST	
Invented	1975	
Invented by	Jon Louis Bentley	
Time complexity in big O notation		
	Average	Worst case
Space	$O(n)$	$O(n)$
Search	$O(\log n)$	$O(n)$
Insert	$O(\log n)$	$O(n)$
Delete	$O(\log n)$	$O(n)$

Generally the algorithm uses squared distances for comparison to avoid computing square roots. Additionally, it can save computation by holding the squared current best distance in a variable for comparison.

CODE:

Data Structure:

A typical node structure with three components, one holding the data, one holding the address of the left child, one holding the address of the right child is defined in the name of Kdnode.

```
typedef struct KdNode
{
    float Data[4];
```

```

struct KdNode *Left;
struct KdNode *Right;
};

```

Main functions:

- (i) Insertion function: This function is used for the insertion into the kD tree. In this function a count is used to compare different coordinates of the node data item, each time. If the tree is empty, a root node is created otherwise the value of the new node's data item is checked for and compared with the root's 0th coordinate. If less, the left address of the node is checked. If empty, new node is inserted there, else the count is incremented by 1 and the second coordinate of the left child and the new node are compared to find a valid position for insertion. The process continues until a position is found vacant. At each level different coordinates are compared depending on the count.

```

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; }
            }
    }
}}

```


- (ii) Inorder function: This function is used for the inorder traversal of the given Kd tree and printing its elements in left->root->right order.

```
void inorder(struct KdNode *root)
{
    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));
    }
}
```

- (iii) Search function: This function is the primary function of the code, as it searches for the approximate nearest neighbours of a given element. User is asked to input a query point in the main code. That point is searched for in the tree, if found its inorder traversal is done, in the process elements being stored in an array. And $k/2$ points before the point and the same lying after the query point in the in order are printed. Otherwise, the point is first inserted into the tree and then the same process is followed.

```
void search(float Item[4], int k, struct KdNode *root)
{ 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"); }

```

- (iv) Main function: In the main function, tree is created based on the training data. Thereafter the code for searching is called for the query point.

OUTPUT SCREENSHOT:

```

C:\Users\Kushagr\Desktop\KD2.exe
Enter the element for which you want to find the nearest neighbours:
Enter first coordinate: 54
Enter second coordinate: 50
Enter the no of nearest neighbours: 20
New tree after insertion:
<0, 91> <2, 53> <3, 11> <4, 30> <5, 45> <6, 1> <7, 91> <9, 58> <10, 59>
<13, 68> <16, 35> <18, 95> <21, 16> <22, 33> <23, 41> <24, 72> <26, 23>
<27, 36> <28, 93> <29, 78> <31, 8> <33, 15> <34, 0> <35, 94> <37, 59> <
39, 58> <40, 42> <41, 67> <44, 39> <45, 9> <46, 5> <47, 26> <48, 83> <50
, 50> <53, 68> <54, 50> <55, 67> <56, 40> <57, 87> <61, 91> <62, 64> <64
, 48> <66, 76> <67, 99> <68, 84> <69, 24> <70, 50> <71, 38> <73, 64> <74
, 20> <77, 6> <78, 58> <81, 27> <84, 54> <86, 90> <88, 6> <90, 42> <91,
4> <92, 82> <93, 48> <95, 42> <96, 21> <97, 12>
The nearest neighbours for the entered element are: <55,67><56,40><57,87><61,91>
<62,64><64,48><66,76><67,99><68,84><69,24><70,50><53,68><50,50><48,83><47,26><46
,5><45,9><44,39><41,67><40,42>
Searching done
Process returned 0 (0x0)   execution time : 62.276 s
Press any key to continue.

```

3.3 KNN using Locality Sensitive Hashing:

An LSH family \mathcal{F} is defined for a metric space $\mathcal{M} = (M, d)$, a threshold $R > 0$ and an approximation factor $c > 1$. This family \mathcal{F} is a family of functions $h : \mathcal{M} \rightarrow S$ which map elements from the metric space to a bucket $s \in S$. The LSH family satisfies the following

conditions for any two points $p, q \in \mathcal{M}$, using a function $h \in \mathcal{F}$ which is chosen uniformly at random:

- if $d(p, q) \leq R$, then $h(p) = h(q)$ (i.e., p and q collide) with probability at least P_1 ,
- if $d(p, q) \geq cR$, then $h(p) \neq h(q)$ with probability at most P_2 .

A family is working right when $P_1 > P_2$.

One of the main applications of LSH is to provide a method for efficient approximate nearest neighbor search algorithms. Consider an LSH family \mathcal{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 \mathcal{G} of hash functions g , where each function g is obtained by concatenating k functions h_1, \dots, h_k from \mathcal{F} , i.e., $g(p) = [h_1(p), \dots, h_k(p)]$. In other words, a random hash function g is obtained by concatenating k randomly chosen hash functions from \mathcal{F} . The algorithm then constructs L hash tables, each corresponding to hash function g .

Given a query point q , 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 q . The process is stopped as soon as a point within distance cR from q is found.

CODE:

Main functions:

- (i) **Read_hash:** In this function the training data is read from the file and based on a hash function (which in our case is Euclidian Distance Measure) the data items are allocated a class with respect to the bucket's initial holding and a radius. Near distance is half the radius and is used as a measure by us here, because we want all points in a bucket to be within the range of the radius from each other, as demanded by the user. So if a point is at near distance from a central point of the bucket, it's maximum distance from the other point in the bucket will be less than equal to r , hence sufficing the requirement.

```

void read_hash()
{
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);
}

```

- (ii) Hash2 function: The hash_2 function is similar to the read hash function, except for the fact that it reads from the file containing the data to be tested, i.e. containing the query points.

```

void hash2()
{
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.....
t=clock()-t;
time_taken=(double)(t)/CLOCKS_PER_SEC;
}
}

```

```

printf("\nTime taken for the search function: %lf",time_taken);
//printf("\n\n\nFinding the actual nearest neighbours of the query point");
//actual_nearest_neighbours(val,s);
}

```

(iii)Main function: In the main function, the buckets are initialised and the query point is asked for by the user. Also the user is asked to set the value of the radius.

OUTPUT SCREENSHOT:

```

"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> <43, 24> <32, 49> <44, 23> <21, 33> <24, 43> <23, 26>
Time taken for the search function: 0.007000

```

CODE FOR LSH FOR STRINGS:

It is a primitive sample code developed only to check the working for Locality Sensitive Hashing with the strings. In this code 26 arrays are created and based on the first few characters the matching for the strings is done and bucket allocation follows.

Main function:

```

void search_neigh()
{
char ch;
int num,len=0,len1;
int num2,num3,num4;

```

```

printf("\nString search is conducted for %s",sstr);
ch=sstr[0];
printf("\nFirst character:%c",ch);
num=(int)ch;
num4=num-65;
printf("\nThe string has been allocated to bucket %d", (num-64));
printf("\nEnter the number of nearest neighbours you would like to print: ");
scanf("%d",&num2);
num3=num2;
if(ch=='A')
{
while(num2!=0)
{

printf("\n%s",alph1[num3-num2]);
num2--;
}
}
else if(ch=='B')
{
while(num2!=0)
{

printf("\n%s",alph2[num3-num2]);
num2--;
}
}
else if(ch=='C')
{

while(num2!=0)
{

printf("\n%s",alph3[num3-num2]);
num2--;
}
}
else if(ch=='D')
{

while(num2!=0)
{

printf("\n%s",alph4[num3-num2]);
num2--;
}
}....

....
}else if(ch=='Y')
{

```

```

while(num2!=0)
{

printf("\n%s",alph25[num3-num2]);
num2--;
}
}else if(ch=='Z')
{

while(num2!=0)
{

printf("\n%s",alph26[num3-num2]);
num2--;
}
}
}
void divide_input()
{
int len,len1=0;
char choice;
for(k=0;k<i;k++)
{
printf("\n %s ",strings[k]);
choice= strings[k][0];
switch(choice)
{
case 'A':
strcpy(alph1[a1],strings[k]);
a1++;
break;

case 'B':
strcpy(alph2[b1],strings[k]);
b1++;
break;.....

..... case 'Y':
strcpy(alph25[y2],strings[k]);
y2++;
break;

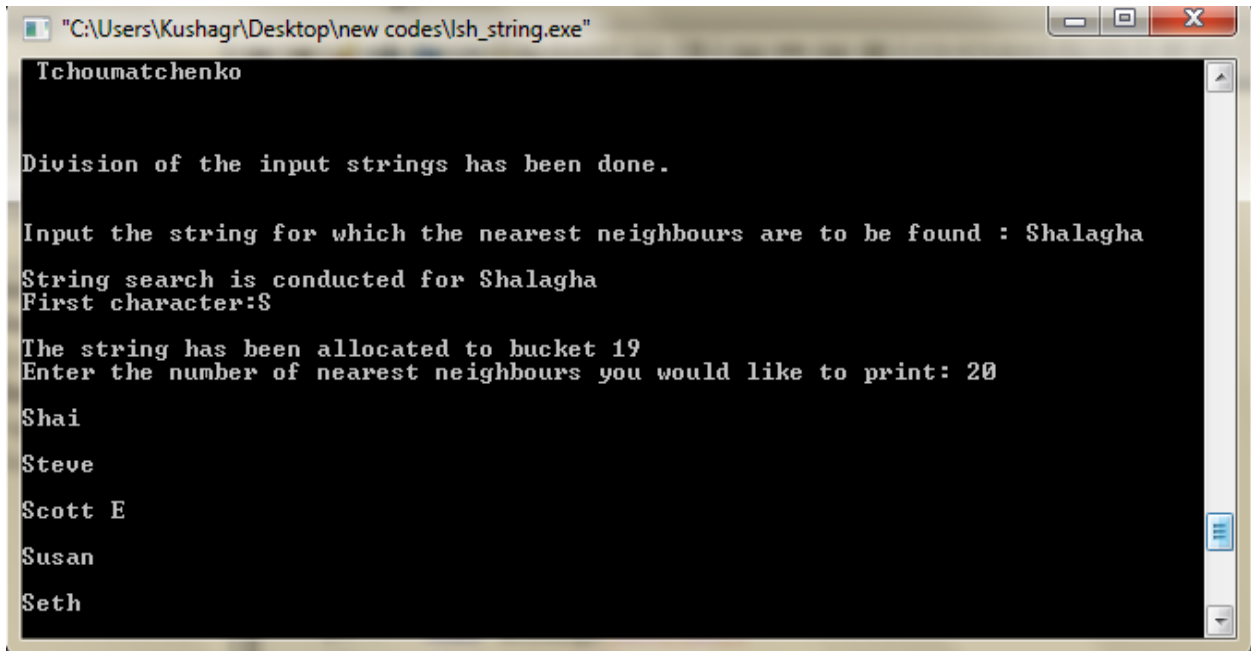
case 'Z':
strcpy(alph26[z1],strings[k]);
z1++;
break;

}}
printf("\n\nDivision of the input strings has been done.\n\n");

}

```


OUTPUT SCREENSHOT:



```
"C:\Users\Kushagr\Desktop\new codes\lsh_string.exe"
Tchoumatchenko

Division of the input strings has been done.

Input the string for which the nearest neighbours are to be found : Shalagha
String search is conducted for Shalagha
First character:S
The string has been allocated to bucket 19
Enter the number of nearest neighbours you would like to print: 20

Shai
Steve
Scott E
Susan
Seth
```

CODE FOR LSH WITH MULTIPLE HASH FUNCTIONS:

Main Functions:

- (i) Hash1 function: This function reads the query point and selects one random point from each bucket and calculates the Euclidian distance between the two. It then sorts the distance array in ascending order and calls the hash 2 function.

```
int hash1(float x[9])
{
    int i=0, j=0;
    float tem,tem2;
    dist[0][0]= sqrt (pow(x[0]-arr1[i][0],2)+pow(x[1]-arr1[i][1],2)+pow(x[2]-arr1[i][2],2)+ pow(x[3]-
arr1[i][3],2)+ pow(x[4]-arr1[i][4],2)+ pow(x[5]-arr1[i][5],2)+ pow(x[6]-arr1[i][6],2)+ pow(x[7]-
arr1[i][7],2)+ pow(x[8]-arr1[i][8],2));
    dist[0][1]=0.00;
    dist[1][0]=sqrt (pow(x[0]-arr2[i][0],2) + pow(x[1]-arr2[i][1],2)+ pow(x[2]-arr2[i][2],2)+
pow(x[3]-arr2[i][3],2)+ pow(x[4]-arr2[i][4],2)+ pow(x[5]-arr2[i][5],2)+ pow(x[6]-arr2[i][6],2)+
pow(x[7]-arr2[i][7],2)+ pow(x[8]-arr2[i][8],2));
    dist[1][1]=1.00;
```

```

    dist[2][0]=sqrt( pow(x[0]-arr3[i][0],2) + pow(x[1]-arr3[i][1],2)+ pow(x[2]-arr3[i][2],2)+
    pow(x[3]-arr3[i][3],2)+ pow(x[4]-arr3[i][4],2)+ pow(x[5]-arr3[i][5],2)+ pow(x[6]-arr3[i][6],2)+
    pow(x[7]-arr3[i][7],2)+ pow(x[8]-arr3[i][8],2));
    dist[2][1]=2.00;
    dist[3][0]=sqrt( pow(x[0]-arr4[i][0],2) + pow(x[1]-arr4[i][1],2)+ pow(x[2]-arr4[i][2],2)+
    pow(x[3]-arr4[i][3],2)+ pow(x[4]-arr4[i][4],2)+ pow(x[5]-arr4[i][5],2)+ pow(x[6]-arr4[i][6],2)+
    pow(x[7]-arr4[i][7],2)+ pow(x[8]-arr4[i][8],2));
    dist[3][1]=3.00;
    dist[4][0]=sqrt( pow(x[0]-arr5[i][0],2) + pow(x[1]-arr5[i][1],2)+ pow(x[2]-
    arr5[i][2],2)+ pow(x[3]-arr5[i][3],2)+ pow(x[4]-arr5[i][4],2)+ pow(x[5]-arr5[i][5],2)+ pow(x[6]-
    arr5[i][6],2)+ pow(x[7]-arr5[i][7],2)+ pow(x[8]-arr5[i][8],2));
    dist[4][1]=4.00;
    dist[5][0]=sqrt( pow(x[0]-arr6[i][0],2) + pow(x[1]-arr6[i][1],2)+ pow(x[2]-
    arr6[i][2],2)+ pow(x[3]-arr6[i][3],2)+ pow(x[4]-arr6[i][4],2)+ pow(x[5]-arr6[i][5],2)+ pow(x[6]-
    arr6[i][6],2)+ pow(x[7]-arr6[i][7],2)+ pow(x[8]-arr6[i][8],2));
    dist[5][1]=5.00;
    tem=100.00;
    for(j=0;j<6;j++)
    {
        for(mini=0;mini<5-j;mini++)
        {
            if(dist[mini][0]>dist[mini+1][0])
            {
                tem=dist[mini][0];
                tem2=dist[mini][1];
                dist[mini][0]=dist[mini+1][0];
                dist[mini][1]=dist[mini+1][1];

                dist[mini+1][0]=tem;
                dist[mini+1][1]=tem2;

            }
        }
    }
    hash2(x);
}

```

- (ii) Hash 2 function: This function computes the linear distance between the query point and a point each from the buckets. This is to ensure that the query point goes to right bucket. The distances so computed are sent to the neighbour function.

```

int hash2(float x[9])
{
    int i=0,j=0;
    float dist1[6];
    dist1[0]= (x[0]-arr1[i][0])+(x[1]-arr1[i][1])+(x[2]-arr1[i][2])+(x[3]-arr1[i][3])+(x[4]-arr1[i][4])+(x[5]-
    arr1[i][5])+(x[6]-arr1[i][6])+(x[7]-arr1[i][7])+(x[8]-arr1[i][8]);
}

```

```

    dist1[1]= (x[0]-arr2[i][0])+(x[1]-arr2[i][1])+(x[2]-arr2[i][2])+(x[3]-arr2[i][3])+(x[4]-arr2[i][4])+(x[5]-
arr2[i][5])+(x[6]-arr2[i][6])+(x[7]-arr2[i][7])+(x[8]-arr2[i][8]);
    dist1[2]= (x[0]-arr3[i][0])+(x[1]-arr3[i][1])+(x[2]-arr3[i][2])+(x[3]-arr3[i][3])+(x[4]-arr3[i][4])+(x[5]-
arr3[i][5])+(x[6]-arr3[i][6])+(x[7]-arr3[i][7])+(x[8]-arr3[i][8]);
    dist1[3]= (x[0]-arr4[i][0])+(x[1]-arr4[i][1])+(x[2]-arr4[i][2])+(x[3]-arr4[i][3])+(x[4]-arr4[i][4])+(x[5]-
arr4[i][5])+(x[6]-arr4[i][6])+(x[7]-arr4[i][7])+(x[8]-arr4[i][8]);
    dist1[4]= (x[0]-arr5[i][0])+(x[1]-arr5[i][1])+(x[2]-arr5[i][2])+(x[3]-arr5[i][3])+(x[4]-arr5[i][4])+(x[5]-
arr5[i][5])+(x[6]-arr5[i][6])+(x[7]-arr5[i][7])+(x[8]-arr5[i][8]);
    dist1[5]= (x[0]-arr6[i][0])+(x[1]-arr6[i][1])+(x[2]-arr6[i][2])+(x[3]-arr6[i][3])+(x[4]-arr6[i][4])+(x[5]-
arr6[i][5])+(x[6]-arr6[i][6])+(x[7]-arr6[i][7])+(x[8]-arr6[i][8]);
    //printf("\n");
    for(j=0;j<6;j++)
    {
        if(dist1[j]<0)
            dist1[j]=0-dist1[j];
    }
    for(j=0;j<6;j++)
    {
        printf(" %f",dist1[j]);
    }

    neigh(x,dist1);
}

```

- (iii) Neighbour function: This function takes the minimum distance from the hash1 function and the the distances computed by the hash2 function and after applying the conditions on the two finds the nearest neighbour with accuracy.

```

void neigh(float x[9],float k[6])
{
    int i;
    int d,h;
    if(dist[0][1]==0.00 && k[0]<=0.4)
    {
        m1++;
        arr1[m1][0]=x[0];arr1[m1][1]=x[1];arr1[m1][2]=x[2];
        arr1[m1][3]=x[3];arr1[m1][4]=x[4];arr1[m1][5]=x[5];
        arr1[m1][6]=x[6];arr1[m1][7]=x[7];arr1[m1][8]=x[8];
        printf(" CLASS 1");
        if(t1>0){
            printf("It's %d nearest neighbours are: ",t1);
            for(h=0;h<t1;h++)
            {
                printf("\n");
                for(d=0;d<6;d++)
                    printf("%f",arr1[h][d]);
                printf("");
            }
        }
        else if(dist[0][1]==1.00 && k[1]<=0.4)

```

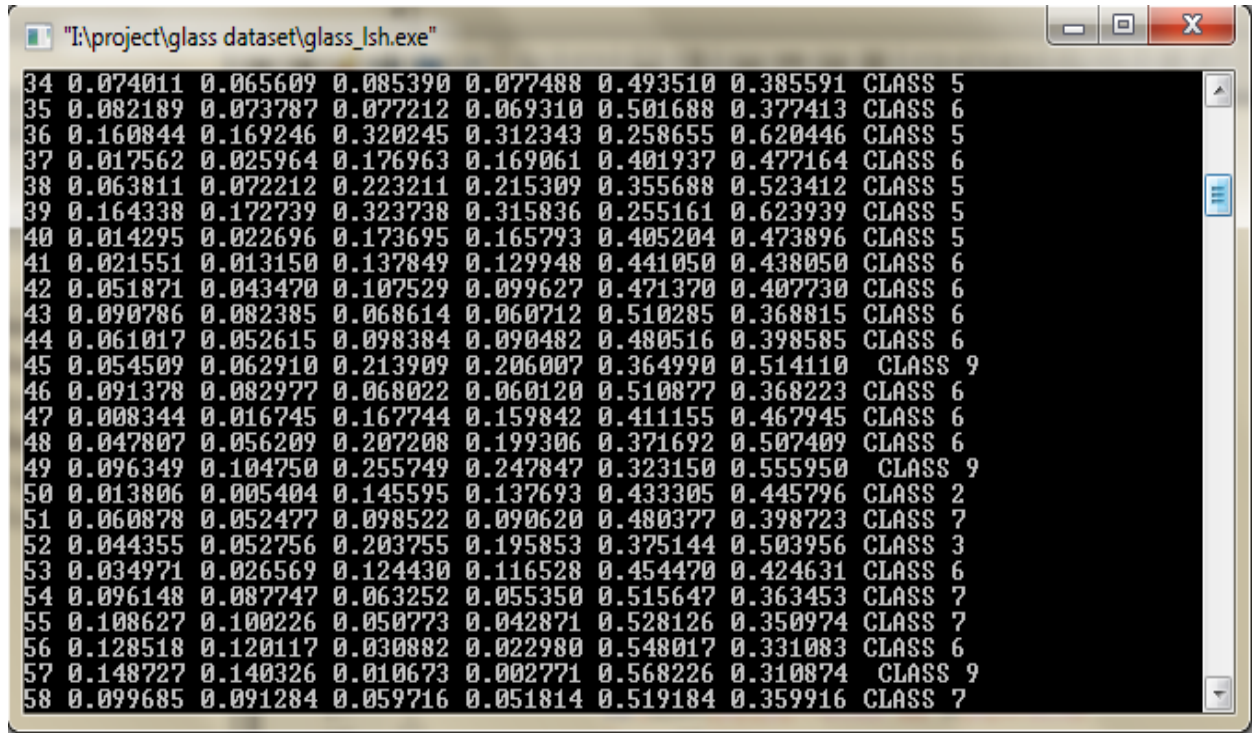
```

{
    m2++;
    arr2[m2][0]=x[0];arr2[m2][1]=x[1];arr2[m2][2]=x[2];
    arr2[m2][3]=x[3];arr2[m2][4]=x[4];arr2[m2][5]=x[5];
    arr2[m2][6]=x[6];arr2[m2][7]=x[7];arr2[m2][8]=x[8];
    printf(" CLASS 2");
    if(t1>0){
printf("It's %d nearest neighbours are: ",t1);
for(h=0;h<t1;h++)
{ printf("\n");
for(d=0;d<6;d++)
printf("%f",arr2[h][d]);
printf("");}
}
else if(dist[0][1]==2.00 && k[2]<=0.4)
{
    m3++;
    arr3[m3][0]=x[0];arr3[m3][1]=x[1];arr3[m3][2]=x[2];
    arr3[m3][3]=x[3];arr3[m3][4]=x[4];arr3[m3][5]=x[5];
    arr3[m3][6]=x[6];arr3[m3][7]=x[7];arr3[m3][8]=x[8];
printf(" CLASS 3");
if(t1>0){
printf("It's %d nearest neighbours are: ",t1);
for(h=0;h<t1;h++)
{ printf("\n");
for(d=0;d<6;d++)
printf("%f",arr3[h][d]);
printf("");}
} }....
.....else if(dist[0][1]==5.00 && k[5]<=0.4)
{
    m6++;
    arr6[m6][0]=x[0];arr6[m6][1]=x[1];arr6[m6][2]=x[2];
    arr6[m6][3]=x[3];arr6[m6][4]=x[4];arr6[m6][5]=x[5];
    arr6[m6][6]=x[6];arr6[m6][7]=x[7];arr6[m6][8]=x[8];
    printf(" CLASS 7");
    if(t1>0){
printf("It's %d nearest neighbours are: ",t1);
for(h=0;h<t1;h++)
{ printf("\n");
for(d=0;d<6;d++)
printf("%f",arr6[h][d]);
printf("");}
}
}
else
{
printf(" CLASS 9");
}
}

```

}

OUTPUT SCREENSHOT:



```
"I:\project\glass dataset\glass_lsh.exe"
34 0.074011 0.065609 0.085390 0.077488 0.493510 0.385591 CLASS 5
35 0.082189 0.073787 0.077212 0.069310 0.501688 0.377413 CLASS 6
36 0.160844 0.169246 0.320245 0.312343 0.258655 0.620446 CLASS 5
37 0.017562 0.025964 0.176963 0.169061 0.401937 0.477164 CLASS 6
38 0.063811 0.072212 0.223211 0.215309 0.355688 0.523412 CLASS 5
39 0.164338 0.172739 0.323738 0.315836 0.255161 0.623939 CLASS 5
40 0.014295 0.022696 0.173695 0.165793 0.405204 0.473896 CLASS 5
41 0.021551 0.013150 0.137849 0.129948 0.441050 0.438050 CLASS 6
42 0.051871 0.043470 0.107529 0.099627 0.471370 0.407730 CLASS 6
43 0.090786 0.082385 0.068614 0.060712 0.510285 0.368815 CLASS 6
44 0.061017 0.052615 0.098384 0.090482 0.480516 0.398585 CLASS 6
45 0.054509 0.062910 0.213909 0.206007 0.364990 0.514110 CLASS 9
46 0.091378 0.082977 0.068022 0.060120 0.510877 0.368223 CLASS 6
47 0.008344 0.016745 0.167744 0.159842 0.411155 0.467945 CLASS 6
48 0.047807 0.056209 0.207208 0.199306 0.371692 0.507409 CLASS 6
49 0.096349 0.104750 0.255749 0.247847 0.323150 0.555950 CLASS 9
50 0.013806 0.005404 0.145595 0.137693 0.433305 0.445796 CLASS 2
51 0.060878 0.052477 0.098522 0.090620 0.480377 0.398723 CLASS 7
52 0.044355 0.052756 0.203755 0.195853 0.375144 0.503956 CLASS 3
53 0.034971 0.026569 0.124430 0.116528 0.454470 0.424631 CLASS 6
54 0.096148 0.087747 0.063252 0.055350 0.515647 0.363453 CLASS 7
55 0.108627 0.100226 0.050773 0.042871 0.528126 0.350974 CLASS 7
56 0.128518 0.120117 0.030882 0.022980 0.548017 0.331083 CLASS 6
57 0.148727 0.140326 0.010673 0.002771 0.568226 0.310874 CLASS 9
58 0.099685 0.091284 0.059716 0.051814 0.519184 0.359916 CLASS 7
```

CHAPTER 4: GRAPHS AND RESULTS

Through this chapter, mainly the differences in performances of the two Approximation Algorithms namely: K Nearest Neighbour Search Algorithms and KNN using Locality Sensitive Hashing will be analysed on the basis of time and accuracy.

4.1 WHEN CODES WERE RUN ON IRIS DATASET:

4.1.1 About the dataset:

Size of Dataset: 150

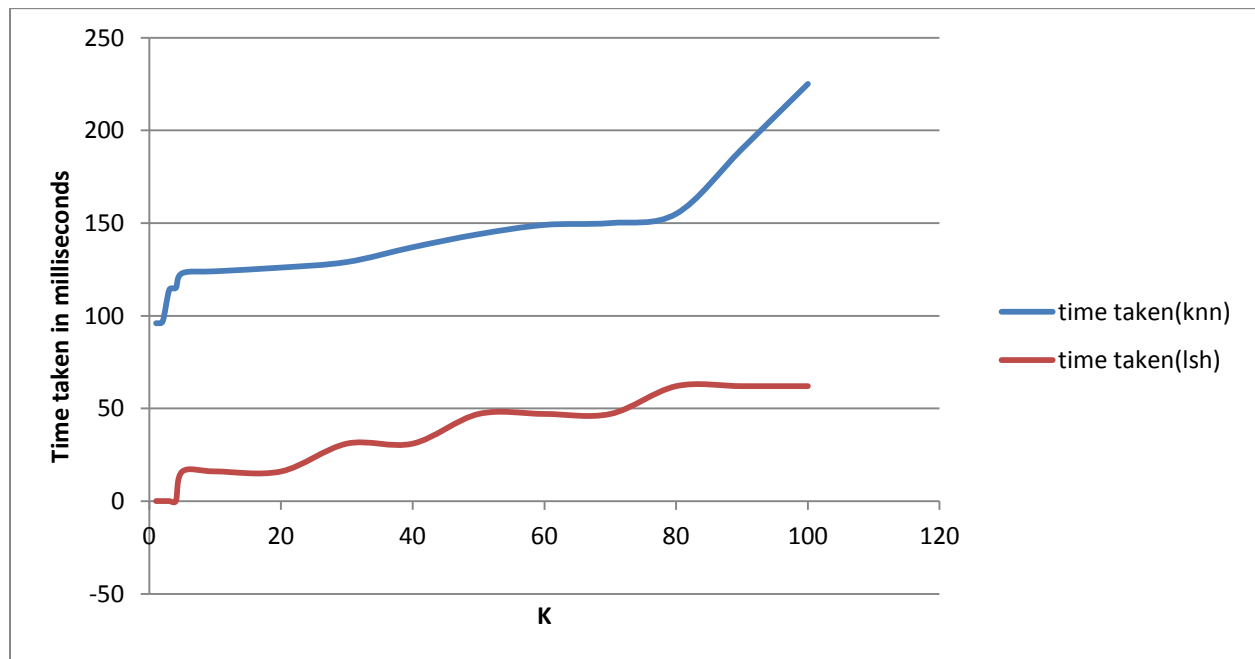
Number of Classes data is divided in: 3 classes of 50 data items each

Number of Attributes: 4

Data Type of Attributes: Integer

4.1.2 Analysis:

(i) Comparison based on time:

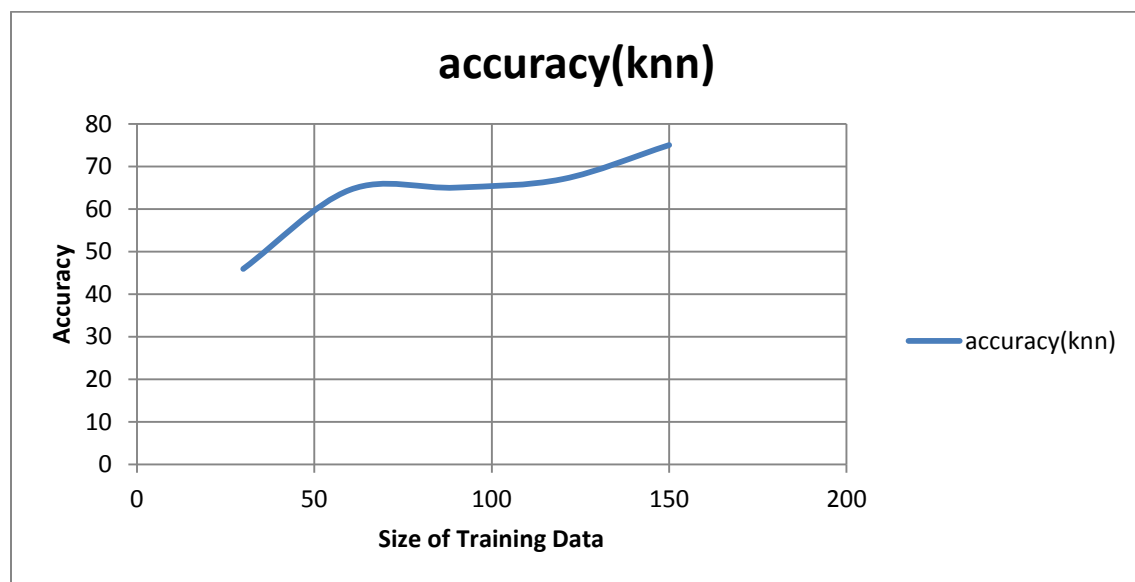


As can be seen above, for evaluating the 'k' nearest neighbours of a given query point, KNN takes substantially more time than the KNN using Locality Sensitive Hashing code. The time

taken by the kNN using LSH code is almost always less than 60 milliseconds, where the time by the kNN code is almost above 100 milliseconds.

The difference in the time taken is huge despite of the fact that the size of dataset is merely 150 and the values of k range between 10 and 100 only. In actual data mining processes, both the size of the dataset and the value of 'k' is a lot more and hence the time difference is bound to rise then.

(ii) Comparison based on Accuracy:

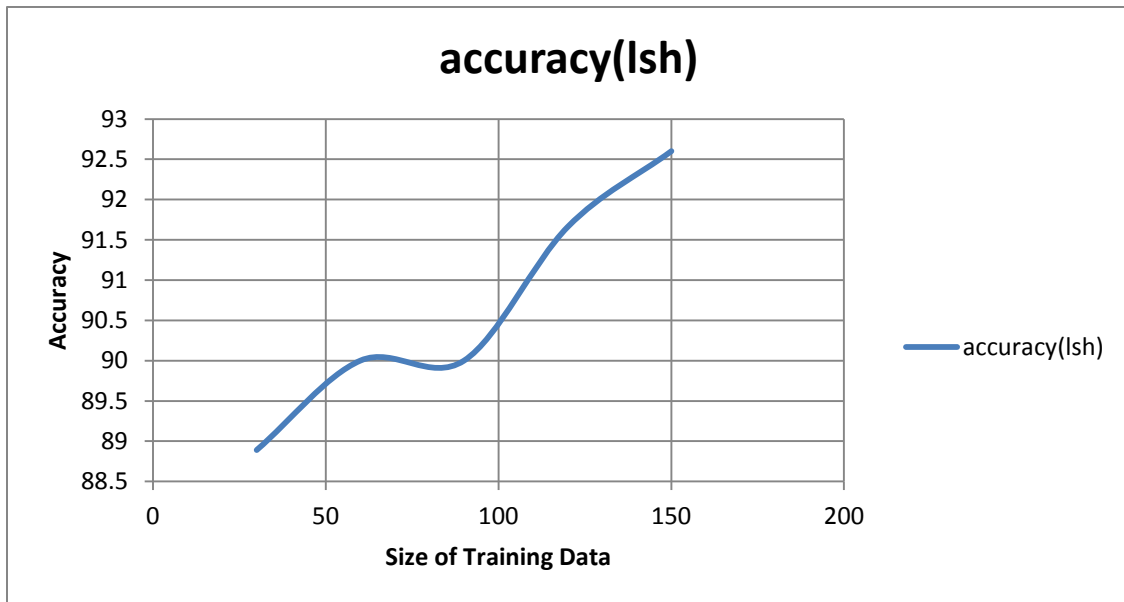


The accuracy for the algorithms in this case was determined by varying the sizes of the training and the testing datasets. When 20 percent of the training data was used for the purpose of training, the remaining 80 percent was tested for using the code. As we already knew the classes of the data in training, we could determine the accuracy of the code efficiently.

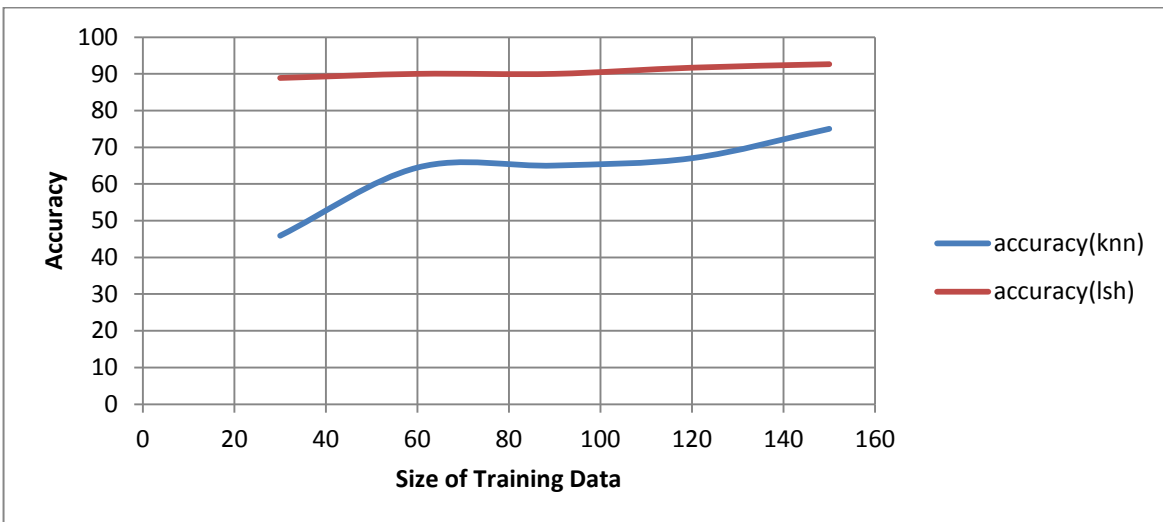
The accuracy of kNN when 30 of 150 elements were used for training and remaining 120 were tested was around 45 percent. Its accuracy, when all the training data was used for training purposes, shot up to 74 percent. The accuracy, thus, is average and not very promising.

The accuracy for kNN using locality sensitive hashing, as shown in the next graph, ranges from 89 percent to 92.5 percent as the size of training data varies from 20% to 100%. The reason for

this trend in LSH is that the classes into which the dataset is already divided does not have much of an effect on the distribution of elements. All the elements, be it in training data or testing data are redistributed into the three buckets using the Euclidian Hash function. The noise in the point to bucket mapping is thus reduced and accuracy increases.



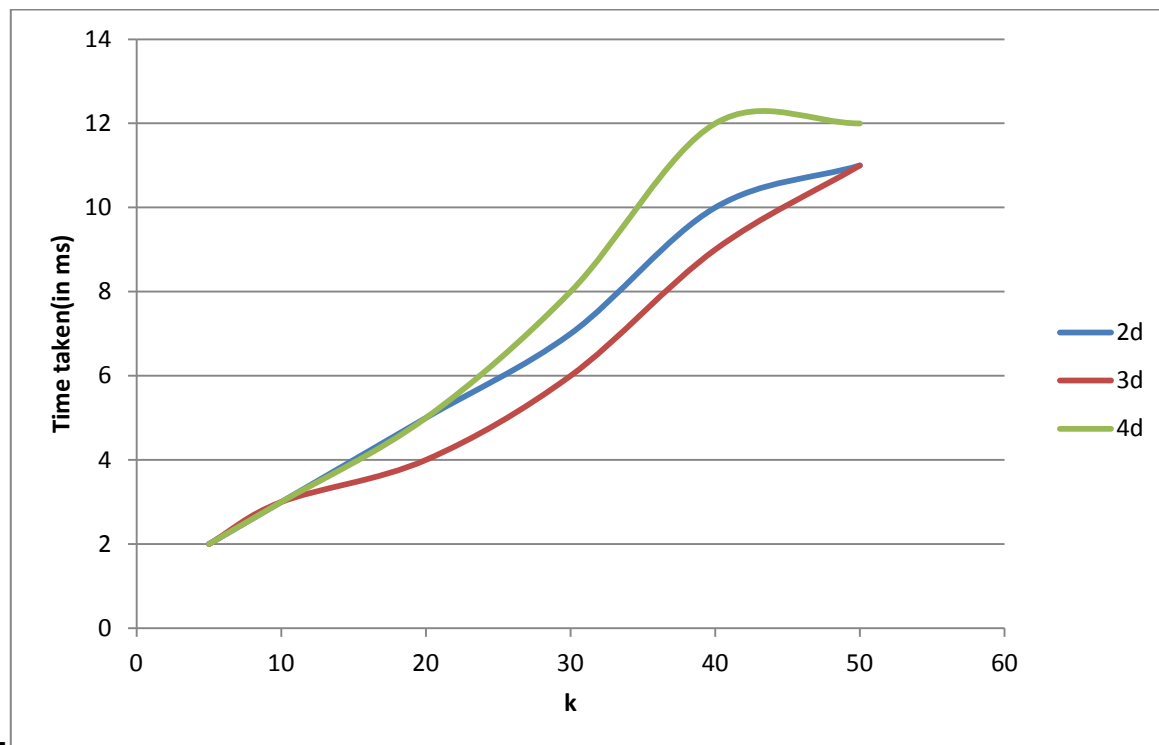
As can be seen from the graph below, the Locality sensitive Hashing algorithm outperforms the kNN algorithm in terms of accuracy as well.



But since no conclusion can be drawn from a single dataset analysis, three more datasets have been analysed next.

(iii) Dependence on Dimension:

Also, a small effort to look into the dependence on dimension was made by running the LSH code for Iris Dataset, by considering only two, then three and then all four attributes while finding neighbours.



As can be seen above there is minimal dependence on the dimension and hence LSH is the code to work on.

4.2 WHEN CODES WERE RUN ON LETTER DATASET:

4.2.1 About the Dataset:

Size of the Dataset: 20,000

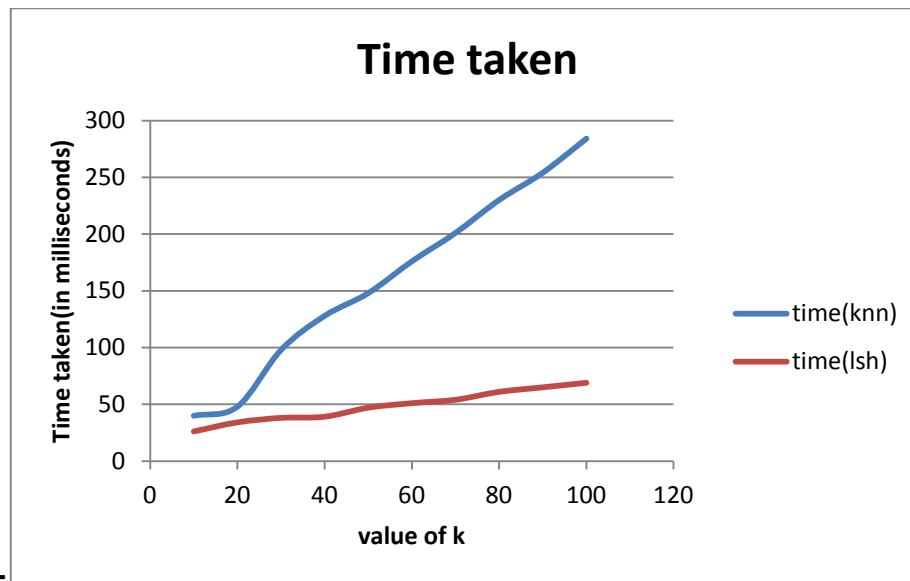
Number of Classes the Dataset has been divided into: 26

Number of Attributes: 16

Data Type of Attributes: Integer

4.2.2 Analysis:

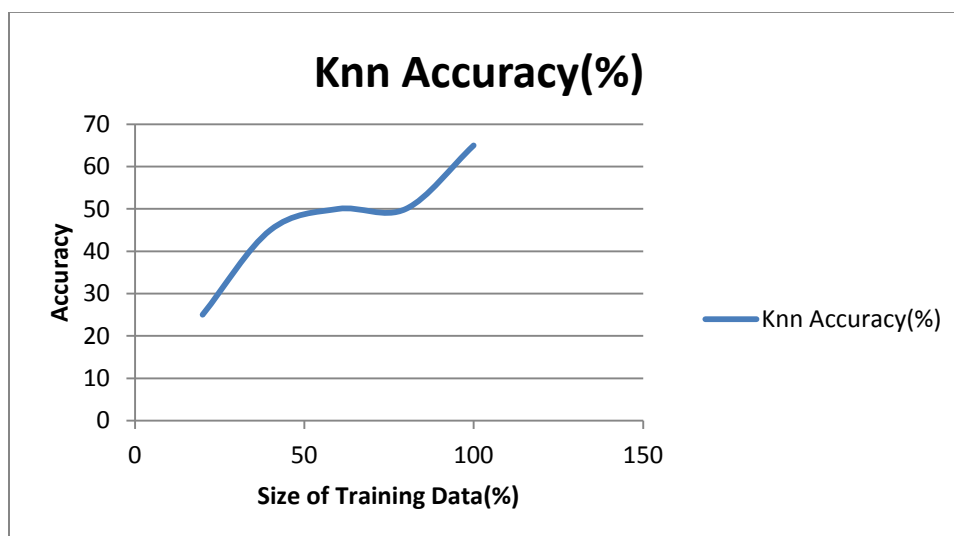
(i) Comparison based on time:



As can be seen from the graph above, as we vary the value of k, the value of time varies as well. The time taken by kNN increase with k, by heaps and bounds. On the contrary, the dependence of LSH on k remains minimal here as well.

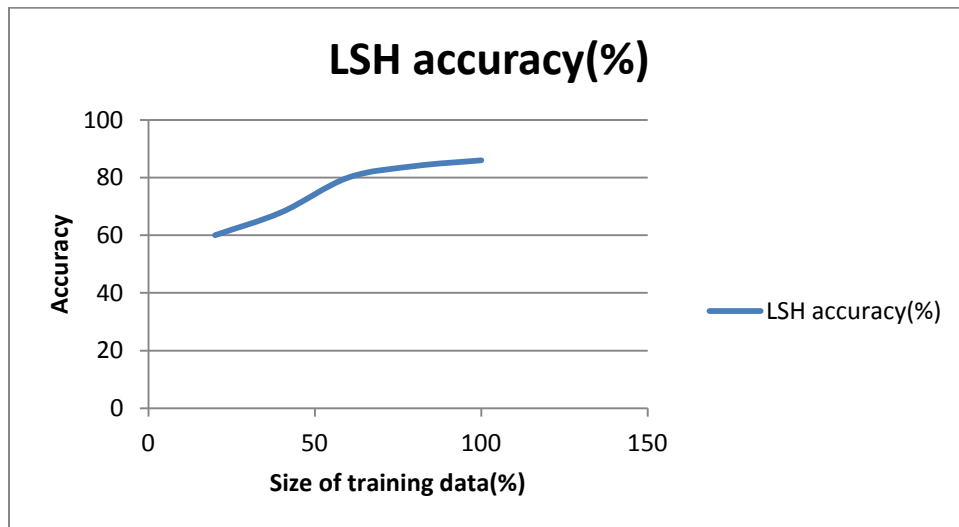
In terms of time, kNN using LSH again outshines simple kNN.

(ii) Comparison based on Accuracy:

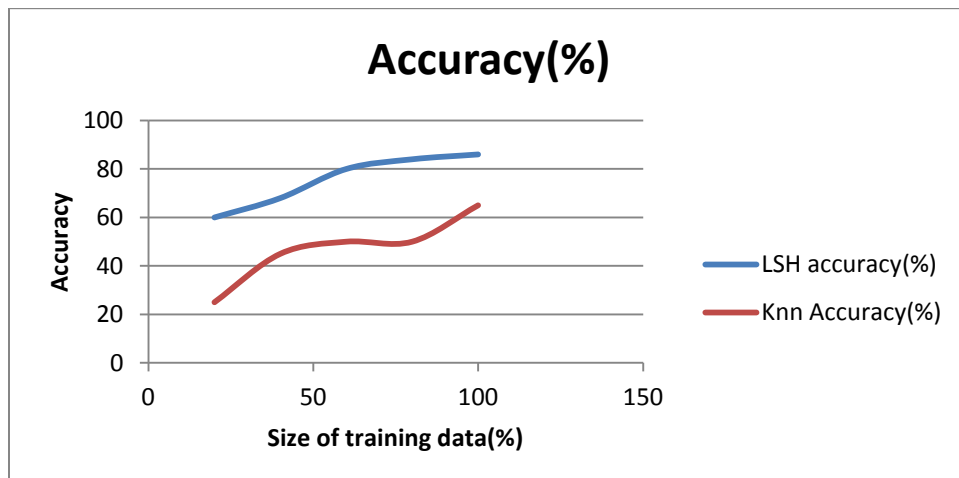


The code for kNN did not work properly when 100% of the data was taken for training purposes. It took 107 minutes for the code to process merely 7 or 8 elements in the testing data. Hence subsets of the dataset of size 5000 and less were taken for training purposes and testing was done on maximum 100 elements each time.

As can be seen from the graph above, the accuracy for kNN in this case is almost always less than 60 percent and there is a huge effect of the size of training data on the accuracy which ranges from 25% to 65% as the data ranges from 20% to 100%.



As for the accuracy for LSH, the accuracy is not very high and promising either, but it is still better than that of simple kNN. There is noise reduction in distribution of elements in kNN using LSH as the random distribution of elements in the dataset is not considered, and elements are rearranged based on their proximity with the elements in the 26 buckets.



4.3 WHEN RUN ON GLASS DATASET:

4.3.1: About the Dataset

Size of the dataset: 214

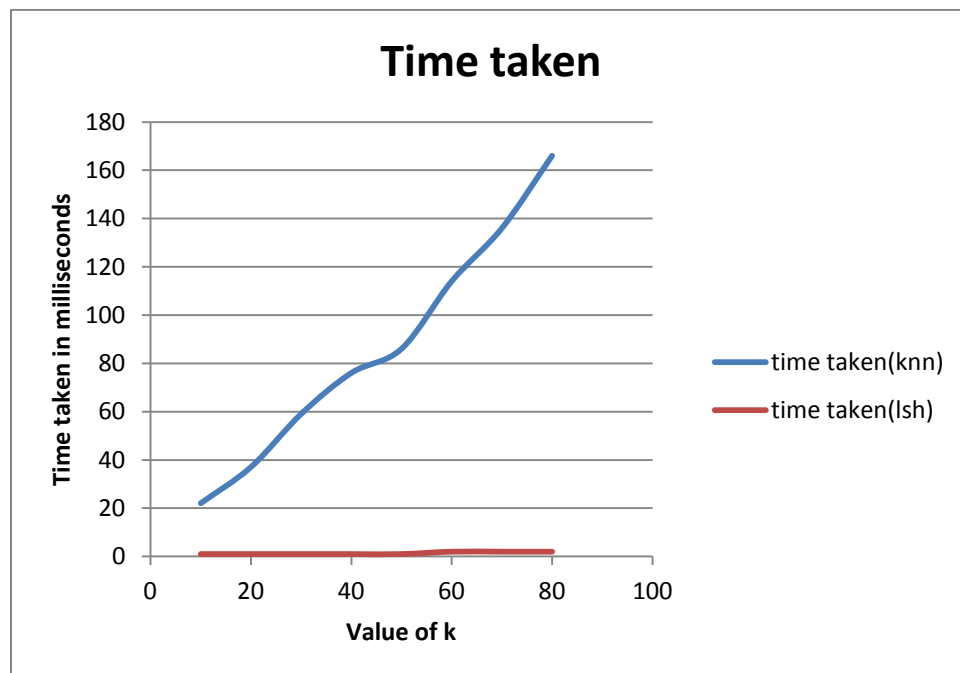
Number of Classes data is divided into: 6

Number of attributes: 9

Data Type of Attributes: Floating point integers

4.3.2: Analysis:

(i) Comparison based on time:

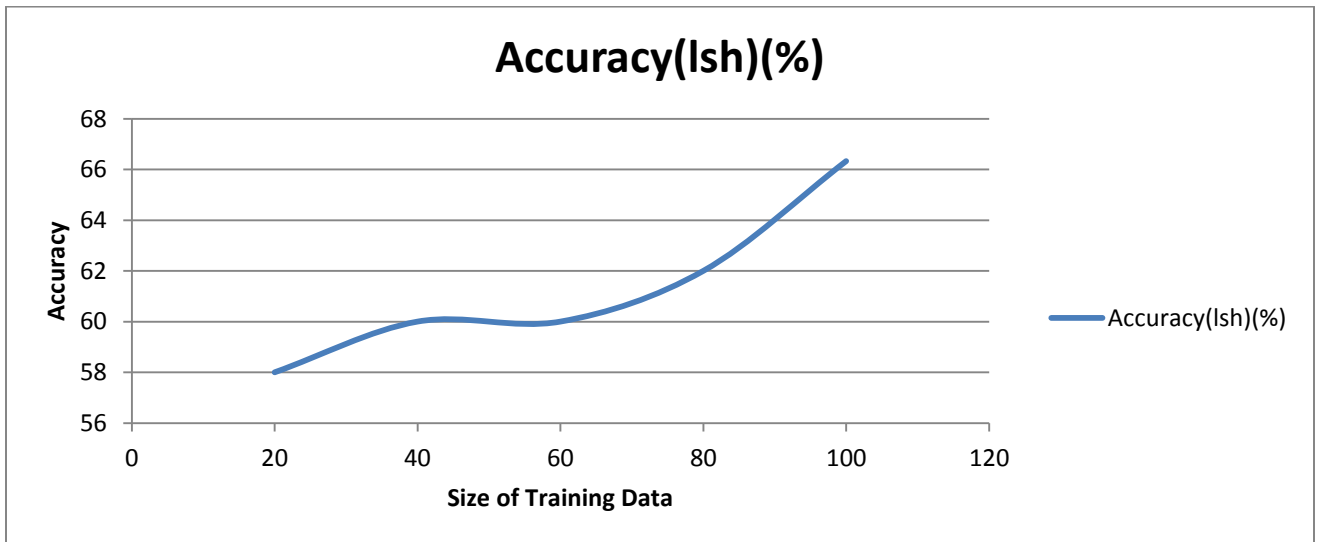
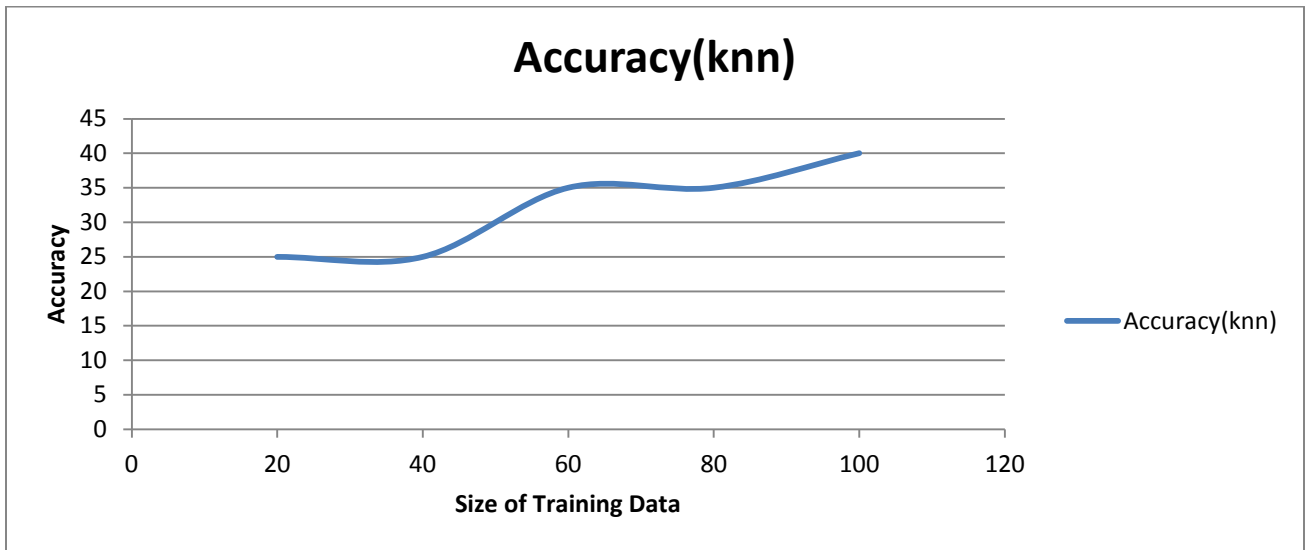


Here the comparison between the performance of the two codes in terms of time makes a clear contribution to our search for the better algorithm. The time take by the kNN using LSH is around 1 millisecond, whereas the time taken by kNN varies from 20 milliseconds to 160 milliseconds as the value of 'k' changes. The difference in performance is too evident to deserve an explanation. There were two hash functions used in the LSH code to increase the accuracy,

thus double the calculation than the previous LSH codes, and still the performance remains outstanding.

(ii) Comparison based on Accuracy:

The data in the dataset was randomly distributed with no sense of being class specific which can be attributed to the low accuracy of both kNN and kNN using LSH in this case.



Although both the codes were to deal with the same dataset, the trend of the accuracy of the LSH code being more than that of the kNN code remains to be. The accuracy of the kNN code goes maximum upto 40 percent due to random classification. In case of LSH, there is some

improvement due to redistribution, but the accuracy still remains to be below 70 percent throughout.



4.4 When run on Chess Dataset:

4.4.1 About the Dataset:

Size of the Dataset: 27,000

Number of Classes the dataset has been divided into: 17

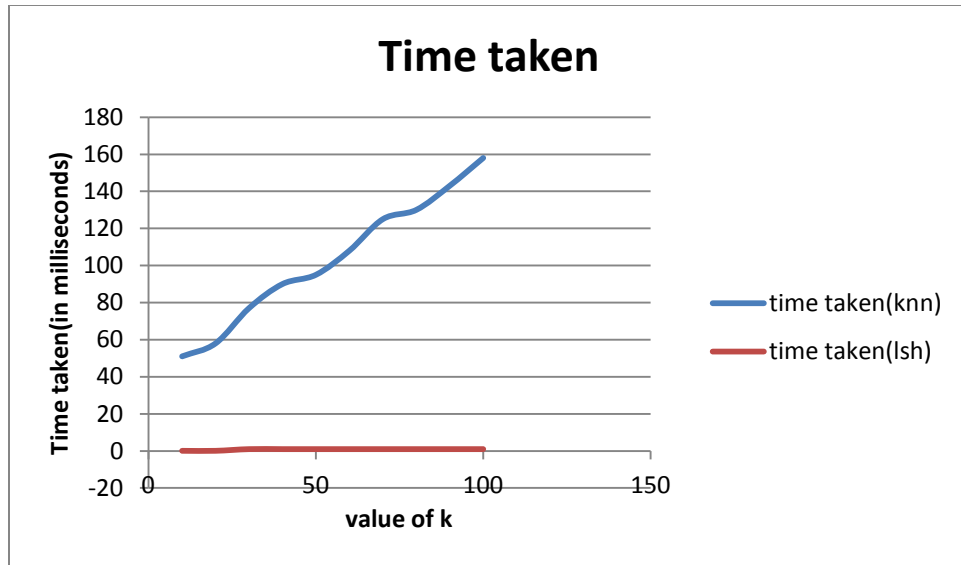
Number of attributes: 6

Type of Attributes: Mixed Data type (Both characters and integers)

4.4.2: Analysis:

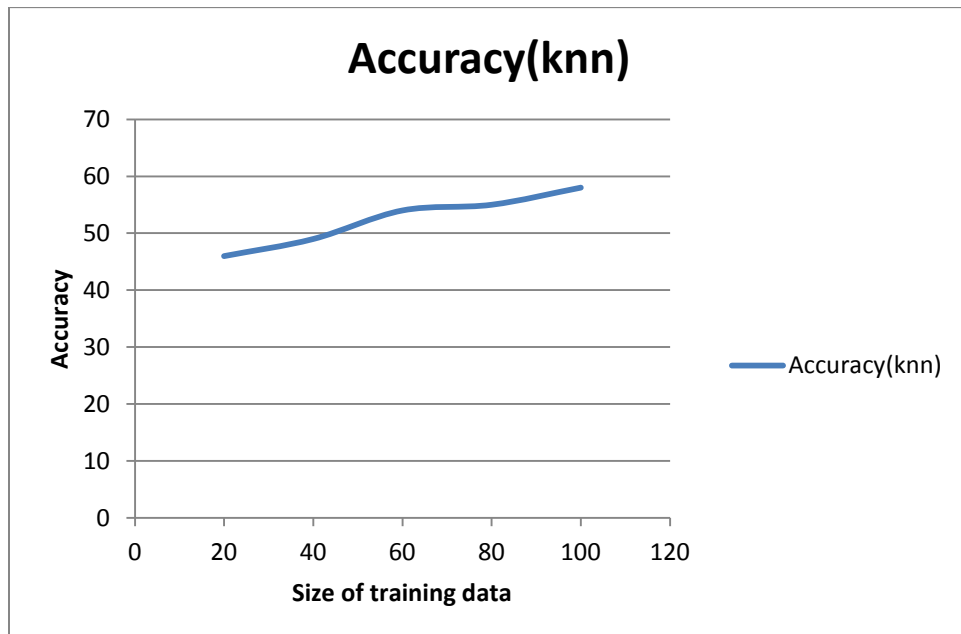
(i) Comparison based on time:

The graph below completely confirms our assumption so far. The performance of LSH in terms of time, irrespective of the size of dataset is better than that of kNN. The kNN code heavily depends on the value of k, whereas the code for LSH is by and large independent of the value of 'k' when computing k nearest neighbours.

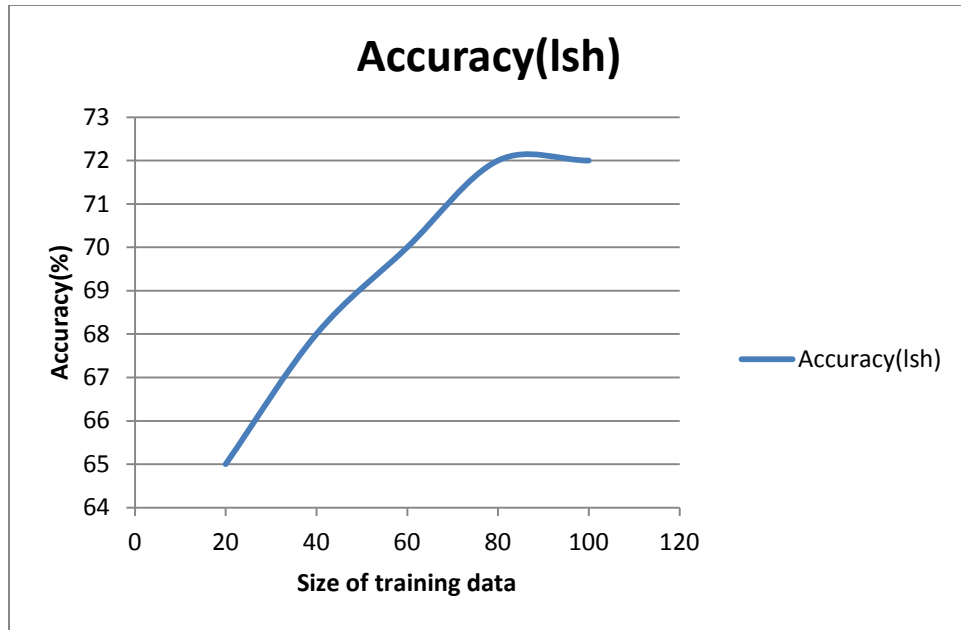


(ii) Comparison based on Accuracy:

The accuracy here again shows a similar trend. It doesn't go very low or very high but remains almost stable with the variance of maximum 10 in both the cases.

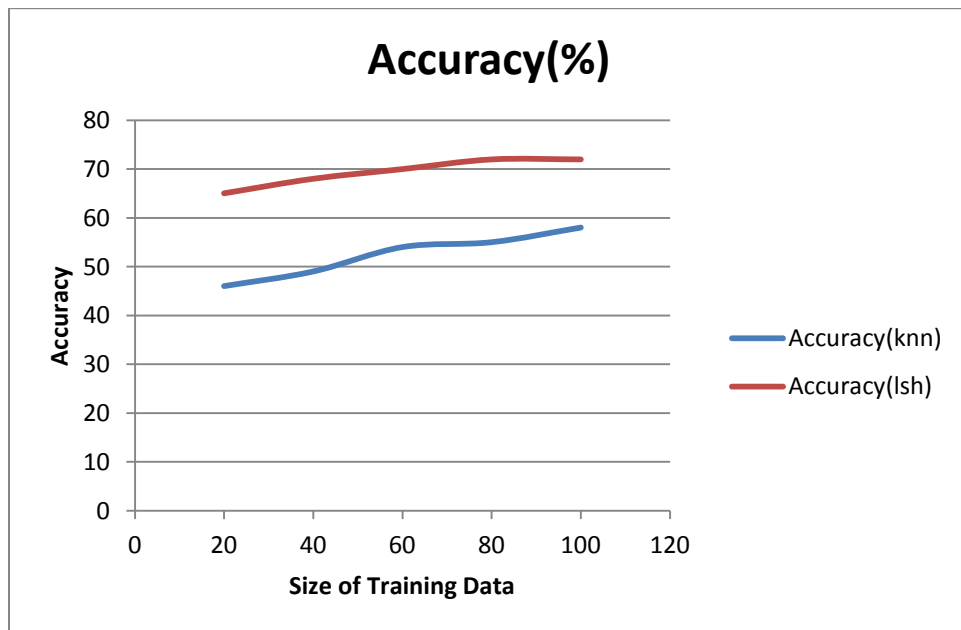


Even in this comparison, the accuracy for LSH is more than that of the simple kNN. This can again be attributed to the noise reduction that takes place in case of LSH.



The datasets and the experimentation with them, provide an insight into the functioning of the two algorithms, where one algorithm gains and where the other one loses.

Here, LSH is in a better position because of lesser computation involved in comparison to the K Nearest Neighbour Search. It has better accuracy because it operates independently and doesn't lay much emphasis on the distribution already present.



CHAPTER 5: CONCLUSION

Through the project, various methods of approximation have been studied. The ones implemented have been studied all the more in detail.

The K Nearest Neighbour search algorithm has been implemented at basic level and then its variations were created to run it for different datasets, and analyse its performance.

Also the k Nearest Neighbour Algorithm using Locality sensitive Hashing was first implemented at basic level and then it was modified to make it suitable for the various datasets.

As seen in the graphs, the performance of kNN is still marred by a variety of factors in terms of time. It is dependent on the size of the dataset and also the value of 'k'. This verifies the complexity of the code which is $O(knd)$.

kNN using Locality sensitive Hashing outperforms the simple k Nearest Neighbours Algorithm in terms of time based performance. Irrespective of the value of 'k' and the dimension of the data items, there is almost constant time taken by the LSH algorithm which proves that when it comes to efficiency in terms of time, LSH is the algorithm to opt for.

One would think that if LSH is cutting down on time, it must be at the cost of Accuracy. Well yes, some amount of accuracy is compromised on, but then when a comparison is drawn between the levels of accuracy between kNN using LSH and simple kNN, LSH stands in a win win situation.

The accuracy of kNN using LSH irrespective of the data distribution in the dataset has constantly remained more than that of the simple kNN. The initialisation of the buckets in LSH with the average of all data items belonging to a class and then allotment of buckets both to items in the training data and testing data, cuts down on the noise in the initial distribution and gives even better results than kNN which completely bases itself on the given division of the classes.

Thus, all in all, for data mining purposes the algorithm which is set to give better performance both in terms of time as well as accuracy irrespective of the size of the dataset, the dimension of the data items is k Nearest Neighbour Search Algorithm using Locality Sensitive Hashing.

CHAPTER 6: REFERENCES

Research papers:-

- **Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions**
-Alexandr Andoni and Piotr Indyk
- **Nearest Neighbor Search: the Old, the New, and the Impossible**
-Alexander Andoni
- **An Optimal Algorithm for Approximate Nearest Neighbor Searching in Fixed Dimensions**
-David M. Mount, Nathan S. Netanyahu, Ruth Silverman and Angela Y. Wu
- **Similarity search in high dimensions via hashing**
-A. Gionis, P. Indyk, and R. Motwani
- **. Fast Approximate Nearest Neighbours with Automatic Algorithm Configuration**
- Marius Muja, David G. Lowe:

Books:

- **Mining of Massive Datasets**
- Jure Leskovec, Stanford Univ. Anand Rajaraman Millway Lab
& Jeffrey D. Ullman Stanford Univ
- **Data Mining : Concepts and Techniques**
- Jiawei Han and. Micheline Kamber. University of Illinois at Urbana-Champaign