Multiple Random Projection For Fast, Approximate Nearest Neighbor Search in High Dimensions

by

Yousuf Shamim Ahmed

A thesis submitted in conformity with the requirements
for the degree of M.Sc.
Graduate Department of Computer Science
University of Toronto

Abstract

Multiple Random Projection For Fast, Approximate Nearest Neighbor Search in High Dimensions

Yousuf Shamim Ahmed

M.Sc.

Graduate Department of Computer Science

University of Toronto

2004

Random Projection has recently been used as a promising dimensionality reduction technique. Using random projection can speed up the finding of approximate nearest neighbors (NN) but it can't easily be used for exact NN. On the other hand, k-d tree and other related data structures can find exact NN, but as the dimensionality of the feature space increases these structures become quickly inefficient. The computational cost of these tree data structures grow almost exponentially with the intrinsic dimensionality of the data. In this thesis, we present experimental results evaluating the performance of exact and approximate methods for NN search on a variety of real and synthetic data sets. Finally, we present a hybrid model of Multiple Random Projection (MRP) and k-d tree to find approximate nearest neighbors in high dimension. The experimental results show that this hybridization results in improved performance w.r.t. number of distance calculations needed to find NN.

Acknowledgements

First of all, I want to thank my supervisor Sam Roweis for being an excellent mentor through out my Masters studies. He not only introduced me to this powerful topic of Random Projection but also carefully guided me all through this path giving right directions and helped me with many useful discussions.

I would also like to thank my second reader Avner Magen for his thorough review of this thesis and inspiring me with some constructive comments about this topic. I am also thankful to the machine learning group and proud to be a part of it.

On a personal note, I would like to thank my good friends Ben, Andriy, Mustansir, Renqiang, Nazrul for many enjoyable discussions. I also thank my parents who are bearing the pain of staying away from me, yet supporting me in every way they can. Finally I thank my wife Lubna who has the courage to start her marital life with a graduate student and for being more than mere inspirational.

Contents

1 Introduction **1**

2 Exact Nearest Neighbor **5**

2.1 Naive Nearest 6

2.1.1 Naive Nearest with Early Break 6

2.1.2 Naive Nearest with Annulus Bound 7

2.2 k-d tree 8

2.2.1 Searching in k-d tree 11

2.3 R-tree 11

2.4 Ball Tree 13

2.4.1 Constructing a Ball tree 14

2.4.2 Searching in a Ball tree 14

2.5 SR-tree 14

2.6 Experiments with Exact NN Algorithms 16

2.6.1 Experimental methods 16

2.6.2 Data sets 18

2.7 Experimental Results 20

2.8 Curse of Dimensionality 27

3 Approximate Nearest Neighbor **29**

3.1 Metric Skip Lists 30

3.1.1 Data Structure . . . 31
3.1.2 Off-line Construction . . . 31
3.1.3 Searching . . . 32
3.1.4 Experimental Results . . . 35
3.2 Algorithm by Kleinberg . . . 44
3.2.1 Data Structure . . . 44
3.2.2 Processing a query . . . 45
3.2.3 Modification of the Algorithm . . . 46
3.2.4 Experimental results . . . 48
3.3 Other methods . . . 55

4 Multiple Random Projection Technique **56**
4.1 Dimensionality Reduction . . . 56
4.1.1 PCA . . . 57
4.1.2 SVD . . . 58
4.1.3 DCT . . . 59
4.2 Random Projection . . . 60
4.2.1 Johnson-Lindenstrauss (JL) Lemma . . . 60
4.2.2 How randomization is done . . . 60
4.2.3 Characteristics of RP . . . 61
4.3 Multiple Random Projections (MRP) in finding Nearest Neighbor . . . 62
4.4 Experimental Results . . . 66

5 Conclusions **75**
5.1 Summary . . . 75
5.1.1 Main Results . . . 76
5.2 Future Work . . . 77
5.2.1 Proof of the new approach . . . 77

5.2.2 Optimal values of different parameters 77
5.2.3 Optimal way of choosing from candidate points 78
5.3 The Last word 78

Bibliography **80**

List of Figures

2.1 Annulus Bound 8

2.2 A 2-dimensional k-d tree 10

2.3 Ball tree 13

2.4 Compare various NN algorithms for Random Uniform data 21

2.5 Compare various NN algorithms for Single Gaussian data with zero mean and unit standard deviation 22

2.6 Compare various NN algorithms for Mixture of 4 and 12 Gaussian data with 0.2 standard deviation 23

2.7 Compare various NN algorithms for Mixture of 4 and 12 Gaussian data with 0.4 standard deviation 24

2.8 Compare various NN algorithms for Mixture of 4 and 12 Gaussian data with 1 standard deviation 25

2.9 Performance of Different Nearest Neighbor on USPS digit data (256D) . 26

3.1 Metric skip list - USPS Digit Data 35

3.2 Metric skip list - Mnist Digit Data 36

3.3 Metric skip list - Feret Data 36

3.4 Metric skip list - Orl Face Data 37

3.5 Metric skip list - Forest Data 37

3.6 Metric skip list - Random Uniform Data 39

3.7 Low intrinsic dimensionality of Random Uniform Data 40

3.8 Metric skip list - Single Gaussian Data 41
3.9 Metric skip list - Mixture of 10 moderately Separated Gaussian Data . . 42
3.10 Kleinberg algorithm - USPS Digit Data 49
3.11 Kleinberg algorithm - Mnist Data 49
3.12 Kleinberg algorithm - Feret Data 50
3.13 Kleinberg algorithm - Forest Data 50
3.14 Kleinberg algorithm - Orl Face Data 51
3.15 Kleinberg algorithm - Random Uniform Data 52
3.16 Kleinberg algorithm - Single Gaussian Data 53
3.17 Kleinberg algorithm - Mixture of 10 Gaussian Data 54

4.1 False Nearest Neighbor in single random projection 63
4.2 Single Random Projection - USPS Digit Data 64
4.3 Single Random Projection - Random Uniform Data 64
4.4 Single Random Projection - Single Gaussian Data 65
4.5 Multiple Random Projection - USPS Digit Data 68
4.6 Multiple Random Projection - Mnist Digit Data 68
4.7 Multiple Random Projection - Feret Data 69
4.8 Multiple Random Projection - Orl Face Data 69
4.9 Multiple Random Projection - Forest Data 70
4.10 Best of MRP VS Early Break : Real Data sets 71
4.11 Best of MRP VS Early Break : Mixture of moderately well separated Gaussians 72
4.12 Best of MRP VS Early Break : Random Uniform Data 72
4.13 Best of MRP VS Early Break : Single Gaussian Data 73

Chapter 1

Introduction

The Nearest Neighbor (NN) Search problem arises in various fields of Computer Science. Basically there are two flavors of the problem: exact NN and approximate NN. The problem is the following: Given a set P of points in a high-dimensional space, construct a data structure which given a query point q finds the point in P closest to q (for exact NN) or a close approximation to the nearest point of q (for approximate NN).

The above problem is of significant importance to pattern recognition, searching in multimedia data, vector compression [1], computational statistics [2], data mining etc. Many of these applications involve data sets which are very large and moreover the dimensionality of the data points can be in the order of hundreds or thousands; both of these factors make it a challenging computational problem.

In this thesis, we have considered the most commonly used settings of nearest neighbor search problem where the points are in $\mathbf{R}^d$ and the distance metric is Euclidean. That is, for points $x = \{x_1, x_2, ..., x_d\}$ and $y = \{y_1, y_2, ..., y_d\}$, the distance between them is defined as

$$d(x, y) = \left(\sum_{i=1}^{d} (x_i - y_i)^2 \right)^{(1/2)}$$

There is quite a handful of literature for the Euclidean nearest-neighbor problem. If the points lie in the plane i.e. $d = 2$, the nearest-neighbor problem can be solved with

$O(\log n)$ per query and with a storage requirement of $O(n)$ [3, 4] using divide-and-conquer paradigm. Unfortunately, as the dimensionality increases such algorithms become less and less efficient. Their space and time requirements grow *exponentially* in the dimension. Many researchers have studied the theoretical aspects of the exact NN problem. For example, Dobkin and Lipton [5] give an upper bound of order $O(2^d \log n)$ with pre-processing time $O(n^{2^{d+1}})$ to answer a nearest neighbor query in $\mathbf{R}^d$ (the term pre-processing refers to the the sum of pre-processing time and storage requirements). Clarkson [6] improved that with a query time $O(2^{\Omega(d)} \cdot \log n)$ and pre-processing $O(n^{\lceil d/2 \rceil (1+\epsilon)})$. Most of the subsequent approaches and extensions (e.g. [7, 8]) have required a query time of at least $\Omega(2^{\Omega(d)} \cdot \log n)$. Meiser [9] has given a solution which is an exception to this phenomenon of the exponential dependence on d. His solution has a query time of $O(d^5 \log n)$ with storage $O(n^{d+\epsilon})$.

The lack of success in removing the exponential dependence on the dimension led researchers to find alternative path for solutions: mainly, whether we can remove the exponential dependence on d if we allow the answers to be *approximate*. The notion of approximation is explained as the following: instead of reporting a point p that is closest to q, the algorithm can report any point within distance $(1 + \epsilon)$ times the distance from q to p. Formally, we say that $p \in P$ is an *ϵ-approximate nearest neighbor* of q if for all $p' \in P$, we have $d(p, q) \leq (1 + \epsilon) d(p', q)$. Another way to define approximate solution is: instead of reporting the closest point p to q, we can report any point if it can be found within any of the K exact NN of q. Settling for an approximate algorithms makes sense as the methods used for mapping features to numerical coordinates in many of the applications have been chosen on heuristic grounds, and so an "approximate" answer may be almost as valuable as an "exact" one.

The ϵ-approximate nearest neighbor problem, for arbitrary small $\epsilon > 0$ has also been studied extensively. Arya, Mount et al. [12, 13] gave an algorithm with query time $O(2^{\Omega(d)} \cdot \epsilon^{-d} \log n)$ and pre-processing $O(n \log n)$. Clarkson [14] obtained an algorithm

where he improves the dependence on ϵ to $2^{\Omega(d)} \cdot \epsilon^{-(d-1)/2}$. But again, these results are still exponentially dependent on d. Kleinberg [15] gave a conceptually simple algorithm with query time $O((\frac{1}{\epsilon^2})(d \log^2 d)(d + \log n))$ and storage $O(n \log d)^{2d}$. Although its query time is polynomial in d, its storage is prohibitively large. An improved version of the algorithm is also presented in the same paper which has a query time $O((\frac{1}{\epsilon^2} \log \delta^{-1})(n + d \log^3 n))$ with near-linear storage but may fail at query time with probability δ.

There are several things to be noted here. First, the brute force (naive nearest) approach which computes the distance from query point to every other point in every dimension has a query time $O(dn)$, is faster (even theoretically) when the dimension d is not less than $\log n$. Also, if we want the storage cost to be polynomial in n (for variable d) then there exists no algorithm that is better than the naive nearest approach once d is comparable to $\log n$ [15]. This phenomenon is known as *curse of dimensionality*. As qouted in Arya, Mount, et al.[13],"... if the dimension is significantly larger than log$n$ (as it for number of practical instances), there are no approaches we know of that are significantly faster than brute-force search."

One another approximate method called Locality Sensitive Hashing (LSH), introduced by Indyk and Motwani, has a worst-case query time of $O(dn^{1/\epsilon})$ [16]. A significant improvement of this technique was later presented in [17] whose running time is $O(dn^{1/(1+\epsilon)})$ which is sublinear in n for any $\epsilon > 0$. The idea of LSH is to hash the points with multiple hash functions so that for each of the hash function, the probability of collision for closer points are much greater than the points which are further apart. The query point is then hashed and NN can be determined by retrieving elements stored at that position. LSH uses l_1 norm rather than l_2 and requires that all coordinates must be positive.

In this thesis, we develop a new approach of finding approximate nearest neighbor by combining the advantages of both exact and approximate nearest neighbor technique. We will use multiple random projections to reduce the dimensionality (an idea from approx. NN) and then use a fast exact nearest neighbor algorithm (such as k-d tree) in

the projections to find the closest point. This scheme has a resemblance with LSH in the sense that it uses multiple random projection instead of multiple hash function to find NN. It appears that this practical method achieves better results in terms of number of distance calculations needed to find the nearest point than other methods including brute force, k-d tree and other approximate algorithms in very high dimensions. The higher the dimension, the greater the savings in distance calculations.

This thesis is organized as follows. Chapter 2 describes the *exact nearest neighbor* techniques and some experimental results applying those methods. Chapter 3 talks about the *approximate nearest neighbors*. In Chapter 4 we present our new idea of combining multiple random projections with one of the exact nearest neighbor approaches and evaluate its performance. Chapter 5 discusses future work and conclusions.

Chapter 2

Exact Nearest Neighbor

The exact nearest neighbor (NN) search problem is defined as the following:

We are given a set P of n sites $\{p_1, p_2, ..., p_n\}$, where each site is a point in $\mathbf{R}^d$. We must pre-process P in such a way so that we can efficiently answer queries of the following form: Given an arbitrary query point $q \in \mathbf{R}^d$, return the site in P that is "closest" to q under a given distance function.

The definition of "closest" we study in this thesis is the Euclidean distance measurement or l_2 norm which is defined in the previous chapter. Mathematically, we want to *find the point $p \in P$ which has the minimum distance from q i.e.*

$$\|q - p\| \leq \|q - p'\| \qquad \forall p' \in P$$

where $\|q - p\|^2 = \sum_{i=1}^{d} (q_i - p_i)^2$.

There are several exact nearest neighbor techniques we will consider:

1. NaiveNearest
2. k-d tree
3. R-tree
4. Ball tree

5. SR-tree

2.1 Naive Nearest

This approach is the most naive approach (it goes by its name). It is a brute force method where we calculate the distance for each point from the query point, in every dimension. The complexity of this method is $O(dn)$. Here, d is the number of dimensions and n is the number of points.

The following is the pseudo-code for finding nearest neighbor of the query point q from the points in P in brute force approach:

Algorithm 1 Nearest Neighbor Search in Naive Nearest (Brute Force) approach

```
nearest ← NA
d_min ← ∞ {initialize minimum distance to ∞ }
for j=1:n do
  d_qj ← 0 {d_qj is the distance between point q and p_j, initialized to 0}
  for k=1:d do
    d_qj ← d_qj + (q^k − p_j^k)^2
  end for
  if (d_qj < d_min) then
    nearest ← p_j {Store the nearest point so far}
    d_min ← d_qj
  end if
end for
output nearest
```

2.1.1 Naive Nearest with Early Break

The Early Break (EB) strategy stems from the observation that, for Euclidean distance metric (or for any distance metric which is additive across dimensions), we do not need to compute distances in all the dimensions. When the distance between a point p and query point q exceeds the minimum distance found so far, we can safely discard that point because it can't be the closest one. In fact, this strategy exhibits significant saving

in distance calculation that is needed to find the nearest point. The complexity of the method is still $O(dn)$ though.

The Early Break strategy calculates distance in the following way:

Algorithm 2 Nearest Neighbor Search in Naive Nearest with Early Break strategy

```
d_min ← ∞ {initialize minimum distance to ∞}
nearest ← NA
for j=1:n do
  d_qj ← 0 {d_qj is the distance between point q and p_j, initialized to 0}
  for k=1:d do
    d_qj ← d_qj + (q^k − p_j^k)^2
    if (d_qj > d_min) then
      break; {break the loop when the distance exceeds the minimum distance}
    end if
  end for
  if (d_qj < d_min) then
    nearest ← p_j {Store the nearest point so far}
    d_min ← d_qj
  end if
end for
output nearest
```

In algorithm 2, the first *if* statement in 2nd *for* loop checks whether the partial distance between query point q and p_j exceeds the minimum distance found so far from q. If it does then it exits the loop.

2.1.2 Naive Nearest with Annulus Bound

The Annulus Bound (AB) method [24] is a nearest neighbor search algorithm designed specifically for Vector Quantization. It is based on geometric observation. As shown in the figure 2.1, any point p_i that is closer to query point q than p_g must satisfy

$$\|q\| - r \leq \|p_i\| \leq \|q\| + r$$

where $r = \|q - p_g\|$. This constraint defines a annular region.

To implement this algorithm, we maintain a list of points sorted by their norm. We then guess a point p_g. To find the closest point, we enumerate those points, whose norms

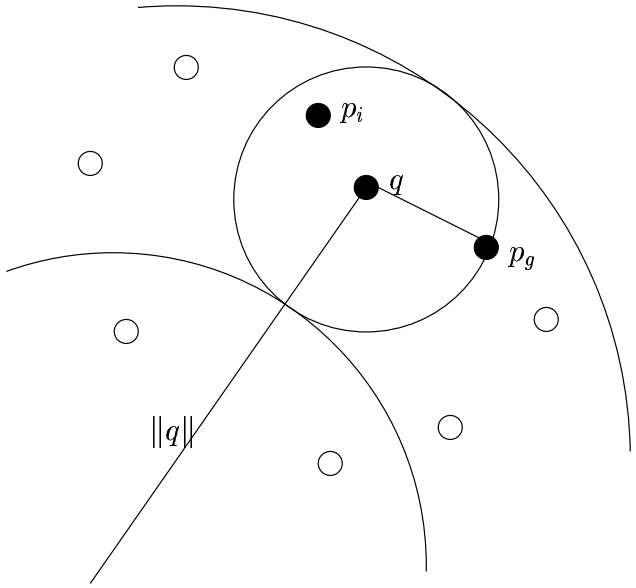


Figure 2.1: In Annulus Bound method if a point p_i is closer to the query point q than another point p_g then $\|p_i\|$ lies between $\|q\| - r$ and $\|q\| + r$ where $r = \|q - p_g\|$.

lie in the range $[\|q\| - r,\ \|q\| + r]$. We shrink the radius r of this search as we find closer points and iterate the procedure again (although, in practice, this shrinking of radius has little impact on the performance [24]).

The running time of this algorithm is still $O(dn)$ but proportional to the number of points which lie in the annulus. Algorithm 3 describes how NN search is performed in Annulus Bound method.

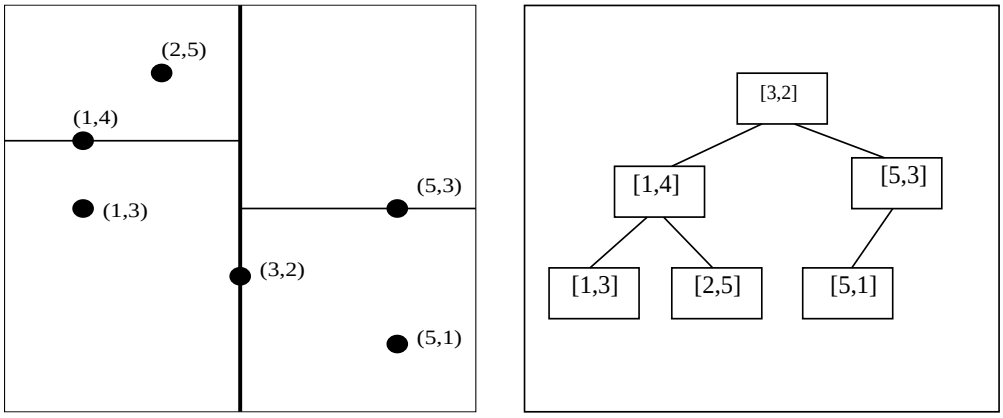
2.2 k-d tree

The k-d tree [18, 19] is a multidimensional binary tree data structure that supports nearest neighbor and other spatial searches.

Both leaf and internal node of the k-d tree store points which are d-dimensional vectors. An internal node also stores an index i (which is the index of the dimension of the point stored in that internal node) and a value s (which is the value of that dimension) that are used to "split" the nodes into left and right subtrees. The left subtree contains those nodes where the value of its ith coordinate is less than or equal to s, the right

Algorithm 3 NN search in Naive Nearest with Annulus Bound

Find norm of all n points from a reference point & sort them.
$r \leftarrow \|q - p_g\|$ {p_g is the guessed point}
$d_{min} \leftarrow \infty$ {initialize minimum distance to ∞}
$nearest \leftarrow$ NA;
done $\leftarrow$ *false*
while (not done) **do**
  $l \leftarrow \|q\| - r$
  $u \leftarrow \|q\| + r$
  Let l_i and u_i denote the lower and upper index of points whose norm lie between l & u and set done $\leftarrow$ *true*
  for $j = l_i : u_i$ **do**
    $d_{qj} \leftarrow 0$ {d_{qj} is the distance between point q and p_j, initialized to 0}
    for k=1:d **do**
      $d_{qj} \leftarrow d_{qj} + (q^k - p_j^k)^2$
      if $(d_{qj} > d_{min})$ **then**
        break;
      end if
    end for
    if $(d_{qj} < d_{min})$ **then**
      $nearest \leftarrow p_j$ {Store the nearest point so far}
      $d_{min} \leftarrow d_{qj}$
      $r = \text{sqrt}(d_{min})$
      done $\leftarrow$ *false*
      break;
    end if
  end for
end while
output *nearest*



(a) (b)

Figure 2.2: A 2-dimensional k-d tree. The filled circles in (a) are the 2 dimensional vectors. The [3,2] node splits the vectors along the line $x = 3$. All vectors in left subtree has x coordinate less or equal to 3 and the right subtree contains all vectors with x coordinate greater than 3. The corresponding tree diagram is shown in (b).

subtree contains those nodes where the value of its ith coordinate is greater than s. For 2-dimensional vectors, all the internal nodes partition the data along a line which is perpendicular to one of the axis. In figure 2.2(a), the internal node [3,2] has $i = 1$ and $s = 3$ and it splits the vectors of the relevant subtree along the line $x = 3$. The left subtree consists of all the vectors whose x-coordinate is less than or equal to x, and the rigth subtree consists of all the vectors that have their x-coordinate greater than x. Likewise, the [1,4] node splits the vectors along the line $y = 4$ and [5,3] node splits along the line $y = 3$. Figure 2.2(b) shows the corresponding tree diagram of these points. In higher dimension, for each internal node, the data is partitioned along a *splitting plane* that is perpendicular to some axis. In this thesis, we've constructed the k-d tree as described in [25]. A balanced (an optimization of a tree which aims to keep equal numbers of items on each subtree of each node so as to minimize the maximum path from the root to any leaf node) k-d tree of height $\lceil \log_2 n \rceil$ can be built in $O(dn\log n)$ time and uses $O(dn)$ space [18].

2.2.1 Searching in k-d tree

The nearest neighbor search algorithm for k-d tree is performed by applying the algorithm presented in [25]. The search starts at the root of the tree and proceeds down towards the leaves recursively. At a leaf node, the distance between the query vector and the vector stored in that leaf is computed. At an internal node, we first search recursively down to find the subtree where the query vector falls in. If the ith coordinate of the query vector is less than or equal to s, we search down the left subtree, and if it is not, we search down the right subtree. When the recursive call returns, we check whether we have to search the other subtree as well. If the distance between the query vector and the splitting plane is less than the distance between the query vector and the closest point found thus far, then it is possible to find a vector which is closer to the query point in the other subtree. If that is the case, then we must search the other subtree as well.

It is clear that on average, at least $\Omega(\log n)$ nodes must be searched because any nearest neighbor searching requires traversal to at least one leaf from root. It is also clear that no more than n nodes are searched in the worst case. Friedman, Bentley and Finkel [18] have shown that, the amount of backtracking needed is independent on n. While this cost is independent of n, it is exponentially dependent on d, the dimensionality of the data vectors [25]. Thus, when the dimensionality of the data vectors is low, the expected number of vectors searched in k-d tree is $O(\log n)$.

2.3 R-tree

An R-tree [21] is a height balanced tree. The index records in the leaf nodes contain pointers to data objects. Every leaf node of an R-tree contains an index record of the form *(I,tuple-identifier)* where I is a d-dimensional rectangle which is the bounding box of the data object indexed $I = (I_1, I_2, ..., I_d)$. Here, d is the number of dimensions and I_i is the closed bounded interval $[a, b]$ describing the extent of the object along dimension

i.

Non-leaf nodes contain entries of the form *(I,child-pointer)* where *child-pointer* points to a lower node in the R-tree and I is the minimum bounding rectangle of all the rectangles of the node's children.

An R-tree satisfies the following property:

- Every leaf node contains maximum of M and minimum of $m \leq \frac{M}{2}$ index records. Every non-leaf node contains between m and M children unless it is the root. The root has at least 2 children.
- For each leaf entry *(I,tuple-identifier)* and for each non-leaf entry *(I,child-pointer)*, I is the smallest rectangle that spatially contains the rectangles in the child node.
- All leaves appear at the same level.

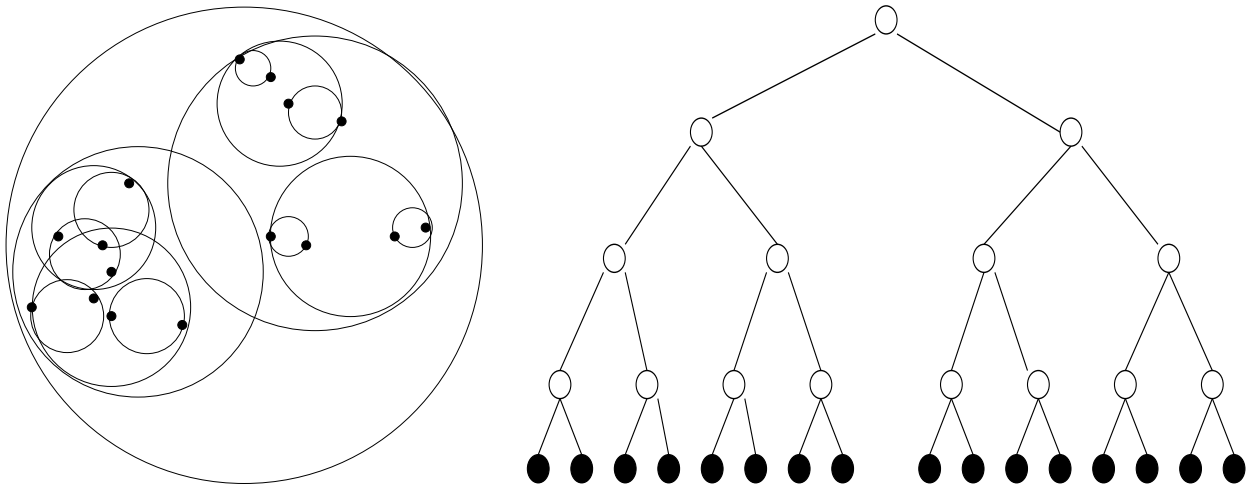
The height of an R-tree containing n index records or n points is at most $\lceil \log_m n \rceil - 1$, because the branching factor of each node is at least m. The number of non-leaf nodes in an R-tree is maximum when all the nodes possess at most m children. So, other than the root, all other non-leaf nodes have worst case space utilization when they have m children i.e. the worst case space utilization for all nodes except root node is $\frac{m}{M}$. When the nodes tend to have more than m entries, the tree height will decrease and improve space utilization.

An algorithm for inserting a new data vector in an R-tree is given in [21]. Inserting a new data vector in an R-tree depends on choosing the leaf node where the vector is to be inserted and invoking of node splitting algorithm if there is no room for the new vector on that chosen leaf node. The choosing of leaf node takes $O(d \log_m n)$ time. The split node algorithm that we have used here is quadratic in M and linear in d [21].

The search algorithm of an R-tree can be implemented as [26]. This is a branch-and-bound traversal algorithm. If the ordering and pruning heuristics are chosen well, they can significantly reduce the number of nodes visited in a large search space [26].

2.4 Ball Tree

The Ball tree [22] is a simple geometric data structure and well suited to geometric learning tasks. The Ball tree structure is related to other hierarchical representations such as k-d trees, octrees [23], SS tree [20]. According to [22], a *ball* is referred to be a region bounded by a hypersphere in d-dimensional Euclidean space $\mathbf{R}^d$. The balls are represented by d+1 floating point values which specify the coordinates of its center and the length of its radius. A *Ball tree* is a complete binary tree. In a Ball tree, every node is represented as a ball in such a way that each interior node's ball is the smallest which contains the balls of its children.



(a) (b)

Figure 2.3: Ball tree structure. The filled circles in (a) are leaves of the tree which contain the d dimensional points. Every other ball represent an interior node which is the smallest ball that contains the balls of its children. The corresponding binary tree diagram is shown in (b).

The interior nodes are used to guide efficient search through the leaf structure. Sibling regions in a Ball tree are allowed to intersect and they do not need to partition the entire space.

2.4.1 Constructing a Ball tree

We've used the k-d construction algorithm of Ball tree as given in [22]. It is an off-line top down construction algorithm. At each stage of the algorithm the leaf balls are split in to two sets from which Ball trees are recursively built. These trees become the left and right children of the parent node. To split the balls, we choose a dimension and a splitting value along this dimension. The left child contains the leaf balls whose center has a coordinate in the given dimension which is less than the splitting value and the right child contains those in which it is greater. The dimension to split on is chosen to be the one in which the balls are most extended and the splitting value is chosen to be the median. Because median finding is linear in number of samples and there are $\log n$ stages, the whole construction algorithm takes $O(dn \log n)$ [22]. The tree that is produced by this method is completely balanced but may not adapt itself well to any hierarchical structure of the leaf balls.

2.4.2 Searching in a Ball tree

In searching the nearest neighbor in a ball tree, a ball "qBall" is maintained whose attribute "ctr" contains query point and attribute "r" contains the minimum distance found so far (it is initialized to ∞ at the start of the algorithm). The internal nodes whose ball doesn't intersect with the "qBall" are pruned away. Otherwise we recursively search that node. In the search algorithm for Ball tree in algorithm 4 the function *dist* calculates the distance between two ball centers. The function NNSearch takes a node parameter "N" which has pointers "lt" and "rt" as its children.

2.5 SR-tree

The SR-tree [27] (Sphere/Rectangle-tree) was proposed as an improvement to SS-tree [20] and R*-tree [28]. While the SS-tree uses bounding sphere and R*-tree uses bound-

Algorithm 4 Nearest Neighbor Search in ball tree

```
function NNSearch(N)
if N is a leaf then
  D ← dist(qBall.ctr, N.ball.ctr)
  if D < qBall.r then
    qBall.r ← D
  end if
else
  LD ← dist(N.lt.ball.ctr, qBall.ctr)
  RD ← dist(N.rt.ball.ctr, qBall.ctr)
  if D > qBall.r and RD > qBall.r then
    do nothing
  else
    if LD ≤ RD then
      NNSearch(N.lt)
      if RD < qBall.r then
        NNSearch(N.rt)
      end if
    else
      NNSearch(N.rt)
      if LD < qBall.r then
        NNSearch(N.lt)
      end if
    end if
  end if
end if
```

ing rectangles to build the tree, a region of the SR-tree is specified by the intersection of a bounding sphere and a bounding rectangle. Hence, SR-tree captures the advantages of both of its predecessors. A node of SR-tree has the following structure $N = (S, R, w, child - pointers)$: a bounding sphere S, a bounding rectangle R, the total number of points w contained in the subtree of the node and pointers to its children *child-pointers*. If the node is a leaf node then it points to entries (or points) $p_1, p_2, ..., p_k$ where $m \leq k \leq M$. Here, m and M are the minimum and maximum number of entries in a leaf. If the node is a non-leaf node then it points to the nodes $N_1, N_2, ..., N_k$ where $m \leq k \leq M$ as before. The algorithm to build the SR-tree is given in algorithm 5 and algorithm 6.

The branch-and-bound algorithm that is used for R-tree searching in [26] can also be used for SR-tree searching with some small modification in the way of computing the distance from a search point to each region in the SR- tree [27]. It is a depth-first search with two stages. At first stage it collects the given number of points to make a candidate set. Secondly, it revises the candidate set with visiting every leaf whose region overlaps the range of the candidate set. For details about the search method see [26, 27].

2.6 Experiments with Exact NN Algorithms

In this section, we will describe the experimental methodology used in this thesis. We also describe various kinds of data sets and discuss some of their properties.

2.6.1 Experimental methods

It is hard to find a completely fair method to compare different nearest neighbor (NN) algorithms. Some algorithms have very large pre-processing overhead. Also, these algorithms use different operations as the principal mean of finding nearest neighbor. One method of comparing different nearest neighbor algorithms is cpu time which is largely

Algorithm 5 Insert a point $p \in R^d$ in SR-tree

function Insert(Point p)
N = ChooseSubTree(*root*, p); {Returns the subtree where the new node will be inserted}
split = AddRecord(N);{ Check if there is any space for new node, otherwise split the parent node}
while $split == true$ **do**
  split = AdjustTree(N);{If split needed then adjust the tree. Return true if further split is required}
  if $split == true$ **then**
    $N \leftarrow N->parent$; {If split is done then move upward}
  end if
end while
End Insert

function ChooseSubTree(Node N, Point p) return Node
while N is not a leaf **do**
  Find the child N' in N which is the nearest to p;
  $N \leftarrow N'$;
end while
return N
End ChooseSubTree

function AddRecord(Node N, Point p) returns boolean
if N has space for another child **then**
  Insert p as N's child;
  Adjust the parent nodes upto root to accomodate changes;
  $split \leftarrow false$;
else
  $split \leftarrow true$;
end if
End AddRecord

Algorithm 6 Algorithm for adjusting the tree in SR-tree

```
function AdjustTree(Node N) returns boolean {Called when a split is necessary}
Find the coordinate variance on each dimension from the centroids of its children and
choose the dimension with highest variance for splitting;
Split the Node into two nodes N1 and N2;
if N == root then
  Make new root and set its childpointers to N1 and N2;
  split ← false;
else
  if N− > parent has no space due to splitting then
    split ← true;
  else
    Adjust the parents upto root to enclose new nodes.
  end if
end if
End AdjustTree
```

implementation dependent. Various authors have used various comparison methods such as number of disk access, cpu time etc. In our experiments, we have used number of distance calculations needed as the method of comparison. Though the algorithms we've tried have other overheads, they mainly use the distance calculation between query point and some of the other points in the data space as one of the basic measurement units.

2.6.2 Data sets

In our thesis, we've used several synthetic data sets as well as real life data sets. The synthetic data sets are:

- Random uniform data in the range $[0, 1]^d$.
- Random Gaussian data with zero mean and unit standard deviation.
- Mixture of Gaussians where the clusters are well separated. The centers of all the clusters are also Gaussian distributed with zero mean and unit standard deviation. The clusters are made well separated by means of generating data points for each cluster from a Gaussian with low standard deviation (e.g. 0.2 or lower). This value

of standard deviation doesn't depend on the dimension of the data set.

- Mixture of Gaussians where the clusters are overlapping and the centers have distributions same as the previous category. Overlapping clusters can be generated by generating points for each cluster from Gaussians with high standard deviation (such as 1). As before, this value of standard deviation doesn't depend on dimension.

Other than these, the real life data sets that we've used are the followings:

- **USPS digit data:** This data set contains pictures of handwritten digits and has 256 dimensions for each of the data points. There are 7291 data points and 2007 query points in the data set. The values of the data and query points are normalized between 0 and 1 on each dimension.

- **Mnist digit data:** This data set also contains pictures of digits. Each data point has 784 dimensions. There are around 6000 data points for each digit and a total of around 60,000 data points. For our experiment, we've taken a total of 8000 data points (800 points from each digit) and 2000 query points (200 from each digit) at random.

- **Feret data:** The Feret data set has 3816 face images. Of these, we've used 3216 images as the data points and the rest (600) as query points. The raw Feret face images were passed through a multi-stage normalization process. For information about how the normalization is done see http:// www.cs.colostate.edu/ evalfacerec/ data/ normalization.html. This normalized data has 18,000 dimensions. This data set is further modified to exclude those dimensions where the corresponding mask values in a mask file have intensity less than some threshold value. This preprocessing reduces the dimensionality of data to 17154. This is the dimension of the data we've worked with.

- **Olivetti Research Laboratory (ORL) faces data:** This data set is rather small. It has 400 face images from which we've chosen 360 images as data points and the rest as query points at random. We took this data set as it is also very high dimensional. Each data point in this data set has 10304 dimensions.

- **ForestCover data:** The data set is taken from http:// kdd.ics.uci.edu/ databases/ covertype/ covertype.html. This data set has 54 dimensions. 40 of them are binary and contain mainly zeros. The rest of the dimensions are normalized so that for each dimension the minimum value is 0 and the maximum is 1. There are approximately half a million data points in the set. For our purpose we've taken 10000 data points and 2000 query points at random.

2.7 Experimental Results

In this section, we've compared various nearest neighbor techniques w.r.t brute force (naive nearest) approach. Naive nearest approach needs the distance calculation to be done for all points in all dimensions. Hence, considering it needs 100 % distance calculation, we check how other exact algorithms fare with the increase in dimensions.

In these experiments, we've mainly used the synthetic data sets. In the figures 2.4 to 2.8, the horizontal axis denotes dimension in log scale and the vertical axis denotes percent distance calculation needed. The maximum dimension of the data we've used here is 512. Beyond this point, some exact nearest neighbor algorithms need prohibitively large memory. We've used uniform random data set in $[0, 1]^d$, single Gaussian data set and mixture of 4 and 12 Gaussians (where we've varied their standard deviation from 0.2 to 1). When the standard deviation is low the Gaussians are considered more separated than their higher standard deviation counterpart. We've generated 5000 data points and 500 query points for all the data sets in each of the dimension we've experimented with. In each specific dimension, the percentage distance calculation is measured by averaging

over 15 data sets.

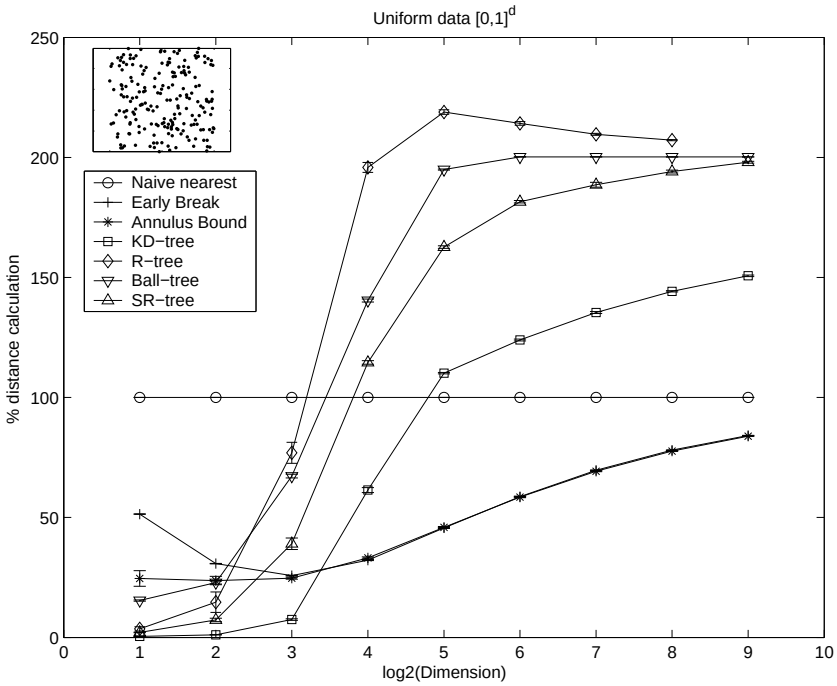


Figure 2.4: Comparing various NN algorithms with Random Uniform data (averaged over 15 data sets for each of the algorithms in the specific dimension). The horizontal axis denotes dimension of the data (in log scale) and vertical axis denotes the cost (in % distance calculation w.r.t. naive nearest algorithm) of various NN algorithms.

In figure 2.4 with random uniform data, we see that in low dimension (upto 8) kd-tree works better than any other methods. Even SR-tree, Ball tree and R-tree work better than Early Break or Annulus Bound when data are not more than 4 dimension. With the increse in dimension, the tree based algorithms tend to deteriorate quickly. And in high dimension, they become more expensive than brute force approach.

Using single Gaussian data with zero mean and unit standard deviation gives approximately the same results. In figure 2.5, as the data points are random Gausssian with no specific structure (i.e. all the dimensiona are equally important and the data is not lying in any low dimensional manifold), the tree algorithms can't do better than the naive nearest approaches in high dimensions.

From the figures 2.6 to 2.8, we can see that for mixture of Gaussians models:

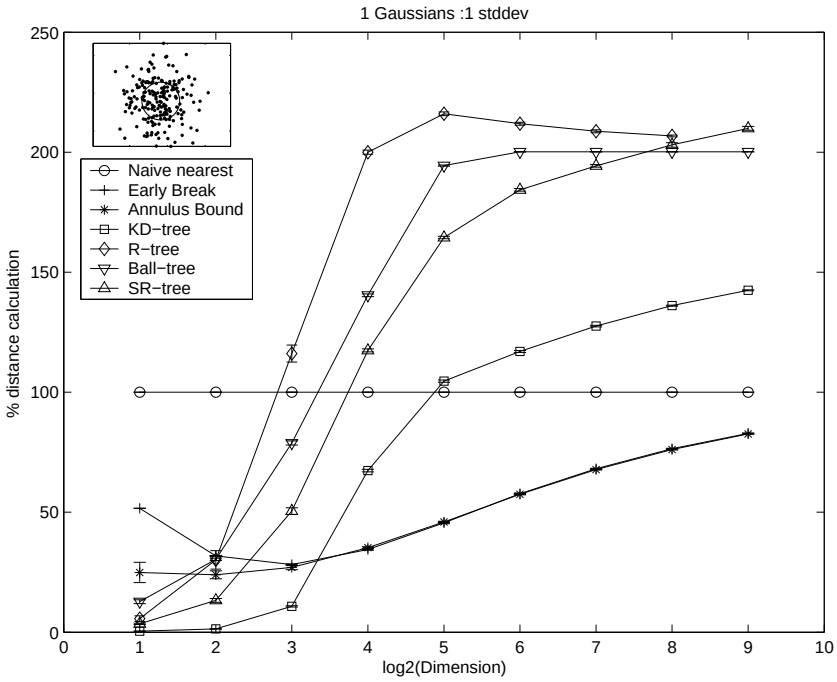
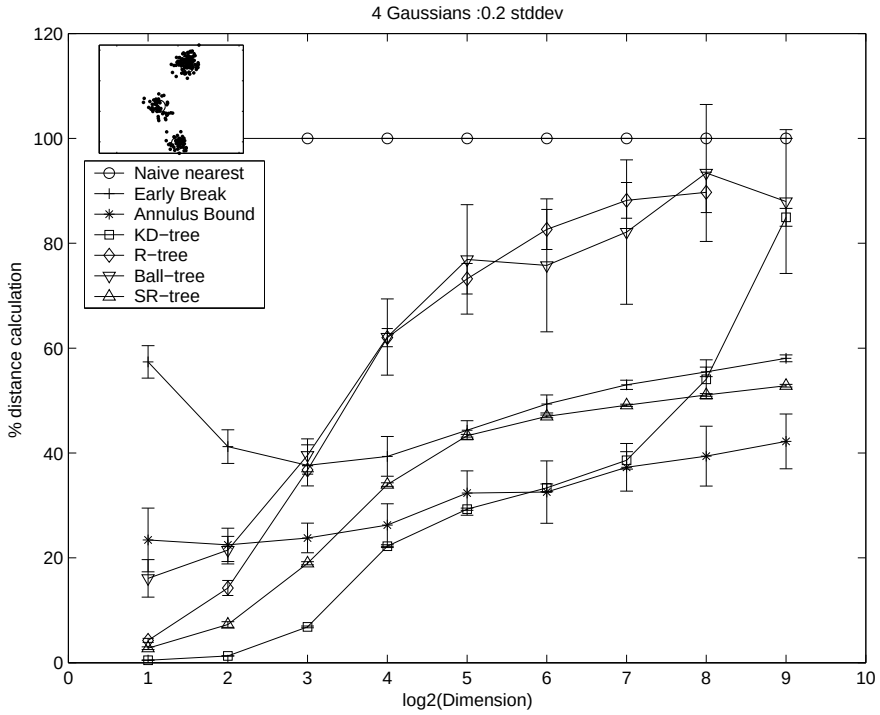
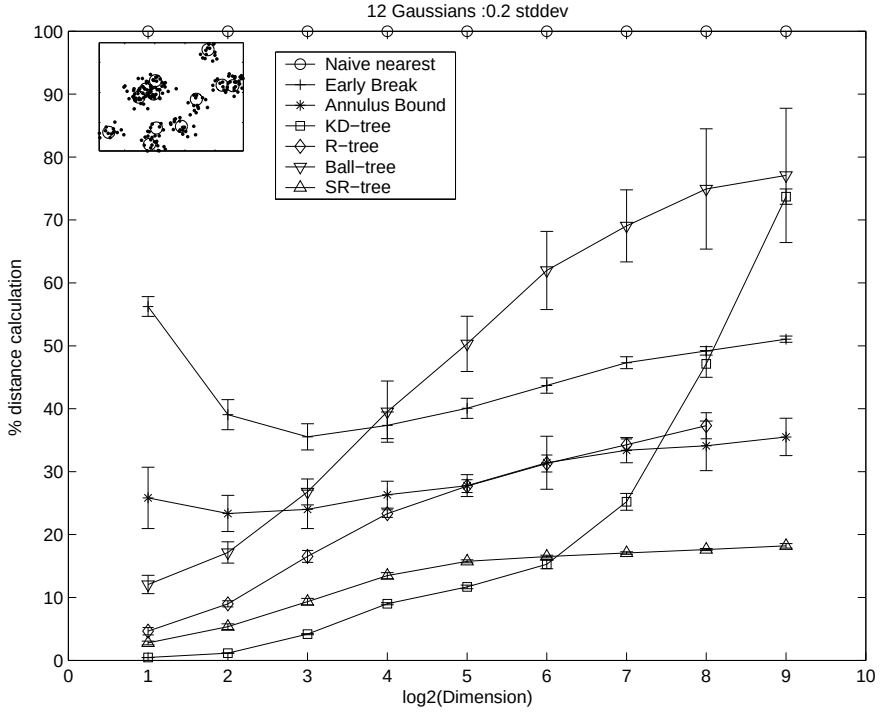


Figure 2.5: Comparing various NN algorithms for Single Gaussian data with zero mean and unit standard deviation (averaged over 15 data sets for each of the algorithms in the specific dimension). The horizontal axis denotes dimension of the data (in log scale) and vertical axis denotes the cost (in % distance calculation w.r.t. naive nearest algorithm) of various NN algorithms.

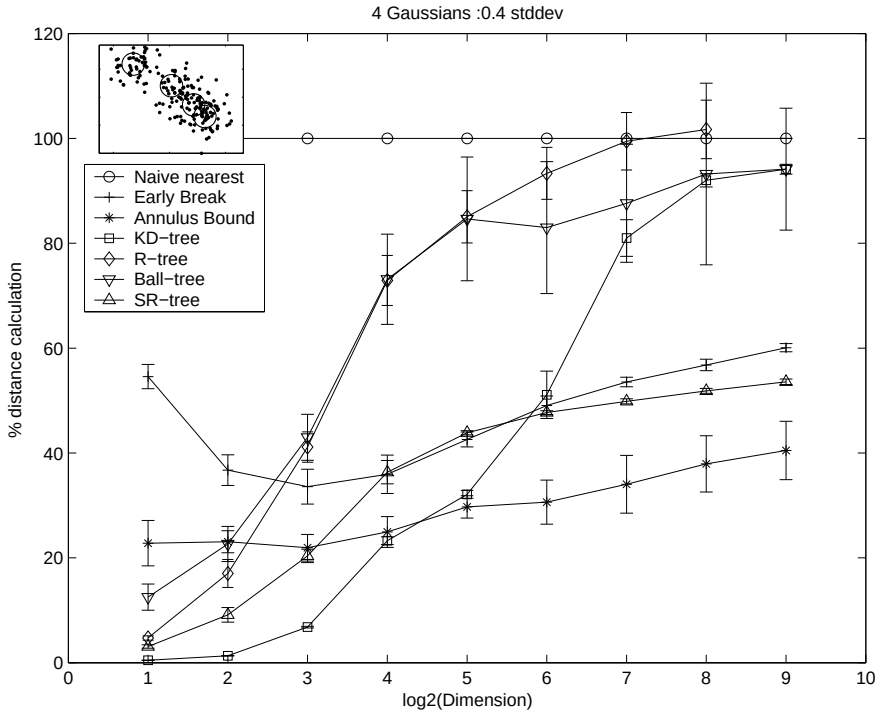


(a) Mixture of 4 Gaussians with 0.2 standard deviation

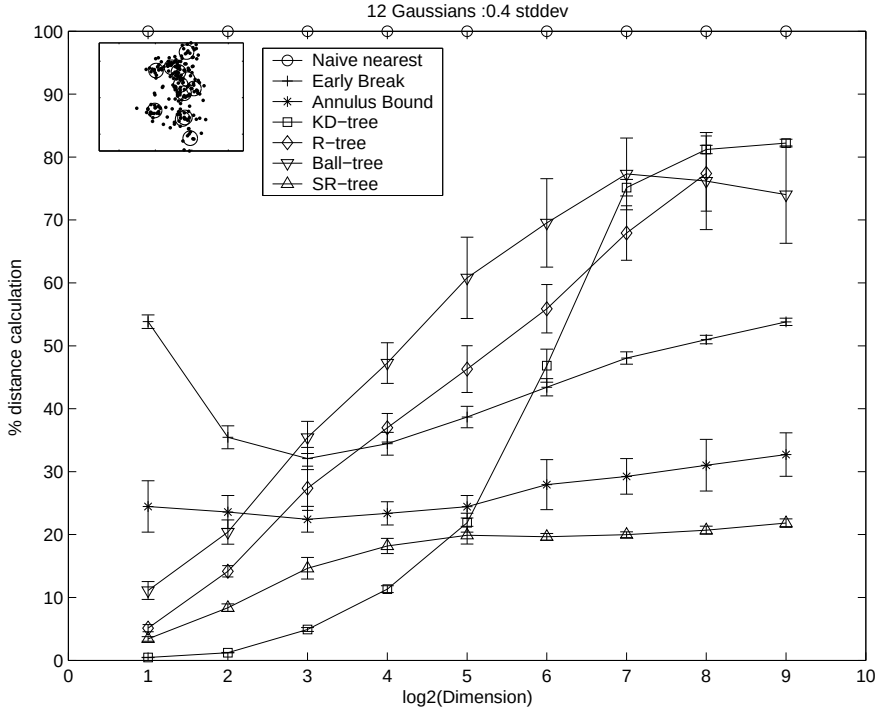


(b) Mixture of 12 Gaussians with 0.2 standard deviation

Figure 2.6: Comparing various NN algorithms for Mixture of a) 4 Gaussian data b) 12 Gaussian data with 0.2 standard deviation (averaged over 15 data sets for each of the algorithms in the specific dimension).

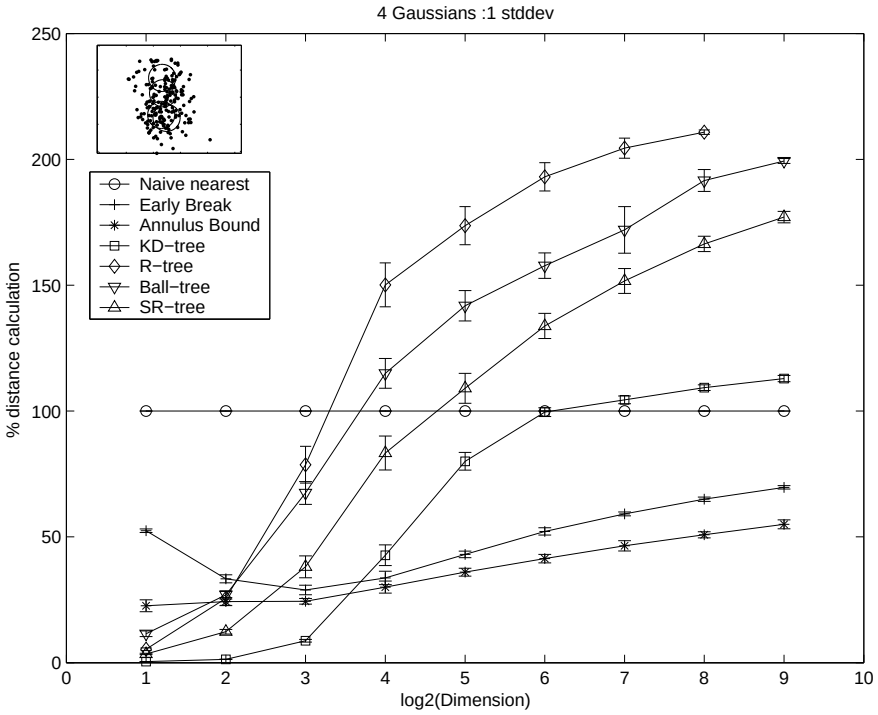


(a) Mixture of 4 Gaussians with 0.4 standard deviation

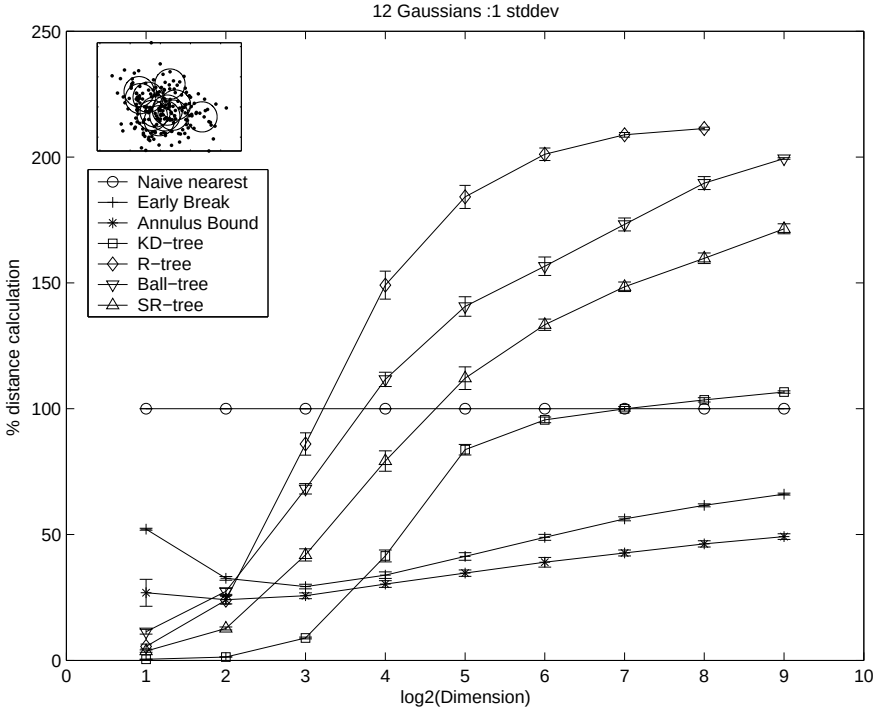


(b) Mixture of 12 Gaussians with 0.4 standard deviation

Figure 2.7: Comparing various NN algorithms for Mixture of a) 4 Gaussian data b) 12 Gaussian data with 0.4 standard deviation (averaged over 15 data sets for each of the algorithms in the specific dimension).



(a) Mixture of 4 Gaussians with 1 standard deviation



(b) Mixture of 12 Gaussians with 1 standard deviation

Figure 2.8: Comparing various NN algorithms for Mixture of a) 4 Gaussian data b) 12 Gaussian data with 1 standard deviation (averaged over 15 data sets for each of the algorithms in the specific dimension).

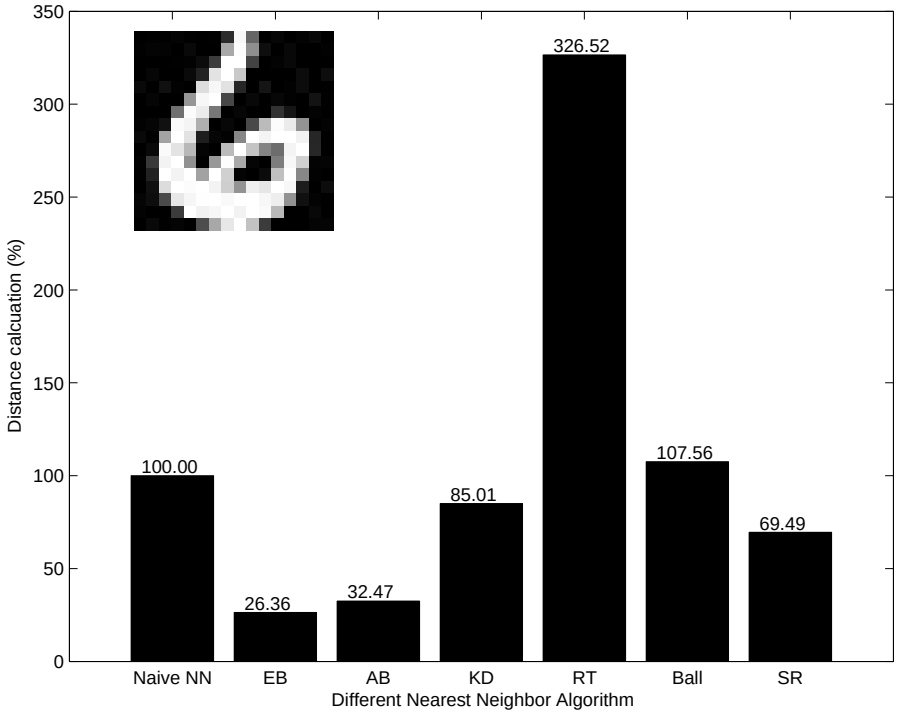


Figure 2.9: Performance of Different Nearest Neighbor on USPS digit data (256D)

- For well separated clusters (e.g. mixture of Gaussians with 0.2 standard deviation) and in low dimension, kd-tree is better than SR-tree and other tree structures, but gets worse than SR-tree in very high dimension (128D or higher). The good performance of the tree structures is due to the fact that when the standard deviation is small, only a small fraction of the 'true' space is filled. Hence, the exact NN techniques of the tree based algorithms show better performance than naive nearest approaches.

- For moderately separated clusters (e.g. mixture of Gaussians with 0.4 standard deviation) kd-tree is still better than SR-tree in low dimension but tends to get more expensive quicker (in approx. 20D - 30D). The fraction of the 'true' space that is filled up by the data set gets bigger and performance of tree structures deteriorates.

- For overlapping clusters (e.g. mixture of Gaussians with 1 standard deviation) kd-

tree works better than all other tree structures. SR-tree behaves badly in higher standard deviation. In these cases, naive nearest approach (e.g Early Break strategy) performs better than the structural tree approaches because the data is no longer clustered in small space. Therefore k-d tree and other tree algorithms can't work well.

We also tried USPS digit data with different exact nearest neighbor techniques. All the other real data sets have very high dimensionality which prevents us from applying the tree based exact nearest neighbor techniques on them. From figure 2.9, we see that in 256 dimensions, the tree structures have very large overhead. The naive nearest approaches (Early Break and Annulus Bound method) are better than all the specialized tree based algorithms.

If the data set is completely random with no special structure then in very high dimension no exact algorithm can be expected to do better than the brute force approach. As we have discussed in the previous chapter, the space and time complexity becomes exponential for exact nearest neighbor techniques where the data actually lies in high dimensions. This phenomenon has been mentioned by many authors, which can be found in various literartures on this topic [13, 10, 15].

2.8 Curse of Dimensionality

From the above results, we see that dimensionality plays a big part on the efficiency of the algorithms. The k-d tree, SR-tree, Ball tree and other related data structures work nicely in small dimension (e.g. in the range of 8-10 dimensions). But these structures start deteriorating when the dimensionality gets higher. Of course, we know that real life data can be of many dimensions (e.g. hundreds, even thousands) and using exact nearest neighbor algorithm on these data would make it computationally very expensive. This is one of the main motivation of looking on *Approximate Nearest Neighbor* rather than

an exact one. Also many real life problems don't actually need to find an exact nearest neighbor. A good approximation is sometimes sufficient enough to do the job. We will look at various approximation technique in the next chapter.

Chapter 3

Approximate Nearest Neighbor

It is natural to try improving the computational requirements by only looking for an approximate nearest neighbor for the query. We define approximate nearest neighbor as follows:

$p \in P$ is an ϵ-approximate nearest neighbor of q if for all $p' \in P$, we have

$$d(p, q) \leq (1 + \epsilon)d(p', q)$$

Here $d(.,.)$ is the Euclidean distance between two points i.e. $d(p, q) = \sqrt{\left(\sum_{i=1}^{d}(p_i - q_i)^2\right)}$

In previous chapter, we said that "curse of dimensionality" is the biggest problem for exact nearest neighbor techniques. When the intrinsic dimension of data is very large, the algorithms for finding exact nearest neighbor become computationally very expensive, and no better than the brute force approach.

One of the common methods of approximation is to reduce the dimension of the data somehow (by random projection, PCA etc). But there are several methods for finding approximate nearest neighbor which do not depend on dimensionality reduction technique. For our experiments, we've tried two such methods for comparison. One is Metric Skip Lists [29] proposed by David Karger and Matthias Ruhl which can answer nearest neighbor queries in $O(\log n)$ time and other is an ϵ-approximate algorithm which has a near linear storage and a query time that improves asymptotically on linear search

in all dimensions by Jon M. Kleinberg [15]. Following is a brief description of both the methods:

3.1 Metric Skip Lists

Metric skip list is a nearest neighbor algorithm designed for general metric spaces. These metrics must have an underlying property that they are *growth-constrained.* By growth-constrained we mean that for any point p and number r, the ratio between number of points in balls (centered at p) of radius $2r$ and r is bounded by a constant which is referred as expansion rate, c.

The algorithm and the data structure can be best understood from [29]. In this thesis, we've used the algorithm with Euclidean metrics, and the data sets are believed to lie on a submanifold of $\mathbf{R}^d$.

According to the terminology used in [29], the concept of growth constrained metrics is the following: Given a metric space $M' = (M, d)$ (here M is the space where the data live and d is a distance function on that space which is symmetric and satisfies the triangle inequality property), and a subset $S \subseteq M$ of n points in the space, and let $B_p(r)$ $:= \{s \in S |\ d(p, s) \leq r\}$ be the ball of radius r around a point p in S, we say that S has a (ρ, c)-expansion iff for all points $p \in M$ and $r > 0$

$$|B_p(r)| \geq \rho \Longrightarrow |B_p(2r)| \leq c \cdot |B_p(r)|$$

So, the expansion property requires that when a ball grows around any point in M, points from S appears in the space at a constant rate.

From the triangle inequality property we can deduce that if $d(p, q) \leq r$, then $B_q(r) \subseteq B_p(2r) \subseteq B_q(4r)$ [29]. The basis of Metric Skip list algorithm lies on the following Sampling Lemma which is quoted from [29].

Sampling Lemma:

Let $M' = (M, d)$ be the sampling space, and $S \subseteq M$ be a subset of size n with (ρ, c)

expansion. Then for all $p, q \in S$ and $r \geq d(p, q)$ with $|B_q(r/2)| \geq \rho$, the following is true: When selecting $3c^3$ points in $B_p(2r)$ uniformly at random, with probability at least $9/10$, one of these points will lie in $B_q(r/2)$.

The metric skip list follows from the sampling paradigm. Below we discuss the construction of the data structure of this algorithm.

3.1.1 Data Structure

Let S be the sample space. The points in the space are randomly ordered and given that ordering the construction of the data structure will be deterministic. Let $S = \{s_1, s_2, ..., s_n\}$ be the random ordering and let s_{i+1} be the successor of s_i and s_1 be the successor of s_n. So, we can imagine that the points are arranged on a circle. For each $s_i \in S$ we have sets of pre-chosen samples. The data structure simply consists of these sets of samples. These sets are called finger-lists. The following definition from Karger *et al.* [29] states how the finger lists are created for each element s_i in S.

"For $r \geq 0$, the radius r finger list for s_i, denoted $F_r(s_i)$, contains the indices of first $24c^3$ elements after s_i in the ordering that have a distance $\leq r$. If we reach the end of ordering we wrap around at beginning. And if there are less than $24c^3$ elements of this kind in S, then $F_r(s_i)$ just contains all of them".

In the next section we'll give an off-line construction of this data structure and describe how searching is performed with this data structure.

3.1.2 Off-line Construction

For off-line construction of the data structure lets suppose we truncate all the finger lists $F_r(s_i)$ so that they do not wrap around at the end of ordering. Rather they only consists of elements after s_i. This structure would still support nearest neighbor searches provided we start the search at s_1, the beginning of the ordering.

The construction starts with an empty data structure and then repeatedly adds

$s_n, s_{n-1}, \cdots, s_1$ to the beginning of the previously constructed data structure. By this way, when we insert s_i in the data structure, we have to compute the finger lists of s_i only. The finger lists of already added elements remain unchanged because they can't contain s_i in this way. At any iteration, the data structure maintains the characteristics of a metric skip list on the subset of items $\{s_{i+1}, s_{i+2}, ..., s_n\}$.

For construction of the finger-list of new element s_i, we start search at s_{i+1}, the successor of s_i and we keep $24c^3$ closest elements seen so far. When we find a new element which is closer to s_i, we drop the element which is furthest from s_i among all the elements in the set. We keep on searching for closer elements until we've done checking all the elements up to s_n. We then add those dropped elements at the end of the list. This yields all the elements of s_i's finger-lists (for proof see [29]). The algorithm for finger list construction of any element is given in algorithm 7 below.

3.1.3 Searching

To find the nearest neighbor for the query point q, we start search at the beginning of the ordering (at s_1), and continue until we finish searching s_n. We initialize s_1 to be the nearest neighbor of the query point q. Then in each iteration, we measure the distance r between q and the current element s_i. We check the finger list of $F_{2r}(s_i)$ to see if there are other elements which are nearer than the current minimum or which halves the distance between q and s_i. If there are multiple elements with any of these properties then we take the one which has the smallest index among them and update the nearest point accordingly. Otherwise, if we can't find any element then we simply choose the element which has the maximum index in $F_{2r}(s_i)$ and iterate again.

This algorithm accesses only $O(\log n)$ finger lists with high probability (for proof see [29]). The unbounded number of possible values of r prevents us from storing the finger lists indexed by r. So, we store the finger lists of an element in an array ordered by r. Therefore, the straight forward approach to find the correct list is to perform a binary

Algorithm 7 Creating Finger Lists for all entries

Require: $c > 0$
  for $i = 1 : n - 1$ **do**
    $start \leftarrow i + 1$ {the finger list of i starts from the i+1th point}
    $end \leftarrow i + 24 * c^3$ {1st $24 * c^3$ points after i}
    if $end > n$ **then**
      $end \leftarrow n$ {Not wrapping around}
    end if
    $j \leftarrow 1$
    for $j = start : end$ **do**
      $FR_{ij} \leftarrow d(s_i, s_j)$ {distance between ith and jth point}
      $F_{ij} \leftarrow s_j$ {store the point in the finger array }
    end for
    $maxR \leftarrow$ maximum value of the FR array
    $j \leftarrow 0$
    for $k = end + 1 : n$ **do**
      if $d(s_i, s_k) < maxR$ **then**
        {kth element closer}
        $j \leftarrow j + 1$
        $f_j \leftarrow$ element of F that has $maxR$
        $F \leftarrow (F \backslash \{f_j\}) \cup \{s_k\}$
      end if
    end for
    $F \leftarrow F \cup \{f_1, f_2, \cdots, f_j\}$
  end for
  output F

search on r in the array. This leads to an additional cost of $O(\log \log n)$ per finger list access. Thus, the running time of this algorithm is $O(\log n \log \log n)$ with high probability (see [29] for details).

The algorithm for nearest neighbor search with metric skip lists is shown in algorithm 8.

Algorithm 8 Finding Nearset Neighbor in Growth Restricted Metrics

$i \leftarrow 1$ {i is the current position}
$m \leftarrow 1$ {m is minimum so far}
$mindist \leftarrow d(s_m, q)$;
$ar\_d\_q(1..n) \leftarrow 0$ {contains distance between q and all the points}
while $i < n$ **do**
  $r \leftarrow d(s_i, q)$ {distance between q and ith point}
  $k \leftarrow$ number of points in the finger list of i within radius $2r$
  for $j = 1 : k$ **do**
    $jj \leftarrow j$th finger of ith point
    if $ar\_d\_q(jj) == 0$ **then**
      $dist \leftarrow ar\_d\_q(jj) \leftarrow d(s_{jj}, q)$ {store distance between q and jj}
    else
      $dist \leftarrow ar\_d\_q(jj)$ {retrieve distance between q and jj from the array}
    end if
    if $dist < mindist$ or $dist \leq r/2$ **then**
      $found \leftarrow$ true;
      Store the point jj in an array ar as it is closer to q
    end if
  end for
  if $found ==$ true **then**
    $i \leftarrow \min(ar)$ {i becomes the smallest index amog the points who are closer to q}
    if $d(s_i, q) < d(s_m, q)$ **then**
      $m \leftarrow i$
    end if
  else
    $i \leftarrow$ maximum index in the fingerlist of i within radius $2r$
  end if
end while
output s_m

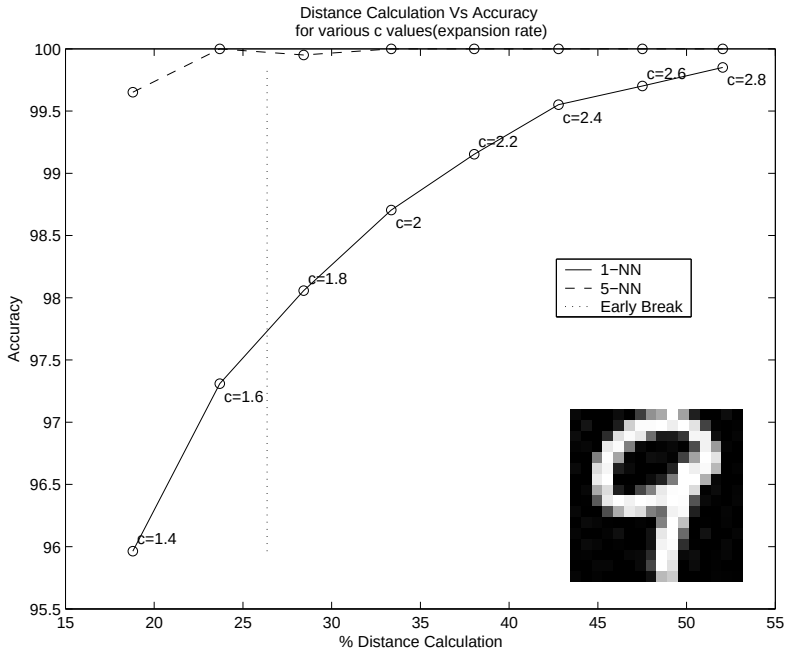


Figure 3.1: Comparing Metric Skip List algorithm with Early Break approach for USPS digit data. 1-NN indicates the approx. answer matches with the exact NN while 5-NN indicates it matches with any of the 5 exact NN.

3.1.4 Experimental Results

The experiments are run over various real life data sets (e.g. USPS digit data, Mnist digit data, Feret images, Forest Cover data, Orl faces data) and various synthetic data sets (random uniform data, single Gaussian data, mixture of Gaussians data). All the synthetic data sets have 8000 data points and 2000 query points. While running the experiments we've varied the assumed expansion rate, c. With the increase in c, the number of fingers in a finger list also increases, which in turn plays a positive role while finding nearest neighbor for a query point in the expense of more calculation. For each of the data sets, we've checked whether the approximate answer returned by the query matches with the exact NN. In the figures, it is termed as 1-NN (shown as straight line). Also, we've checked if the result matches with any of the 5 exact NN (termed 5-NN and shown as broken lines in the figures).

The figures are compared with the distance calculation needed for Early Break ap-

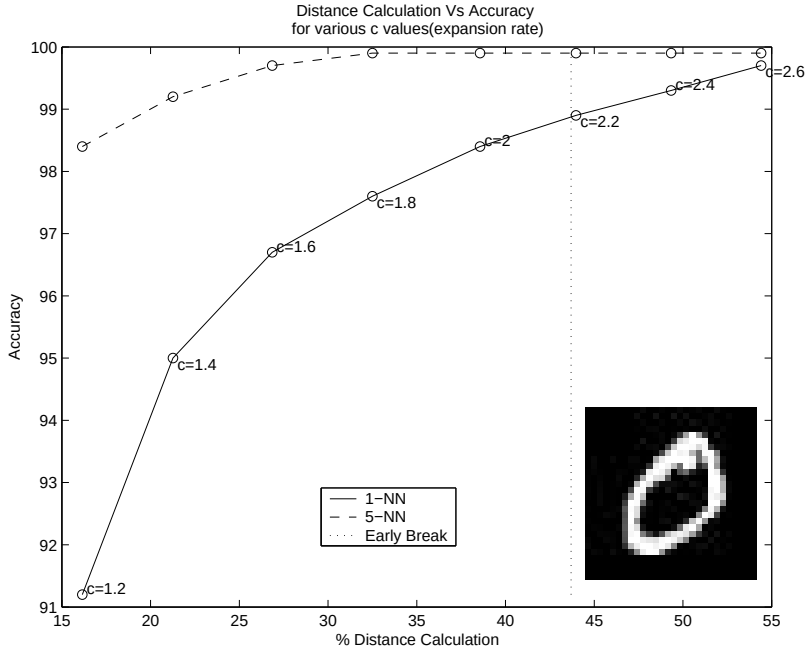


Figure 3.2: Comparing Metric Skip List algorithm with Early Break approach for Mnist digit data. 1-NN indicates the approx. answer matches with the exact NN while 5-NN indicates it matches with any of the 5 exact NN.

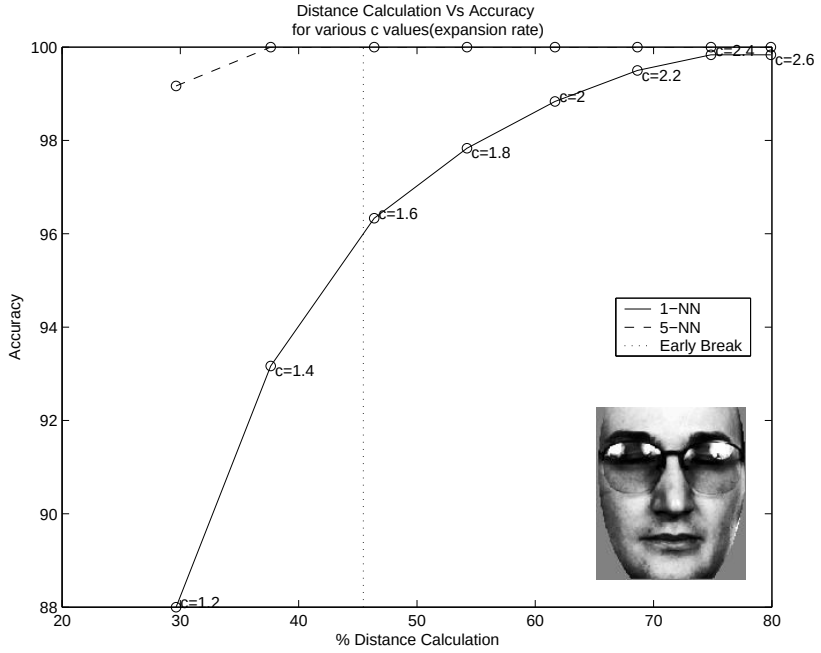


Figure 3.3: Comparing Metric Skip List algorithm with Early Break approach for Feret digit data. 1-NN indicates the approx. answer matches with the exact NN while 5-NN indicates it matches with any of the 5 exact NN.

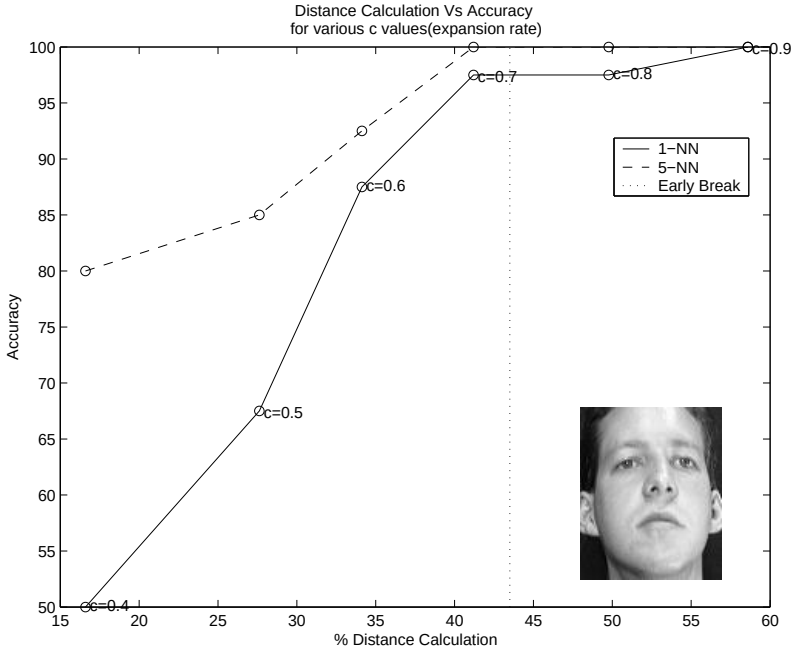


Figure 3.4: Comparing Metric Skip List algorithm with Early Break approach for Orl Face data. 1-NN indicates the approx. answer matches with the exact NN while 5-NN indicates it matches with any of the 5 exact NN.

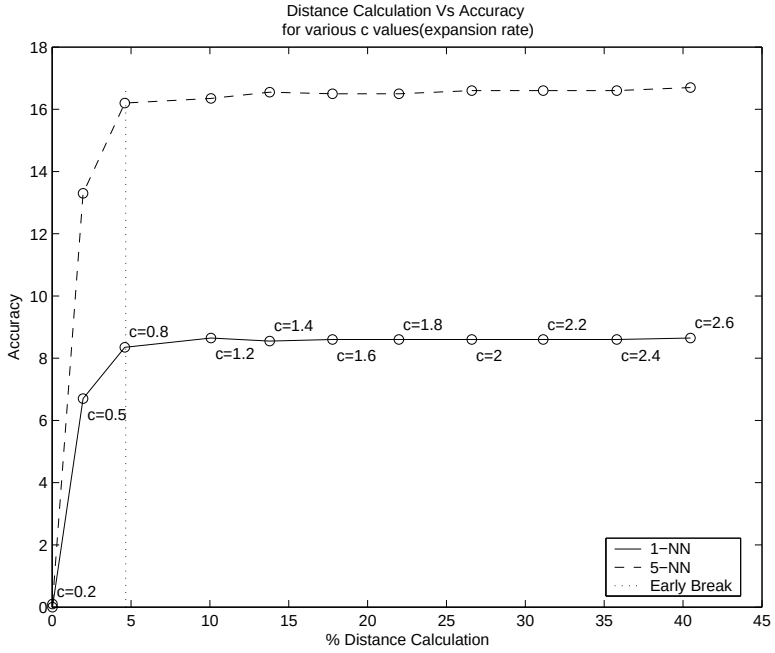
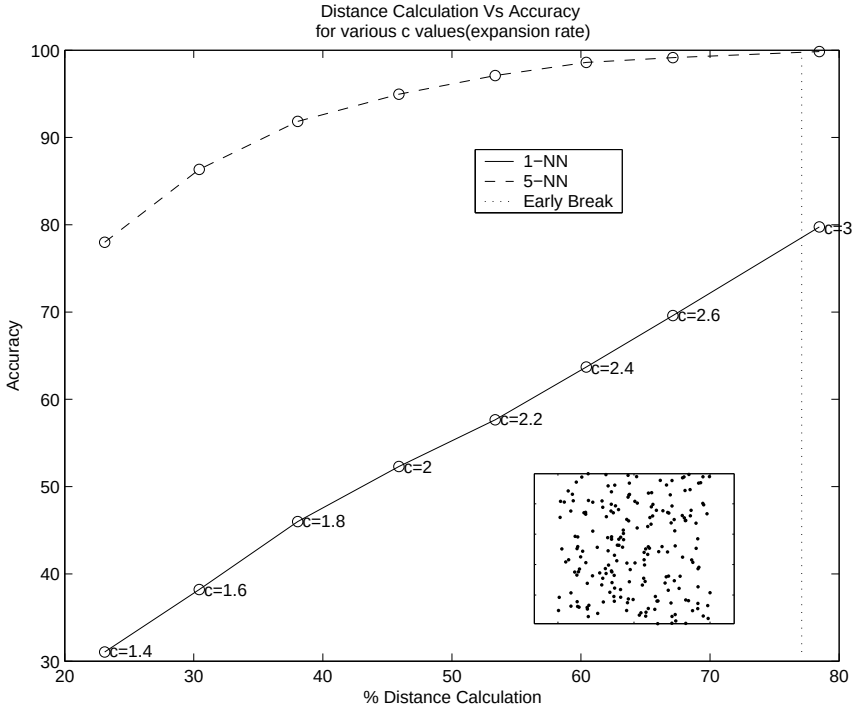


Figure 3.5: Comparing Metric Skip List algorithm with Early Break approach for Forest Cover data. 1-NN indicates the approx. answer matches with the exact NN while 5-NN indicates it matches with any of the 5 exact NN.

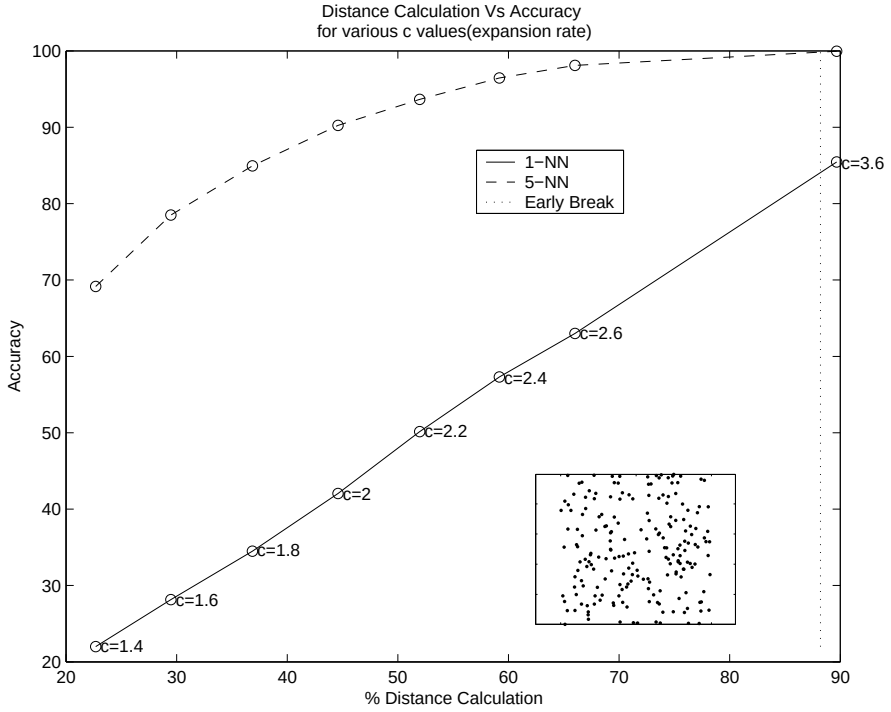
proach. It is shown as a vertical straight line which clearly shows which portion of the approximate algorithms are better than the Early Break approach. Our actual interest are on the left side of the Early Break's distance calculation line which maximizes the accuracy. In Figures 3.1, 3.2, 3.3 the approximate nearest neighbor found by the algorithm are more than 90% accurate. If we imposed the requirement and check whether the answer match with any of the first 5 exact nearest neighbors then the accuracy is more than 98%. That means the approximate answer is close to the vicinity of the actual nearest neighbor, which is sufficient enough for some practical purposes. The Forest Cover data in figure 3.5 behaves badly because of the nature of the data which has binary values in most of its dimensions. As a result, Early Break strategy works best in these kind of data (Early Break approach is approximately 4.5% of brute force approach).

For synthetic data sets, We have used 256 dimensional and 1024 dimensional data sets. The random uniform data are in the range $[0,1]^d$. From figure 3.6, we find that for this type of data, getting good accuracy with 1-NN is very difficult (it is roughly below 90%), but considering 5-NN we can get fairly good accuracy (close to 100%) with lesser cost than Early Break. These data sets have expansion rate 2^d which makes them inappropriate for metric skip list algorithm as they are not growth constrained.

Therefore, we've generated few synthetic data sets which are originally high dimensional (256D) but lie in various low dimensional manifold. The data sets are generated as follows: We randomly (uniform) distributed first 2 dimensions in the range [0,1] and made all other 254 dimensions to be Gaussian distributed with zero mean and unit standard deviation multiplied by various multiplying factors. Then we rotate the data using a random $d \times d$ matrix whose entries are Gaussian with zero mean and unit standard deviation. In figure 3.7(a), if we use very small multiplying factor (e.g. 0.001) to multiply the Gaussian distribution of the other 254 dimensions then the value of these dimensions become very negligble which makes this data set virtually 2-dimensional. This data set has expanson rate $2^2 = 4$ and is growth restricted. As we can see in the figure, metric

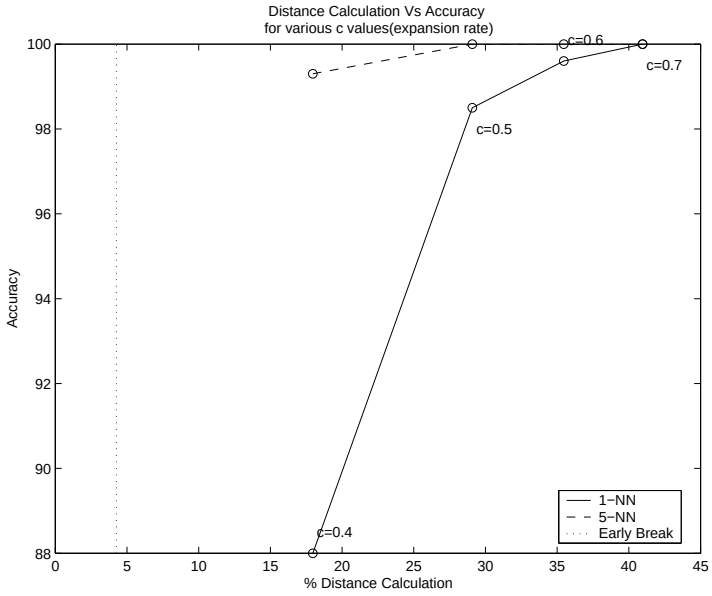


(a) Uniform Random Data - 256D.

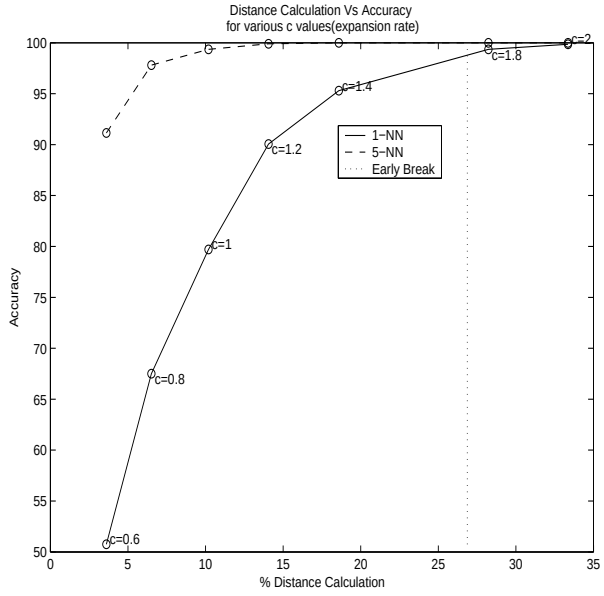


(b) Uniform Random Data - 1024D.

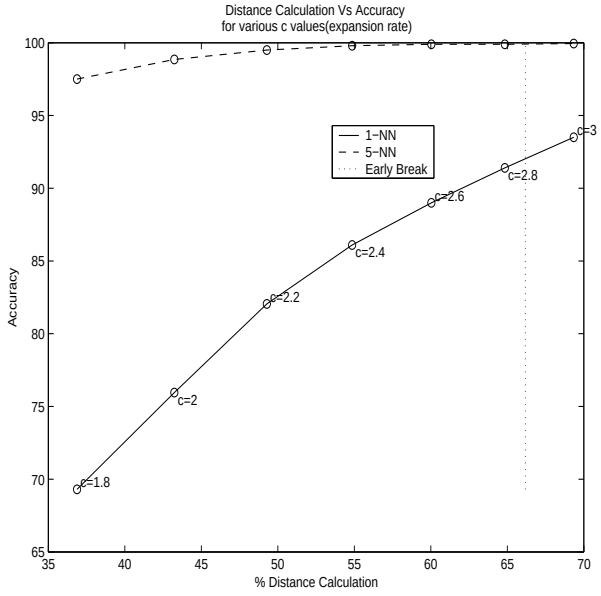
Figure 3.6: Comparing Metric Skip List algorithm with Early Break approach for Random Uniform ($[0,1]^d$) data (a) with 256D (b) with 1024D for various expansion rates.



(a)

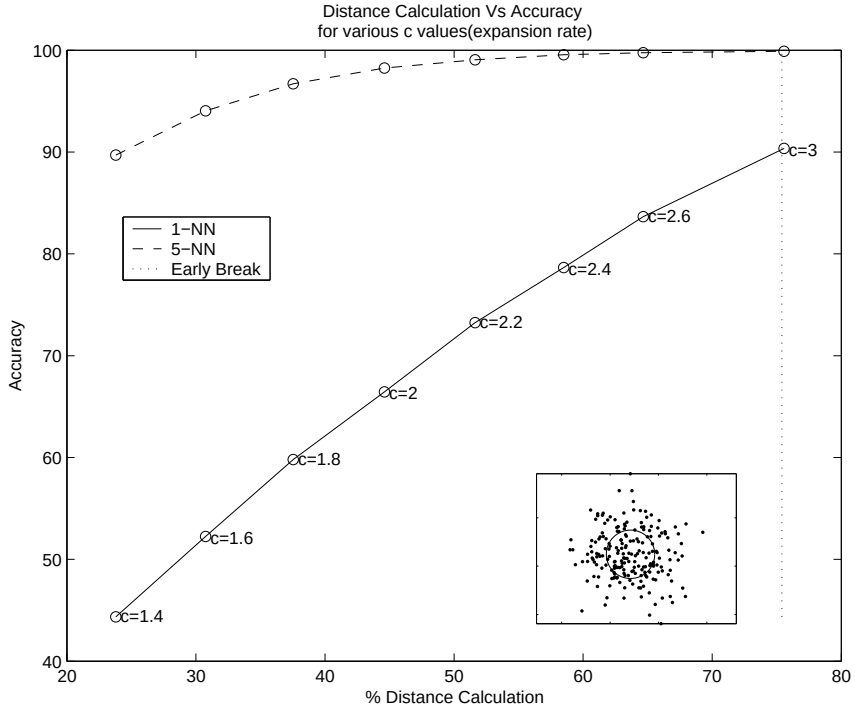


(b)

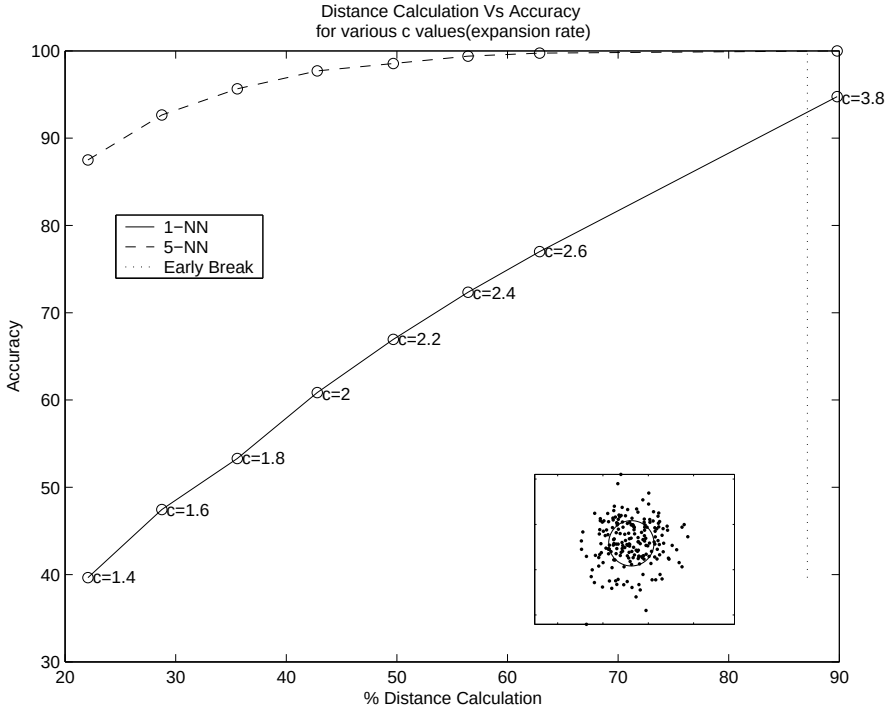


(c)

Figure 3.7: Comparing Metric Skip List algorithm with Early Break approach for 256D data where 2 of the dimensions are uniform random in [0,1] and others are Gaussian distributed (mean 0,variance 1) multipled by (a) 0.001. Data set lives in 2 dimension. (b) 0.01. Data set still lives in low dimension but greater than 2 dimension. (c) 0.1. Data set lives in bigger dimension. All 3 data sets are then rotated using a 256×256 dimensional random Gaussian matrix (mean 0, variance 1).

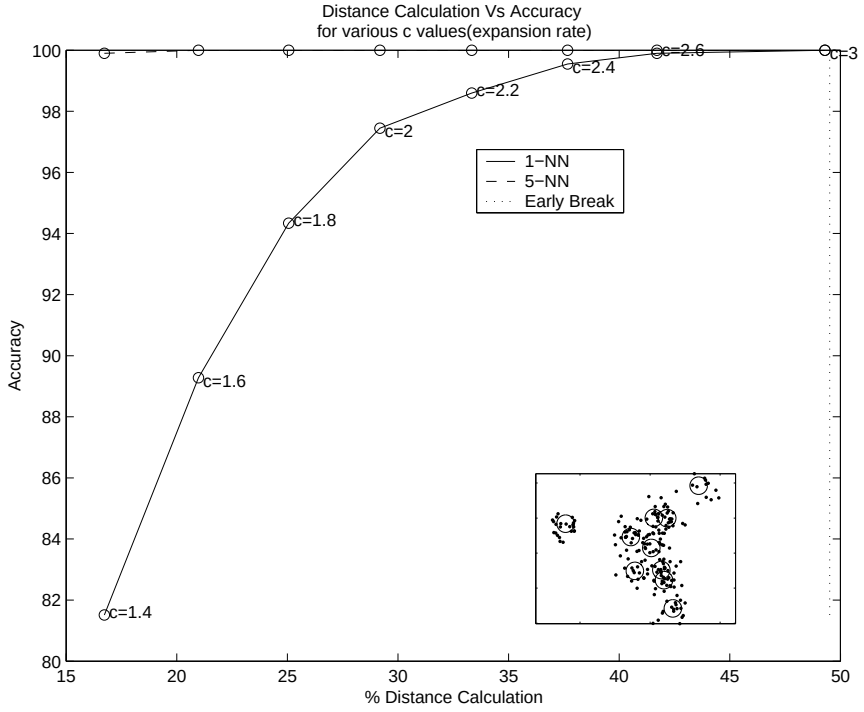


(a) 256 dimensional Single Gaussian data

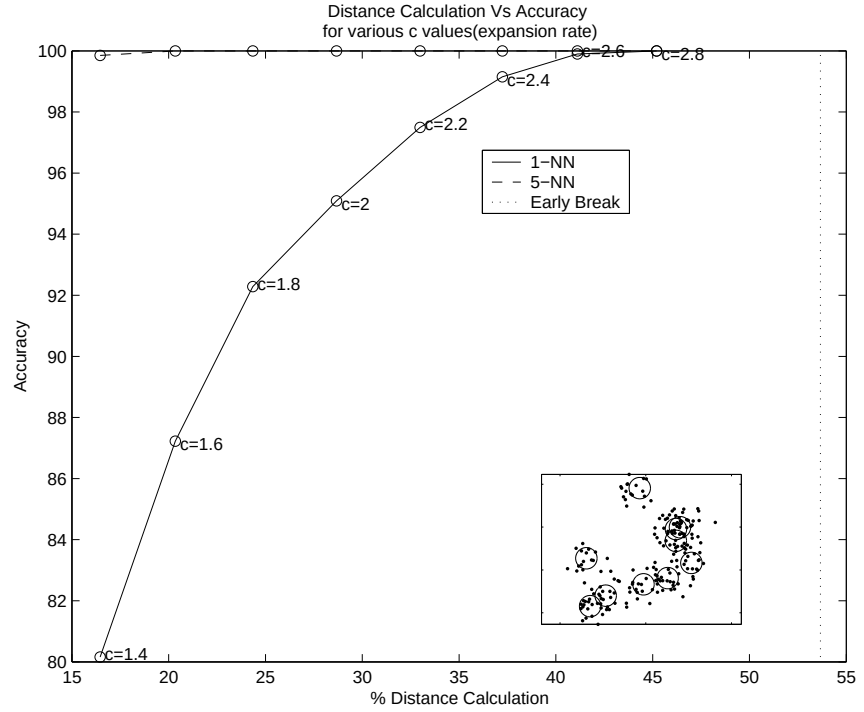


(b) 1024 dimensional Single Gaussian data

Figure 3.8: Comparing Metric Skip List algorithm with Early Break approach for Single Gaussian (mean 0, std. deviation 1) data (a) with 256D (b) with 1024D for various expansion rates.



(a) 256 dimensional Mixture of 10 Gaussian



(b) 1024 dimensional Mixture of 10 Gaussian

Figure 3.9: Comparing Metric Skip List algorithm with Early Break approach for Mixture of 10 moderately Separated Gaussian Data (a) with 256D (b) with 1024D for various expansion rates.

skip list algorithm has very good accuracy rate in low c values. The reason the distance calculation cost is high because the search point s_i can't move faster towards the query point (q). As the data set is very dense, s_i may find multiple points that are closer to the q than itself and according to the algorithm it chooses the one with the smallest index i as the next search point. As a result, it moves towards the query point very slowly. Early Break also performs reasonably well in this data set.

In figure 3.7(b), we added noise in the data set by setting the multiplying factor to be 0.01. The data set still lives in low dimension so the accuracy of metric skip list remains high. The distance calculation cost gets lower because the algorithm doesn't suffer from the previous problem of not moving faster towards the query point. Early Break approach also starts deteriorating in this data set. If we go on increasing the multiplying factor then the cost of metric skip list (to attain reasonably good accuracy) and Early break both get higher, as we can see from figure 3.7(c) where we set the multiplying factor to be 0.1.

The single Gaussian data (with zero mean and unit standard deviation) sets like random uniform data, also don't exhibit good accuracy with 1-NN but provide nearly 100% accuarcy with 5-NN (see figure 3.8). The figure is very similar to the one for random uniform data set (figure 3.6).

The mixture of 10 moderately well separated Gaussian data is generated as the following: the cluster centers are Gaussian distributed with zero mean and 1 standard deviation and for each of the clusters the data is generated with low (0.25) standard deviation so they are grouped together. From figure 3.9, we see that this data set performs much better than the uniform random and single Gaussian data sets. With both 1-NN and 5-NN, we can get 100% accuracy with much lesser cost than Early Break.

3.2 Algorithm by Kleinberg

Jon M. Kleinberg presented two algorithms for nearest neighbor search in high-dimension in [15]. The algorithms are based on a method for combining randomly chosen one-dimensional projections of the underlying point set. The algorithms are the followings:

1. An algorithm for finding ϵ-approximate nearest neighbors with a query time of $O((\frac{1}{\epsilon^2})(d \log^2 d)(d + \log n))$ but storage of $O(n \log d)^{2d}$.

2. An ϵ-approximate nearest neighbor algorithm with near linear storage and a query time that improves asymptotically on linear search in all dimensions.

We have tried the second approach for our experiment. Because the pre-processing time and storage requirements for the first approach is prohibitively high, whereas the second approach has near linear storage (though the query time of the second approach is little higher than the first). The second approach returns an ϵ-approximate nearest neighbor with probability $1 - \delta$ and has a query time $O((\frac{1}{\epsilon^2} \log \delta^{-1})(n + d \log^3 n))$, pre-processing time $O^*(d^2 n)$, and storage $O^*(dn)$ [1]. One important thing about the algorithm is, when $d = O(n/\log^3 n)$ then only we can answer a query with a net constant number of operations per point, rather than the d operations per point required by the brute force approach.

3.2.1 Data Structure

For the data structure we need the following constants:

- γ_2 is chosen so that $e^{\gamma_2} \leq 1 + \epsilon$
- c_1 is chosen s.t. $e^{-\frac{1}{64} c_1 \gamma_2^2} \leq \frac{1}{4}(\frac{1}{3}\delta)^{1/\log n}$
- c_2 is chosen s.t. $e^{-\frac{1}{64} c_2 \epsilon^2} \leq \frac{1}{2}$

[1] the $O^*()$ also suppresses terms that are polynomial in $\log n$

- c_2' is chosen s.t. $e^{-\frac{1}{64}c_2'\epsilon^2} \leq \frac{1}{3}\delta$
- γ_1 is chosen s.t. $(1 - \frac{c_1 \log^3 n}{n})^{\frac{\gamma_1 n}{\log^3 n}} \geq 1 - \frac{1}{3}\delta$
- c_3 is chosen s.t. $(1 - \frac{\gamma_1}{\log^3 n})^{c_3 \log^3 n} \leq \delta$

Set $\epsilon_0 = \frac{\gamma_2}{\log n}$ and define

$$L = f(\frac{1}{6}\epsilon_0, \delta) = \Theta(d \log^2 n(\log^2 d + \log d \log\log n))$$

where the function $f(.,.)$ is defined as in [15].

We choose a set V of L vectors $v_1, v_2, ..., v_L$. Here $V \subset S^{d-1}$ where $S^{d-1} \subset R^d$ denote the unit $(d-1)$ sphere $\{v \in R^d : \|v\| = 1\}$. The term $\|v\|$ is the norm of vector v and defined as $\|v\| = \sqrt{v \cdot v}$. The data structure is simply an $L \times n$ matrix M (here n is the numper of points in the set P). The entry $M[i,j]$ is set equal to $v_i \cdot p_j$.

3.2.2 Processing a query

Let $q \in R^d$ be a query point and let p^* be a point which minimizes the distance $d(p,q)$ over all $p \in P$. The purpose of the algorithm is to output a point in the set, $Z = \{p_i \in P : d(p_i, q) \leq (1+\epsilon)d(p^*, q)\}$ with probability $1 - \delta$.

According to the terminology used by Kleinberg [15], if $p_i, p_j \in P$ and $v_k \in V$, we say p_i dominates p_j with respect to v_k if $|v_k \cdot p_i - v_k \cdot q| < |v_k \cdot p_j - v_k \cdot q|$.

Now, for a subset V' of V, we say that p_i dominates p_j with respect to V' if p_i dominates p_j for more than half of the vectors in V'. Kleinberg referred it as V'-comparison of p_i and p_j.

For simplicity let the number of points n be 2^m. With these n points, we can construct a complete binary tree T of height m. Also let T_h be the height of the tree at height $h \leq m$. The leaves are situated at height 0.

Let $L_1 = c_1 \log^3 n$ where c_1 is a constant defined earlier. A multiset Γ of L_1 vectors are drawn uniformly at random with replacement from V. Then the inner product with

q is computed for each $v \in \Gamma$. Assuming (for simplicity) L_1 is a power of 2, we write $b = \log L_1$.

The algorithm has 2 parts which are described below.

Part A:

- A sub-multiset Γ_x of Γ of size $c_2' + c_2 h$ is chosen for each node x at height $h \leq b$.
- By assumption T has n leaves. We randomly assign each point $p \in P$ to a leaf of tree T.
- From the leaves upto height b, for each node x, we make Γ_x comparisons of the points assigned to its two children p_i and p_j. If p_i dominates p_j with respect to Γ_x then we assign p_i to x. Otherwise we assign p_j to x.
- From height b upto root we perform the same procedure using Γ-comparisons. Let p_A denote the point assigned to the root.

Part B:

- We choose randomly a set $P' \subset P$ of size $c_3 \log^3 n$.
- Then we compute the distance from q to each $p \in P'$. Let p_B be the point which has the smallest distance from the query point q.

From p_A and p_B, we determine which one is closer to q, and return that point as the answer to the query.

3.2.3 Modification of the Algorithm

When implementing the above algorithm, we haven't used the constants (which depend on ϵ, δ and n) that are defined by Kleinberg [15] because the values of the constants are very large for an actual implementation of this method. The constants are used to find the values of L - the number of vectors in V , L_1 - the number of vectors in Γ, and

the number of vectors in Γ_x which is a sub-multiset of Γ. In our implementation, we've used L_1 to be 90% of L and the number of vectors in sub-multiset Γ_x to be 75% of L as arbitrary constants. Though these values are arbitrary, we have chosen them high enough and close to L so that we get better performance from the algorithm. We then run the algorithm for various values of L.

Algorithm 9 An ϵ-approximate nearest neighbor algorithm by Kleinberg

{ Part A}
$V \leftarrow L$ random vectors each of length d s.t. $\{v \in V : \|v\| = 1\}$
for $i = 1{:}L$ **do**
  for $j = 1{:}n$ **do**
    $M_{ij} \leftarrow v_i \cdot p_j$ {$L \times n$ matrix which is our preprocessed data structure}
  end for
end for
$\Gamma \leftarrow$ random subset of $L_1 \in L$ vectors from V
for i=1:L_1 **do**
  $vq_i \leftarrow v_i \cdot q$ {for each $v_i \in \Gamma$}
end for
Assign each point in P to a distinct leaf of T {T is a binary tree of height $m = \lceil \log n \rceil$}
while $h \leq$ height of tree **do**
  if $h \leq \log L_1$ **then**
    $\Gamma_x \leftarrow$ random sub multiset of Γ
  else
    $\Gamma_x \leftarrow \Gamma$
  end if
  From leaf to root, assign $p \in P$ to $x \in T$ by Γ_x-comparison of the children of x and assigning the winner to x
end while
$p_A \leftarrow$ point assigned to the root

{Part B}
Choose a random set $P' \subset P$ of size $O(\log^3 n)$
Find the point in P' which has got the minimum distance from q.
Set p_B to be this point.
if p_A is closer to q than p_B **then**
  set answer to p_A
else
  set answer to p_B
end if

3.2.4 Experimental results

We have used the USPS digit, Mnist digit, Feret images, Orl face and Forest Cover data sets and some synthetic datas sets to test the algorithm's performance. The synthetic data sets are uniform random, single Gaussian and mixture of 10 moderately well separated Gaussian data sets. The generation of these synthetic data sets are described in previous section and in Chapter 2. All the synthetic data sets have 8000 data points and 2000 query points.

Kleinberg's algorithm uses matrix multiplication as its basic measurement unit, which is roughly the same as distance calculation unit. Other than matrix multiplication, the algorithm also uses large number of boolean comparisons between the points as the other measurement unit.

From the figures 3.10 to 3.12 and figure 3.14, we can see that though it is hard to get significant accuracy w.r.t. to 1-NN (i.e. whether the result of the algorithm matches with the exact nearest neighbor), the output of the algorithm shows a good accuracy w.r.t 5-NN (i.e. whether the approximate result matches with any of the 5 exact nearest neighbor). The Forest Cover data set in figure 3.13 suffers because of its bad intrinsic structure as before.

If we consider only the query time of the algorithm, it shows significant improvement in case of number of calculation needed to be done. But there is a large pre-processing factor that needed to be considered. The pre-processing is of the order $O(Ldn)$. But because of large constant factors and the high value of L, in practice this pre-processing cost is very expensive. Also, as we've said earlier the algorithm has considerable amount of boolean comparisons which is also a factor in the query time that we didn't include.

From the figure 3.10 to 3.12 and 3.14, we can see that the query time needed to find reasonable accuracy w.r.t 5-NN is less than Early Break stategy. But including the pre processing time shows that cost is far beyond the calculation required for Early Break. Forest data set in figure 3.13, is nowhere near any acceptable accuracy because of the

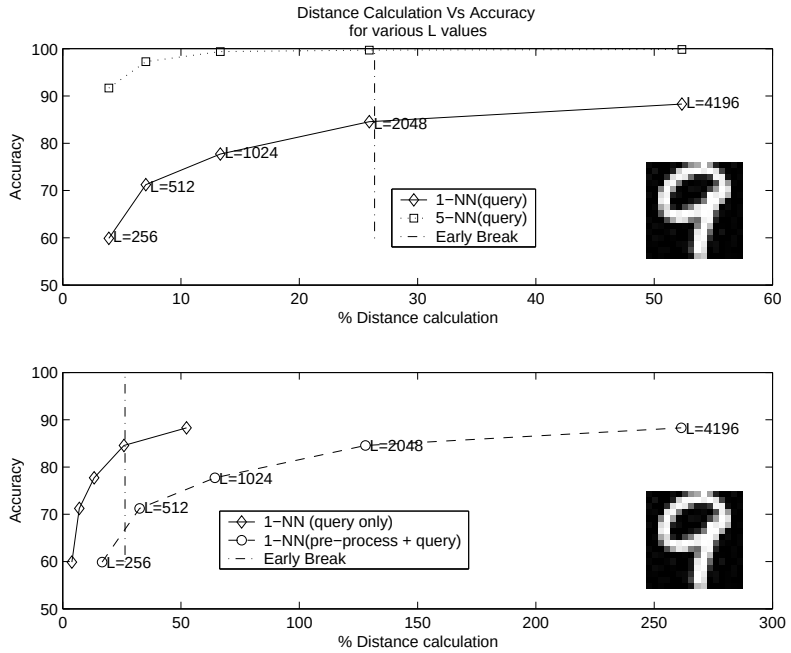


Figure 3.10: Comparing Kleinberg algorithm with Early Break for USPS Digit data. 1-NN indicates the approx. answer matches with the exact NN while 5-NN indicates it matches with any of the 5 exact NN. The upper figure compares 1-NN and 5-NN with Early Break. The lower figure compares 1-NN without preprocessing and 1-NN with preprocessing with Early Break.

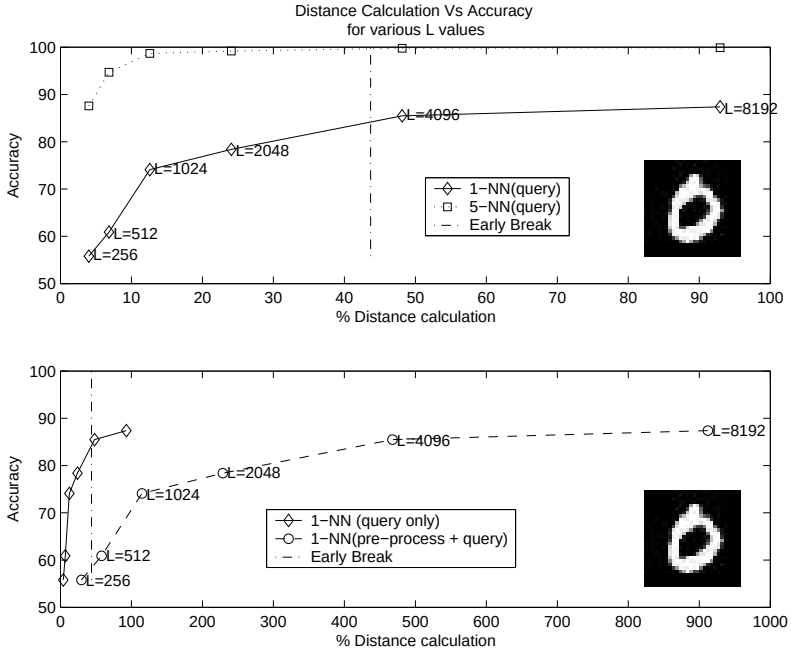


Figure 3.11: Comparing Kleinberg algorithm with Early Break for Mnist Digit data. The upper figure compares 1-NN and 5-NN with Early Break. The lower figure compares 1-NN without preprocessing and 1-NN with preprocessing with Early Break.

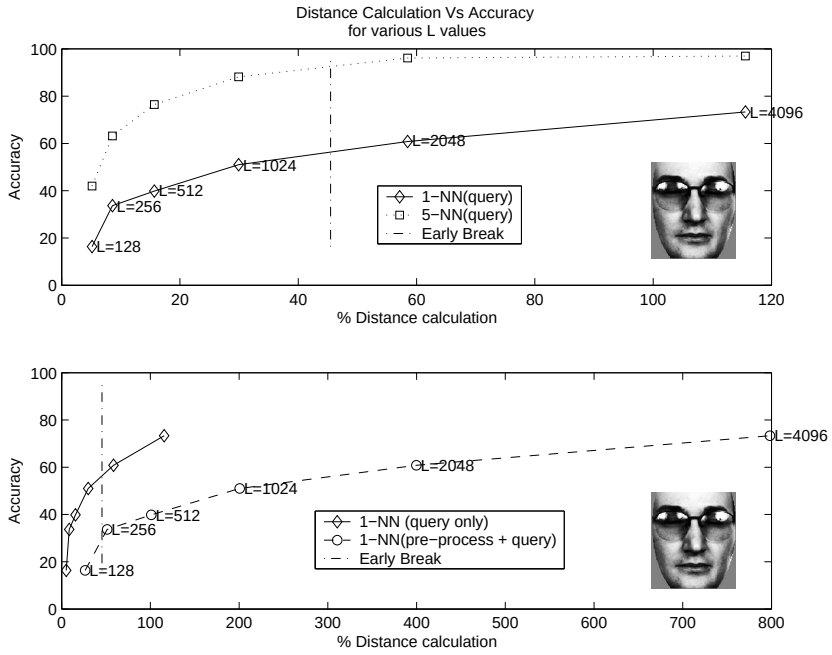


Figure 3.12: Comparing Kleinberg algorithm with Early Break for Feret image data. The upper figure compares 1-NN and 5-NN with Early Break. The lower figure compares 1-NN without preprocessing and 1-NN with preprocessing with Early Break.

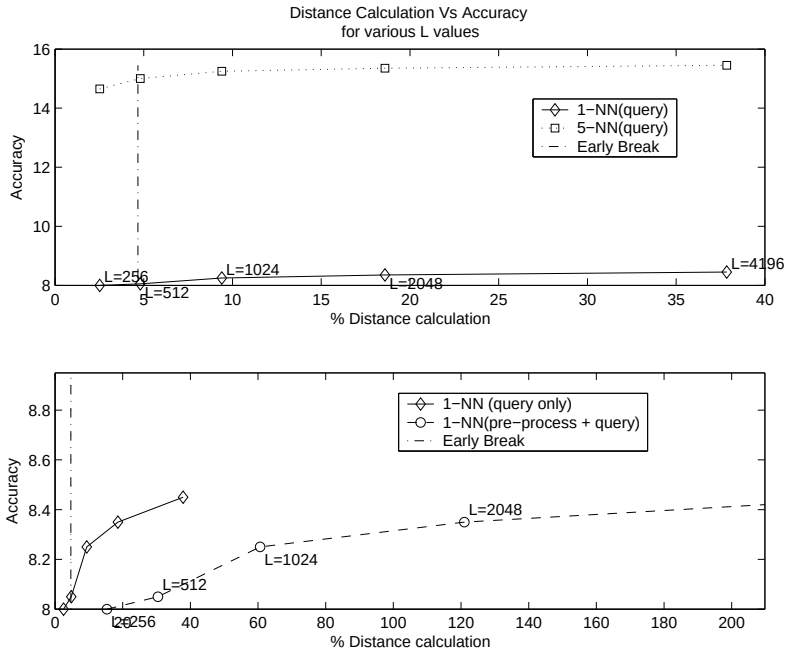


Figure 3.13: Comparing Kleinberg algorithm with Early Break for Forest Cover data. The upper figure compares 1-NN and 5-NN with Early Break. The lower figure compares 1-NN without preprocessing and 1-NN with preprocessing with Early Break.

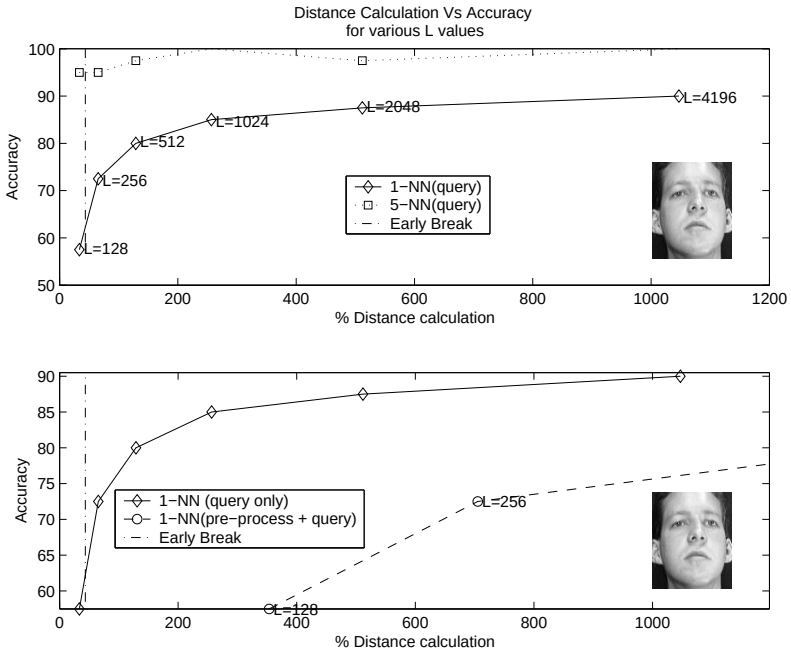
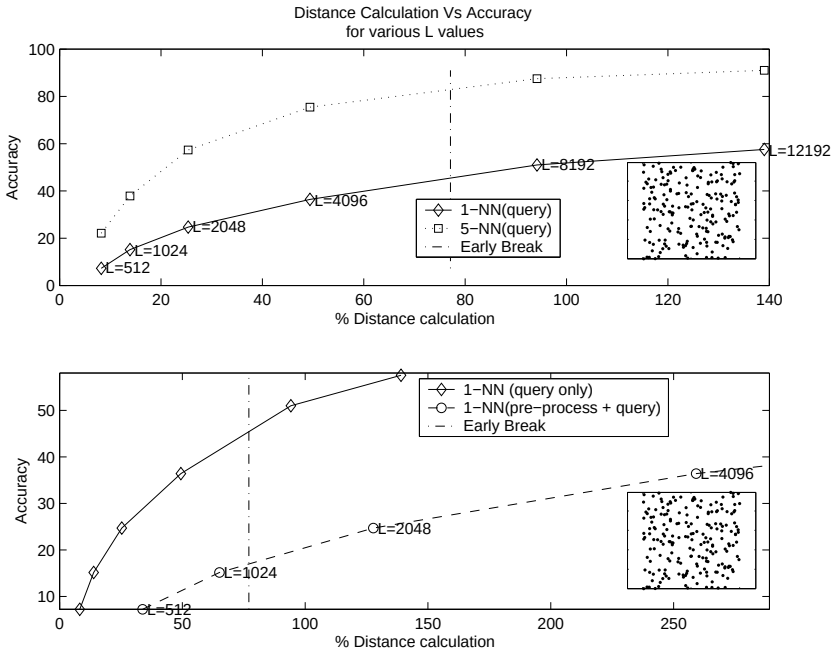


Figure 3.14: Comparing Kleinberg algorithm with Early Break for Orl Face data. The upper figure compares 1-NN and 5-NN with Early Break. The lower figure compares 1-NN without preprocessing and 1-NN with preprocessing with Early Break.

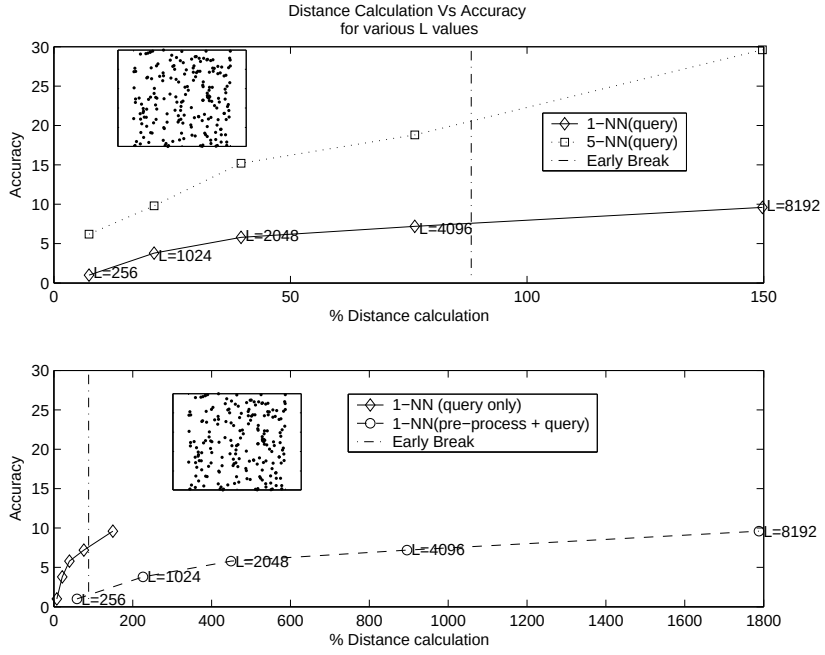
reasons earlier stated.

For synthetic data sets, We have used 256 and 1024 dimensional data sets. From figure 3.15 to figure 3.17, we can see that we can't get significant accuracy rate w.r.t 1-NN. We can get higher accuracy rate with 5-NN for 256 dimensional data sets, but the accuracy is not that significant and also the cost of the query surpasses that of Early Break. For 1024 dimensional data sets it is hard to achieve good accuracy rate even with 5-NN (see figure 3.15(b), 3.16(b),3.17(b)). Moreover, adding pre-processing time with it make the computational cost really high as before.

The Kleinberg algorithm can answer a query with a net constant number of operations per point when $d = O(n/\log^3 n)$. The data sets we've considered here have $d >> n/\log^3 n$, which is the main reason for such inaccurate results. So, when d is large we need n to be large enough to take advantage of this algorithm.

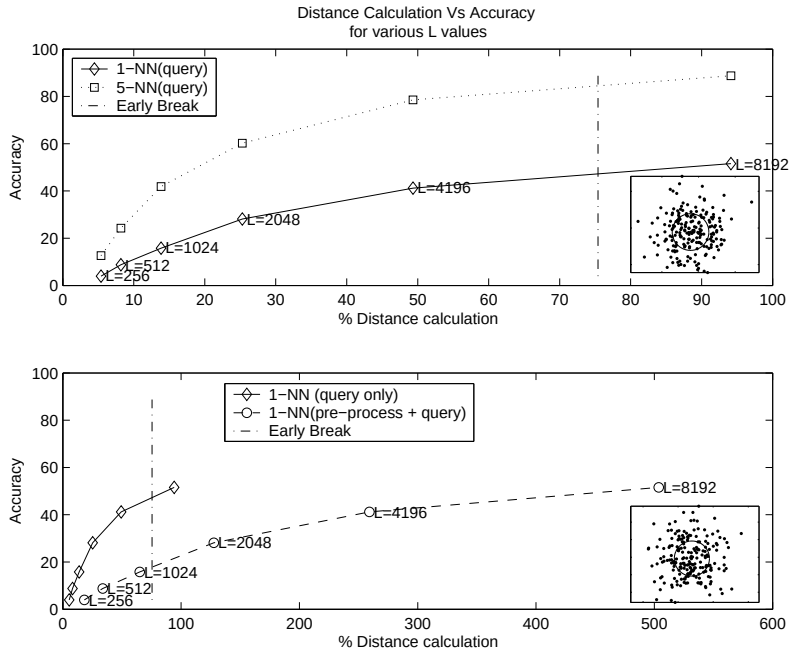


(a) 256 dimensional Random Uniform data

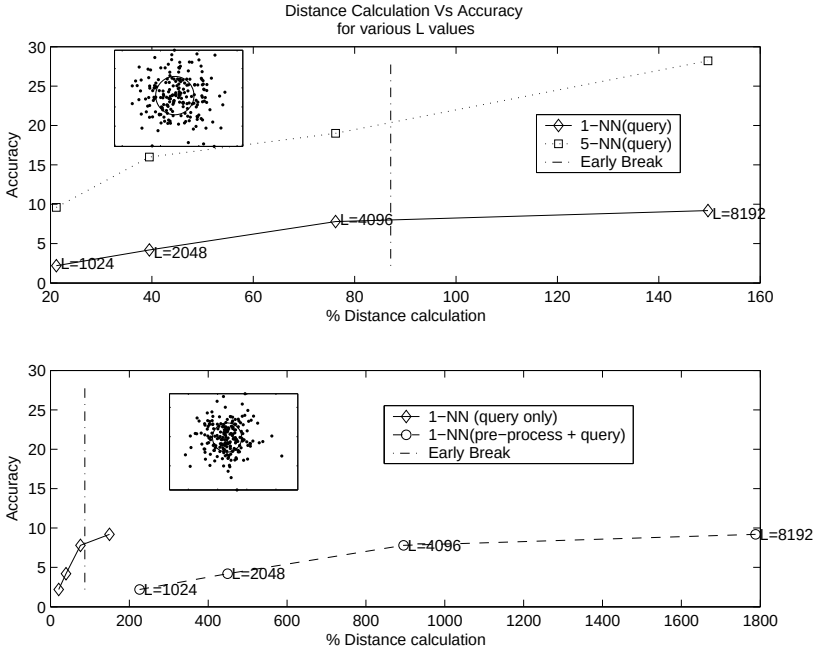


(b) 1024 dimensional Random Uniform data

Figure 3.15: Comparing Kleinberg algorithm with Early Break for Random Uniform Data (a) with 256D (b) with 1024D for various L values.

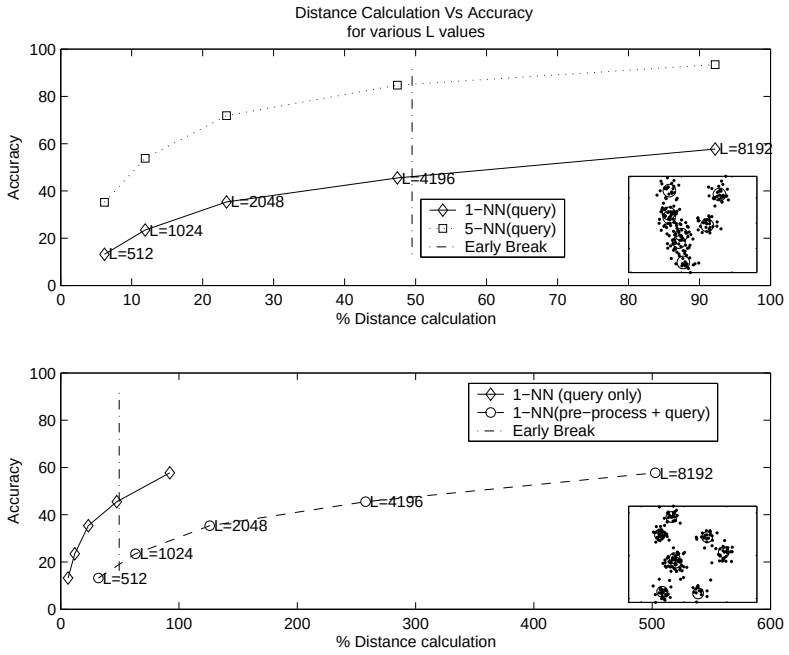


(a) 256 dimensional Single Gaussian data

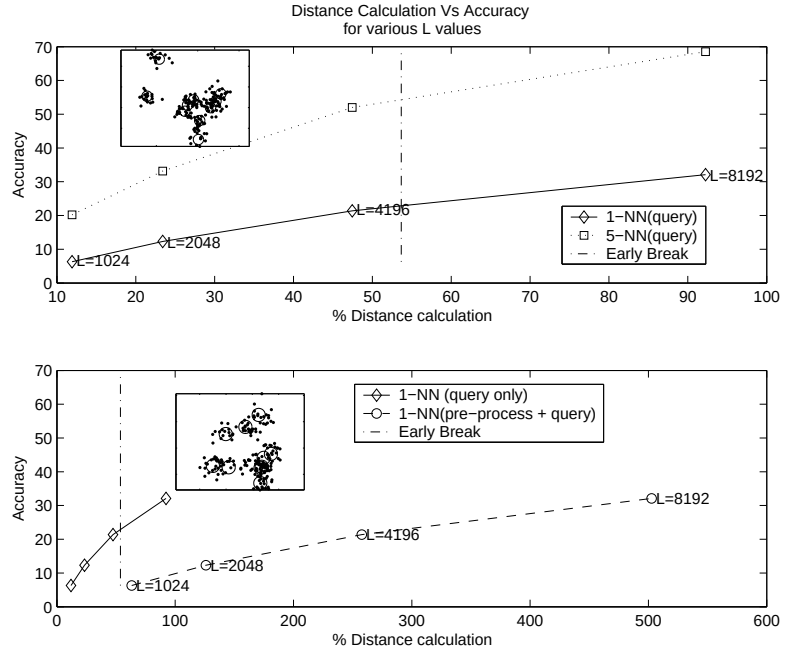


(b) 1024 dimensional Single Gaussian data

Figure 3.16: Comparing Kleinberg algorithm with Early Break for Single Gaussian Data (a) with 256D (b) with 1024D for various L values.



(a) 256 dimensional Mixture of 10 Gaussian Data



(b) 1024 dimensional Mixture of 10 Gaussian Data

Figure 3.17: Comparing Kleinberg algorithm with Early Break for Mixture of 10 moderately well separated Gaussian Data (a) with 256D (b) with 1024D for various L values.

3.3 Other methods

There are several other nearest neighbor search algorithms that we haven't covered. Robert Kraugthgamer and James Lee have given a deterministic data structure for maintaining a set S of points in a general metric space, while supporting nearest neighbor queries [30].

Another very recent algorithm for finding nearest neighbor is "Cover Tree" [31]. It is a tree data structure for fast nearest neighbor operations in general n-point metric spaces. Cover tree can be used both for exact and approximate NN search.

These data structures are fairly new and we haven't studied them in detail.

Chapter 4

Multiple Random Projection Technique

In this chapter we'll discuss in detail our technique of finding nearest neighbors. First we'll talk about different dimensionality reduction techniques then we'll move to random projection (RP): what is random projection, how randomization is done and its characteristics. Next we'll describe how the multiple random projection (MRP) technique fits in our approach of finding nearest neighbor. Lastly, we'll show some experimental results using multiple random projection technique on various data sets.

4.1 Dimensionality Reduction

To avoid the "curse of dimensionality", it is natural to think about reducing the dimension to a certain degree where its effect is small (it can't be eliminated totally). But at the same time, we'll need to be careful about how much information we are sacrificing. One way of performing dimensionality reduction is to project data into a lower dimensional linear subspace that captures as much of the variation of the data as possible. The most widely used technique to do this is Principal Component Analysis (PCA). But PCA is computationally quite expensive for high dimensional data sets. It needs to compute a

data covariance matrix which is $d \times d$ for d-dimensional data. Also, PCA is not guaranteed to give the best results. Thus for NN, it is desirable to find a computationally simple method of dimensionality reduction technique that does not cause data to be distorted significantly.

In random projection (RP), the original high dimensional data is projected into a lower dimensional subspace using a random matrix (each entry of this random matrix is Gaussian distributed). Random projection is computationally efficient, yet sufficiently accurate method for dimensionality reduction of high dimensional data sets.

Before continuing with RP, in the remainder of this section, we'll briefly discuss some alternative methods of dimensionality reduction. The alternative methods we will look at are PCA, Singular Value Decomposition (SVD) and Discrete Cosine Transform (DCT).

4.1.1 PCA

Principal Component Analysis (PCA) is a powerful tool in analyzing data. It is a way of identifying patterns in data, and expressing the data to highlight their similarities and differences. Technically, it finds the k-dimensional linear subspace of $\mathbf{R}^d$ that captures the variation of the data as much as possible.

Specifically, given the data set $X = \{x_1, x_2, ..., x_n\}$ it finds the linear projection to $\mathbf{R}^k$ for which

$$\sum_{i=1}^{n} \|x_i^* - \mu^*\|^2$$

is maximized. Here x_i^* is the projection of point x_i and μ^* is the mean of the projected data.

PCA can be viewed as a procedure of finding projections of high dimensional data. It can solve an optimization problem exactly and efficiently, via eigenvalue computation. The eigenvalue decomposition of the data covariance matrix is computed as $E\{(XX^T)/n\}$ $= Q\Delta Q^T$ where $X_{d \times n}$ is the original set of n d-dimensional observation and the columns of matrix Q are the eigenvectors of the data covariance matrix $E\{(XX^T)/n\}$ and Δ is a

diagonal matrix containing the respective eigenvalues.

For reducing dimensionality, the data is projected into a subspace spanned by the most important eigenvectors :

$$X^{PCA} = Q_k^T X$$

where Q_k is a $d \times k$ matrix that contains the k eigenvectors corresponding to the k largest eigenvalues [44]. Unfortunately, the eigenvalue decomposition of the data covariance matrix whose size is $d \times d$ for d-dimensional data is very expensive to compute. The running time of PCA is polynomial, but is rather high. It is $O(d^2n) + O(d^3)$ [32] for d-dimensional data. Actually, Roweis [33], Tipping and Bishop [34] have shown that for a large matrix there exists computationally less expensive methods for finding only few eigenvectors and eigenvalues.

4.1.2 SVD

A very powerful set of techniques dealing with sets of equations or matrices that are either singular or numerically very close to singular is the so-called singular value decomposition (SVD). In SVD, any matrix $X_{d\times n}$ can be decomposed into $X = USV^T$ where U and V are orthogonal matrices that contain the left and right singular vectors of X, and the diagonal of S contains the singular values of X. Reducing the dimensionality using SVD can be done by projecting the data into the subspace spanned by the left singular vectors corresponding to the k largest singular values:

$$X^{SVD} = U_k^T X$$

where $d \times k$ matrix U_k contains these k singular vectors [44]. Like PCA, SVD is also computationally expensive. But for sparse data matrices SVD is useful because there exists numerical routines such as power or Lanczos method that are more efficient [45]. For a sparse matrix $X_{d\times n}$ with about c nonzero entries, the computational complexity of SVD is of order $O(dcn)$ [35].

One of the most prominent use of sparse SVD is Latent Semantic Indexing [35, 36]. LSI is an information retrieval technique based on the spectral analysis of the term-document matrix. It is a dimensionality reduction method for text-document data. Using LSI, the document data is presented in a lower-dimensional "topic" space: the documents are characterized by some underlying (latent, hidden) concepts referred to by the terms. Also, random projection can be used as a way of speeding up LSI [35].

4.1.3 DCT

The discrete cosine transform (DCT) is a Fourier-related transform similar to the discrete Fourier transform (DFT), but using only real numbers. It is equivalent to a DFT of roughly twice the length, operating on real data with even symmetry (since the Fourier transform of a real and even function is real and even), where in some variants the input and/or output data are shifted by half a sample.

DCT is a method mainly used for image compression. Hence it can also be used as a dimensionality reduction technique. DCT is computationally less expensive than PCA, but its performance approaches that of PCA. In DCT, the distortion of image occurs at highest frequencies only and human eye tends to neglect these as noise. Dimensionality reduction in DCT can be done as follows: An image is transformed into DCT space and in the inverse transform, the transform co-efficient corresponding to the highest frequencies are discarded to reduce the dimension [37, 38].

The computation of DCT doesn't depend on data matrix, contrast to PCA that needs eigenvalue decomposition of data covariance matrix. For this reason, DCT is cheaper to compute. The computational complexity of DCT is of the order $O(dn \log_2(dn))$ for a data matrix with d-dimension and n data points [38].

4.2 Random Projection

In Random Projection(RP), the original d-dimensional data is projected into k-dimensional subspace ($k << d$) using a random $k \times d$ matrix M, whose entries are Gaussian. The key idea of random mapping arises from the Johnson-Lindenstrauss (JL) lemma [43], which is given below:

4.2.1 Johnson-Lindenstrauss (JL) Lemma

The JL lemma states that if points in a vector space are projected to a random subspace of suitably high dimension, then the pair-wise Euclidean distances between the points are approximately preserved. To be more precise, any n point set in the Euclidean space can be embedded in a space of $O(\log n/\epsilon^2)$ without distorting the distances between any pair of points by more than a factor of $(1 \pm \epsilon)$ for any $0 < \epsilon < 1$ [39].

When a unit vector is projected into a random k dimensional subspace, its squared length is concentrated around its mean which is k/d, and is not distorted by more than $(1 \pm \epsilon)$ with probability $O(1/n^2)$. Normalizing this and applying the trivial union bound then gives the lemma. For a detailed proof of this lemma see [39, 40].

According to JL lemma, points in higher dimensional spaces can still retain an approximation level of separation when they are embedded (or projected) in lower dimensional space. The idea of random embedding of points in lower dimensional space stems from this lemma.

4.2.2 How randomization is done

A random projection from d dimension to k dimension is represented by $k \times d$ matrix M and it doesn't depend on the data. The following method is used to generate a random projection:

1. Set each entry of the matrix M to an i.i.d. $N(0, 1)$ value, i.e. let $M = randn(k, d)$.

2. Let X be the original data matrix with dimension $d \times n$ where n is the number of data points and d is the actual dimension. Then $X^{RP}_{k\times n} = M_{k\times d} X_{d\times n}$ will give the matrix of the projected points.

4.2.3 Characteristics of RP

Random projection is computationally very simple. Forming the random matrix M and projecting the data $X_{d\times n}$ into k dimension is of order $O(dkn)$. If the data matrix is sparse with c non-zero entries per data point then the complexity is of the order $O(ckn)$.

There are different ways of making the random matrix M. To have unitary projection, we have to make the k rows of the matrix M orthogonal (for example, by using Gram-Schmidt algorithm). Unfortunately, orthogonalizing M is computationally quite expensive. A result proposed by Hect-Nielsen [41] state that in high dimension, there exists much larger number of almost orthogonal than orthogonal directions. So, orthogonalizing M is not necessary for random projection. In some variants of random projection, the k rows of the random matrix M is normalized to have unit length.

In Euclidean distance measure, we write the original distance between two data vectors, x_1 and x_2 as $\|x_1 - x_2\|$. After projecting the data in random space we can approximate the distance between these vectors by the scaled distance measure in the reduced space:

$$\sqrt{d/k}\|Mx_1 - Mx_2\|$$

where d is the original and k is the reduced dimensionality of the data set. The scaling term $\sqrt{d/k}$ takes into account the reduction in dimensionality. According to Johnson-Lindenstrauss lemma, the expected norm of a projection of a unit vector into a random subspace through the origin is $\sqrt{k/d}$ [43].

Another point of interest is the entries in the random matrix M. In our implementation, each entry m_{ij} is normally distributed. But this need not be the case. Achlioptas [42] has shown that a much simpler distribution will work well and can replace the Gaussian

distribution. The distribution according to Achlioptas is:

$$m_{ij} = \sqrt{3} \cdot \begin{cases} +1 & \text{with probability } \frac{1}{6} \\ 0 & \text{with probability } \frac{2}{3} \\ -1 & \text{with probability } \frac{1}{6} \end{cases}$$

Practically, all zero mean, unit variance distributions of m_{ij} would give a mapping that will satisfy Johnson-Lindenstrauss lemma.

Random projection preserves the similiraties of the data vectors well even when the data is projected to moderate numbers of dimensions and the projection is fast to compute. For some comparisons between random projection and other dimensionality reduction methods see [11, 44].

In the following section, we'll discuss how projecting randomly multiple times assists in finding nearest neighbor (NN) in high dimension quickly with a high accuracy.

4.3 Multiple Random Projections (MRP) in finding Nearest Neighbor

From the previous discussions of RP, it can be realized that random projections are faster yet useful way of reducing dimensions. Taking a random projection of a high dimensional data with dimension d to a much smaller dimension k, obviously makes the computation much less but finding the nearest neighbor (NN) with that small amount of information is not always possible (because in Euclidean distance every dimension is equally important and independent of others). Figure 4.1 depict this picture. In this figure, the query point q is nearer to p_i than p_j. But after taking a single random projection of these points, p_j becomes closer to q and thus we may falsely select it as the nearest neighbor of q. If we take single random projection from d to k ($d >> k$) then there exists high probability of picking false points as the nearest neighbors. Also the figures from 4.2 to 4.4 reflect this behaviour. The data sets used here are all 256 dimensional. In these

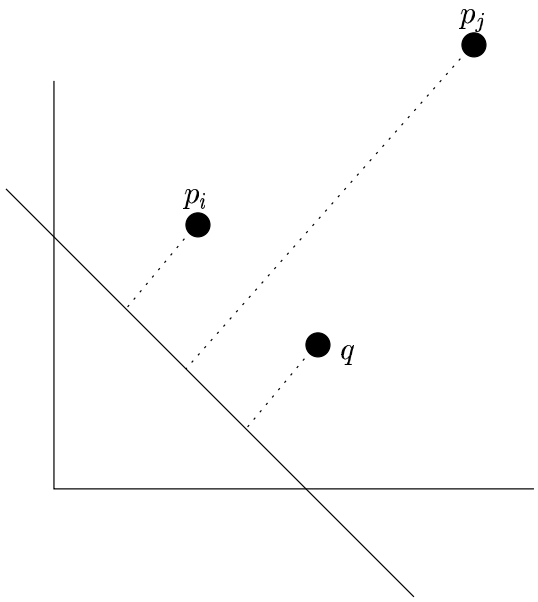


Figure 4.1: False Nearest Neighbor in single random projection. The point p_i is closer to the query point q than p_j. But in this projection p_j will be wrongly selected as the nearest neighbor.

figures, we've projected the data into various low dimensional subspace (less than the orginal dimension) and computed NN in that projection. From the figures, it is clear that single random projection is very much susceptible to error and it can't yield good accuracy rate in nearest neighbor search.

So one natural idea is to get more information from the projected data, and for this we need more projections. If we take enough random projections then there is a good probability that not all of them will give us false results. For each of the projections in low dimension, we calculate the exact NN from the projected data set (this can be fast since exact methods like k-d tree, SR-tree etc works well in low dimension). This exact NN may not be actual NN but it may be close to actual NN. If we take J projections then we'll have at most J exact NN as candidates of the actual answer. When we have exact NN for all the projections we hope that we've got better chance of finding the actual NN in them or at least these points may be close to the actual NN.

But sometimes taking only one NN per projection is not adequate. There is chance

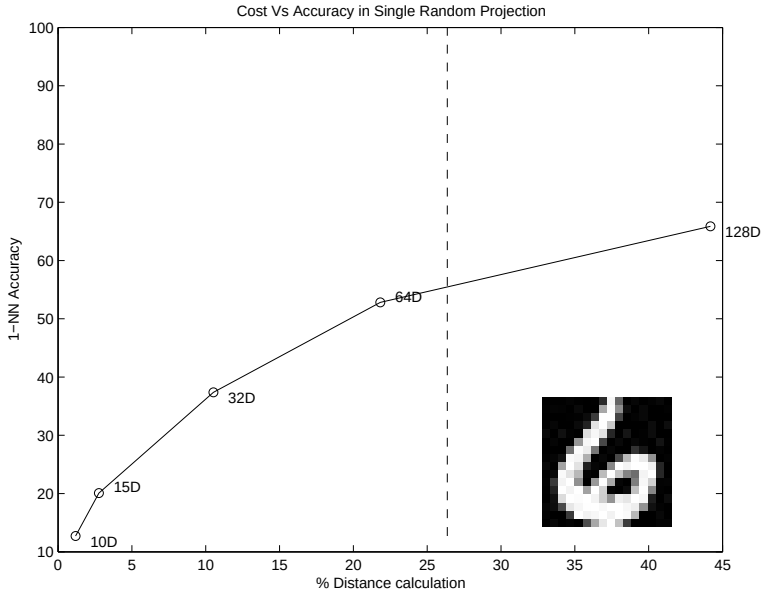


Figure 4.2: Single Random Projection - USPS Digit Data (256D). The horizontal axis denotes cost (in % distance calculation of Naive NN) and vertical axis denotes matching percentage with exact NN.

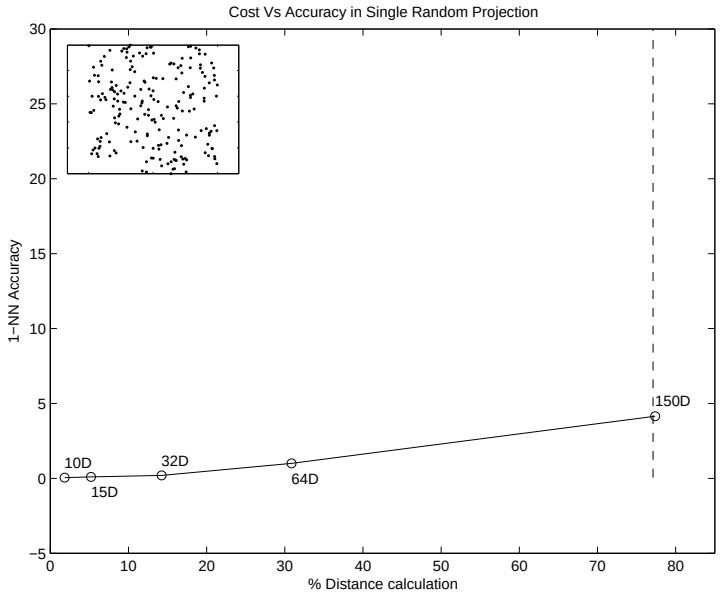


Figure 4.3: Single Random Projection - Random Uniform Data (256D) in the range $[0,1]^d$. The horizontal axis denotes cost (in % distance calculation of Naive NN) and vertical axis denotes matching percentage with 1-NN.

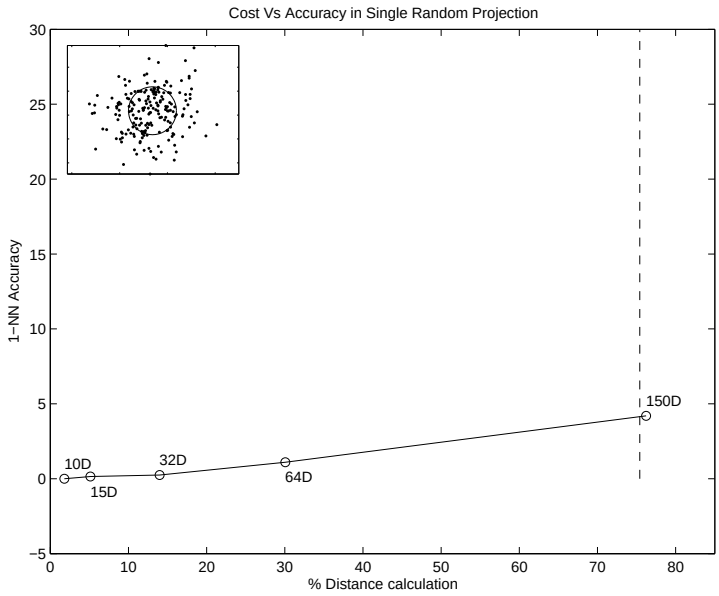


Figure 4.4: Single Random Projection - Single Gaussian Data (256D) with zero mean, standard deviation 1. The horizontal axis denotes cost (in % distance calculation of Naive NN) and vertical axis denotes matching percentage with 1-NN.

that we'll miss the actual NN in them. So, to be more robust, when we are finding NN in low dimensions we take first K-NN i.e. for each of the low dimensional projections we find the closest K-NN. We take the union of these to be the candidate set C of exact NN. If we have J projections and for each of them we have K-NN, then the candidate set will consist at most $J \times K$ points (some of them may appear multiple times). After that, we find the nearest neighbor of the query point q from only those points in C. In searching nearest neighbor in the set C, we use naive nearest with Early Break in original i.e. in all d dimension. As there are only few candidate points, which are much more less than actual data points, the computational cost of finding NN in actual high dimension among those points is not that much.

Choosing which exact NN technique should be used in low dimensional projections is another question. There are lots of efficient techniques for finding NN in low dimension and from our discussions in Chapter 2, we know that in lower dimension, k-d tree is the least expensive method than all other tree (SR-tree, R-tree, Ball tree) methods as well

as naive nearest methods. So, to make use of its high performance in lower dimension, we have used k-d tree as the exact NN technique in reduced dimension.

The pseudo-code of finding approx. NN in MRP technique is given in Algorithm 10.

Algorithm 10 finding approx. NN in MRP technique for query point q

Let the reduced dimension be k and original dimension be d

{Pre-Processing Stage}

for each of the J projections **do**

Make a Random Matrix $M^J_{k\times d}$ with each entry m_{ij} being set to an i.i.d. $N(0,1)$ value, i.e. let $M = randn(k,d)$.

Get the reduced data set $X^J_{k\times n} = M^J_{k\times d} X_{d\times n}$

Build k-d tree on the data set X^J.

end for

{Query Stage}

for each of the J projections **do**

Project the query point q into k dimension using the random matrix M^J constructed before.

Call k-d tree and return K-NN for the query point q using projected data set X^J

Store those K points as candidate points in a set C

end for

Take only the unique points in C and find the nearest point of q among those points using naive nearest method. Let p^{MRP} be the closest point of q in that set.

Output p^{MRP}

4.4 Experimental Results

For evaluating the performance of MRP, we'll use the following data sets :

- Real Life Data : Forest Cover Data(54D), USPS digit data (256D), Mnist digit data (784D), Orl Face data(10304D) and Feret Image Data (17154D).
- Synthetic Gaussian Data: Single Gaussian data with zero mean and unit standard deviation and Mixture of 10 Gaussians where the clusters are moderately well sep-

arated. Dimension of the data are 128, 256, 512, 1024. The generations of these data sets are given in previous Chapters. Each of these data sets have 8000 data points and 2000 query points generated from the same distribution.

- Synthetic Uniform Data: Random uniform data in the range $[0, 1]^d$. The dimensions are 128, 256, 512, 1024. For these data sets also we've used 8000 data points and 2000 query points.

Since there is no way of knowing beforehand what should be the optimal number of projections or what should be the projected dimension or what is the value of K per projection (i.e. how many NN per projection should be considered), we'll vary them and compare the cost associated with them with naive nearest with Early Break strategy. The reason we're using naive NN with Early Break (EB) because it is approximately half the cost of brute force approach and has similar cost but much less overhead than naive NN with Annulus Bound.

In figures 4.5 to 4.9, we've used different reduced dimensions (e.g. different k values) and for each of those dimension we have used multiple projections. Also the K value (number of NN/projection) is being varied in each of the projections. The broken vertical line indicates the cost of naive NN with Early Break (EB). The different reduced dimensions are arbitrarily chosen. The only constraint is that, k-d tree will be run on those low dimensional data set and we know that k-d tree's cost goes exponential when the dimension exceeds a certain threshold (in our experiment, we've used 20 as the maximum projected dimension). For each of the projected dimension and for each of the K-NN per projection, we've used different numbers of projection in the figures. The accuracy is measured w.r.t 1-NN i.e. whether the approximate answer of the query matches with the exact NN.

The area to the left of the straight vertical line indicates that in this region MRP's cost is less than Early Break's. Ideally we want the accuracy in this region to be close to

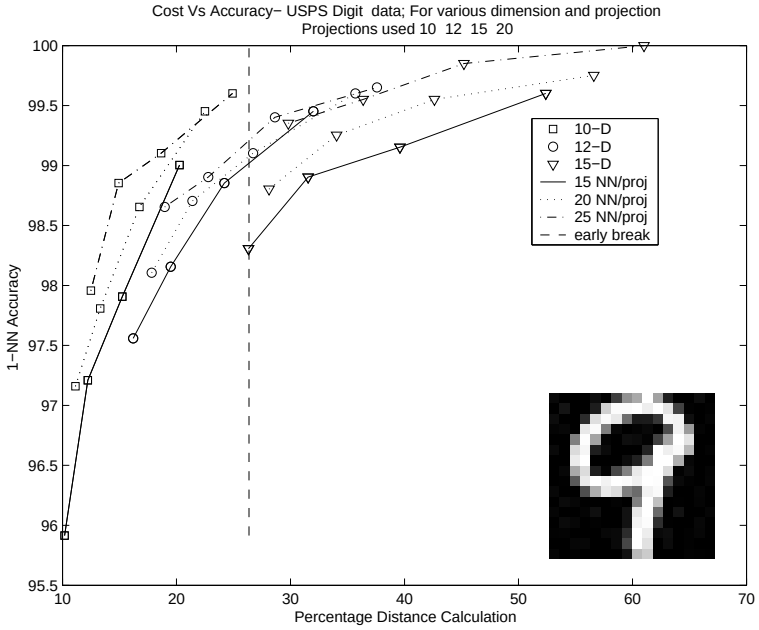


Figure 4.5: Multiple Random Projection - USPS Digit Data. The horizontal axis denotes cost (in % distance calculation of Naive NN) and vertical axis denotes matching percentage with 1-NN. For each dimension and for each K-NN/proj, we've used 4 different projections: 10, 12, 15, and 20 .

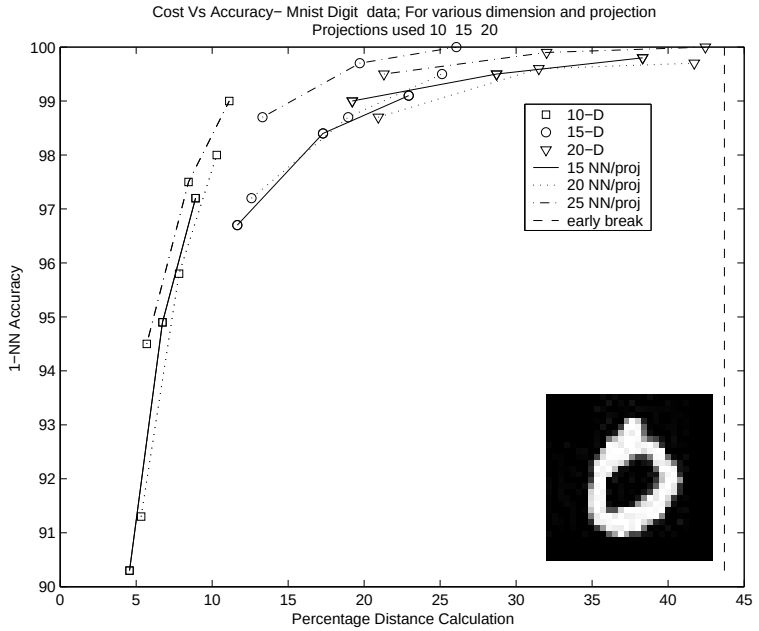


Figure 4.6: Multiple Random Projection - Mnist Digit Data. The horizontal axis denotes cost (in % distance calculation of Naive NN) and vertical axis denotes matching percentage with 1-NN. For each dimension and for each K-NN/proj, we've used 3 different projections: 10, 15 and 20.

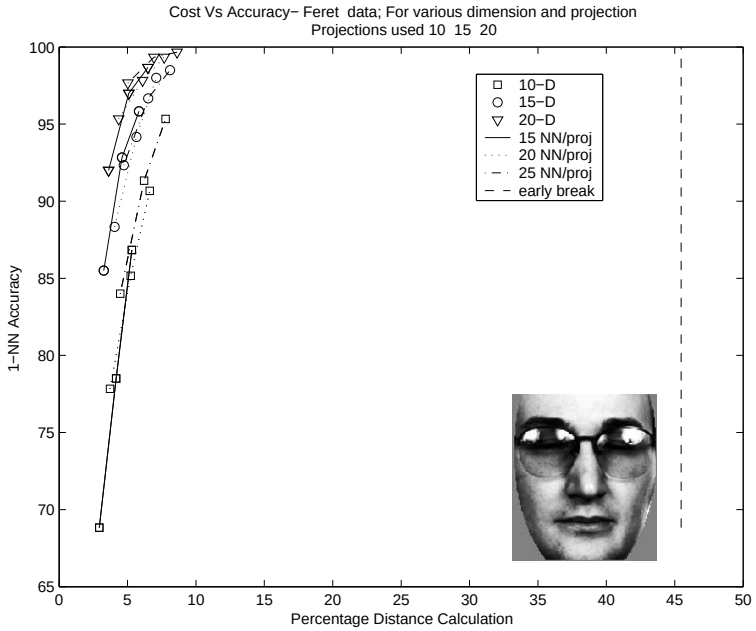


Figure 4.7: Multiple Random Projection - Feret Data. The horizontal axis denotes cost (in % distance calculation of Naive NN) and vertical axis denotes matching percentage with 1-NN. For each dimension and for each K-NN/proj, we've used 3 different projections: 10, 15 and 20.

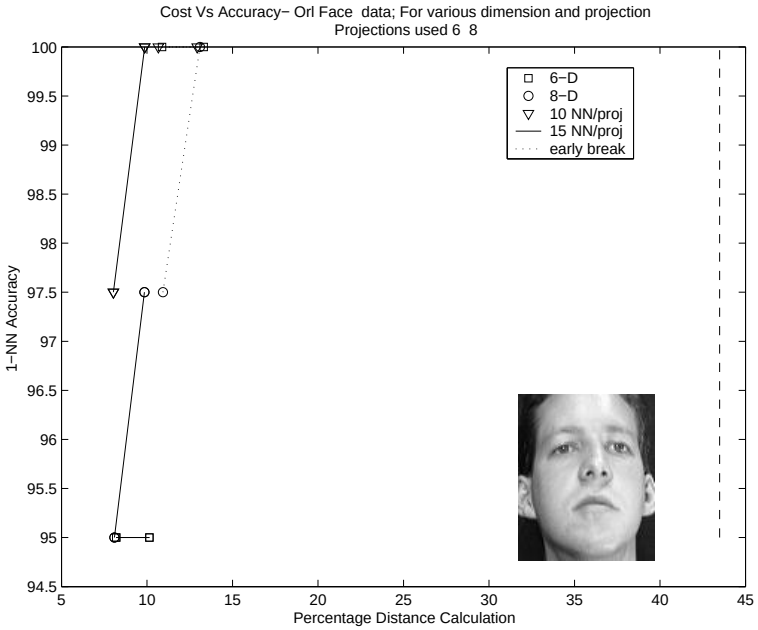


Figure 4.8: Multiple Random Projection - Orl Face Data. The horizontal axis denotes cost (in % distance calculation of Naive NN) and vertical axis denotes matching percentage with 1-NN.For each dimension and for each K-NN/proj, we've used 2 different projections: 6 and 8. The query set in this data set is very small (40) and with small number of projections the data set attains high accuracy.

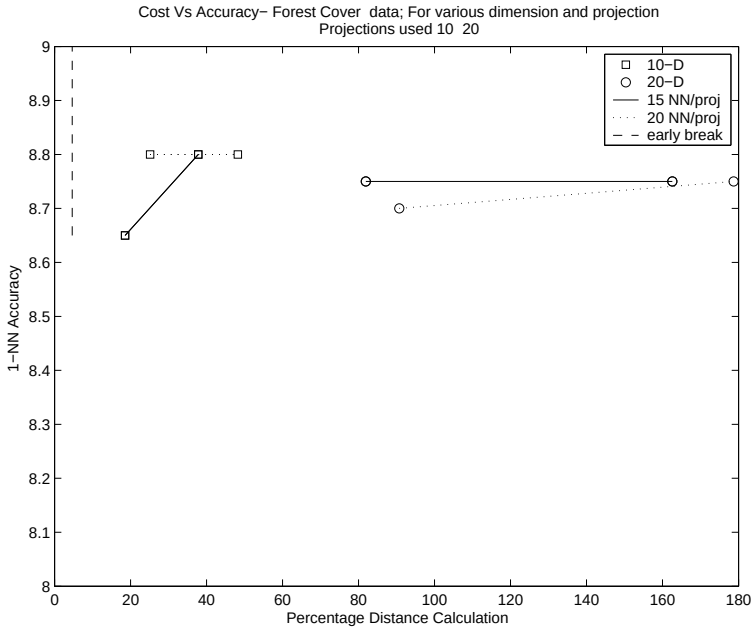


Figure 4.9: Multiple Random Projection - Forest Data. The horizontal axis denotes cost (in % distance calculation of Naive NN) and vertical axis denotes matching percentage with 1-NN. For each dimension and for each K-NN/proj, we've used 2 different projections: 10 and 20.

100%. We see that for most of the figures they are actually close to the perfect accuracy.

One of the reasons behind this good result is that these real life data sets have lower intrinsic dimensionality than their original dimensionality i.e. they lie on a low dimensional submanifold or they are clustered in some small fractions of the true space. When the intrinsic dimensionality is low, the amount of information loss becomes less. For example, if the dimension of a data set is 1000 and true dimensionality is 100, then by reducing the dimensionality to 20 dimension will not loose that much information as it was supposed to loose when the true dimensionality is the same as the original. So, when the difference between original and intrinsic dimensionality gets higher, MRP gets more useful.

We also want to compare the best result from MRP with that of Early Break (EB). The figures from 4.10 to 4.13 depict the results. In these figures, the (o) symbol indicates matching percentage with 1-NN and the (+) symbol indicates the matching percentage

with 5-NN. In figure 4.10, it shows that the for real data set the best result of MRP outperforms EB except for Forest data set. Forest data set has 40 binary dimensions (most of them are 0) out of 54 total dimensions. So, k-d tree can't get enough information to build the tree correctly and thus can't work well on this data set. But other than this data set, all the other data sets show good performance. As the dimension increases, the savings in computational cost become more prominent. Figure 4.10 describes what percentage of distance calculation is needed to achieve at least 99% accuracy from these data sets. For Forest data set, we can't get the desired accuracy with the projected dimensions that we've used, so we've just mentioned the best accuracy that can be achieved. In the figures from 4.10 to 4.13, the percentage numbers at the body of the figures indicate this lower accuracy rate (which is the best one that can be achieved) within the different reduced dimensions we've used.

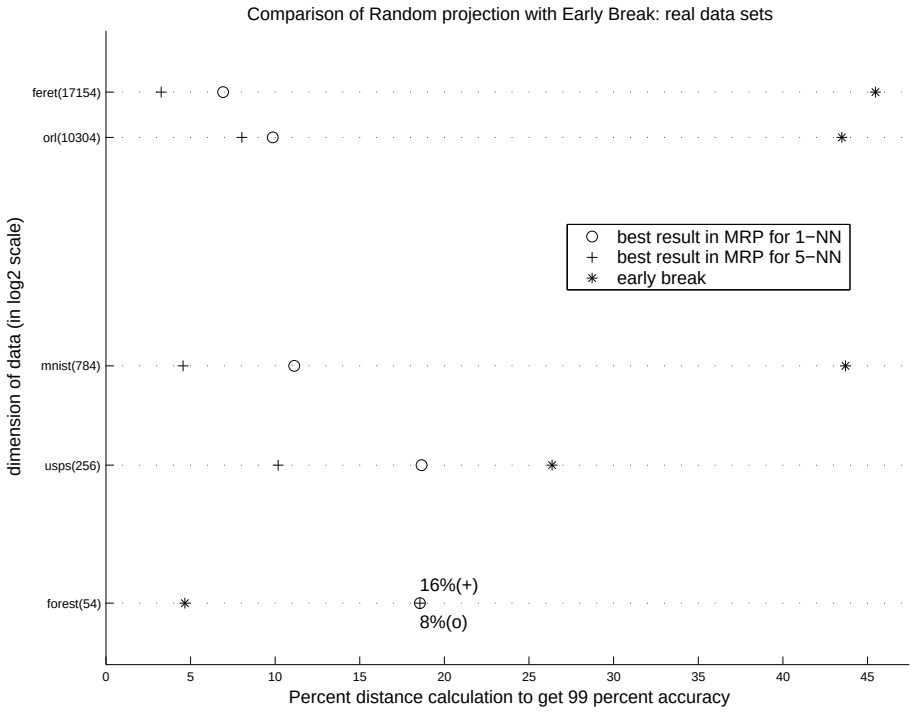


Figure 4.10: Comparison between best of MRP and Early Break. The least cost needed to get desired accuracy (99%) with 1-NN is shown by (o) and with 5-NN is shown by (+) sign. 1-NN indicates the approx result matches with exact NN and 5-NN indicates it matches with any of the 5 exact NN.

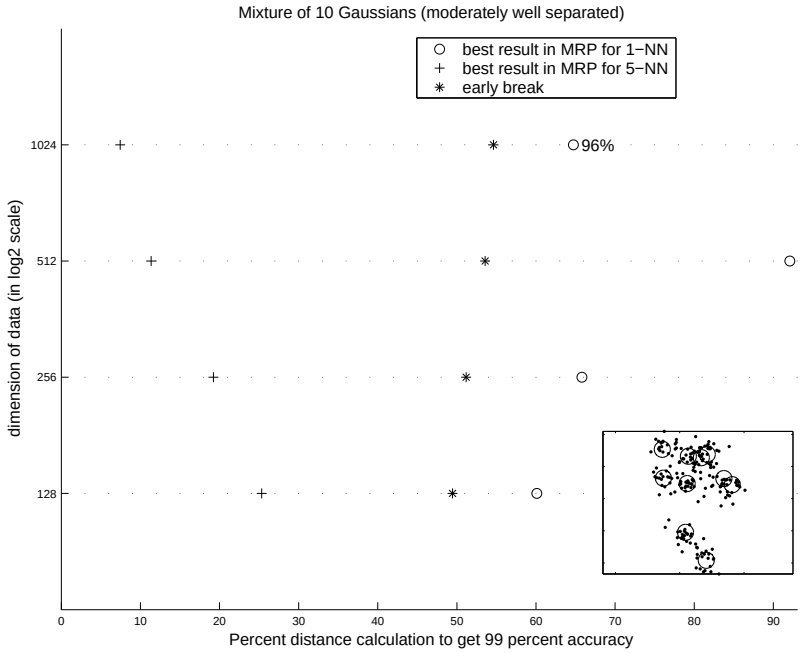


Figure 4.11: Comparison between best of MRP for mixture of well separated Gaussians and Early Break. The least cost needed to get desired accuracy (99%) with 1-NN is shown by (o) and with 5-NN is shown by (+) sign.

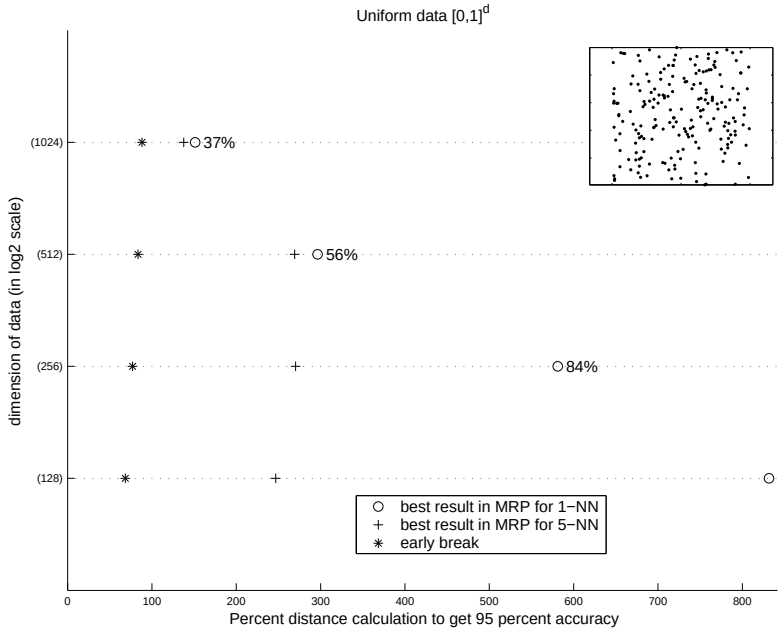


Figure 4.12: Comparison between best of MRP for random uniform data and Early Break. The least cost needed to get desired accuracy (95%) with 1-NN is shown by (o) and with 5-NN is shown by (+) sign.

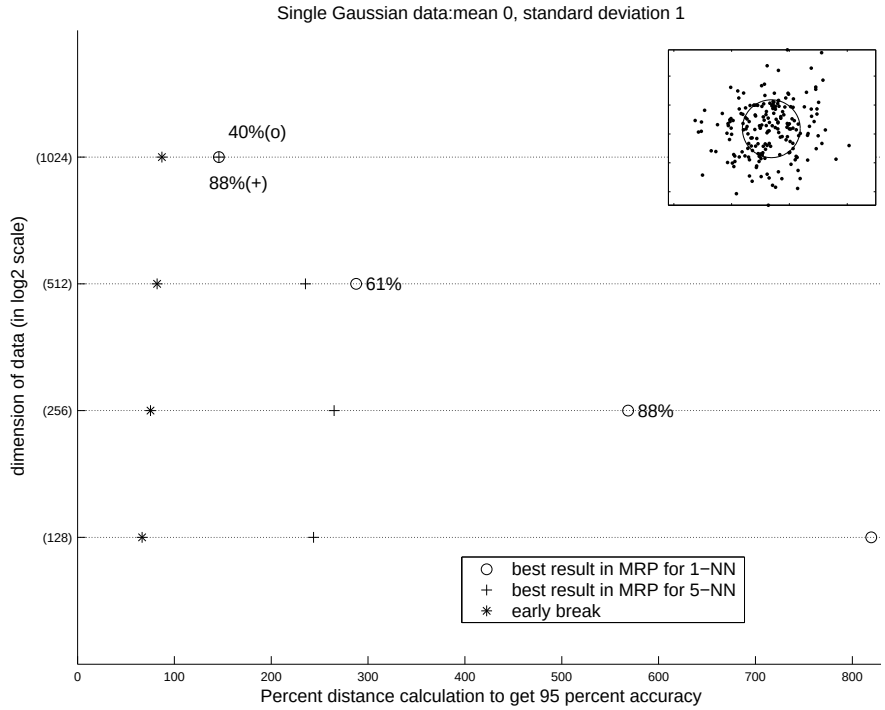


Figure 4.13: Comparison between best of MRP for single Gaussian data and Early Break. The least cost needed to get desired accuracy (95%) with 1-NN is shown by (o) and with 5-NN is shown by (+) sign.

In figure 4.11 we have used a mixture of 10 well separated Gaussian data. These data sets also perform reasonably well. Although it is tough to get the desired accuracy of 99% for 1-NN (i.e. the output of MRP matches with exact NN) with the projected dimensions that we've used, it is comparatively easy to achieve that accuracy if we consider 5-NN (i.e. the output of MRP matches with first 5 exact NN). In this experiment, we increased the projected dimension until we get the desired accuracy or the projected dimension reaches a certain threshold. If we can't get the desired accuracy within this threshold then we mention the best accuracy (in percentage) that is achieved. From our discussions in Chapter 2, we know that for well separated mixture of Gaussians, in high dimension the exact NN technique such as k-d tree is more expensive than EB. That means, even for well separated Gaussian data sets, in high dimension no exact NN algorithm can outperform EB. So, using MRP for these kind of data sets will give us results that are very close to exact results.

Figure 4.12 is for the uniform random data sets. These data sets are unstructured and purely random. So, with these data sets it is even hard to get reasonable accuracy w.r.t. 5-NN. For these kinds of data sets it is difficult to outperform naive NN approach. In this figure, it is shown that it takes considerable amount of computational cost to get a lower accuracy rate (95%).

The last figure we've considered is for single Gaussian data with zero mean and unit standard deviation (figure 4.12). These data sets are also purely random and not structured. The results for these data sets are very similar to the uniform data sets because of the same reason.

Chapter 5

Conclusions

We'll begin by summing up the work done in the thesis, indicating the main result and its performance and feasibility. We'll also describe some of the findings for various nearest neighbor techniques that we've explored through out the thesis. Next, we'll comment on some interesting directions for future research.

5.1 Summary

In this thesis, we have presented a fast approximation algorithm for finding nearest neighbors in high dimension using multiple random projections. Though the concept of single random projection is not new, it is interesting to see its effectiveness with multiple projections.

The principal method that we've used for finding nearest neighbor in this thesis is the multiple random projection approach in combination with a fast exact nearest neighbor technique (e.g. k-d tree) and a naive NN-technique (e.g. Early Break). The random projection to a few dimension from a large dimensional data sets makes it feasible for k-d tree to work on that data set in each projection. The k-d tree finds some candidate nearest neighbors for each projection of a particular query. When we take multiple projections of the data set and run k-d tree on each of the projected data set then we get more candidate

points for that query and the probability of finding the exact nearest neighbor among those candidate points gets higher. Then we use one of the naive nearest approach (Early Break strategy) on the union of those candidate points to find the nearest neighbor.

Also in this thesis, we've compared different exact nearest neighbor techniques and compared the performance of random projection technique with them. We've concluded that k-d tree is less costly than other tree structures (e.g. SR-tree, Ball tree and R-tree) when the dimension is small. With the increase in dimension, k-d tree and other tree structures deteriorate. SR-tree works better than k-d tree in moderate dimension but the computational cost is close to the naive nearest with Early Break method.

We've also used two approximation algorithms and compared those with our method. One of them is the Metric skip list and other is an ϵ-approximate algorithm by Kleinberg. Both the methods have large pre-processing overhead which limits their application to some of the data sets. The results returned by the algorithms are very close approximation of the actual nearest neighbor (i.e. they provide good accuracy).

5.1.1 Main Results

The main finding of this thesis is, Multiple Random Projection (MRP) works well when the data sets are high dimensional and have some underlying structure rather than purely random. That is, if the data set actually lies on some low dimensional manifold or is clustered in some fraction of the 'true' space then, by reducing the dimension with random projection, we can still get points that are close to the actual nearest neighbor (NN) or in some case we can find the actual NN. By applying naive nearest approach on these points which are in the close vicinity of the actual result, we can get the best one among them which has got a fairly high chance of being the exact NN. By increasing the number of reduced dimensions, number of projections and number of K-NN per projection, we can get higher accuracy at the expense of more computational cost.

We've used MRP technique on some real life data sets (such as digits, face images)

and got good accuracy rate on them. For well separated mixture of Gaussian data, this technique gets results that are very close to actual NN. But for uniform random data or single Gaussian data set, where all the dimensions are equally important (i.e. they don't live in some low dimensional subspace) this technique suffers from both accuracy and computational cost.

5.2 Future Work

In this section we'll consider possible directions for future work with the Multiple Random Projection approach. We'll discuss the things that are left undone and which will strengthen the effectiveness of random projections.

5.2.1 Proof of the new approach

In this thesis, we've explored the practical implementation of MRP. We didn't give any theoretical proof of this approach. It'll be very fruitful if we can give a proof of why the random projection works and what is the probability of finding the exact or approximate solution in this approach.

5.2.2 Optimal values of different parameters

While implementing MRP, we varied the dimension value, number of projections and also the number of candidate nearest neighbors for each of the projections. Some of these combinations exhibit good results i.e. they need lesser computational cost than brute force approach or any other exact nearest neighbor approach. But we didn't specify any method to find those combinations. It'll be another interesting project to find the optimal values of these parameters and prove them mathematically.

5.2.3 Optimal way of choosing from candidate points

A good amount of computation is needed when we try to find the nearest neighbor from the candidate points. This amount depends on the number of projection J and number of NN per projection K (the computation increases with the increase in J and K). In this implementation, we are using naive nearest method for finding NN among those candidate points. There could be other optimal ways of doing this rather than using naive nearest method. We store, for each of the J projections, K-NN. So, there should be at most $J \times K$ candidate points (as some of the points may appear multiple times). If we can find a weighted scheme to find the NN from those candidate points without using naive nearest method then we can save lots of computations. We've tried couple of weighted schemes on those points. One scheme is to output the point which has appeared most on the J projections (in case of ties output any of them). One another scheme is to give some weight for the K-NN in each of the J projections with the 1st NN having the highest weight (in our case it is 1) and gradually decreasing the weight till the Kth one. The second approach is little better than the first one but neither of them has given very satisfactory results.

5.3 The Last word

A huge number of literature about nearest neighbor search can be found in computer science research field because it is the very basis of some advance research in this field. Both exact and approximate nearest neighbor techniques find their usefulness in various areas. Depending on the nature of the application we can choose which technique will be useful. When the data is low dimensional very efficient exact NN-techniques exist, but for high dimension these algorithms are intractable. And for some of the applications, we do not need the exact point rather a close approximation is sufficient.

Multiple Random Projection (MRP) approach combining with k-d tree and Naive

Nearest with Early Break strategy provide us with some good approximation that can be extended to get near perfect accuracy. It works well with the data sets that have some underlying structure and lie in some low dimensional subspace. This promising scheme needs much closer study.

Bibliography

[1] A. Gersho, R.M. Gray. Vector Quantization and Data Compression. *Kluwer*, 1991.

[2] L. Devroye, T.J. Wagner. Nearest Neighbor Methods in Discrimination. *Handbook of Statistics*, volume 2, P.R. Krishnaiah and L.N. kanal, editors, North Holland, 1982.

[3] R. Lipton, R. Tarzan. Application of a Planar Separator Theorem. *SIAM J. Computing*, 9(1980), pp. 615-627, 1980.

[4] M.I. Shamos, D. Hoey. Closest Point Problems. *Proc. 16th Annu. IEEE Sympos. Found. Comput. Sci.*, 1975, pp. 152-162.

[5] D. Dobkin, R. Lipton. Multidimensional Search Problems. *SIAM J. Computing*, 5(1976), pp. 181-186.

[6] K. Clarkson. A Randomized Algorithm for Closest-point Queries. *SIAM J. Computing*, 17(1988), pp. 830-847.

[7] J. Matousek. Reporting Points in Halfspaces. *Proc. 32nd IEEE FOCS*, 1991.

[8] A.C Yao, F.F. Yao. A General Approach to d-dimensional Geometric Queries. *Proc. 17th ACM STOC* , 1985.

[9] S. Meiser. Point Location in Arrangements of Hyperplanes. *Information and Computation* , 106(1993), pp. 286-303.

[10] P. Indyk. Nearest Neighbors in High-dimensional Spaces. To appear in *Handbook of Discrete and Computational Geometry (2nd edition)*, J. E. Goodman and J. O'Rourke, editors. CRC Press LLC.

[11] S. Dasgupta. Experiments with random projection. In *Uncertainty in Artificial Intelligence: Proceedings of the Sixteenth Conference (UAI-2000)*, pp. 143-151, San Francisco, CA, 2000.

[12] S. Arya. Nearest Neighbor Searching and Applications. *PhD thesis, University of Maryland technical report CS-TR-3490*, June 1995.

[13] S. Arya, D. Mount, N. Netanyahu, R. Silverman, A. Wu. An Optimal Algorithm for Approximate Nearest Neighbor Searching in High Dimensions. *Proc. 5th ACM-SIAM SODA*, 1994.

[14] K. Clarkson. An Algorithm for Approximate Closest-point Queries. *Proc. 10th ACM Symp. on Computational Geometry*,1994.

[15] J. Kleinberg . Two Algorithms for Nearest-neighbor Search in High Dimensions. *Proc. 29th Annu. ACM Sympos. Theory of Computating*, 1997.

[16] P. Indyk, R. Motwani. Approximate Nearest Neighbor - Towards Removing the Curse of Dimensionality. In *Proceedings of the 30th Symposium on Theory of Computing*, pp. 604-613, 1998.

[17] A. Gionis, P. Indyk, R. Motwani. Similarity Search in High Dimensions via Hashing. In *Proceedings of the 25th VLDB Conference*, 1999.

[18] J. Friedman, J. Bentley, R. Finkel.An Algorithm for Finding Best Matches in Logarithmic Expected Time. *ACM Trans. on Mathematical Software*, 3(1977), pp. 209-226.

[19] J. Bentley. Multidimensional Binary Search Trees used for associative searching. *Communications of the ACM*, 18(1975), pp. 509-517.

[20] D. White, R. Jain. Similarity Indexing with the SS-tree. *Proc. of the 12th Int. Conf. on Data Engineering, New Orleans, USA*, 1996, pp. 516-523.

[21] A. Guttman. R-trees: A Dynamic Index Structure for Spatial Ssearching. In *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pp. 47-57, 1984.

[22] S. Omohundro. Five Balltree Construction Algorithms. *Technical report 89-063, International Computer Science Institute, Berkeley, California*, 1989.

[23] H. Samet. The Quadtree and Related Hierarchical Data Structures. *ACM Computing Surveys*, 16(1984), pp. 187-206.

[24] K. Zatloukal, M.H. Johnson, R.E. Ladner.Nearest Neighbor search for Data Compression. *Data Structures, Near Neighbor Searches, and Methodology: Fifth and Sixth DIMACS Implementation Challenges*, edited by M.H. Goldwasser, David S. Johnson, and Catherine C. McGeoch, American Mathematical Society, 2002, pp. 69-86.

[25] A.W. Moore. Efficient Memory-based Learning for Robot Control.*PhD. Thesis; Technical Report No. 209*, Computer Laboratory, University of Cambridge, 1991.

[26] N. Roussopoulos, S. Kelly, F. Vincent.Nearest Neighbor Queries. In *Proceedings of ACM SIGMOD Conference on Management of Data*, May 1995.

[27] N. Katayama, S. Satoh. The SR-tree: An Index Structure for High-Dimensional Nearest Neighbor Queries. *ACM Sigmod Conference*, pp. 369-380,1997.

[28] N. Beckmann, H. Kriegel, R. Schneider, B. Seeger. The R*- tree : An Efficient and Robust Access Method for Points and Rectangles. In *Proc. of ACM SIGMOD*, pp. 322-331, Atlantic City, NJ, May 1990.

[29] D.R. Karger, M. Ruhl. Finding Nearest Neighbor in Growth-restricted Metrics. In *34th Annual ACM Symposium on the Theory of Computing*, pp. 63-66, 2002.

[30] R. Krauthgamer, J. Lee. Navigating nets: Simple algorithms for proximity search. *Proceedings of the 15th Annual Symposium on Discrete Algorithms (SODA)*, pp. 791-801, 2004.

[31] A. Beygelzimer, S. Kakade, J. Langford. Cover Trees for Nearest Neighbor. Preprint, 2004. Available at http://hunch.net/~jl/projects/cover_tree/cover_tree.html

[32] G.H. Golub, C.F. van Loan. Matrix Computations. *North Oxford Academic*, Oxford, UK, 1993.

[33] S. Roweis. EM Algorithms for PCA and SPCA. In *Neural Information Processing Systems 10*, pp. 626-632,1997.

[34] M. Tipping, C. Bishop. Probabilistic principal component analysis. *Technical Report Technical Report NCRG/97/010, Neural Computing Research Group*, Aston University, Birmingham, UK, September 1997.

[35] C.H. Papadimitriou, P.Raghavan, H. Tamaki, S. Vempala. Latent Semantic Indexing: A probabilistic analysis. *Proc. 17th ACM Symp. on the Principles of Database Systems*, pp. 159-168,1998.

[36] S. Deerwester, S.T. Dumais, G.W. Furnus, T.K. Landauer. Indexing by latent semantic analysis. *Journal of the Am. Soc. for Information Science*,41(6):391-407, 1990.

[37] K.R. Rao, P.Yip. Discrete Cosine Transform: Algorithms, Advantages, Applications. *Academic Press*, 1990.

[38] M. Sonka, V. Hlavac, R. Boyle. Image Processing, Analysis and Machine Vision. *PWS publishing*, 1998.

[39] S. Dasgupta, A. Gupta. An Elementary Proof of the Johnson-Lindenstrauss lemma. *Technical Report TR-99-006*, International Computer Science Institute, Berkeley, California, USA, 1999.

[40] P. Frankl, H. Maehara. The Johnson-Lindenstrauss lemma and the sphericity of some graphs. *Journal of Combinatorial Theory, Ser. B*, 44:355-362, 1988.

[41] R. Hect-Nielsen. Context Vectors: general purpose approximate meaning representations self organized from raw data. In *Computational Intelligence: Imitating Life*, J.M. Zurada, R.J. Marks II, C.J. Robinson, editors, pp. 43-56, IEEE Press, 1994.

[42] D. Achlioptas. Database-friendly Random Projections. *Proc. ACM Symp. on the Principles of Database Systems*, pp. 274-281, 2001.

[43] W.B. Johnson, J. Lindenstrauss. Extensions of Lipshitz Mapping into Hilbert Space. *Conference in modern analysis and probability*, vol. 26 of *Contemporary Mathematics*, pp. 189-206, American Math. Soc., 1984.

[44] E. Bingham, H.Mannila. Random Projection in Dimensionality Reduction: Application to Image and Text Data. *Proc. 7th ACM SIGKDD Int. Conf. on Knowledge Discovery and Data Mining (KDD-2001)*, pp. 245-250, 2001.

[45] M.W. Berry. Large-scale sparse singular value computations. *International Journal for Super-Computer Applications*, 6(1):13-49, 1992.