UNIVERSITY OF CALIFORNIA, SAN DIEGO

Fast Tree Based Nearest Neighbor Search

A Thesis submitted in partial satisfaction of the requirements
for the degree Master of Science

in

Computer Science

by

Zhen Zhai

Committee in charge:

Professor Sanjoy Dasgupta, Chair
Professor Yoav Freund
Professor Ramamohan Paturi

2017

The Thesis of Zhen Zhai is approved and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____

Chair

University of California, San Diego

2017

DEDICATION

To my family

EPIGRAPH

How long should you try? Until.

—Jim Rohn

TABLE OF CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to acknowledge Professor Sanjoy Dasgupta for his guidance and advises, for being my very first mentor who taught me research and machine learning, and for his patience over my past three years of research studies. Many thanks to Professor Yoav Freund for teaching me how to do good research and how to present my work clearly, and for giving me a wonderful space in his lab where many of my projects happen. I would also like to thank Professor Ramamohan Paturi for supporting and encouraging me to pursue my degree and my research projects. His words have made me stronger and reminded me to never give up. I wouldn't be who I am without any of my mentors' support.

Chapter 4, in part, has been submitted for publication of the material as it may appear in Nearest Neighbor Search Using Twin-Vantages Forests, Zhai, Zhen; Dasgupta, Sanjoy, 2017. The thesis author was the primary investigator and author of this material.

ABSTRACT OF THE THESIS

Fast Tree Based Nearest Neighbor Search

by

Zhen Zhai

Master of Science in Computer Science

University of California, San Diego, 2017

Professor Sanjoy Dasgupta, Chair

Nearest neighbor search is a basic primitive method used for machine learning and information retrieval. We look at exact nearest neighbor search algorithms using tree structures. The most basic tree structure used for fast nearest neighbor search is k-d trees. This thesis will look at k-d tree's shortcomings and explore various ways to improve its performance. First, we look at PCA trees, which give good performance but is time-expensive. We then study randomized trees, which are very efficient data structures and are flexible in space complexity. Then we introduce a new randomized tree structure, two-vantage-point tree, which outperforms all other tree structures including PCA trees, r-k-d trees, and RP trees. At last, we look at spillover on trees, which can be

used to improve the performance of any tree structures. We then compare randomized trees with spillover and show that spill trees only work well with very small spill factor. If more space is allowed, two-vantage-point trees are preferred over spill trees.

Chapter 1

Nearest Neighbor Search

1.1 Introduction

In a nearest neighbor(NN) search problem, we are given a dataset S of size n, $S = \{x^{(1)}, x^{(2)}, ..., x^{(n)}\}$, where $x^{(i)}$ are vectors in d dimensional space $\mathbb{R}^d$. Then given a query data q, NN search will output a data point $x^{(i)}$ for $1 \leq i \leq n$, that $x^{(i)}$ and q has the shortest distance comparing to all other data points in S. The distance between query q and data $x^{(i)}$ can be measured by different distance function. We used Euclidean distance function D

$$D(q, x^{(i)}) = \sum_{k=0}^{d} (x_k^{(i)} - q_k)^2$$

The NN search will be

$$\text{NN}(q, S) = \arg\min_{x \in S} D(q, x)$$

NN search is used in many areas of study, including computer vision, machine learning, and data mining. Although NN search can be broadly used across different fields, it is computationally intensive due to the distance computation of n data points in $\mathbb{R}^d$ where both n and d could be arbitrarily large. A linear search would take time $O(nd)$

for each query. Therefore, a significant amount of research devoting to speed up NN search has been conducted in the past decades.

1.2 K-D Tree

One of the most popular structures used by NN search is k-dimensional(k-d) tree, a multidimensional binary search tree[1] [6]. K-d trees do binary partition recursively. It starts by selecting one feature direction to partition the data points from the root into two subsets evenly. Then, it repeats the partition recursively until the size of the subset reaches certain size limit. See Algorithm 1.

Algorithm 1 Building k-d tree

Input: dataset S, minimum leaf size m
Output: tree node

function BUILD_TREE(S, m)
  if $|S| < m$ **then**
    return $leaf \leftarrow$ leaf node created with points in S
  else
    $T \leftarrow$ tree node
    $T, S_L, S_R =$ KD_PARTITION(T) $\leftarrow$ call a partition function
    $T.L \leftarrow$ left subtree of T
    $T.R \leftarrow$ right subtree of T
    $T.L =$ BUILD_TREE($S_L, T.split\_value$)
    $T.R =$ BUILD_TREE($S_R, T.split\_value$)
    return T
  end if
end function
function KD_PARTITION(T)
  $T.split\_coord = c \leftarrow$ select coordinate with max variance
  $T.split\_value = median\{x_c : x_c \in S\}$
  $x_c \leftarrow$ value at split coordinate c of data x.
  $S_L = \{x \in S : x_c \leq T.split\_value\}$
  $S_R = \{x \in S : x_c > T.split\_value\}$
  return T, S_L, S_R
end function

K-d trees select partition directions by finding the feature coordinate with max-

imum variance. Then the data points are split at the median of the chosen direction. It always split evenly so that the left subtree and the right subtree have the same size. However, this can sometimes be hard to achieve in practice, because we could have multiple data points with the same value as the median in the chosen direction. Therefore, we need to deal with these data points separately.

Here, we introduce a tie-breaking method to make sure an even split at each tree level. First, we collect all the data points that have the median value in the split direction, then project all these data points to a random vector with normal distribution $\mathcal{N}(0,1)$. The projection will map each data point to a value. We sort the projected value and find the k^{th} smallest pivot value, where k is the number of data points needed in left subtree. The data points which has projected value smaller than the pivot will go to the left subtree, and the rest will go to the right subtree. Therefore the size of both subtrees is equal. It is guaranteed an even split at each tree level. See Algorithm 2.

Algorithm 2 Break Tie

function KD_PARTITION(T)
    $T.split\_coord = c \leftarrow$ coordinate with max variance
    $s_c \leftarrow$ value at coordinate c of data s.
    $T.split\_value = median\{s_c : s_c \in S\}$
    $S_L, S_R =$ BREAK_TIE_PARTITION(S, T)
    return T, S_L, S_R
end function
function BREAK_TIE_PARTITION(S, T)
    $S_L = \{x \in S : x_c < T.split\_value\}$
    $S_R = \{x \in S : x_c > T.split\_value\}$
    $P = \{x \in S : x_c = T.split\_value\}$
    $T.tie\_breaker = v \leftarrow$ random vector from $\mathcal{N}(0,1)$
    $T.tie\_pivot = pivot \leftarrow$ the split value for projected data
    $S_L = S_L + \{x \in P : v \cdot x \leq pivot\}$
    $S_R = S_R + \{x \in P : v \cdot x > pivot\}$
    return S_L, S_R
end function

After we build a k-d tree from the training set, we pass a query data q into a

search algorithm which will return the NN of query q. Searching starts from the root down to the leaves. We compare query q with the split median of each tree node. The query q will then go to either the left subtree or the right subtree depending on whether it is smaller than or greater than the median. We repeat the above step until query q falls into a leaf node l. Then we do a linear search with all the data points within the leaf to find the nearest neighbor n.

$$n = \mathrm{NN}(q,l) = \arg\min_{s \in l} D(q,s)$$

One problem is that the query may end up falling into a leaf node that doesn't contain the true nearest neighbor n^*. To solve this, we need to search more than one leaf node to find the true NN. We do the comprehensive search, where multiple leaves are searched in a tree. The search algorithm decides whether to search more than one leaf node by comparing the distance from query to the returned neighbor n_0 with the distance to the split boundary. If the distance to the split boundary is smaller than the distance to the returned neighbor, it searches the leaf on the other side of the boundary as well. After searching the other leaf node, another nearest neighbor n_1 will be returned. Then we compare the distance from query q to n_0 and n_1, then return the data with smaller distance.

$$n = \arg\min_{i \in \{n_0, n_1\}} D(q,i)$$

We apply this algorithm recursively until the distance from query to the returned neighbor is smaller than the distance to the split boundary. This can guarantee we find the true NN of q in the dataset. See Algorithm 3 and Figure 1.1.

Although the comprehensive search algorithm guarantees to find the true NN, it could end up doing a linear search on the entire dataset S. Imagine the algorithm compares distance $D(q,n)$ with distance $D(q,b)$, where b is the split border, we may

Algorithm 3 K-D Tree Comprehensive Search

Input: query q, k-d tree node T
Output: nearest neighbor

function K-D_TREE_SEARCH(q, T)
  if T is leaf **then**
    return $\arg\min_{s \in T} D(q,s)$
  else
    $c = T.split\_coord$
    $q_c \leftarrow$ value at coordinate c of query q.
    if $q_c \geq T.split\_value$ **then**
      $nn_R =$ K-D_TREE_SEARCH($q, T.R$)
      $d_{split} \leftarrow$ distance between q and the split
      $d_{nn} \leftarrow$ distance between q and nn_R
      if $d_{split} \leq d_{nn}$ **then**
        $nn_L =$ K-D_TREE_SEARCH($q, T.L$)
        return $\arg\min_{s \in \{nn_L, nn_R\}} D(q,s)$
      else
        return nn_R
      end if
    else
      $nn_L =$ K-D_TREE_SEARCH($q, T.L$)
      $d_{split} \leftarrow$ distance between q and the split
      $d_{nn} \leftarrow$ distance between q and nn_L
      if $d_{split} \leq d_{nn}$ **then**
        $nn_R =$ K-D_TREE_SEARCH($q, T.R$)
        return $\arg\min_{s \in \{nn_L, nn_R\}} D(q,s)$
      else
        return nn_L
      end if
    end if
  end if
end function

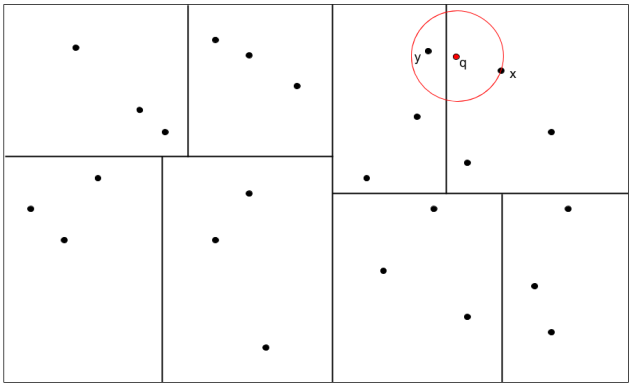

Figure 1.1: K-d tree partition with leaf size equals to three. The red point is query q. The first returned neighbor is point x. The comprehensive search algorithm measures and compares the distance between q and the split boundary with the distance between q and x. Because the distance from q to the split boundary is smaller, the leaf node on the other side of the split will be searched as well. Then the true nearest neighbor y will be returned after searching the other leaf.

have $D(q,n) > D(q,b)$ all the time. In this case, it will apply linear search on all the tree nodes, which is the same as doing a linear search on the entire dataset. Therefore, it could take time $O(n)$, where n is the size of the dataset S.

1.3 Defeatist Search

It could be very time consuming to find the true NN using comprehensive search. Therefore, in practice, the defeatist search is used broadly. The defeatist search algorithm only searches one leaf for each query. It will only take time $O(log\ n)$ comparing to $O(n)$ in the comprehensive search. See Algorithm 4. However, defeatist search can't guarantee to find the true NN. Therefore, we have the trade-off between time complexity and the accuracy rate.

The reason that defeatist search can't guarantee to find the true NN is that the query q may fall into a leaf node that doesn't contain the true NN. See Figure 1.2. The

Algorithm 4 K-D tree Defeatist Search

Input: query q, k-d tree node T
Output: nearest neighbor

```
1: function DEFEATIST_SEARCH(q, T)
2:     if T is leaf then
3:         return arg min_{s∈T} D(q, s)
4:     else
5:         c = T.split_coord ← Split direction
6:         q_c ← value at coordinate c of query q.
7:         if q_c ≥ T.split_value then
8:             return DEFEATIST_SEARCH(q, T.R)
9:         else
10:            return DEFEATIST_SEARCH(q, T.L)
11:        end if
12:    end if
13: end function
```

defeatist search could end up searching the wrong node when the query point is very close to the border of the split. If query q traces down to the node where its true NN doesn't belong, the true NN of q will not be returned from the algorithm. Therefore, the true NN accuracy is essential in the defeatist search. There has been much research devoting to find a way to improve the true NN accuracy in defeatist search; it is also what we focus on in this paper. We will look at different variations of the k-d tree and compare the NN accuracy.

1.4 Experiments

We evaluate the performance of k-d trees and other tree structures using training set and test set. The training set is the set of points that are used to build the model. The test set is the set containing queries. K-d trees, for example, first use training set to build k-d trees, then use data points from the test set as queries and search for NN in the trees.

We find the accuracy of a data structure by comparing the returned NN with the true NN. We first find the true NN of each query in the test set using linear search over

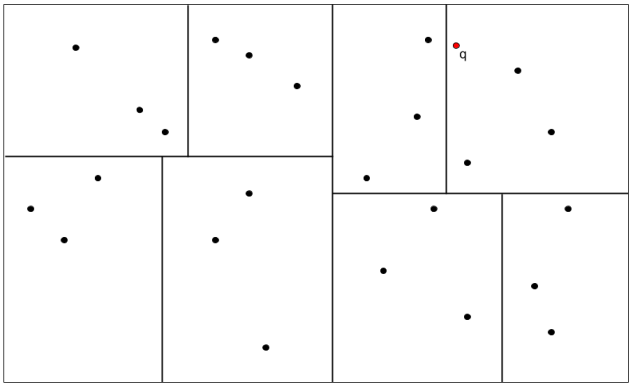

Figure 1.2: Defeatist search in a k-d tree will not find the true NN in this graph. The graph shows a k-d tree partition with leaf size 3. The red point is query q. We see that the true NN of q is not in the same leaf node as q. Therefore, defeatist search won't find the true NN.

all data in training set. Then we build models from the training set and use the models to search for NN of query points. We compare the returned NN with the true NN and we measure the number of queries that don't have their true NN returned from the algorithm. The error rate will then be the fraction of correct NN returned.

$$\text{Fraction of correct NN} = \frac{\text{number of true NN returned}}{\text{number of queries}}$$

We use six different datasets to verify the performance. We have CIFAR, MNIST, SIFT, BIG5, the Million Songs, and Word2Vec. We do pre-process on each dataset by removing duplicate data. Having too many duplicates in the training set causes problems when we try to split a set into two at the lower level of trees.

CIFAR

We first have CIFAR, a dataset of small images of ten different classes, collected by Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. CIFAR's classes include airplane,

automobile, bird, cat, deer, dog, frog, horse, ship, and truck. See Figure 1.3. It has 50,000 color images of size 32 by 32 in the training set and 10,000 images in the test set.

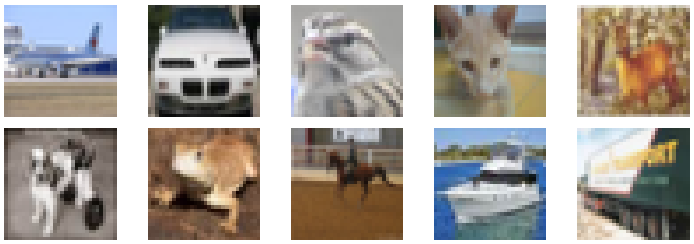

Figure 1.3: These are ten small images from CIFAR. Each of them belongs to one of the classes in CIFAR. The first row has classes airplane, automobile, bird, cat, and deer. The second row has classes dog, frog, horse, ship, and truck.

Using the raw pixel data from CIFAR gives us poor true NN accuracy because we use Euclidean distance as our distance function. Therefore, we use k-means clustering to extract feature vectors that will better represent the images in CIFAR. We extract 1000 features from CIFAR using k-means clustering with the ”triangle” activation function introduced by [3]. The extracted features are shown to yield good performance comparing to other feature extraction methods in [3].

MNIST

We then have MNIST, a dataset containing small images of handwritten digits of 0 to 9. See Figure 1.4. This dataset is collected from NIST by Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. The images are of size 28 by 28. The features are simply the pixels of the images. Therefore, the dimension of the data is 784. There are 60,000 such gray-scaled images in the training set and 10,000 in the test set. Unlike CIFAR, raw pixel data from MNIST yield good true NN performance in the NN search algorithm. Therefore we simply use the raw pixel data in our experiments.

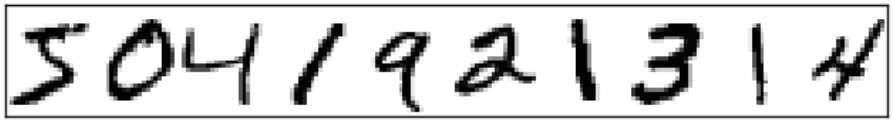

Figure 1.4: Here are a few images from MNIST training set. Each image represents a hand written number.

SIFT

We also have SIFT dataset[9][8]. Scale-invariant feature transform(SIFT) is an algorithm used to extract useful feature vectors from images. The training and test datasets we use are SIFT vectors derived from INRIA Holidays images, a dataset containing personal holiday photos[8]. The INRIA dataset also contains photos that are taken on purpose with rotations and blurring used to test the robustness of algorithms. Therefore it is a good idea to have SIFT vectors extracted and used those for our experiments.

There is a total of 1000,000 SIFT vectors with dimension 128 in the training set. After we removed the duplicate data, we have around 985,000 left in the training set and 10,000 in the test set. The SIFT dataset is one of the largest dataset used to evaluate NN search[9].

BIG5

BIG5, also known as Five Factor Model, is a model used to describe human personality. The five factors include openness, conscientiousness, extroversion, agreeableness, and neuroticism. NEO-PI-R(NEO Personality Inventory, Revised) questionnaire is a measure of these five factors; it's the revised version of NEO Personality Inventory by Paul Costa, Jr. and Robert McCrae[12]. The dataset we use here is the 100-item IPIP(International Personality Item Pool) proxy of the NEO-PI-R [10]. In another word, each data contains 100 questions with the answer of scale 1-5. We preprocess the data set by filtering out the invalid responses, for example, the responses with "Agree" selected

throughout all 100 questions[10].

The BIG5 dataset is of size one million. Each data point represents one human response to the 100 questions. Therefore, each data point has a dimension of 100. After we remove the duplicate data points, we randomly draw 10,000 as our test set and the rest 996,171 data as our training set.

Million Songs

Furthermore, we have the Million Songs dataset, a dataset containing many different features of a million different songs[2]. We extract the timbre feature of the introduction section of each song, which includes the first 50 segments of a song and each segment has 12 timbre values. The timbre data are all floats, therefore we have 600 floats value for each song. We randomly draw 990,000 data points from a million points as training set and the test set has 10,000 data points. After removing the duplicates, we have 951,593 in the training set.

Word2Vec

Finally, we have Word2Vec[14]. The Word2vec model is a group of two-layer neural networks trained for word embedding. The Word2vec aims at giving semantically related words similar embeddings. It is shown that vector of ”King” + ”Man” + ”Women” is a vector closest to the vector of ”Queen”[15].

The model we use is pre-trained on a selection of the original Google News dataset and produces 300-dimensional word embeddings. About 100 billion words from the original Google News dataset are used to train the Word2vec dataset. The Word2vec dataset has 3 million vectors each of dimension 300. Each vector represents a word or a phase obtained from [14].

Figure 1.5 shows the plot of the fraction of correct NN with the number of distance computation for the k-d tree. The number of distance computation is the number of points we perform the linear search on for each query point. We control the size of distance computation by limiting the depth of the tree each query can reach. The deeper the tree the query goes, the smaller the tree node it will reach. Therefore, the number of distance computation increases by a power of two along the curves in the graphs. We can see from the graphs that as the number of distance computation increase, the accuracy of the true NN search increase accordingly. In another word, the more data we use for linear search, the higher the true NN accuracy. In later sections of the paper, we will use the performance of k-d tree defeatist search as the baseline to validate the performance of other data structures.

1.5 Curse of Dimension

In practice, datasets increasingly tend to have high dimension. The high dimensional causes problems in many data structures that are used for the nearest neighbor search. For example, k-d trees partition the data points in one direction at a time. In another word, k-d trees use one feature for each split. However, one single feature will be less and less informative as the dimension of the data increases. As a result, the accuracy of NN search will decrease as the dimension increase. Such problem is referred as the curse of dimensional[1].

In Figure 1.6, we generate 1000 training data and 200 test data from Gaussian distribution with different dimensions d. We perform k-d tree on each of the datasets. When the dataset is of dimension two and three, the true NN accuracy can be around 90%. However, increasing the dimension to five makes the accuracy drop to between

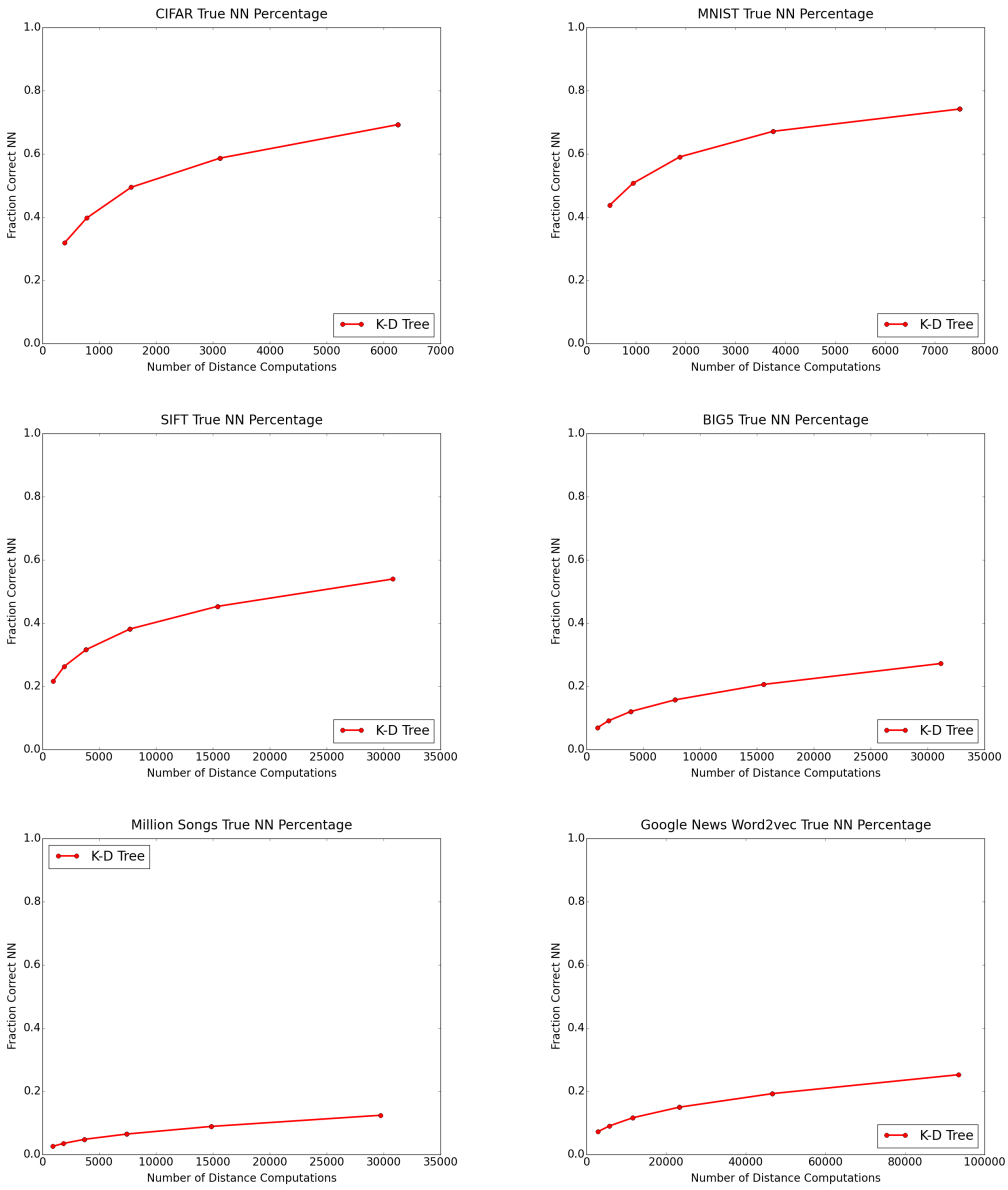

Figure 1.5: These are the k-d tree accuracy for each of the datasets. Both CIFAR and MNIST have true NN accuracy around 50% using k-d trees. And SIFT has around 40%. However, BIG5, the million songs, and Word2vec have very low true NN accuracy using k-d trees, around 10%. We will use k-d tree true NN accuracy as our baseline for our experiments on other tree structures.

60% and 70%. Increasing the dimension to 10 makes the accuracy drop to 30%-50%. When we have dimension bigger than 100, the accuracy drops below 20%. As shown in the Figure 1.6, the true NN accuracy drops very fast as we increase the dimension of the

dataset.

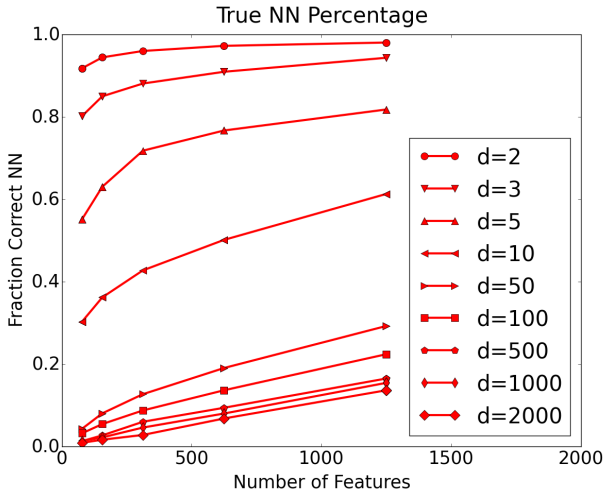

Figure 1.6: This is a demonstration of the curse of dimension. We generate 1000 training data with different dimensions from 2 to 2000. We then apply k-d tree NN search for 200 queries. As shown in the plot, the true NN accuracy decreases rapidly as we increase the dimension of the data.

Chapter 2

Randomized K-D Tree and PCA Tree

One of the important factors of tree structures is to choose split directions. K-d trees select split directions by finding features with maximum variance. Other tree structures choose split directions differently. Randomized k-d(r-k-d) trees select split directions randomly from the first few features with maximum variance, and Principal Component Analysis(PCA) trees select split directions through principal component analysis[16][17].

2.1 Randomized K-D Tree

2.1.1 Data Structure

R-k-d trees are binary search trees with different split directions at each tree node. While k-d tree splits at a direction with maximum feature variance, r-k-d trees select split directions randomly from the first few maximum variance features. We use the first five features with maximum variance in our experiment similar to [16].

R-k-d trees are built by first having all data points in the root. Then randomly select one feature vector from the first five features with maximum variance as the split

direction. The data points are therefore divided into two subsets evenly at the median along the selected feature axis. It recursively split the data points until the size of the tree nodes reaches the leaf limit m. See Algorithm 5.

Algorithm 5 Building r-k-d tree

1: **function** RKD_PARTITION(T)
2: $V = \{c_1, c_2, c_3, c_4, c_5\} \leftarrow$ set of first five coordinates with max variance
3: $T.split\_coord = c \leftarrow$ randomly select one coordinate from V
4: $s_c \leftarrow$ value at coordinate c of data s.
5: $T.split\_value = median\{s_c : s_c \in S\}$
6: $S_L, S_R =$ BREAK_TIE_PARTITION(S, T)
7: **return** T, S_L, S_R
8: **end function**

The quality of a single r-k-d tree may not be as good as a k-d tree, because r-k-d trees may not split a node along the feature direction with maximum variance. However, if we collect more than one r-k-d tree, the performance will get boosted because of the randomization of the trees. The tree randomness could solve the boundary problem in defeatist search as described in Figure 1.2. If query q lies near the splitting boundary in one of the r-k-d trees, it is not likely to lie near the boundary in another r-k-d tree. Therefore, as we build more trees, the probability of the query lying near the boundary become smaller. In other words, the more trees we build, the higher the probability of finding the true NN.

R-k-d tree search is similar to the k-d tree search, where a query point q traverses down the tree from the root and fall into one of the leaf nodes. We build r different r-k-d trees from the training set and perform the query search in all trees. Each tree returns one leaf node which we perform the linear search on. Therefore, we will have r leaf nodes, and the linear search for each of the leaves will return one nearest neighbor. We then select the nearest neighbor that has the closest distance to our query q from these r returned NN. See Algorithm 6.

We start by building two r-k-d trees for each training set and search both trees.

Algorithm 6 Searching r-k-d trees

1: **function** SINGLE_RKD_SEARCH(q, T)
2: **if** T is leaf **then**
3: **return** $\arg\min_{s \in T} D(q, s)$
4: **else**
5: $c = T.split\_coord \leftarrow$ Split direction
6: $q_c \leftarrow$ value at coordinate c of query q.
7: **if** $q_c \geq T.split\_value$ **then**
8: SINGLE_RKD_SEARCH($q, T.R$)
9: **else**
10: SINGLE_RKD_SEARCH($q, T.L$)
11: **end if**
12: **end if**
13: **end function**
14: **function** MULTIPLE_TREES_SEARCH($q, T_1, T_2, \ldots, T_r$)
15: **for** $i \leftarrow [1, r]$ **do**
16: n_i = SINGLE_RKD_SEARCH(q, T_i)
17: **end for**
18: $n^* = \arg\min_{n \in \{n_1, n_2, \ldots, n_r\}} D(q, n)$
19: **return** n^*
20: **end function**

Our experiment shows that the NN accuracy of two r-k-d trees is higher than the k-d tree in almost all of our datasets except BIG5. BIG5 has very low true NN accuracy using the k-d tree, and two r-k-d trees yield similar results.

The accuracy can be further improved by using more than two trees but results in increasing space complexity. There is also the increase of time complexity when searching trees. Each tree returns one leaf node of size m. Therefore, r different r-k-d trees will result in linear search of a total size of $r \cdot m$ data points. One way to limit the distance computation is to decrease the size of leaf node to $\frac{m}{r}$. This will increase the depth of the r-k-d tree, which again increase the space complexity. The benefit of having deeper trees is that we could do the linear search in a smaller subset of points. If the leaf size is $\frac{m}{r}$ and we have r trees, we will only need to do the linear search on a total of $r \cdot \frac{m}{r}$ data points.

2.1.2 Experiments

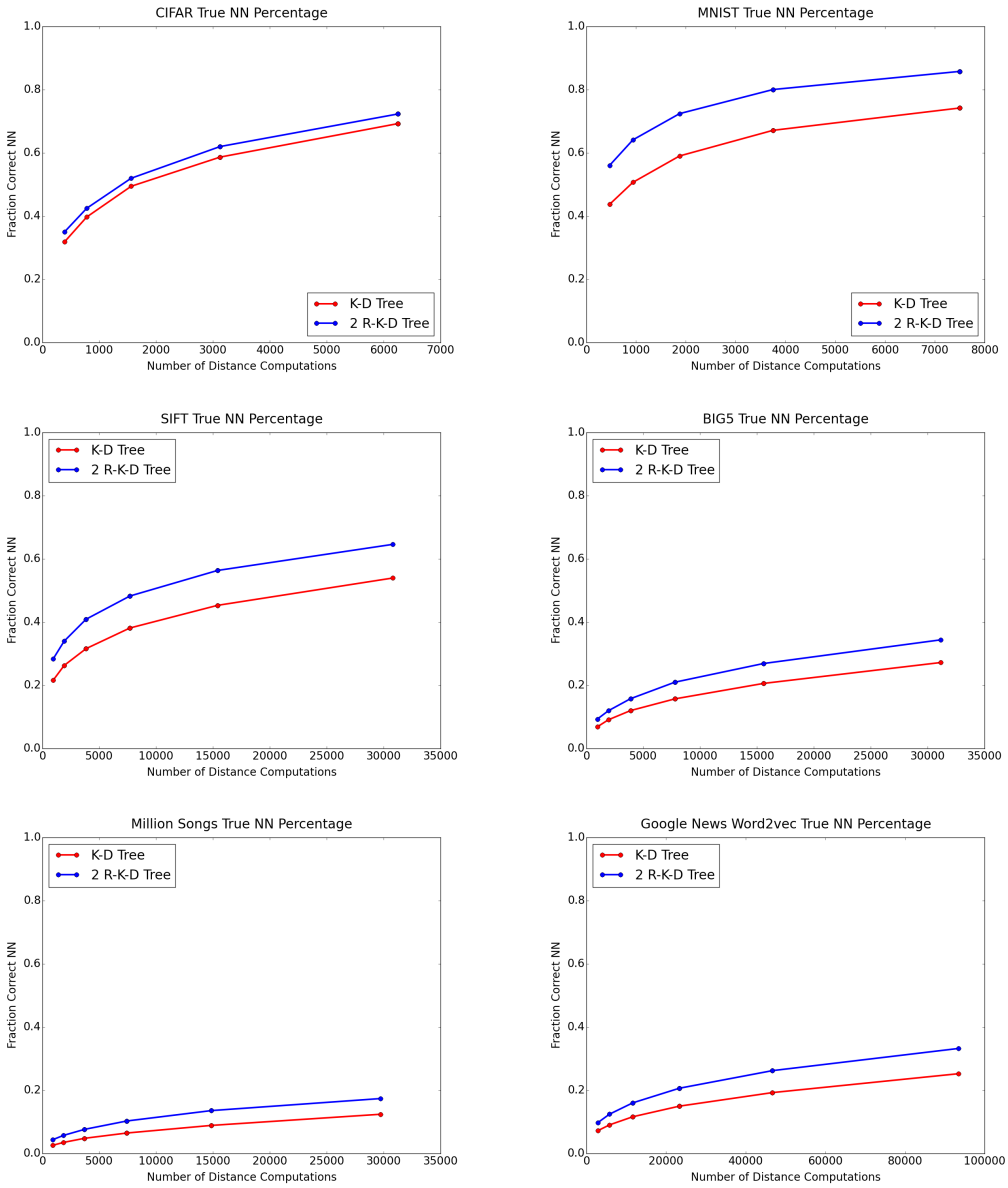

Figure 2.1: We apply k-d trees and two r-k-d trees on all six datasets. Two r-k-d trees give higher NN accuracy than the k-d tree in almost all datasets except BIG5. Both k-d trees and r-k-d trees yield very low true NN accuracy in BIG5. We can't tell whether one is better than the other in BIG5, but all the other datasets show that two r-k-d trees perform better than the k-d tree.

We apply r-k-d trees and k-d trees on all six datasets. Figure 2.1 shows that r-k-d trees have a higher nearest neighbor accuracy than the k-d tree in almost all six datasets

except BIG5. We make sure that we have the same number of distance comparison between k-d trees and r-k-d trees by having the leaf size of the r-k-d tree being half of the leaf size of k-d trees. Therefore, both tree structures perform linear search in subsets with the same size. Given almost the same time complexity, r-k-d trees boost the true NN accuracy of the k-d tree by around 10% in MNIST, SIFT, the Million Songs, and the Word2Vec dataset. Two r-k-d trees have around 3% higher accuracy than k-d trees in CIFAR. However, we have very similar performance between the k-d tree and two r-k-d trees in the BIG5 dataset.

2.2 Multiple R-K-D Trees

Our experiment in the previous sections shows that two r-k-d trees can boost up the NN accuracy by as much as 20% in some datasets while having similar accuracy with the k-d tree in the others. Our experiments show that the more trees we have, the higher the chance we find the true NN.

The build function of r-k-d trees will return a different tree every time it's called because we randomly select our split directions at each split level in the tree. In Algorithm 5, we have five directions to choose from at every split. Therefore, we have 5^s possible different trees with s splits in each tree. The probability of two trees being the same is as small as 0.2^s. Therefore, it is unlikely that we have two same trees returned from the build function.

The reason we want more trees in the algorithm is that we can decrease the failure rate. If the probability of not finding the true NN in one tree is p, we will have a failure rate of p^r with r trees. The more trees we have, the more likely we find the true NN when we run queries through the trees.

Figure 2.2 shows the experiment of the k-d tree with two, four, and eight r-k-d

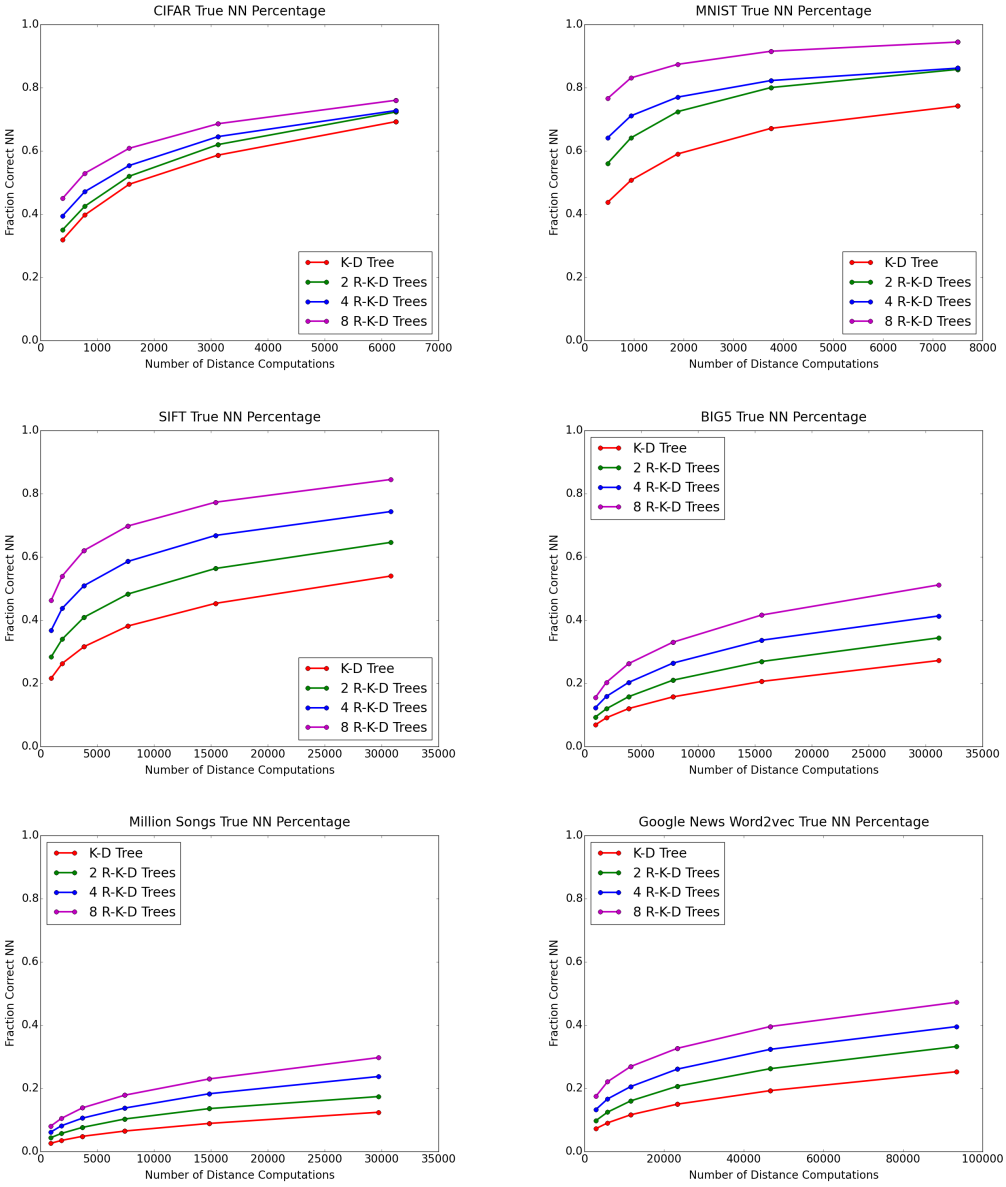

Figure 2.2: We apply two, four, and eight r-k-d trees on all six datasets. The graphs show a consistency ranking of the true NN accuracy. From highest accuracy we have eight r-k-d trees, then four r-k-d trees, followed by two r-k-d trees. K-D trees yield a true NN accuracy that is lower than any of the r-k-d trees.

trees. We can see from the graphs that the accuracy increases as the number of trees go up. We found that the more r-k-d trees we have, the better the performance of finding the true nearest neighbor. Also, two r-k-d trees can almost beat our baseline, k-d trees.

Having more trees will result in the increase of space complexity from building and storing r-k-d trees as we mentioned in 2.1.1. Therefore, we have this trade-off between true NN performance and space complexity. The higher the space complexity, the higher the true NN performance.

2.3 PCA Tree

2.3.1 Data Structure

PCA trees are another kind of binary search trees that partition data at each level using principal component analysis[18]. PCA is used to extract the dominant principal component with the largest variance from the high-dimensional data matrix[7]. PCA trees then use the extracted dominant principal component as the split direction at each tree level. The algorithm first projects all the data points of dataset S into the selected split direction. Then it splits the projected points at the median. See Algorithm 7. PCA trees don't have a space complexity as high as r-k-d trees. However, it suffers from the increase of time complexity due to the PCA, which we need to apply at every split of the tree.

Algorithm 7 Building PCA tree

1: Call BUILD_TREE(S, m) with partition function PCA_PARTITION(T)
2: **function** PCA_PARTITION(T)
3: Apply PCA on all the data points in S.
4: $T.v = v \leftarrow$ first principal component with largest eigenvalue
5: $T.split\_value = median\{s \cdot v : s \in S\}$
6: $S_L = \{s \in S : s \cdot v \leq center\}$
7: $S_R = \{s \in S : s \cdot v > center\}$
8: **return** T, S_L, S_R
9: **end function**

PCA trees always split the data evenly along the largest variance direction of the dataset instead of selecting a feature vector. The limitation of selecting a feature vector

as split direction is that the split can only capture the variance along one feature axis. Therefore the k-d tree will always have its splits being axis-aligned. PCA trees overcome this limitation by applying the PCA and capture the largest variance direction of the datasets. This direction can capture many feature information of the data.

The searching algorithm of PCA trees is similar to k-d trees. The only difference is that PCA trees do a projection before comparing the query to the splitting median. The search of PCA trees projects a query q along the split directory at each split. Then it compares the projected query point with the split median to determine which branch of the tree the query should go down to. See Algorithm 8.

Algorithm 8 Searching PCA tree

Input: query q, PCA tree node T
Output: nearest neighbor

```
1:  function PCA_SEARCH(q, T)
2:      if T is leaf then
3:          return arg min_{s∈T} D(q, s)
4:      else
5:          v = T.v
6:          q_c = q · v
7:          if q_c ≥ T.split_value then
8:              PCA_SEARCH(q, T.R)
9:          else
10:             PCA_SEARCH(q, T.L)
11:         end if
12:     end if
13: end function
```

2.3.2 Experiments

We apply PCA trees along with k-d trees on the datasets. We see from Figure 2.3 that PCA tree gives a higher value of true nearest neighbor accuracy comparing to k-d tree across all six datasets.

The accuracy improvement varies by datasets. From Figure 2.3, we see that PCA

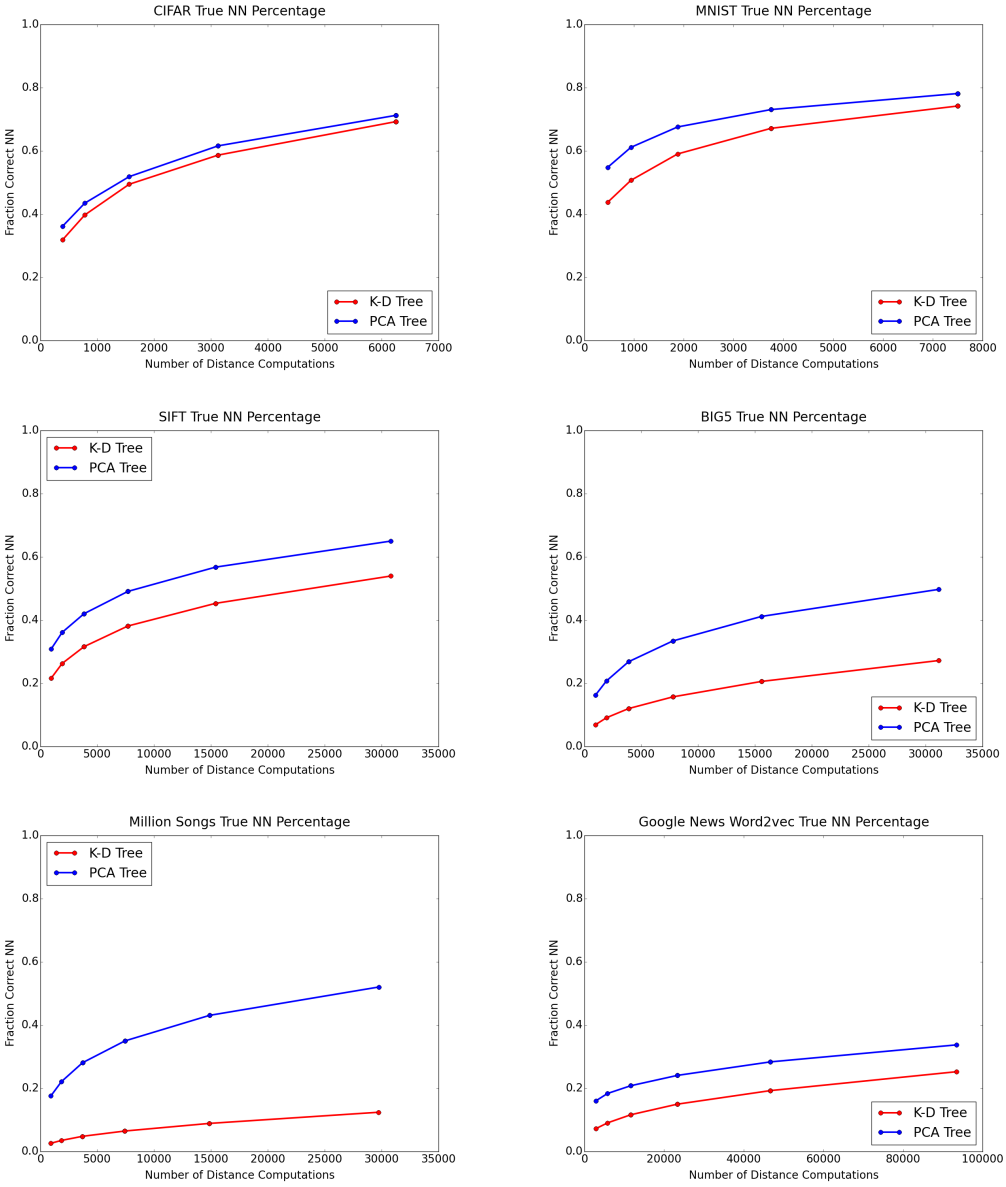

Figure 2.3: We apply PCA trees and k-d trees on six different datasets. We found that PCA trees outperform k-d trees in all six datasets.

trees improve the true NN accuracy by 20-30% in BIG5 and the Million Songs while it only increases by less than 5% in CIFAR. This is because PCA trees give a large boost in the accuracy of the dataset where one feature vector can't capture much information. If the largest variance feature axis, which is used in the k-d tree as the split direction, is very close to the dominant principal component captured by PCA, then PCA trees will not give

a large boost of NN accuracy comparing to k-d trees. However, if the largest variance feature axis is not representative of the dataset comparing to the dominant principal component, then PCA trees will have much higher quality comparing to k-d trees. In this case, PCA trees will have a much higher true NN accuracy.

2.4 R-K-D Tree vs. PCA Tree

Our experiments indicate that both r-k-d trees and PCA trees give higher nearest neighbor accuracy comparing to k-d trees. Therefore, we explore which of these two data structure performs better than the other. Figure 2.4 shows the k-d tree, r-k-d trees, and PCA trees all in one graph.

We see eight r-k-d trees perform better than PCA trees in CIFAR, MNIST, SIFT, and Word2Vec. However, PCA trees perform better than the eight r-k-d trees in the Million Songs dataset. Also, PCA trees have almost the same true NN accuracy as eight r-k-d trees in the BIG5. The graphs show that no one tree structure dominates the other in all six datasets.

The difference between PCA trees and r-k-d trees is the split direction. PCA trees project all the data points onto a direction before split while r-k-d trees select one feature vector as the split. Therefore, PCA trees capture the largest variance direction of the dataset instead of focusing on certain feature vector, while r-k-d trees focus on one feature vector. For datasets where many features have high variance, PCA can have a projection that best captures many of these features. Some datasets, however, only have a few features with high variance, r-k-d trees will focus on these features and will give high true NN accuracy.

MNIST, for example, have pixel values from the images as feature vectors. Handwritten digit images will only have a few pixels that change between different digits.

Also, many of the surrounding pixels remain unchanged throughout the dataset. These pixels will be the white space around the digit in each image. R-k-d trees will ignore all the white space because those are not pixels with big variance. Instead, r-k-d trees will focus on the pixels that change the most often between different digits. Therefore, r-k-d trees give higher true NN accuracy comparing to PCA trees in MNIST.

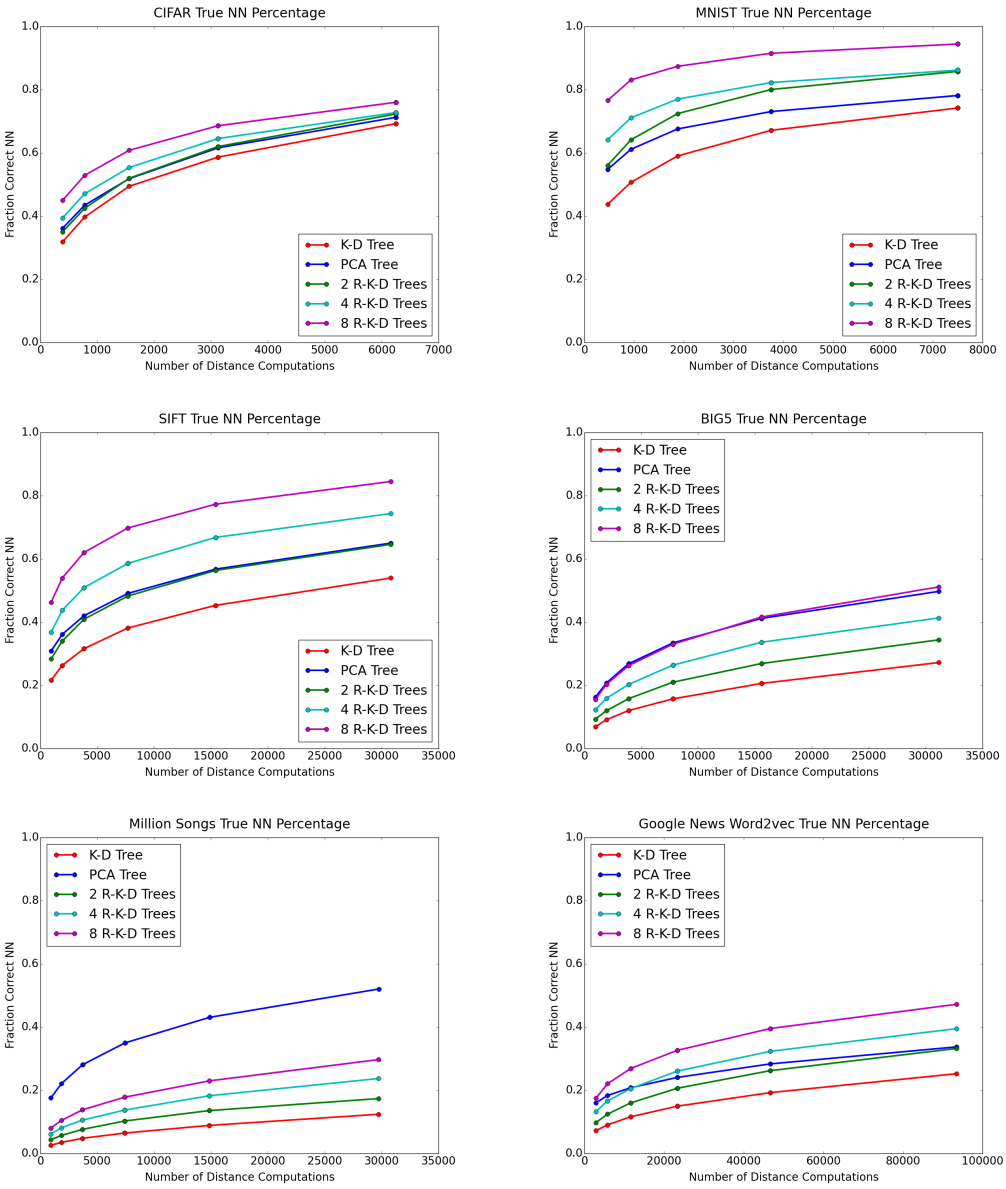

Figure 2.4: We apply PCA trees, r-k-d trees, and k-d trees on all six datasets. We see that PCA trees outperform eight r-k-d trees in the Million Songs dataset. Also, PCA trees have similar true NN accuracy in the BIG5 dataset. However, eight r-k-d trees have better performance than PCA trees in CIFAR, MNIST, SIFT, and Word2Vec.

Chapter 3

Random Projection Tree

In the previous chapter, we looked at r-k-d trees and PCA trees. We found that both these two structures give better accuracy than the k-d trees, however, one requires large space capacity and one requires large time complexity. We now look at another tree algorithm, random projection(RP) trees [4]. RP trees are similar to PCA trees that it project all the data down to a split direction at each tree level. RP trees are also similar to the r-k-d tree that it is randomized and therefore requires multiple trees to obtain high performance. We will first look at the data structure of RP trees and then we will make a comparison between PCA trees, r-k-d trees, and RP trees. We use the same number of trees as r-k-d trees when we do the comparison. Therefore we will look at two, four, and eight RP trees and compare with the data shown in Section 2.4.

3.1 Data Structure

RP trees are binary search trees similar to k-d trees that a tree splits a parent node into a left child node and a right child node recursively. The only difference between RP trees and k-d trees is again the split direction. RP trees select the split direction randomly. See Figure 3.1 All the data points are projected down to the split direction before they

are divided into two subsets. For a dataset of dimension k, we have a random vector v^k of size k at each split where v_i is selected uniformly at random from standard normal distribution $\mathcal{N}(0,1)$. We then do a dot product for all the data points to project the data to v^k. Finally, we split the projected data at the median. See Algorithm 9.

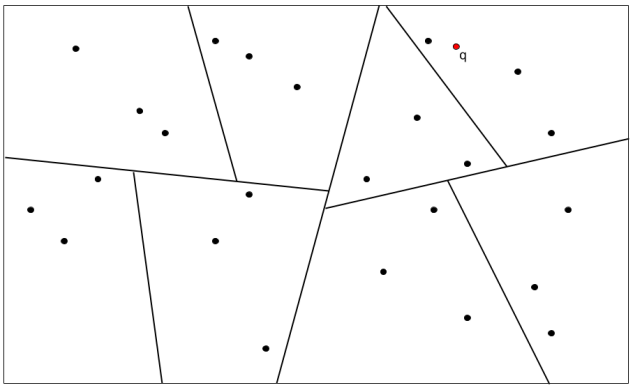

Figure 3.1: This is an RP tree partition with leaf size three. The red star is query q. We see that the true nearest neighbor of q is in the same leaf node as q. Therefore, it will return the true nearest neighbor when the query traverse down the tree.

Algorithm 9 Building RP trees

Call BUILD_TREE(S,m) with partition function RP_PARTITION(T)
function RP_PARTITION(T)
    $T.v \leftarrow$ split direction of node T
    $T.v = v \leftarrow$ random vector
    $T.split\_value = median\{s \cdot v : s \in S\}$
    $S_L = \{s \in S : s \cdot v \leq center\}$
    $S_R = \{s \in S : s \cdot v > center\}$
    return T, S_L, S_R
end function

The benefit of taking a random projection is similar to PCA trees that we can capture the values of more than one feature in the projection. Furthermore, choosing a random direction is time efficient comparing to the PCA. It also introduces randomness into the tree and allows to build multiple different trees from the dataset. The quality

of an RP tree is not as good as a PCA tree because PCA tree projects each split with a direction with dominating variance instead of a random direction. We could, however, have multiple RP trees to boost up the true NN accuracy.

The searching of RP trees is similar to r-k-d trees, where we build r different trees and search all the trees to find the nearest neighbor. See Algorithm 10.

Algorithm 10 Searching RP tree

```
1: Call MULTIPLE_TREES_SEARCH(q, T_1, T_2, ... , T_r)
       with function SINGLE_RP_SEARCH(q, T)
2: function SINGLE_RP_SEARCH(q, T)
3:     if T is leaf then
4:         return arg min_{s∈T} D(q, s)
5:     else
6:         c = T.split_coord ← Split direction
7:         q_c ← value at coordinate c of query q.
8:         if q_c ≥ T.split_value then
9:             SINGLE_RP_SEARCH(q, T.R)
10:        else
11:            SINGLE_RP_SEARCH(q, T.L)
12:        end if
13:    end if
14: end function
```

3.2 Experiments

We apply RP trees on all our datasets and compare it with our baseline, k-d trees. Here we build two RP trees and search both of the trees to find the nearest neighbor. See Figure 3.2.

We select r to be 2, 4, and 8, similar to r-k-d trees. We will compare the performance of RP trees and r-k-d trees with similar space complexity. In other words, we will compare RP trees and r-k-d trees given the same number of trees.

We found that two RP trees can't outperform the true NN accuracy of the k-d trees. In fact, k-d trees is a lot more accurate than two RP trees in CIFAR and MNIST

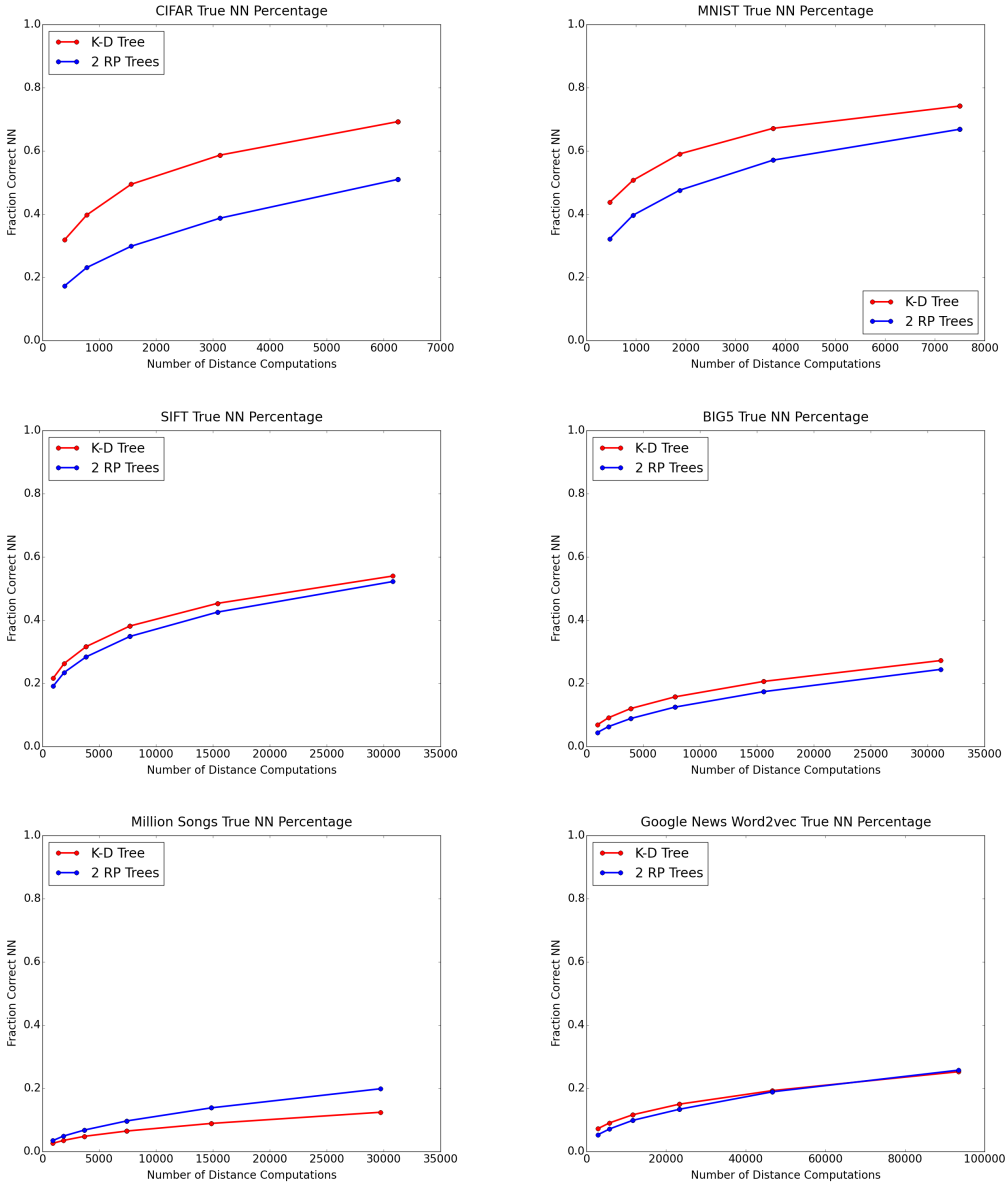

Figure 3.2: We apply two RP trees and k-d tree on four different datasets. We found that two RP trees can't outperform k-d trees in all datasets except the million songs. Two RP trees have true NN accuracy barely lower than k-d trees in SIFT, BIG5, and the Word2Vec.

datasets. K-d trees have true NN accuracy a little bit higher than two RP trees in SIFT, BIG5, and Word2Vec. However, we do have two RP trees outperform k-d trees in the Million Songs dataset. Therefore, we try to increase the number of RP trees we use. The

more RP trees we use, the better chance we have at finding the true nearest neighbor. Figure 3.3 shows the performance of two, four, and eight RP trees along with our baseline k-d trees.

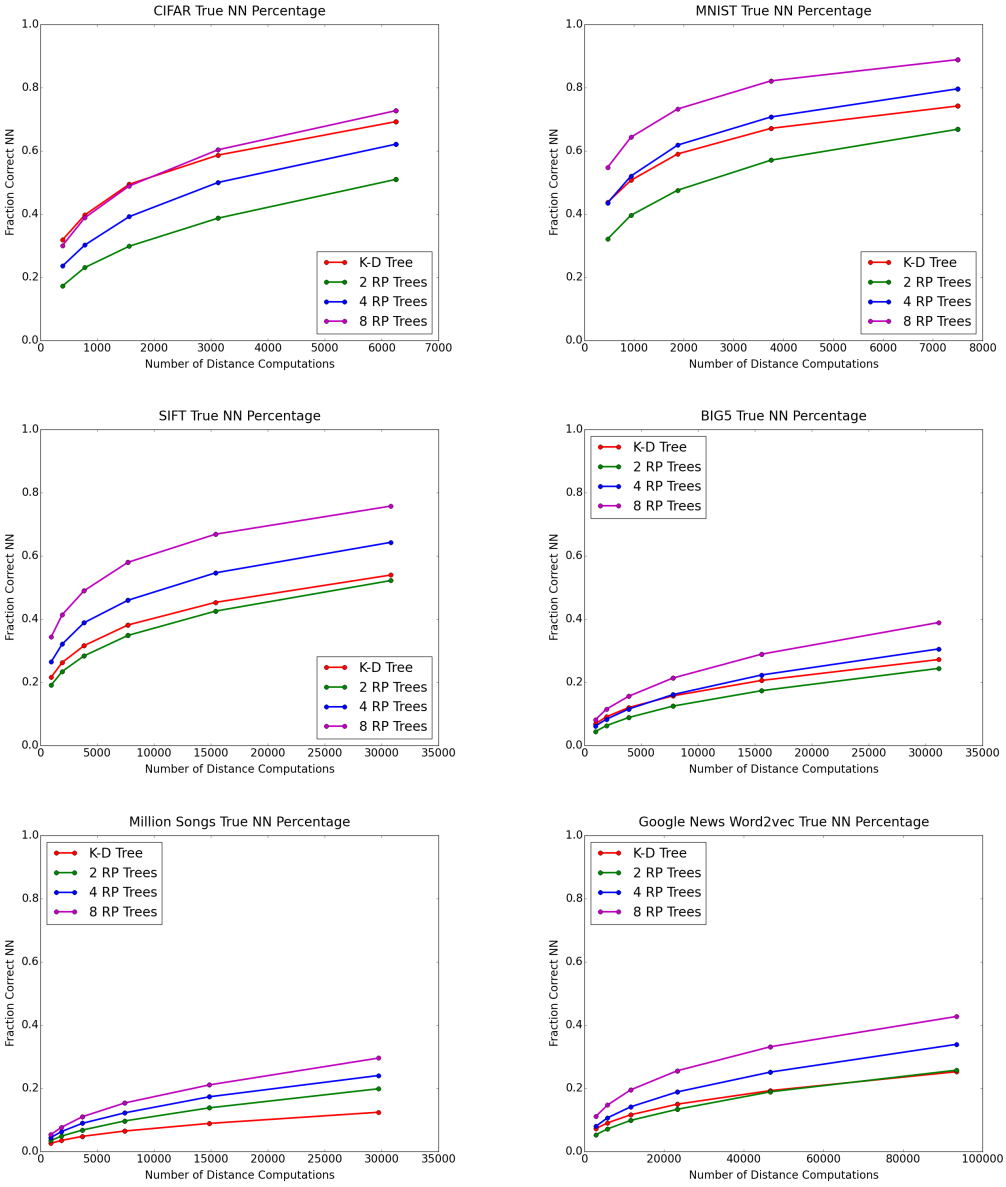

Figure 3.3: We apply two, four, and eight RP trees along with k-d tree on the datasets. We found that eight RP trees can outperform k-d tree in almost all the datasets.

From Figure 3.3 we see that eight RP trees outperform k-d trees in all six datasets. In fact, four RP trees already have higher true NN accuracy than k-d trees in all datasets

except CIFAR. RP trees don't seem to work too well in CIFAR. K-D trees in CIFAR give better NN accuracy than four RP trees, and eight RP trees can barely beat it. From the graph, we see that eight RP trees only give a higher accuracy in CIFAR when we increase the number of distance computation to 2000. In other words, RP trees work better when we allow a larger number of distance computations. This holds true in other datasets as well. RP trees seem to work better and better as the number of distance computation increase.

3.3 RP Tree with R-K-D Tree and PCA Tree

RP trees lower the dimension of the data at each split, which is similar to the idea of PCA tree. It is less time consuming compared to PCA tree because the split directory is generated randomly. And because of the randomness in the RP trees, we can build multiple RP trees and search all the trees, which is very similar to r-k-d trees. We compared r-k-d trees with PCA trees in section 2.4, we want to see how RP trees' performance fit into the graph.

From Figure 3.3 and Figure 2.2, we see that the more randomized trees we use, the higher the true NN accuracy. Therefore, we only show eight RP trees and eight r-k-d trees in Figure 3.4.

We see that eight r-k-d trees outperform all other tree structures on some datasets while R-K-D trees have leading performance on the others. However, RP trees have second leading performance in almost all the datasets. RP trees perform better than PCA trees in MNIST, SIFT, and Word2Vec. Also, RP trees have similar performance with r-k-d trees in the Million Songs. Therefore, we aim to improve the performance of RP trees with a new tree structure two-vantage-point trees.

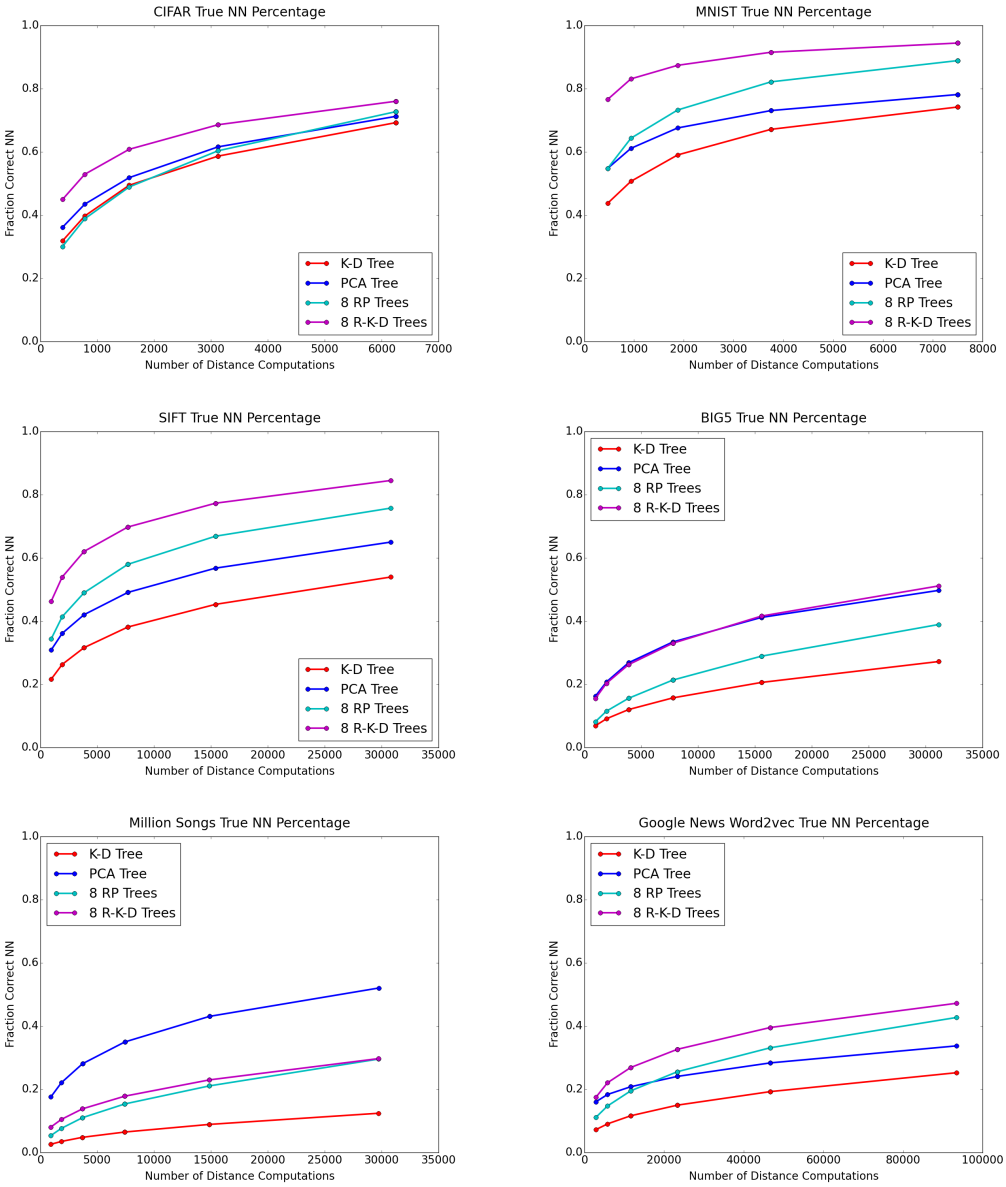

Figure 3.4: We apply eight RP trees, eight r-k-d trees, k-d trees, and PCA trees on the datasets. We found that eight r-k-d trees give the best NN accuracy on CIFAR, MNIST, and SIFT, while PCA tree gives the best NN accuracy on the million songs and Word2Vec. PCA trees and eight r-k-d trees share very similar true NN accuracy on BIG5. RP trees always rank the second best performance in the graphs.

Chapter 4

Two-Vantage-Point Tree

After we examine PCA trees, RP trees, and r-k-d trees, we conclude that PCA and randomization are two techniques that can boost up the accuracy of the NN search. See Figure 3.4. PCA focuses on the structure of the dataset but requires very high time complexity. Randomization is very useful that it allows multiple trees to boost up the true NN accuracy. We here introduce a new tree structure that can capture the structure of the dataset and also use randomization that allows multiple trees, Two-Vantage-Point(V^2) tree or Twin Vantage trees.

4.1 Data Structure

V^2 trees are binary trees similar to RP trees that a parent node is split into left child node and right child node recursively by projecting data onto a random split direction. However, we select the random direction by first selecting two points from the dataset randomly and then take the vector between the chosen points as our split direction. For a dataset S of dimension k, we have a vector $\mathbf{v} = [v_1 v_2 ... v_i ... v_k]$ at each split. V^2 trees randomly select two data points $\mathbf{s_i}, \mathbf{s_j} \in S$ and have $\mathbf{v} = \mathbf{s_i} - \mathbf{s_j}$. We then project the data down to $\mathbf{v}$ before we split the tree node at each tree level. See Algorithm 11.

Algorithm 11 Building V^2 tree

Call BUILD_TREE(S, m) with partition function V2_PARTITION(T)
function V2_PARTITION(T)
    Randomly select $s_i, s_j \in S$
    $T.v \leftarrow$ split direction of node T
    $T.v = s_i - s_j$
    $T.split\_value = median\{s \cdot v : s \in S\}$
    $S_L = \{s \in S : s \cdot v \leq center\}$
    $S_R = \{s \in S : s \cdot v > center\}$
    return T, S_L, S_R
end function

We select the split direction by taking the vector between two random data points from the dataset to make sure that the projection direction lies in the same intrinsic space as our dataset. We capture the space information of the dataset and also have randomness in the trees. Therefore, we can again use multiple trees to increase the NN search accuracy.

Searching of the trees is the same as the searching of RP trees. We stored the projection direction in $T.v$ and will then be used the same way as in RP tree search. The query point q will be projected down to the split direction at every tree level when traversing down the tree to search for NN. See Algorithm 10.

4.2 Experiments

Similar to RP trees, we use r trees to evaluate the performance. We again select r to be two, four, and eight. We want to see whether V^2 trees can outperform RP trees or r-k-d trees. See Figure 4.1.

From Figure 4.1 we see that V^2 trees always give high accuracy than RP trees and r-k-d trees. Furthermore, with only two V^2 trees, we get a higher true NN accuracy than eight RP trees and eight r-k-d trees in the Million Songs dataset. Two V^2 trees boost

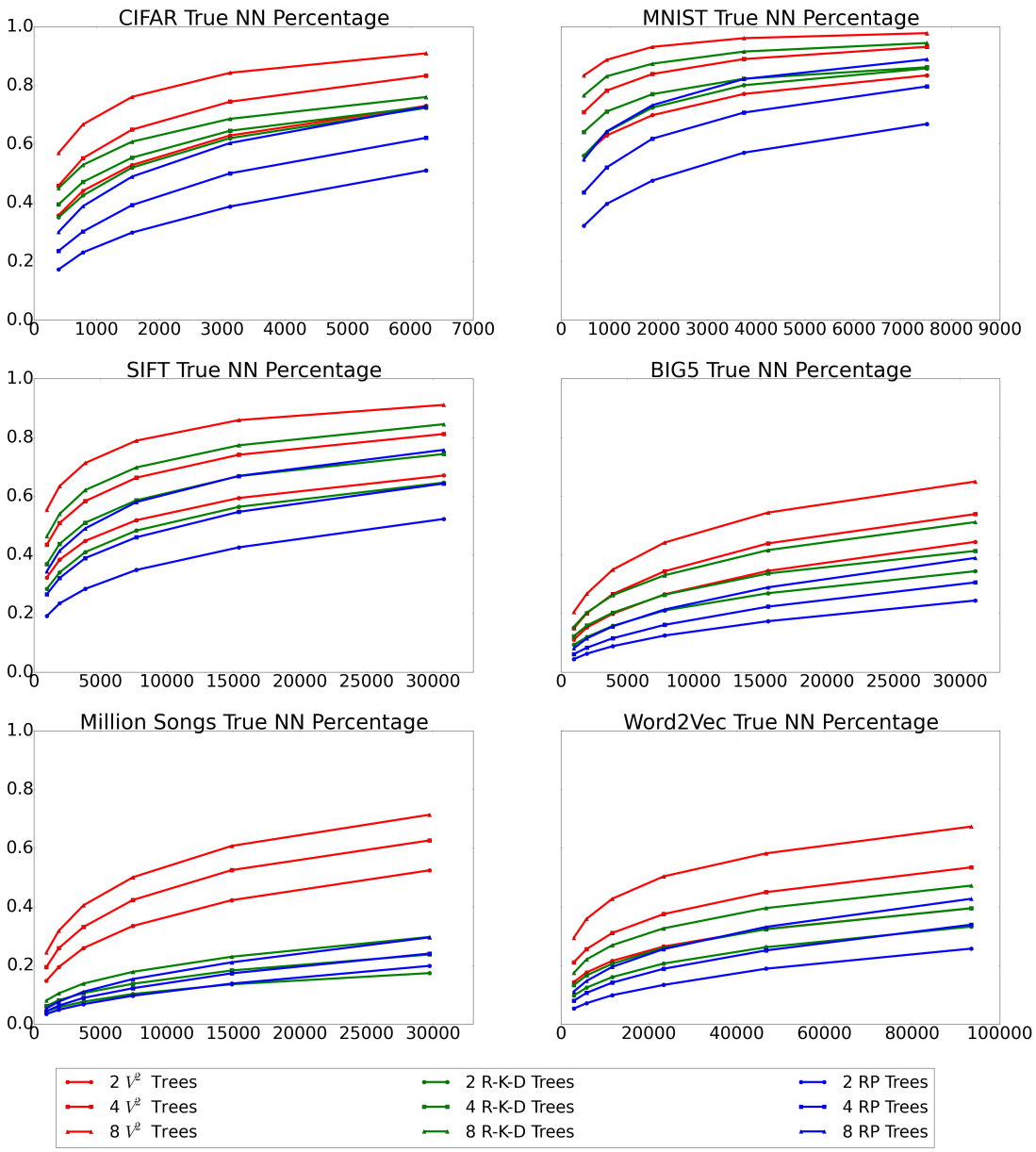

Figure 4.1: We apply two, four, and eight V^2 trees along with RP trees and r-k-d trees on the six datasets. We found that V^2 trees always have higher true NN accuracy than RP trees and r-k-d trees for all different values of r. V^2 trees perform extremely well in the Million Songs dataset that two V^2 trees can outperform eight RP trees and eight r-k-d trees.

up the true NN accuracy by almost 20% in the Million Songs. This is because the data points in the Million Songs all lie in a simple intrinsic dimension which is easy to capture by taking a vector between two random data points. RP trees won't be able to obtain this

information because the projection is completely random. R-k-d trees only look at one feature vector at a time, so won't be able to capture more than one feature at each split. Therefore, V^2 trees perform a lot better than RP trees and r-k-d trees in the Million Songs dataset.

Given that V^2 tree outperform RP trees and r-k-d trees, we want to see the performance of the V^2 trees comparing to all other tree structures including the leading PCA trees. See Figure 4.2.

We use eight trees in multiple trees structure and compare the performance of V^2 trees with RP trees, r-k-d trees, PCA trees and k-d trees. We found that eight V^2 trees outperform all the other tree structures in Figure 4.2. We see that eight V^2 trees boost the true NN accuracy of the second best tree structure by almost 20% in CIFAR, the Million Songs, and Word2Vec datasets.

Chapter 4, in part, has been submitted for publication of the material as it may appear in Nearest Neighbor Search Using Twin-Vantages Forests, Zhai, Zhen; Dasgupta, Sanjoy, 2017. The thesis author was the primary investigator and author of this material.

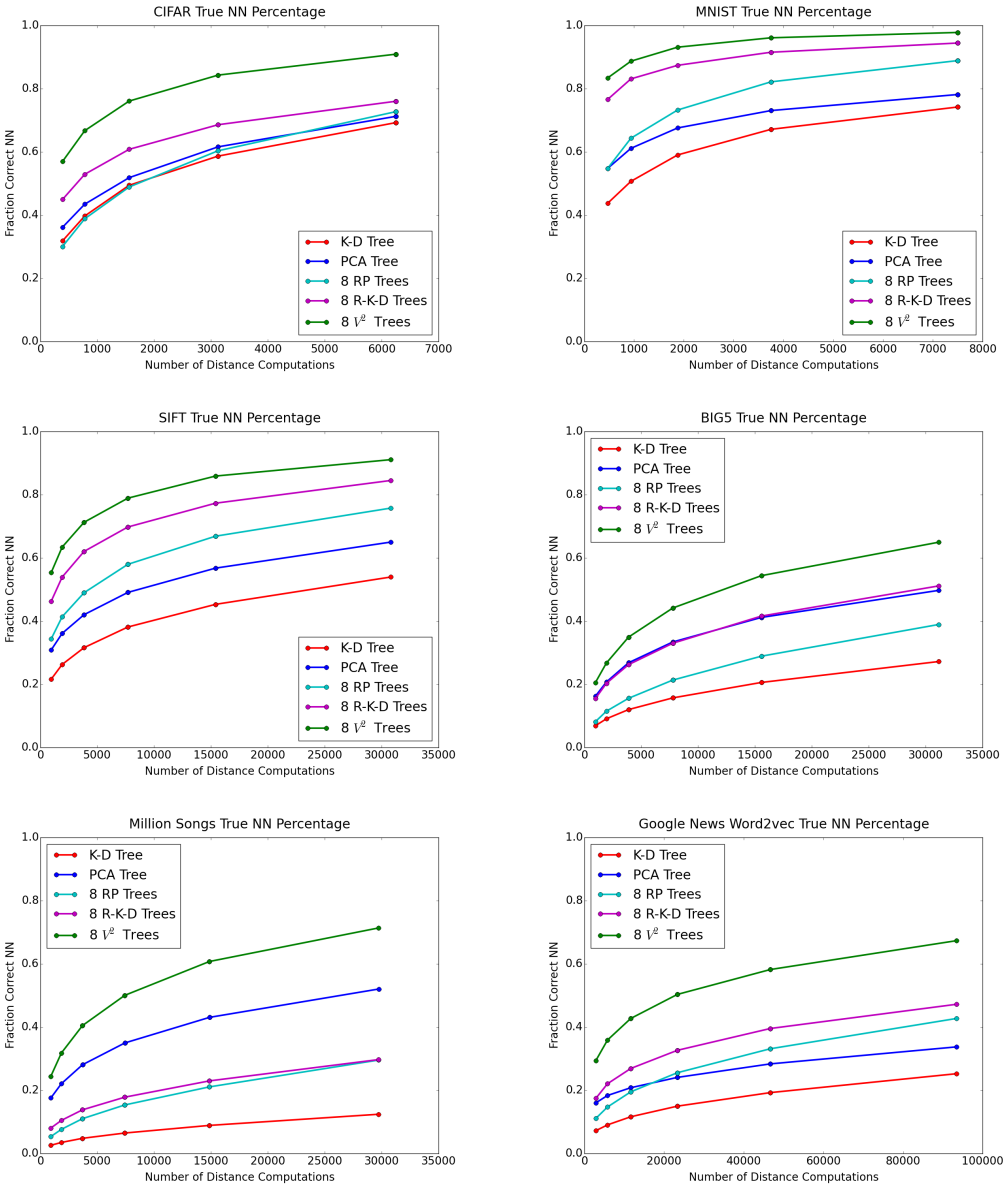

Figure 4.2: We apply eight V^2 trees, eight RP trees, eight r-k-d trees, k-d trees, and PCA trees on all the six datasets. We found that eight V^2 trees always have a leading true NN accuracy comparing to all other trees.

Chapter 5

Spillover in Search Trees

Solving the boundary problem from Figure 1.2 is one of the important ways to improve the performance of trees. Spillover is a technique that aims to solve the boundary problem of trees.[11][13]. Spillover allows part of the data points near the splitting boundary to be included in both of the left and right child nodes, see Figure 5.1. We can apply spillover on all the tree structures. We apply spillover on k-d trees to get the k-d spill trees; we also look at PCA spill trees. We will compare the performance of spill trees and the original trees to see whether spillover increases the true nearest neighbor accuracy.

5.1 Data Structure

The number of spilled data points are determined by the spill factor $\alpha \in \{0, 0.5\}$, which stands for the percentage of the data points that we included in the spill at each tree level. At each level, we first split at the median to have half of the data points in the left tree node and the other half in the right node. We then want to make sure that the spillover area is very close to the boundary. Therefore we capture $\alpha * |S_p|$ many points that are located to the left and right of the median, where S_p is the parent node and $|S_p|$

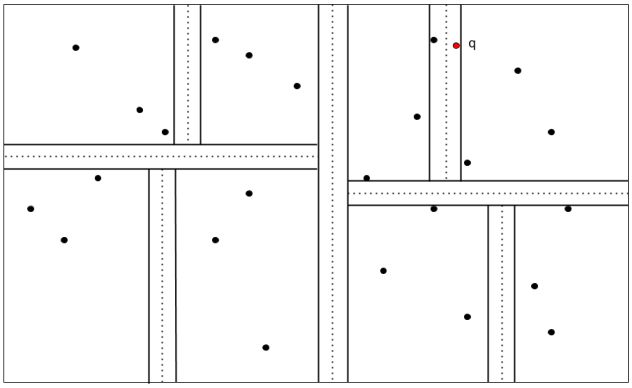

Figure 5.1: This is a k-d spill tree partitioned of leaf size three. The red star is query q. We see that the true nearest neighbor of q is in the spill area. Therefore, the spilled k-d tree will return the true nearest neighbor when the query traverses down the tree. However, it won't return the true NN if we don't spill, because the true NN is in a different node with the query.

is the size of the parent node. Then, we will have a spillover of size $2 * \alpha * |S_p|$. The spillover area will be included in both the left and right tree node. See Algorithm 12 and Algorithm 13.

Algorithm 12 Building k-d spill tree

Call BUILD_TREE(S, m) with partition function KD_SPILL_PARTITION(T)
function KD_SPILL_PARTITION(T)
    $T.split\_coord = c \leftarrow$ coordinate with max variance
    $s_c \leftarrow$ value at coordinate c of data s.
    T.bound_left $= (0.5 - \alpha)$ percentile $\{s_c : s_c \in S\}$
    T.bound_right $= (0.5 + \alpha)$ percentile $\{s_c : s_c \in S\}$
    $T.split\_value = median\{s_c \cdot v : s_c \in S\}$
    $S_L = \{s \in S : s_c \leq$bound_right$\}$
    $S_R = \{s \in S : s_c >$bound_left$\}$
    return T, S_L, S_R
end function

If we allow spillover with spill factor α, we then have the size of each child nodes to be $(0.5 + \alpha) * |S_p|$. This will increase the space complexity because we are storing duplicate data in each level of the tree. Given the size of the dataset is $|S|$; the leaf node

Algorithm 13 Building PCA spill tree

1: Call BUILD_TREE(S, m) with partition function PCA_SPILL_PARTITION(T)
2: **function** PCA_SPILL_PARTITION(T)
3: $T.v \leftarrow$ split direction of node T
4: Apply PCA on all the data points in S.
5: $T.v = v$ first principal component with largest eigenvalue
6: T.bound_left $= (0.5 - \alpha)$ percentile $\{s \cdot v : s \in S\}$
7: T.bound_right $= (0.5 + \alpha)$ percentile $\{s \cdot v : s \in S\}$
8: $T.split\_value = median\{s \cdot v : s_c \in S\}$
9: $S_L = \{s \in S : s \cdot v \leq$bound_right$\}$
10: $S_R = \{s \in S : s \cdot v >$bound_left$\}$
11: **return** T, S_L, S_R
12: **end function**

of trees is of size m. Let $\beta = 1/2 + \alpha$. We will have spillover of size $m(\frac{|S|}{m})^{lg(1+2\alpha)}$[5]. The spillover size is large, comparing to linear space complexity of other trees.

Searching the spill trees is the same as searching the original tree. Notice that Algorithm 12 line 7 and Algorithm 13 line 8 both store the median of the split in $T.split\_value$. Although we are not using $T.split\_value$ when we build trees, we will use it when we perform searching. Algorithm 4 and Algorithm 8 both search left sub-tree $T.L$ if the query at coordinate c has a value that's smaller than the median $T.split\_value$. Otherwise, the right sub-tree $T.R$ will be searched. In spill trees, the query will follow the same path as in k-d trees and PCA trees. However, there are more data points in the leaves of spill trees than the original trees.

The time complexity of searching will not increase a lot. If the minimum leaf size m is greater than $2^{-i}|S|$, which is the leaf size of the tree with depth i. We will have $(1+\alpha)2^{-i}|S|$ many points for the linear search in spill trees.

5.2 Experiments

We apply different spill factors on k-d trees and also PCA trees to see whether spilling can boost up the nearest neighbor accuracy. See Figure 5.3 and Figure 5.2. Our experiments show that spillover will improve the NN accuracy of both k-d trees and PCA trees. Figure 5.3 shows that the bigger the spill factor α in PCA spill trees, the better the performance. However, when we look at k-d spill trees in Figure 5.2 we see that bigger spill factor not necessary improve the true NN accuracy. K-d spill can improve the performance of CIFAR, SIFT, BIG5, and Word2Vec similar to PCA spill trees. However, different spill factors don't seem to make many differences in the million songs dataset and MNIST.

We also show the space blowup on both Figure 5.3 and 5.2. The space blowup is the factor of space of the original tree. A space blowup of 3 in k-d spill trees means it takes three times the space of a k-d tree. A space blowup of 10 in PCA spill trees represents ten times the space of a PCA tree. The space blowup factor increases when the spill factor increases. Also, the deeper the tree we reach, the bigger the space blowup factor. In the figures, we see that the smaller the number of distance computation the bigger the space blowup.

5.3 Virtual Spill Tree

We found that spilling is very useful on tree structures. However, spill trees increase space complexity very fast. We want to limit the space complexity of the spill trees and keep the good performance.

One idea is to directly use the k-d trees and the PCA trees, and search the trees using the methodology of spilling, the virtual spill trees [5]. Virtual spill trees perform the spilling when searching the trees, that if a query q falls into the spill area, both the

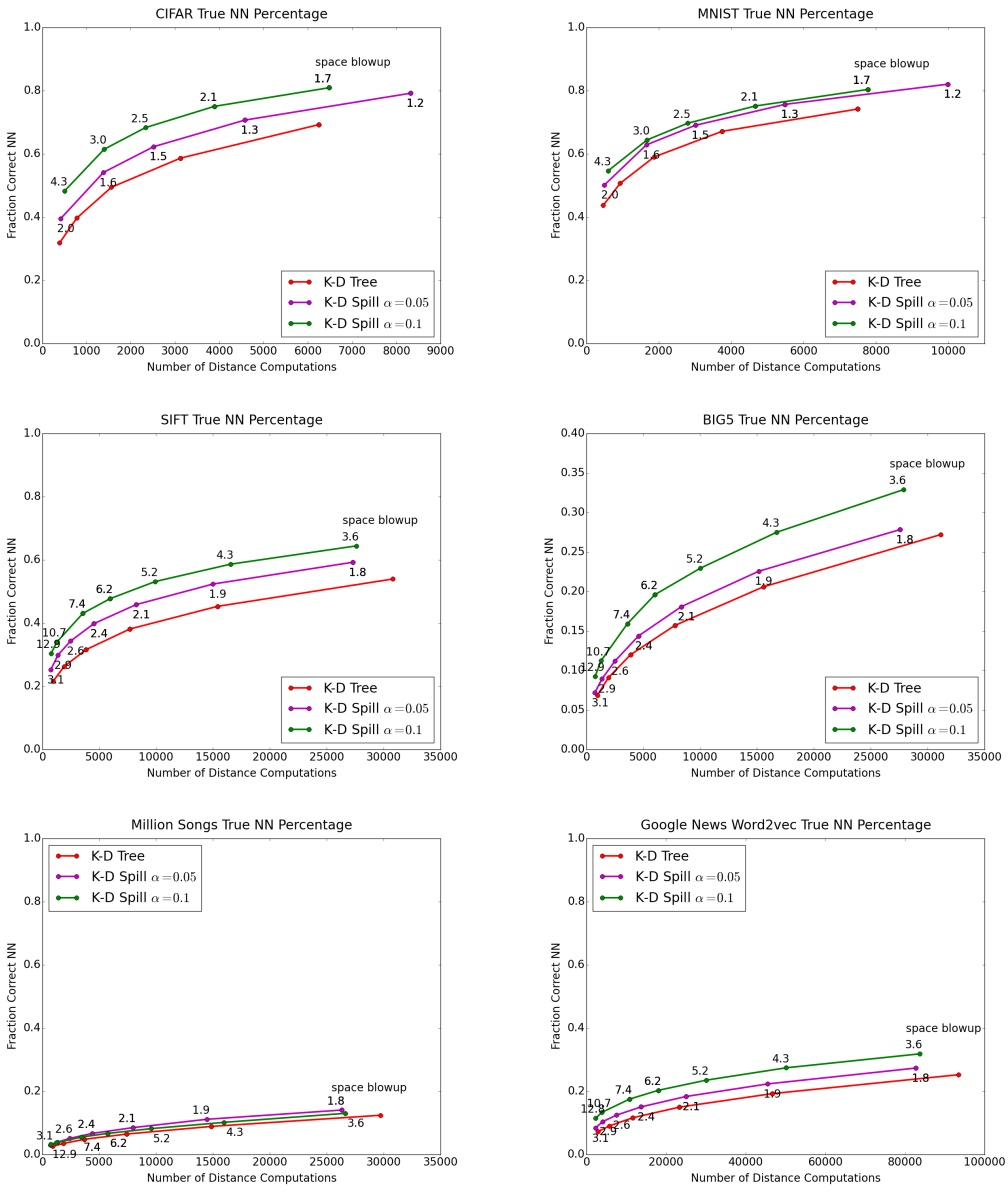

Figure 5.2: We used $\alpha = 0.1$ and 0.05 on k-d trees. We see that spilling increases the true NN accuracy of k-d trees. Spill factors don't make much difference in MNIST and the Million Songs. However, bigger spill factors yield better performance in CIFAR, SIFT, BIG5, and Word2Vec. We also label each data point with the space blowup factor. The bigger space blowup factor, the larger the space the tree takes.

left and right sub-trees will be searched. See Algorithm 14.

We do need to store T.bound_left and T.bound_right when building the trees. We will not use the left and right bound when we build the trees, but we need to use it when

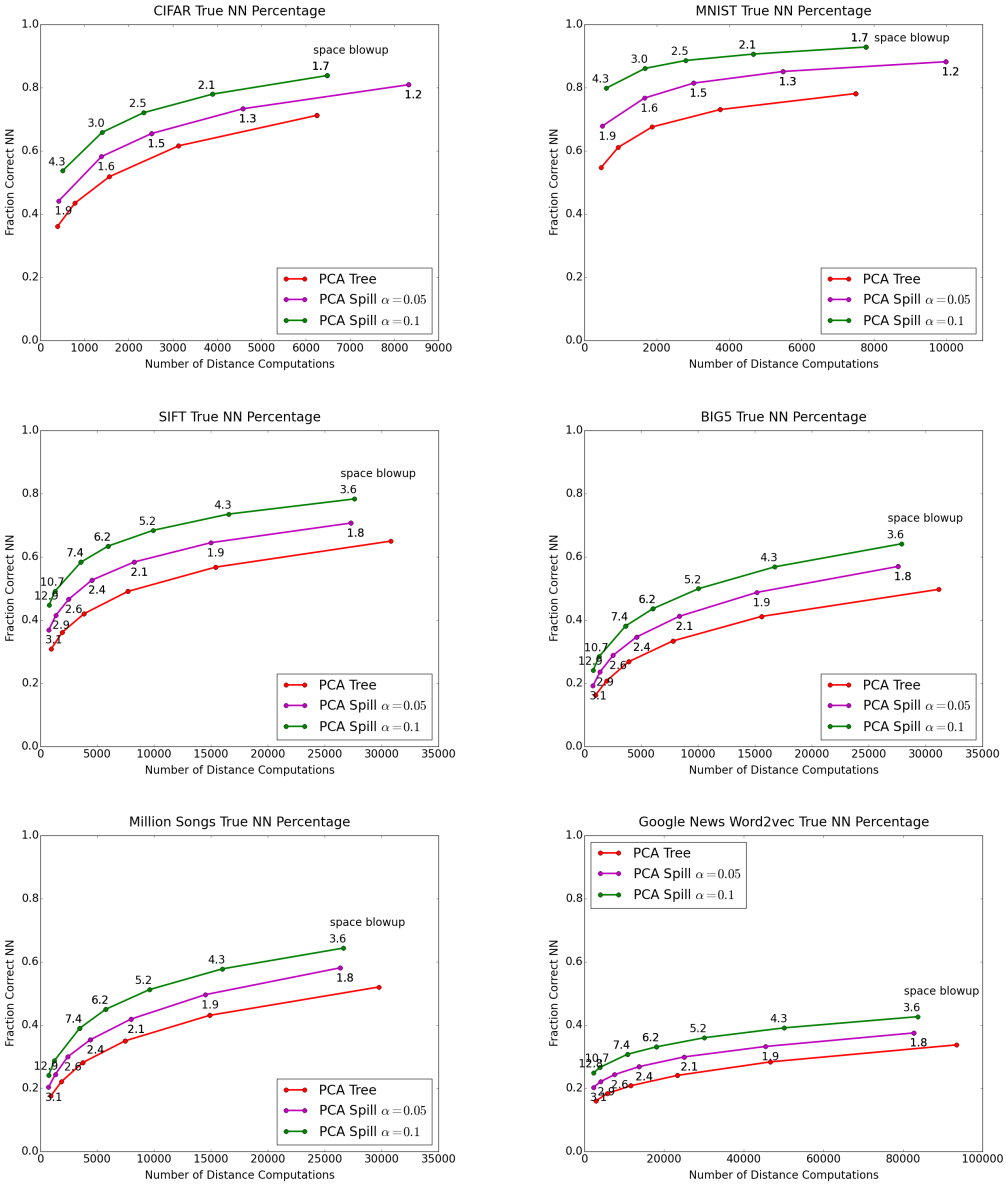

Figure 5.3: We applied $\alpha = 0.1$ and 0.05 on PCA trees. We see that spill factor of 0.05 give higher NN accuracy than the PCA trees, and 0.1 spill factor gives an even higher accuracy than spill factor of 0.05. We again label each data point with the space blowup factor. Space blowup factor represents the multiples of the space of the original trees. Space blowup 12 represents the PCA spill tree takes 12 times the space of the PCA tree.

searching. Therefore, we can build virtual spill trees similar to the original tree, but we need to calculate and store the left and right bound of the spill area at each tree level.

Virtual spill trees will not increase the space complexity. However, we could

Algorithm 14 Searching virtual spill tree

```
1:  function VIRTUAL_SPILLTREE_SEARCH(q, T)
2:      if T is leaf then
3:          return arg min_{s∈T} D(q,s)
4:      else
5:          c = T.split_coord ← Split direction q_c ← value at coordinate c of query q.
6:          if q_c < T.bound_left then
7:              return VIRTUAL_SPILLTREE_SEARCH(q, T.L)
8:          else if q_c > T.bound_right then
9:              return VIRTUAL_SPILLTREE_SEARCH(q, T.R)
10:         else
11:             NN_left = VIRTUAL_SPILLTREE_SEARCH(q, T.L)
12:             NN_right = VIRTUAL_SPILLTREE_SEARCH(q, T.R)
13:             n* = arg min n ∈ {NN_left, NN_right} D(q,n)
14:             return n*
15:         end if
16:         return T
17:     end if
18: end function
```

potential search many tree nodes. If query q falls into the spill zone at every splits, we could end up searching the entire tree. This will increase the time complexity tremendously. Therefore, virtual spill tree is more like a comprehensive search than a defeatist search.

5.4 Randomized Trees vs. Spill Trees

Both randomized trees and k-d spill trees take up more space than k-d trees. Our goal is to see which of these two types of trees give better performance given similar space blowup. We, therefore, compare the performance of k-d spill trees and V^2 trees, both of which have low time complexity but high space complexity. See Figure 5.4.

Looking at spills tree in Figure 5.4, we see that 0.05 spill trees have space blowup between 1 and 3, and 0.1 spill trees have space blowup rate higher than 3, some reach 12. If we look at the V^2 trees, the number of trees is the space blowup rate. Therefore,

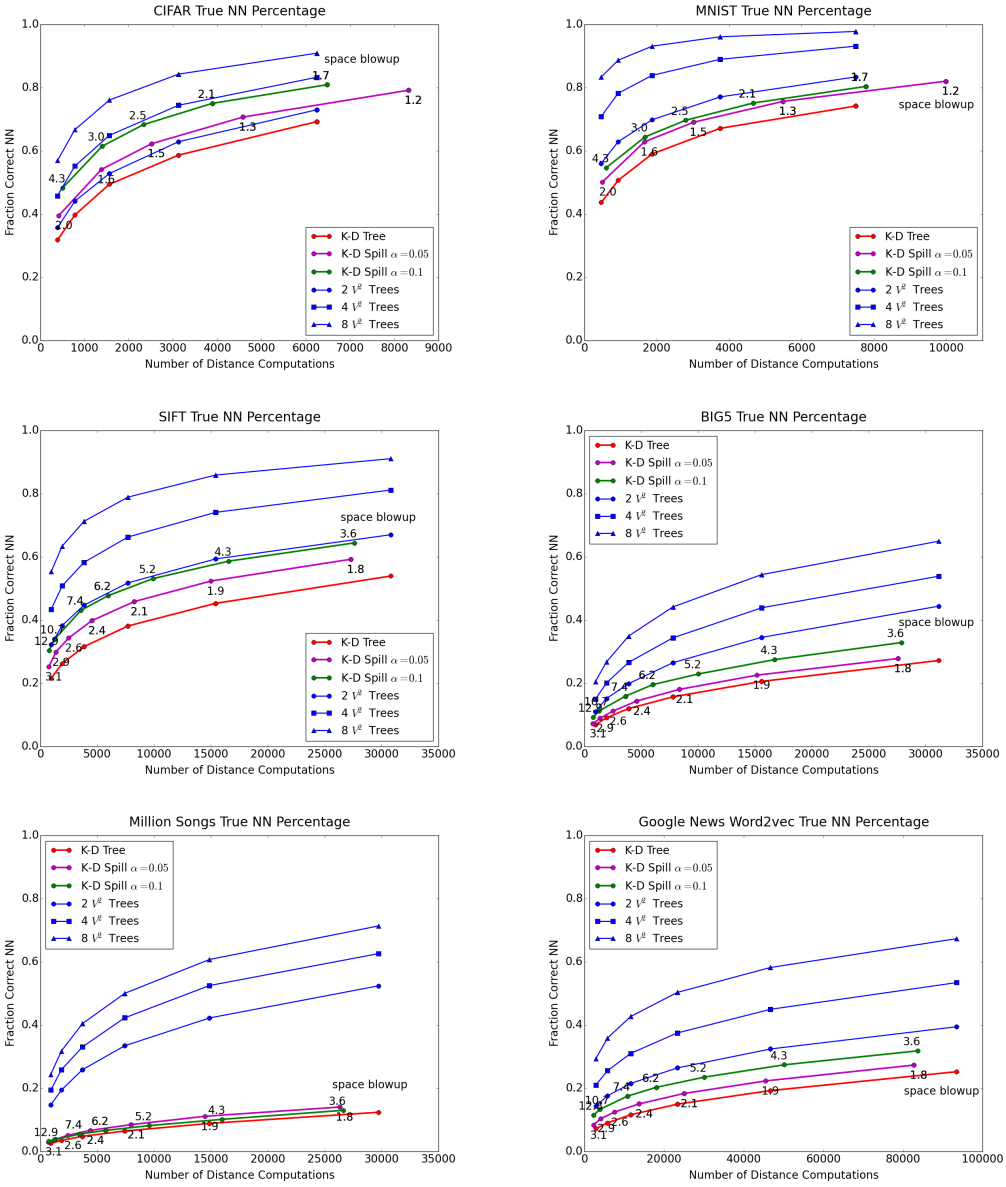

Figure 5.4: We compare k-d spill trees with V^2 trees. We see that two V^2 trees can beat the performance of spill trees in MNIST, SIFT, BIG5, Word2Vec, and Songs. Therefore, given the space blowup rate of 2, it is better we do V^2 trees comparing to do spillover. However, if we look at CIFAR, k-d spill trees with both 0.05 and 0.1 spill factors outperform two V^2 trees. However, when more space is given, V^2 trees outperform the k-d spill trees.

we want to look at 0.05-spill trees against 2 V^2 trees, and 0.1-spill trees against four and eight V^2 trees.

Figure 5.4 shows us that two V^2 trees outperform 0.05-spill trees in most of the datasets. V^2 trees work very well in the Million Songs that two V^2 trees boost up the true NN performance of 0.05-spill trees by almost 20%. If we look at Big5, SIFT, Word2Vec, and MNIST, two V^2 trees outperform 0.05-spill trees by around 5%. However, both 0.05-spill trees beat two V^2 trees in CIFAR. It is not clear which of the 0.05-spill trees and the two V^2 trees are better.

If we look at 0.1-spill trees, we see that four V^2 trees outperform 0.1-spill tree barely in CIFAR. Eight V^2 trees boost up the accuracy of 0.1-spill trees by almost 10%. If we look at eight V^2 trees on all other datasets, we see it outperforms k-d spill trees by almost 20% in MNIST, SIFT, BIG5, and Word2Vec. Looking at the Million Songs, eight V^2 trees boost up the true NN accuracy by almost 50%. It is shown that eight V^2 trees work better than 0.1-spill trees.

We do the same experiment on PCA spill trees. See Figure 5.5. PCA spill trees with spill factor 0.1 have similar true NN accuracy with four V^2 trees in CIFAR, MNIST, SIFT, Million Songs, and Word2Vec. Eight V^2 trees, however, outperform PCA spills in these five datasets. Looking at BIG5, we see that eight V^2 trees have true NN accuracy slightly lower than PCA spill with 0.1 spill factor. PCA spill trees with 0.1 spill factor have space blowup range from 3 to 12. In this case, we think that having more than eight V^2 trees can give higher accuracy than PCA spill trees. Notice that PCA trees and PCA spill trees require very high time complexity due to PCA on every tree level.

We concluded in the previous chapter that randomized trees give good true NN accuracy when we use more trees. Therefore, when more space blowup is allowed, randomized trees work better than spillover. But when the space blowup rate is limited to a small number, spillover may be a better choice. Therefore, when the space blowup rate is low, spillover with small spill factor is a better option comparing to randomized trees.

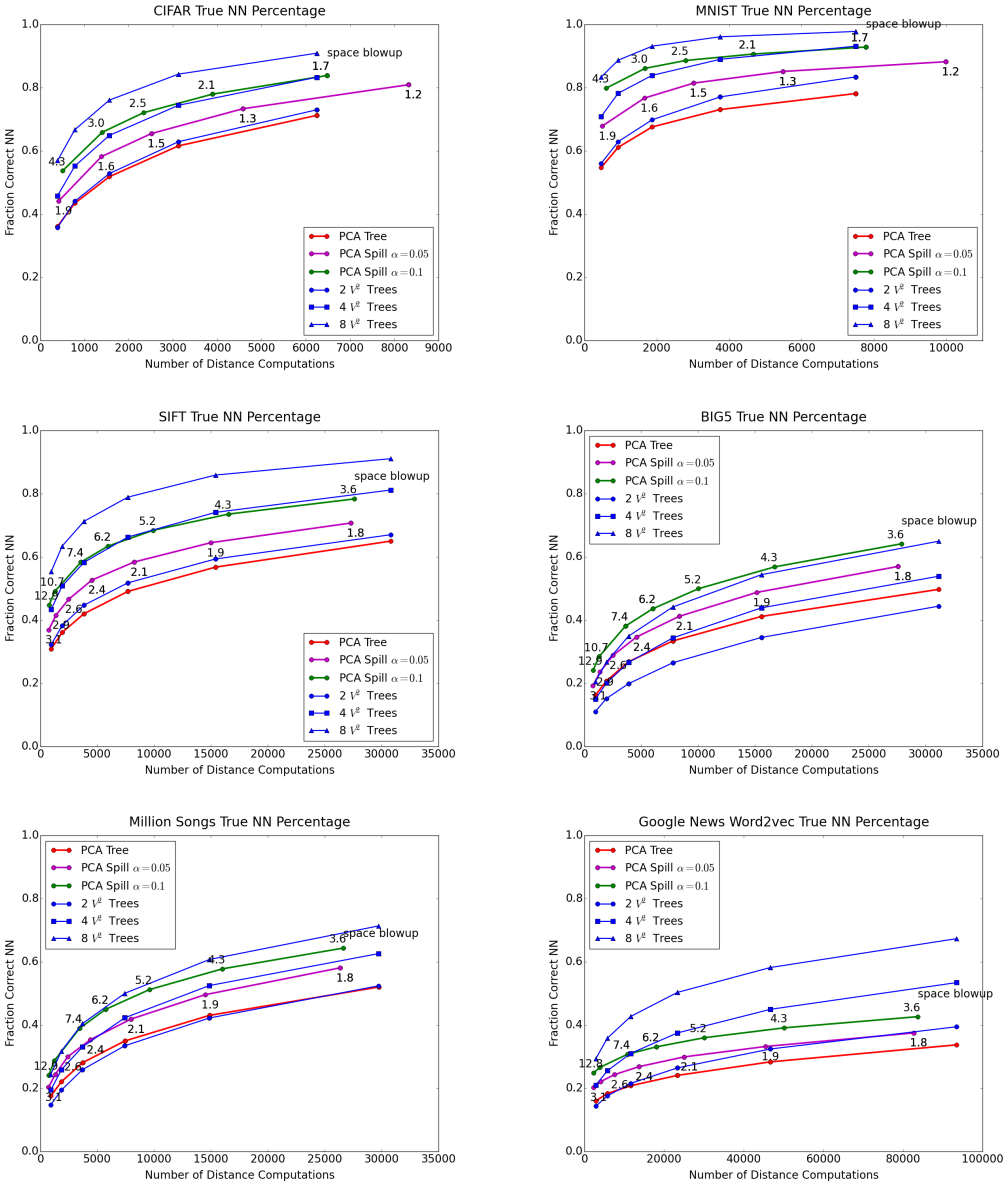

Figure 5.5: We compare PCA spill trees with V^2 trees. Eight V^2 trees outperform PCA spill trees in almost all dataset, except BIG5. PCA spill trees with spill factor 0.1 have slightly higher true NN accuracy than eight V^2 trees. It has a space blowup rate range from 3 to 12.

Chapter 6

Perspective

We looked at three categories of tree structures, PCA trees, randomized trees, and spill trees. We showed that PCA trees are time-consuming but yield good true NN accuracy. Randomized trees are flexible that one can scale up the performance by introducing multiple trees. Spill trees boost up performance by allowing spillover and work well with small spill factors.

We introduced a new randomized tree structure, V^2 trees. V^2 tree is a randomized tree that can capture spatial information of the train data and has better true NN accuracy than any other randomized trees. V^2 trees can also outperform PCA trees and only require a small time complexity compared with PCA.

Furthermore, we examined V^2 trees with spill trees. Spillover is a technique that can be applied to any tree structures. It expands the tree to include more data points in each tree node and therefore also increases space complexity. We compare spill trees with V^2 trees given similar space complexity. We conclude that V^2 trees work better when large space blowup is allowed, and spillover works better when the space blowup rate is limited.

Bibliography

[1] J. L. Bentley. Multidimensional binary search trees used for associative searching. *CACM*, 18(9):509–517, 1975.

[2] T. Bertin-Mahieux, D. P. Ellis, B. Whitman, and P. Lamere. The million song dataset. In *Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR 2011)*, 2011.

[3] A. Coates, H. Lee, and A. Ng. An analysis of single-layer networks in unsupervised feature learning. 15:215–223, 2011.

[4] S. Dasgupta and Y. Freund. Random projection trees and low dimensional manifolds. In *Proceedings of the fortieth annual ACM symposium on Theory of computing*, pages 537–546. ACM, 2008.

[5] S. Dasgupta and K. Sinha. Randomized partition trees for exact nearest neighbor search. *CoRR*, abs/1302.1948, 2013.

[6] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Trans. Math. Softw.*, 3(3):209–226, 1977.

[7] H. Hotelling. Analysis of a complex of statistical variables into principal components. *Journal of Educational Psychology*, 24(6):417–441, 1933.

[8] H. Jegou, M. Douze, and C. Schmid. *Hamming Embedding and Weak Geometric Consistency for Large Scale Image Search*, pages 304–317. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[9] H. Jégou, M. Douze, and C. Schmid. Product Quantization for Nearest Neighbor Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(1):117–128, Jan. 2011.

[10] M. Kosinski, S. C. Matz, S. D. Gosling, V. Popov, and D. Stillwell. Facebook as a research tool for the social sciences: Opportunities, challenges, ethical considerations, and practical guidelines. *American Psychologist*, 70(6):543–556, 9 2015.

[11] T. Liu, A. W. Moore, K. Yang, and A. G. Gray. An investigation of practical approximate nearest neighbor algorithms. pages 825–832, 2005.

[12] W. Lord. *NEO PI-R - A Guide to Interpretation and Feedback in a Work Context.* Hogrefe Ltd, Oxford, 2007.

[13] S. Maneewongvatana and D. M. Mount. On the efficiency of nearest neighbor searching with data clustered in lower dimensions. In *Proceedings of the International Conference on Computational Sciences-Part I*, ICCS '01, pages 842–851, London, UK, UK, 2001. Springer-Verlag.

[14] T. Mikolov, I. Sutskever, K. Chen, G. Corrado, and J. Dean. Distributed representations of words and phrases and their compositionality. *CoRR*, abs/1310.4546, 2013.

[15] T. Mikolov, S. W.-t. Yih, and G. Zweig. Linguistic regularities in continuous space word representations. In *Proceedings of the 2013 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (NAACL-HLT-2013)*. Association for Computational Linguistics, May 2013.

[16] M. Muja and D. G. Lowe. Scalable nearest neighbor algorithms for high dimensional data. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36(11):2227–2240, Nov 2014.

[17] R. F. Sproull. Refinements to nearest-neighbor searching in k-dimensional trees. *Algorithmica*, 6(1):579–589, 1991.

[18] N. Verma, S. Kpotufe, and S. Dasgupta. Which spatial partition trees are adaptive to intrinsic dimension? In *Proceedings of the twenty-fifth conference on uncertainty in artificial intelligence*, pages 565–574. AUAI Press, 2009.