# Fast Nonparametric Machine Learning Algorithms for High-dimensional Massive Data and Applications — a Thesis Proposal

Ting Liu

January 4, 2005


## Abstract

Nonparametric methods have become increasingly popular in the statistics communities and probabilistic AI communities. A nonparametric model is a set that cannot be parameterized by a finite number of parameters. These models are extremely useful when the underlying distribution of the data is unknown except that which can be inferred from samples. One simple and well-known nonparametric method is called "nearest-neighbor classification". The nearest-neighbor method uses those observations in the training set $\mathcal{T}$ closest in input space to a query $\mathsf{q}$ to form the prediction of $\mathsf{q}$. Specifically, when $k$ of those observations in $\mathcal{T}$ are considered, it is called $k$-nearest-neighbor.

Despite its simplicity, $k$-nearest-neighbor (or $k$-NN) and its variants have been successful in a large number of machine learning problems, including pattern recognition, text categorization, information retrieval, database and data mining. It is also used for estimating sample distributions and Bayes error. Theoretically, a famous result of Cover and Hart (1967) shows that asymptotically the error rate of the 1-nearest-neighbor classifier is never more than twice the Bayes rate.

However, the $k$-nearest-neighbor method and many related nonparametric methods remain hampered by their computational complexity. A naïve implementation of $k$-NN has a running time proportional to the number of training samples and the number of dimensions of feature vectors. Many spatial methods, such as metric-trees, have been proposed and tested as ways of alleviating the computational cost of such statistics without resorting to approximate answers. But the effectiveness of these methods decreases as the number of dimensions of feature vectors increases. From another direction, researchers are trying to develop ways to find *approximate* nearest neighbors, which can typically be done more efficiently and in many cases are enough for classification or other tasks. Among the approximate algorithms, Locality Sensitive Hashing (LSH) stands out as being successful both theoretically and practically. But since the algorithm is designed to work well even in the worst-case scenario, it might not be very efficient in "benign" cases. In particular, due to its simplicity, it does not automatically adapt to the distribution of data – a feature metric-tree enjoys.

I propose four new spatial methods for fast $k$-NN and its variants, namely *KNS2, KNS3, IOC* and *spill-tree*. KNS2 and KNS3 are designed especially to speed up exact $k$-NN classification problems,

and the IOC algorithm is designed to speed up a variant of $k$-NN classification on large many-class high-dimensional datasets. These three algorithms all share the same insight: that finding the majority class among the $k$-nearest-neighbors of a query need not require us to explicitly find those $k$-nearest-neighbors. Our last algorithm, spill-tree, is designed for approximate-nearest-neighbor search. By combining metric-tree and LSH, the algorithm is able to achieve the advantages of both, and thus enjoys greater efficiency.

Our future plan includes four major parts:

- Further improve spill-tree structure and its searching algorithm.
- Theoretical analysis of the spill-tree algorithm.
- Explore a new trick, namely dual-tree searching, to further speed up $k$-NN.
- Apply the new algorithms to real-world problems of multimedia and computer vision.

In the following thesis proposal, I will present several partial progress notes and lay out a concrete plan of attack for the future.

# Contents

# 1 Introduction

We introduce the $k$-nearest-neighbor algorithm. We also discuss its problem and related work.

## 1.1 Nearest-Neighbor Problem

The first formulation of a rule of the nearest-neighbor type was proposed in 1951 by Fix and Hodges in the papers[21, 22], where they also gave a preliminary analysis of its properties. The inception of the method opened a rich field of research, with applications in a broad spectrum of areas. Cover and Hart (1967) further strengthened the idea of $k$-nearest-neighbor, or $k$-NN by showing that asymptotically the error rate of the 1-nearest-neighbor classifier is never more than twice the Bayes rate [10]. This result makes nearest-neighbor methods very appealing and widely accepted. Some examples of the applications of $k$-NN are: pattern recognition [17, 16], text categorization [55], database and data mining [27, 28], information retrieval [12, 19, 52], image and multimedia search [18, 50, 23, 53], machine learning [9], and statistics and data analysis [14, 36].

The nearest-neighbor method is depicted as follows: Assume the dataset consists of points in a $d$-dimensional Euclidean space. Let $\mathcal{T} = \{x_1, x_2, \ldots, x_n\}$ be the set of training data, and $\mathsf{q}$ be a query point, the nearest-neighbor algorithm is to find the closest point in $\mathcal{T}$ to $\mathsf{q}$. It can be extended to the $k$-nearest-neighbor case, in which $k$ closest points are returned by the algorithm. More formally, we can denote the procedure by $\mathcal{N} = \mathsf{kNN}(\mathsf{q}, k, \mathcal{T}) = \{w_1, w_2, \ldots, w_k\}$. Nearest-neighbor is a nonparametric method, and has been widely used in solving classification problems. In this case, each point $x_i$ in the training set $\mathcal{T}$ also comes with a lable $y_i \in \mathcal{L}$, where we define $m = |\mathcal{L}|$ to be the total number of classes. Especially, when $m = 2$, we call the problem *binary* $k$-NN classification. For $k$-nearest-neighbor classification, one first finds the $k$-nearest-neighbors of $\mathsf{q}$ from $\mathcal{T}$, namely set $\mathcal{N}$, and then labels $\mathsf{q}$ with the class that appears most frequently in $\mathcal{N}$. As a notational convention, we call the most frequent class the "winner" and all other classes the "losers". Thus, the $k$-NN classification amounts to finding the winner class and assigning it to the query point $\mathsf{q}$.

Many researchers have done work to make variations of $k$-NN algorithms improve their performance for different applications [55, 28] or to combine it with other methods [57]. We will not go into detail on all these variants of $k$-NN methods, since it is not the focus of this work.

## 1.2 Speedup Nearest-Neighbor

Nearest-neighbor searching is a fundamental problem in computational geometry. Despite numerous advances in computing technology and algorithm design $k$-NN, with all its variants, remains hampered by its computational complexity. Given a training set $\mathcal{T}$ with $n$ points, and each point in a $d$ dimensional space, a naive $k$-NN search needs to do a linear scan of $\mathcal{T}$ for every single query $\mathsf{q}$, and thus the computational time is $O(dn)$ per query. When both $n$ and $d$ are huge, the algorithm becomes very slow, and sometimes even impractical. There exist many high-dimensional massive problems in real-world applications. For instance, in multimedia applications such as IBM's QBIC (Query by Image Content), the number of features could be several hundreds [18, 50]. In information retrieval for text documents, vector-space representations involve several thousands of dimensions. In drug activity detection, the fingerprints of each compound can go up to $10^6$ dimensions. All these problems require one to search in a huge database containing hundreds of thousands of points. Thus, a naive linear search of $k$-NN is unrealistic, and an effective nearest-neighbor searching algorithm with sublinear running time is needed.

Several effective methods exist for this problem when the dimension $d$ is small, such as Voronoi diagrams [51], which work for 1 or 2 dimensions. Djouadi and Bouktache [15] proposed a method to decrease the number of training samples that are needed for distance calculation by dividing space. However this method is not effective when the number of dimensions is greater than seven. Other methods are designed to work for the problem when the dimension is moderate (i.g. up to the 10's), such as $k$-$D$ tree [24, 51], R-tree [27], and metric-tree [47, 56, 7]. Among these tree structures, metric-tree, or ball tree [56] so far represents the practical state of the art for achieving efficiency in the largest dimensionalities possible [44, 8] without resorting to approximate answers. They have been used in many different ways, and a variety of tree search algorithms and with a variety of "cached sufficient statistics" decorating the internal leaves, for example in [46, 13, 58, 49, 26]. Fast searches are achieved by skipping unnecessary sub-searches. However, many real-world problems are posed with very large dimensionalities that are beyond the capability of such search structures to achieve sub-linear efficiency, this is known as "the curse of dimensionality". In fact, for large enough $d$, in theory or in practice, all these spatial algorithms provide little improvement over the naive linear search. Thus, the high-dimensional case is the long-standing frontier of the nearest-neighbor problem.

### 1.3 Approximate Nearest-Neighbor Searching

Since the $k$-NN problem is very difficult to solve exactly in high dimensions, another avenue of researches focuses on investigating the approximate-nearest-neighbor problem. The premise of these researches is that in many cases it is not necessary to insist on the exact answer; instead, determining an approximate answer should suffice. This observation underlies a large body of recent research, including using random sampling for histogram estimation [6] and median approximation [41], using wavelets for selectivity estimation [42] and approximate SVD [34].

Many researchers work on designing approximate algorithms with a certificate property. Here we define an approximate-nearest-neighbor problem formally: Given an error bound $\epsilon > 0$, we say that a point $w^\epsilon \in \mathcal{T}$ is a $(1+\epsilon)$-NN of $\mathsf{q}$ if $||w^\epsilon - \mathsf{q}|| \leq (1+\epsilon)||w - \mathsf{q}||$, where $w$ is the true nearest-neighbor of $\mathsf{q}$. For the $k$-nearest-neighbor case, the $k^{th}$ point returned by the algorithm is no more than $(1+\epsilon)$ times the distance of the true $k^{th}$ nearest-neighbor. Again, we can represent the procedure by $\mathcal{N}^\epsilon = \mathsf{akNN}(\mathsf{q}, k, \epsilon, \mathcal{T}) = \{w_1^\epsilon, w_2^\epsilon, ..., w_k^\epsilon\}$. Further, the problem is often relaxed to only do this with high probability. Most of these approximate algorithms are still based on space partitioning (similar to the spatial-tree methods) with some flexibility, and people studied the problem from a theoretical perspective [2, 3, 37]. For instance, Arya and Fu [1] studied an expected-case complexity of an algorithm based on partition trees with priority search, and give an expected query time $O((1/\epsilon)^d \log n)$. But the constant in the $O((1/\epsilon)^d \log n)$ contains a term as large as $(1+2\sqrt{d}/\epsilon)^d$, which is huge when $d$ is large. Therefore, although these algorithms have very nice logarithmic dependence on $n$, they tend to be rather inefficient in practice.

Indyk and Motwani [29] proposed a new $(1+\epsilon)$-NN algorithm in 1998. Instead of using space partitioning, their algorithm relies on a new method called *locality sensitive hashing* (LSH). The key idea is to hash the points using several hash functions so as to ensure that, for each function, the probability of collision is much higher for objects which are close to each other than for those which are far apart. Then, one can determine near neighbors by hashing the query point and retrieving elements stored in buckets containing that point (refer to Section 5 for a detailed description). LSH turns out to be very successful both theoretically [29] and practically [25, 4]. However, notice that LSH is designed with very simple data structure and with the goal of achieving competitive results even in the worst case, which rarely happens in practice. This leaves space for improvement of efficiency by using more sophisticated data structures for "practical", namely more "benign" scenarios. See Section 5.1 for more details.

### 1.4 A Brief Summary of Results

I propose four new $k$-NN-related algorithms: KNS2, KNS3, IOC and Spill-tree. All these four algorithms are based on a space partitioning tree structure: metric-trees [47, 56, 7]. We observe that metric-tree does have the advantage in their flexibility, and thus it is easy to capture the intrinsic distribution of the dataset. Also, the triangle inequality can be used for metric-tree search to prune away nodes which are far away from the query. The problem with a metric-tree is that, in general, like all other spatial tree structures, it is hurt by the "curse of dimensionality". Empirical results show that, when $d$ is large, the volume of a metric-tree node at each level tends to decrease very slowly, so that lots of nodes overlap with each other. In this case, it is very hard to prune away any node, and the advantage of metric-tree disappears.

To circumvent the curse of dimensionality, we do not use metric-tree to perform exact $k$-NN search. Rather, we examine the precise statistical question to find additional opportunities for saving computation.

#### 1.4.1 KNS2 and KNS3

We call the first two algorithms KNS2 and KNS3 [38]. In fact, they are both designed for binary $k$-NN classification. Here binary denotes the case where the output label $y_i$ only has two possible values:$\{+1, -1\}$. KNS2 and KNS3 share the same insight that the task of $k$-NN classification of a query $\mathsf{q}$ *need not require us to explicitly find those $k$-nearest-neighbors.* To be more specific, there are three similar but in fact different questions: (a) *"What are the $k$-nearest-neighbors of $\mathsf{q}$?"* (b) *"How many of the $k$-nearest-neighbors of $\mathsf{q}$ are from the positive class?"* and (c) *"Are at least $\mathsf{t}$ of the $k$-nearest-neighbors from the positive class?"* Obviously, the answer to question (a) can be used to answer question (b), and (b) does to (c). However the reverse direction is not true. In other words, (b) and (c) are simpler questions to answer. People have been focusing on question (a), but uses of proximity queries in statistics far more frequently require (b) and (c) types of computations. In fact, for $k$-NN classification problem, when the threshold $\mathsf{t}$ is set, it is sufficient to just answer the much simpler question (c). The triangle inequality of metric-tree has the advantage of bounding the distances between data points, and thus can help us estimate the nearest-neighbors without explicitly finding them. Omachi and Aso [45] proposed a fast $k$-NN classifier based on the branch and bound method. The algorithm shares a similar idea with KNS2, but it did not fully explore the idea of doing $k$-NN classification without explicitly finding the $k$-nearest-neighbor set, and the speedup the algorithm achieved is limited. In section 3, we address Omachi's method in more detail. We test our algorithms on 17 synthetic and real-world datasets, with dimensions ranging from 2 to $1.1 \times 10^6$ and number of data points ranging from $10^4$ to $4.9 \times 10^5$. We observed up to a 100-fold speedup as compared to highly optimized traditional metric-tree-based $k$-NN.

### 1.4.2 IOC

KNS2 and KNS3 can only deal with the binary class case. Our third method, IOC (standing for the **I**nternational **O**lympic **C**ommittee), can apply to the case of $m$ classes where $m$, the number of classes, is greater than 2 [40]. IOC assumes a slightly different processing of the datapoints in the neighborhood of the query. This allows it to search a set of metric-trees, one for each class. During the searches it is possible to quickly prune away classes that cannot possibly be the majority. IOC takes the same leverage point as KNS2 and KNS3 that $k$-NN classification had, whereas the more general problem of $k$-NN does not have: all we need to do is to find the majority class of the $k$ nearest eighbors – not the neighbors themselves. In section 4, we show why it is hard to exploit this leverage point for the case of conventional $k$-NN. We therefore introduce a modified form of $k$-NN called IOC (explained later in section 4) that selects the predicted class by a kind of elimination tournament instead of a direct majority vote. Interestingly, this alternative scheme exhibits no general degradation in empirical performance, and we also prove that the asymptotic behavior of IOC must be very close to that of conventional $k$-NN. We give experimental results on datasets of up to $5.8 \times 10^3$ records and $1.5 \times 10^3$ attributes, frequently showing an order of magnitude acceleration compared with each of (i) conventional linear scan, (ii) a well-known independent SR-tree implementation of conventional $k$-NN and (iii) a highly optimized conventional $k$-NN metric-tree search.

### 1.4.3 Spill-tree

In this algorithm [39], we introduce a new data structure, namely spill-tree. It is a variant of metric-tree. For metric-tree, the point sets contained by two children nodes must be disjoint, while a spill-tree does not have this restriction. Furthermore, we deliberately allowes certain datapoints in a spill-tree to be shared between children nodes. In this way, we are able to combine the advantages of both metric-tree and LSH. We show a new approximate $k$-NN search algorithm based on spill-tree, the algorithm exploits the same random-projection-based approximation that LSH has, but by using a more sophisticated data structure, it achieves greater efficiency. We then provide a detailed empirical evaluation on five large, high-dimensional datasets which show accelerations one to three orders of magnitude over LSH. This result holds true throughout the spectrum of approximation levels.

### 1.4.4 Summary

All the results and detailed descriptions of our new algorithms can be found in the following three papers: KNS2 and KNS3 is introduced in [38], IOC comes from [40], and Spill-tree is from [39].

## 2 Metric-tree

We review the metric-tree data structure, which is the basic data structure our algorithms rely on.

### 2.1 Properties

The *metric-tree* [47, 56, 7, 44]is a data structure that supports efficient nearest-neighbor search. We briefly review some of its properties: A metric-tree organizes a set of points in a spatial hierarchical manner. It is a binary tree whose nodes represent a set of points. The root node represents all points, and the points represented by an internal node $\mathsf{v}$ is partitioned into two subsets, represented by its two children. Formally, if we use $N(\mathsf{v})$ to denote the set of points represented by node $\mathsf{v}$, and use $\mathsf{v.lc}$ and $\mathsf{v.rc}$ to denote the left child and the right child of node $\mathsf{v}$, then we have

$$N(\mathsf{v}) = N(\mathsf{v.lc}) \cup N(\mathsf{v.rc}) \tag{1}$$
$$\emptyset = N(\mathsf{v.lc}) \cap N(\mathsf{v.rc}) \tag{2}$$

for all the non-leaf nodes. At the lowest level, each leaf node contains only a few points.

### 2.2 Partitioning

The key to building a metric-tree is how to partition a node $\mathsf{v}$. One typical way is as follows: We first choose two *pivot* points from $N(\mathsf{v})$, denoted as $\mathsf{v.lpv}$ and $\mathsf{v.rpv}$. Ideally, $\mathsf{v.lpv}$ and $\mathsf{v.rpv}$ are chosen so that the distance between them is the largest of all-pair distances within $N(\mathsf{v})$. More specifically, $||\mathsf{v.lpv} - \mathsf{v.rpv}|| = \max_{p1,p2 \in N(\mathsf{v})} ||p1 - p2||$. However, it takes $O(n^2)$ time to find the optimal $\mathsf{v.lpv}$ and $\mathsf{v.rpv}$. In practice, we resort to a linear-time heuristic that is still able to find reasonable pivot points. Basically, we first randomly pick a point $p$ from $\mathsf{v}$. Then we search for the point that is the farthest from $p$ and set it to be $\mathsf{v.lpv}$. Next we find a third point that is farthest from $\mathsf{v.lpv}$ and set it as $\mathsf{v.rpv}$. After $\mathsf{v.lpv}$ and $\mathsf{v.rpv}$ are found, we can partition node $\mathsf{v}$.

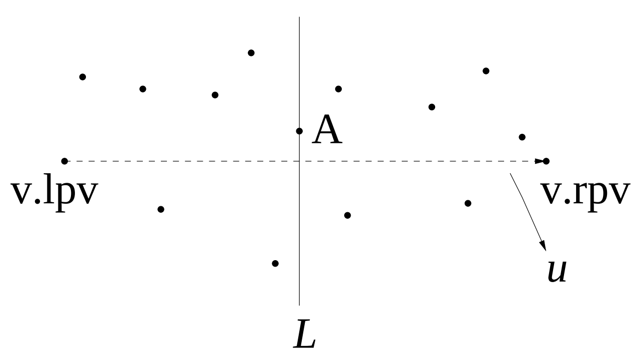


Figure 1: partitioning in a metric-tree.

There are many ways to partition $\mathsf{v}$, and we focus on a particular strategy. We first project all the points down to the vector $\vec{u} = \mathsf{v.\vec{rpv}} - \mathsf{v.\vec{lpv}}$, and then find the median point $A$ along $\vec{u}$. Next, we assign all the points projected to the left of $A$ to $\mathsf{v.lc}$, and all the points projected to the right of $A$ to $\mathsf{v.rc}$. We use $L$ to denote the $d-1$ dimensional plane that is orthogonal to $\vec{u}$ and passes through $A$. $L$ is known as the *decision boundary* since all points to the left of $L$ belong to $\mathsf{v.lc}$ and all points to the right of $L$ belong to $\mathsf{v.rc}$ (see Figure 1). By using a median point to split the datapoints, we can ensure an even partitioning and that the depth of a metric-tree is $\log n$. However, in our implementation, we use a mid point (i.e. the point at $\frac{1}{2}(\mathsf{v}.\vec{\mathsf{lpv}} + \mathsf{v}.\vec{\mathsf{rpv}})$) instead, since it is more efficient to compute, and in practice, we can still have a metric-tree of depth $O(\log n)$.

Each node $\mathsf{v}$ also has a hypersphere $\mathbb{B}$, such that all points represented by $\mathsf{v}$ fall in the ball centered at $\mathsf{v.center}$ with radius $\mathsf{v.r}$, *i.e.* we have

$$N(\mathsf{v}) \subseteq \mathbb{B}(\mathsf{v.center}, \mathsf{v.r}). \tag{3}$$

Notice that the balls of the two children nodes are not necessarily disjoint.

In fact, $\mathsf{v.center}$ is typically the centroid of $N(\mathsf{v})$ and $\mathsf{v.r}$ is chosen to be the minimal value satisfying (3). As a consequence, we know that the radius of any node is always greater than the radius of any of its children nodes. The leaf nodes have very small radii.

### 2.3 Searching

A search on a metric-tree is simply a guided DFS (for simplicity, we assume that $k = 1$). The decision boundary $L$ is used to decide which child node to search first. If the query $\mathsf{q}$ is on the left of $L$, then $\mathsf{v.lc}$ is searched first, otherwise, $\mathsf{v.rc}$ is searched first. At all times, the algorithm maintains a "candidate NN", which is the nearest-neighbor it has found so far while traversing the tree. We call this point $x$, and denote the distance between $\mathsf{q}$ and $x$ by $r$. If DFS is about to exploit a node $\mathsf{v}$, but discovers that no member of $\mathsf{v}$ can be within distance $r$ of $\mathsf{q}$, then it *prunes* this node (i.e., skips searching on this node, along with all its descendants). This happens whenever $\|\mathsf{v.center} - \mathsf{q}\| - \mathsf{v.r} \geq r$. We call this DFS search algorithm MT-DFS hereafter.

Metric-trees have some advantages. First, they can easily capture the clustering property of the dataset and automatically adapt to the local resolution of the data points. Second, since the partitioning plane (i.e. decision boundary $L$) of a metric-tree can be along any direction, they are generated according to the distribution of datapoints. This flexibility makes metric-trees more efficient than $k-D$ trees. In practice, the MT-DFS algorithm is very efficient for $k$-NN search, and particularly when the dimensionality of a dataset is low (say, less than 30). In the best case, the complexity of MT-DFS can be as good as $O(d \log n)$ per query, where $d$ is the dimensions and $n$ is the number of points. Typically for MT-DFS we observe an order of magnitude speed-up over naïve linear scan and other popular data structures such as SR-trees. However, MT-DFS starts to slow down as the dimensionality of the datasets increases. In the worst case, the complexity of MT-DFS can be as bad as $O(dn)$ per query. Sometimes it is even slower than naïve $k$-NN search.

## 3 Fast K-Nearest-Neighbor Classification

In this section, we describe our first two new algorithms KNS2 and KNS3. Both algorithms are based on metric-tree structure, but use different searching strategies.

### 3.1 KNS2

In many binary classification domains, one class is much more frequent than the other. For example, in High Throughput Screening datasets, it is far more common for the result of an experiment to be negative than positive. In detection of fraud telephone calls [20] or credit card transactions [54], the number of legitimate transactions is far more common than fraudulent ones. In insurance risk modeling [48], a very small percentage of the policyholders file one or more claims in a given time period. There are many other examples of domains with similar intrinsic imbalance, and therefore, classification with a skewed distribution is important. Various researchers have focused on designing clever methods to solve this type of problem [5, 43]. The new algorithm introduced in this section, KNS2, is designed to accelerate $k$-NN based classification in such skewed data scenarios.

We rely on the fact that the following two problems are not the same: (a) "Find the $k$-nearest-neighbors." and (b) "How many of the $k$-nearest-neighbors are in the positive class?" Answering (b) exactly does not necessarily require us to answer (a), and in fact it is an easier question to answer. Here, we explicitly investigate whether this fact can be exploited computationally.

KNS2 attacks the problem by building two metric-trees. A *Postree* for the points from the positive (small) class, a *Negtree* for the points from the negative (large) class. Since the number of points from the positive class is so small, it is quite cheap to find the exact $k$-nearest positive points of $\mathsf{q}$ by using MT-DFS. Generally speaking, the idea of KNS2 is as follows: First search *Postree* using MT-DFS to find the $k$-nearest positive neighbors set $Posset_k$. Then search *Negtree* using $Posset_k$ as the initial $k$-NN candidates. We can estimate the number of negative points within the $k$-NN set and prune away unrelated nodes as the same time. The search can be stopped as soon as we get the answer to question (b). Empirically, much more pruning can be achieved by KNS2 than MT-DFS, the concrete description of the algorithm is as following:

Let $Root_{pos}$ be the root of *Postree*, and $Root_{neg}$ be the root of *Negtree*. Then, we classify a new query point $\mathsf{q}$ in the following fashion

- Step 1 — " **Find positive**": Find the $k$ nearest positive class neighbors of $\mathsf{q}$ (and their distances to $\mathsf{q}$) using MT-DFS.
- Step 2 — **"Insert negative"**: Do sufficient search on the negative tree to prove that the number of positive datapoints among $k$-nearest-neighbors is $p$ for some value of $p$.

Step 2 is achieved using a new recursive search called *NegCount*. In order to describe *NegCount* we introduce a set of quantities.

- **Dists.**

  $\mathsf{Dists}$ is an array of elements $\mathsf{Dists}_1, \ldots, \mathsf{Dists}_k$ consisting of the distances to the $k$ nearest positive neighbors found so far of $\mathsf{q}$, sorted in increasing order of distance. For notational convenience we will also write $\mathsf{Dists}_0 = 0$ and $\mathsf{Dists}_{k+1} = \infty$.

- **Pointset.**

  Pointset $V$ is the set of points in the negative nodes visited so far in the search.

- $(n,\ C)$ $(n \le k+1)$.

  C is an array of counts containing p+1 array elements $C_0, C_1, \ldots, C_n$. Say (n, C) summarize interesting negative points for pointset $V$ if and only if

  1. $\forall i = 0, 1, \ldots, n$,
  $$C_i = \mid V \cap \{x : \mid x - q \mid < \mathsf{Dists}_i\} \mid \tag{4}$$

     Intuitively $C_i$ is the number of points in $V$ whose distances to $\mathsf{q}$ are closer than $\mathsf{Dists}_i$. In other words, $C_i$ is the number of negative points in $V$ closer than the $i^{th}$ positive neighbor to $\mathsf{q}$.

  2. $$\begin{cases} C_i + i \le k & \forall i = 0, 1, \ldots, n-1 \\ C_i + i > k & i = n \end{cases} \tag{5}$$

     This simply declares that the length $n$ of the $C$ array is as short as possible while accounting for the $k$ members of $V$ that are nearest to $\mathsf{q}$. Such an $n$ exists since $C_0 = 0$ and $C_{k+1}$ = Total number of negative points. To make the problem interesting, we assume that the number of negative points and the number of positive points are both greater than $k$.

- $D^{\mathsf{v}}_{\mathsf{minp}}$ and $D^{\mathsf{v}}_{\mathsf{maxp}}$

  Here we define two quantities $D^{\mathsf{v}}_{\mathsf{minp}}$ and $D^{\mathsf{v}}_{\mathsf{maxp}}$ to represent the minimum and maximum possible distance from $\mathsf{q}$ to a current node $\mathsf{v}$.

$$\text{Let } D^{\mathsf{v}}_{\mathsf{minp}} = \begin{cases} \max(||\mathsf{q} - \mathsf{v.center}|| - \mathsf{v.r}, D^{\mathsf{v.parent}}_{\mathsf{minp}}) & \text{if } \mathsf{v} \ne root \\ ||\mathsf{q} - \mathsf{v.center}|| - \mathsf{v.r} & \text{if } \mathsf{v} == root \end{cases} \tag{6}$$

$$\text{Let } D^{\mathsf{v}}_{\mathsf{maxp}} = \begin{cases} \min(||\mathsf{q} - \mathsf{v.center}|| + \mathsf{v.r}, D^{\mathsf{v.parent}}_{\mathsf{maxp}}) & \text{if } \mathsf{v} \ne root \\ ||\mathsf{q} - \mathsf{v.center}|| + \mathsf{v.r} & \text{if } \mathsf{v} == root \end{cases} \tag{7}$$

$D^{\mathsf{v}}_{\mathsf{minp}}$ and $D^{\mathsf{v}}_{\mathsf{maxp}}$ are computed using the triangle inequality and the property of a metric-tree that all the points covered by a node must be covered by its parent. This property implies that $D^{\mathsf{v}}_{\mathsf{minp}}$ will never be smaller than the minimum possible distance of its ancestors. Symmetrically, $D^{\mathsf{v}}_{\mathsf{maxp}}$ will never be greater than the maximum possible distance of its ancestors.

Figure 2 gives a good example. There are 3 nodes $p$, $c1$ and $c2$, where $c1$ and $c2$ are $p$'s children, and $\mathsf{q}$ is the query point. In order to compute $D^{c1}_{minp}$, first we compute $|\mathsf{q} - c1.\mathsf{center}| - c1.\mathsf{r}$, which is the length of the dotted line in the figure, but $D^{c1}_{minp}$ can be also bounded by $D^{p}_{minp}$, since it is impossible for any point to be in the shaded area. Similarly, we get the equation for $D^{c1}_{maxp}$.

$D^{\mathsf{v}}_{\mathsf{minp}}$ and $D^{\mathsf{v}}_{\mathsf{maxp}}$ are used to estimate the counts array $(n, C)$. Again we take advantage of the triangle inequality of metric-tree. For any node $\mathsf{v}$, if there exists an $i$ ($i \in [1, n]$), such that $Dists_{i-1} \le D^{\mathsf{v}}_{\mathsf{maxp}} < Dists_i$, then for $\forall x \in Points(\mathsf{v})$, $Dists_{i-1} \le \mid x - \mathsf{q} \mid < Dists_i$. According to the definition of $C$, we can add $\mid Points(\mathsf{v}) \mid$ to $C_i, C_{i+1}, \ldots C_n$. The function of $D^{\mathsf{v}}_{\mathsf{minp}}$ similar to MT-DFS, is used to help prune uninteresting nodes.

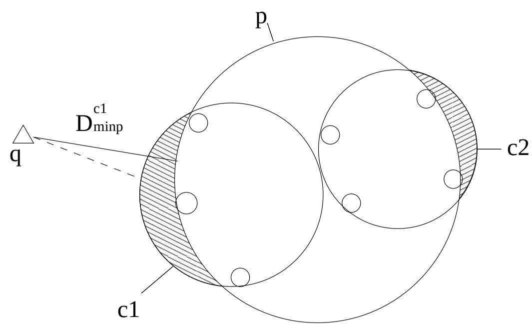


Figure 2: An example to illustrate how to compute $D^{\mathsf{v}}_{\mathsf{minp}}$

Step 2 of KNS2 is implemented by the recursive function below:

$$(n^{out}, C^{out}) = NegCount(n^{in}, C^{in}, Node, j_{parent}, Dists)$$

Assume that on entry $(n^{in}, C^{in})$ summarize interesting negative points for pointset $V$, where $V$ is the set of points visited so far during the search. This algorithm efficiently ensures that on exit $(n^{out}, C^{out})$ summarize interesting negative points for $V \cup Points(\mathsf{v})$. In addition, $j_{parent}$ is a temporary variable used to prevent multiple counts for the same point. This variable relates to the implementation of KNS2, and we do not want to go into the details here.

---

**Procedure** $NegCount$ $(n^{in}, C^{in}, Node, j_{parent}, \mathsf{Dists})$
**begin**

$n^{out}$ := $n^{in}$; $C^{out}$ := $C^{in}$
Compute $D^{\mathsf{v}}_{\mathsf{minp}}$ and $D^{\mathsf{v}}_{\mathsf{maxp}}$
Search for $i, j \in [1, n^{out}]$, such that
$\mathsf{Dists}_{i-1} \leq D^{\mathsf{v}}_{\mathsf{minp}} < \mathsf{Dists}_i$
$\mathsf{Dists}_{j-1} \leq D^{\mathsf{v}}_{\mathsf{maxp}} < \mathsf{Dists}_j$

For all index $\in [j, j_{parent})$ /* Re-estimate $C^{out}$ */
Update $C^{out}_{index} := C^{out}_{index} + \mid Points(\mathsf{v}) \mid$ /* We only update the count less than $j_{parent}$ to avoid counting twice for each point*/
Update $n^{out}$, such that
$C^{out}_{n^{out}-1} + (n^{out} - 1) \leq k$, $C^{out}_{n^{out}} + n^{out} > k$

Set $Dists_{n^{out}}$ := $\infty$

(1) **if** $(n^{out} == 1)$ **return**$(1, C^{out})$ /* At least $k$ negative points closer to $\mathsf{q}$ than the closest positive one */

(2) **if** $(i == j)$ **return**$(n^{out}, C^{out})$ /* Node is located between two adjacent positive points, no need to split */

(3) **if** (Node is a leaf)
Forall $x \in Points(Node)$, compute $\mid x - \mathsf{q} \mid$
Update and return $(n^{out}, C^{out})$

(4) **else**

$node_1$ := child of Node closest to $\mathsf{q}$

$node_2$ := child of Node furthest from $\mathsf{q}$

$(n^{temp}, C^{temp})$ := NegCount$(n^{in}, C^{in}, node_1, j, \mathsf{Dists})$

**if** $(n^{temp} == 1)$ **return** $(1, C^{out})$

**else**

$(n^{out}, C^{out})$ := $NegCount(n^{temp}, C^{temp}, node_2, j, \mathsf{Dists})$

**end**

---

We can stop the procedure when $n^{out}$ becomes 1 (which means all the $k$-nearest-neighbors of $\mathsf{q}$ are in the negative class) or when we run out of nodes. $n^{out}$ represents the number of positive points in the $k$-nearest-neighbors of $\mathsf{q}$. The top-level call is

$$NegCount(k, C^0, NegTree.Root, k+1, Dists)$$

where $C^0$ is an array of zeroes and Dists are defined in step 2 and obtained by applying MT-DFS to the *Postree*.

There are at least two situations in which KNS2 can run faster than MT-DFS. First, when we have found at least $k$ negative points closer than the nearest positive point to $\mathsf{q}$, we can stop. Notice that the $k$ negative points we found are not necessarily the exact $k$-nearest-neighbors to $\mathsf{q}$, in this case, MT-DFS will continue on, but this will not change the answer to our question. This situation happens frequently for skewed datasets. The second situation is as follows: A node can also be pruned if it is located exactly between two adjacent positive points, or it is farther away than the $n^{th}$ positive point. This is because in these situations, there is no need to figure out which negative point is closer within the Node. Especially as $n$ gets smaller, we have more chance to prune a node, because $Dists_{n^{in}}$ decreases as $n^{in}$ decreases.

### 3.2 KNS3

In our second new algorithm KNS3, we remove KNS2's constraint of an assumed skewedness in the class distribution. And we answer an even weaker question: "are at least $t$ of the $k$-nearest-neighbors positive?" (where $t$, a "threshold" value, is supplied by the questioner). This is often the most statistically relevant question, for example during classification with known false positive and false negative costs.

In KNS3, we define two important quantities:

$$D_{\mathsf{t}}^{\mathsf{pos}} = \textit{distance of the } t^{th} \textit{ nearest positive neighbor of } \mathsf{q} \quad (8)$$

$$D_{\mathsf{t'}}^{\mathsf{neg}} = \textit{distance of the } (t')^{th} \textit{ nearest negative neighbor of } \mathsf{q} \quad (9)$$

where $t + t' = k + 1$.

An important property of the two quantities is as follows:

$D_{\mathsf{t}}^{\mathsf{pos}} \le D_{\mathsf{t'}}^{\mathsf{neg}}$ *if and only if at least* $t$ *of the* $k$*-nearest-neighbors of* $\mathsf{q}$ *are from the positive class.*

Figure 3 provides an illustration. In this example, $k = 5$, $t = 3$. We use black dots to denote positive points, and white dots to denote negative points. We first compute $t' = k - t + 1 = 3$, then compute $D^{\mathsf{pos}}_{\mathsf{t}}$ and $D^{\mathsf{neg}}_{\mathsf{t'}}$. In the given example, $D^{\mathsf{pos}}_{\mathsf{t}} = ||\mathsf{q} - A||$, $D^{\mathsf{neg}}_{\mathsf{t'}} = ||\mathsf{q} - B||$, and it is obvious that $D^{\mathsf{pos}}_{\mathsf{t}} \leq D^{\mathsf{neg}}_{\mathsf{t'}}$. So within the circle centers at $\mathsf{q}$, with radius equals to $||\mathsf{q} - B||$, there are exactly three negative points and at least three positive points. In other words, at least 3 of the 5 nearest neighbors of $\mathsf{q}$ are from the positive class.

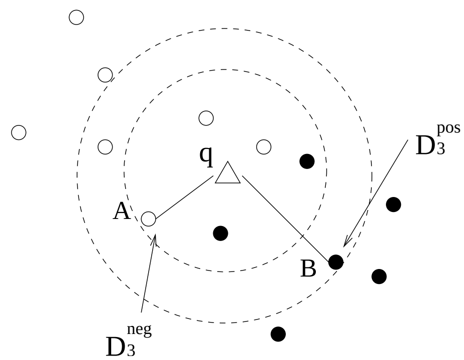


Figure 3: An example of $D^{\mathsf{pos}}_{\mathsf{t}}$ and $D^{\mathsf{neg}}_{\mathsf{t'}}$

The reason to redefine the problem of KNS3 is to transform a $k$-NN searching problem to a much simpler counting problem. In fact, in order to answer the question, we do not even have to compute the exact value of $D^{\mathsf{pos}}_{\mathsf{t}}$ and $D^{\mathsf{neg}}_{\mathsf{t'}}$, instead, we can estimate them. We define $Lo(D^{\mathsf{pos}}_{t})$ and $Up(D^{\mathsf{pos}}_{t})$ as the lower and upper bounds of $D^{\mathsf{pos}}_{\mathsf{t}}$; and similarly we define $Lo(D^{\mathsf{neg}}_{t'})$ and $Up(D^{\mathsf{neg}}_{t'})$ as the lower and upper bounds of $D^{\mathsf{neg}}_{\mathsf{t'}}$. If at any point, we know $Up(D^{\mathsf{pos}}_{t}) \leq Lo(D^{\mathsf{neg}}_{t'})$), then we can conclude that $D^{\mathsf{pos}}_{\mathsf{t}} \leq D^{\mathsf{neg}}_{\mathsf{t'}}$. On the other hand, if $Up(D^{\mathsf{neg}}_{t'}) \leq Lo(D^{\mathsf{pos}}_{t})$, we know $D^{\mathsf{neg}}_{\mathsf{t'}} \leq D^{\mathsf{pos}}_{\mathsf{t}}$. Now our computational task is to efficiently estimate $Lo(D^{\mathsf{pos}}_{t})$, $Up(D^{\mathsf{pos}}_{t})$, $Lo(D^{\mathsf{neg}}_{t'})$ and $Up(D^{\mathsf{neg}}_{t'})$. And it is very convenient for a metric-tree to do so. Please refer to [38] for detailed description.

## 3.3 Experimental Results

We report our experimental results based on KNS2 and KNS3 on 11 real-world datasets, with the number of datapoints ranging from $2.6 \times 10^4$ to $4.9 \times 10^5$, and dimensions from 10 to $1.1 \times 10^6$. Below is a summary of the datasets.

Table 1: Datasets

| **Dataset** | **Num. records** | **Num. Dimensions** | **Num. pos.** | **Dataset** | **Num. records** | **Num. Dimensions** | **Num. pos.** |
|---|---|---|---|---|---|---|---|
| ds1 | 26733 | 6348 | 804 | ds1.10pca | 26733 | 10 | 804 |
| ds1.100pca | 26733 | 100 | 804 | ds2.100anchor | 88358 | 100 | 211 |
| ds2 | 88358 | 1100000 | 211 | J_Lee.100pca | 181395 | 100 | 299 |
| Letter | 20000 | 16 | 790 | Blanc__Mel | 186414 | 10 | 824 |
| Movie | 38943 | 62 | 7620 | Kdd99(10%) | 494021 | 176 | 97278 |
| Ipums | 70187 | 60 | 119 | | | | |

For each dataset, we tested $k = 9$ and $k = 101$. For KNS3, we used $t = \lceil \mathrm{k}/2 \rceil$: a data-point is classified as positive if and only if the majority of its $k$-nearest-neighbors are positive. Each experiment performed 10-fold cross-validation. Thus, each experiment required $n$ $k$-NN classification queries (where $n$ is the total number of points in the dataset) and each query involved the $k$-NN among $0.9n$ records. A naïve implementation with no metric-tree would thus require $0.9n^2$ distance computations.

Table 2 shows the computational cost of naïve $k$-NN both in terms of the number of distance computations and the wall-clock time on an unloaded 2GHz Pentium. We then examine the speedups of MT-DFS and our two new methods (KNS2 and KNS3). It is notable that for some high dimensional datasets, MT-DFS does not produce an acceleration over naïve. On the other hand, KNS2 and KNS3 do, however, and in some cases they are hundreds of times faster than MT-DFS.

Table 2: Number of distance computations and wall-clock-time for Naïve $k$-NN classification. Acceleration for MT-DFS KNS2 and KNS3 (in terms of num. distances and time).

| | | **Naïve** | | MT-DFS | | **KNS2** | | **KNS3** | |
|---|---|---|---|---|---|---|---|---|---|
| | | dists | time (secs) | dists speedup | time speedup | dists speedup | time speedup | dists speedup | time speedup |
| ds1 | k=9 | $6.4 \times 10^8$ | 4830 | 1.6 | 1.0 | 4.7 | 3.1 | 12.8 | 5.8 |
| | k=101 | | | 1.0 | 0.7 | 1.6 | 1.1 | 10 | 4.2 |
| ds1.10pca | k=9 | $6.4 \times 10^8$ | 420 | 11.8 | 11.0 | 33.6 | 21.4 | 71 | 20 |
| | k=101 | | | 4.6 | 3.4 | 6.5 | 4.0 | 40 | 6.1 |
| ds1.100pca | k=9 | $6.4 \times 10^8$ | 2190 | 1.7 | 1.8 | 7.6 | 7.4 | 23.7 | 29.6 |
| | k=101 | | | 0.97 | 1.0 | 1.6 | 1.6 | 16.4 | 6.8 |
| ds2 | k=9 | $8.5 \times 10^9$ | 105500 | 0.64 | 0.24 | 14.0 | 2.8 | 25.6 | 3.0 |
| | k=101 | | | 0.61 | 0.24 | 2.4 | 0.83 | 28.7 | 3.3 |
| ds2.100- | k=9 | $7.0 \times 10^9$ | 24210 | 15.8 | 14.3 | 185.3 | 144 | 580 | 311 |
| | k=101 | | | 10.9 | 14.3 | 23.0 | 19.4 | 612 | 248 |
| J_Lee.100- | k=9 | $3.6 \times 10^{10}$ | 142000 | 2.6 | 2.4 | 28.4 | 27.2 | 15.6 | 12.6 |
| | k=101 | | | 2.2 | 1.9 | 12.6 | 11.6 | 37.4 | 27.2 |
| Blanc_Mel | k=9 | $3.8 \times 10^{10}$ | 44300 | 3.0 | 3.0 | 47.5 | 60.8 | 51.9 | 60.7 |
| | k=101 | | | 2.9 | 3.1 | 7.1 | 33 | 203 | 134.0 |
| Letter | k=9 | $3.6 \times 10^8$ | 290 | 8.5 | 7.1 | 42.9 | 26.4 | 94.2 | 25.5 |
| | k=101 | | | 3.5 | 2.6 | 9.0 | 5.7 | 45.9 | 9.4 |
| Movie | k=9 | $1.4 \times 10^9$ | 3100 | 16.1 | 13.8 | 29.8 | 24.8 | 50.5 | 22.4 |
| | k=101 | | | 9.1 | 7.7 | 10.5 | 8.1 | 33.3 | 11.6 |
| Ipums | k=9 | $4.4 \times 10^9$ | 9520 | 195 | 136 | 665 | 501 | 1003 | 515 |
| | k=101 | | | 69.1 | 50.4 | 144.6 | 121 | 5264 | 544 |
| Kddcup99 | k=9 | $2.7 \times 10^{11}$ | 1670000 | 4.2 | 4.2 | 574 | 702 | 4 | 4.1 |
| (10%) | k=101 | | | 4.2 | 4.2 | 187.7 | 226.2 | 3.9 | 3.9 |

# 4 IOC

KNS2 and KNS3 work well for binary $k$-NN classification. Unfortunately, the insight used in these algorithms does not work directly in the many-class case. Consider a query point $\mathsf{q}$ and its $k$-nearest-neighbor set $\mathcal{N}$. For binary classification, $\mathsf{q}$ is classified as class $i$, if and only if $\mathcal{N}$ contains more than $\lfloor k/2 \rfloor$ points of class $i$. Thus the task of finding the winner (majority class) is reduced to a counting problem. In the case of many-classes, where there are $m$ classes in total, the situation is very different. We no longer have a fixed threshold that allows us to reduce the search-for-winner problem to a counting problem. We know that for a class to be the winner, it is necessary to contain more than $\lfloor k/m \rfloor$ points in $\mathcal{N}$, and it is sufficient to contain more than $\lfloor k/2 \rfloor$ points. However, for numbers between $\lfloor k/m \rfloor$ and $\lfloor k/2 \rfloor$, we cannot prove anything. Therefore, we cannot reduce the $k$-NN search problem to a simple counting problem. This is the reason why the previous techniques do not extend directly to the many-class case.

### 4.1 IOC: high-level description

The IOC algorithm is a variant to the $k$-NN algorithm that allows speedup using metric-trees. The motivation behind IOC is to modify $k$-NN in such a way that it can be reduced to a *sequence* of counting problems. One important observation is that despite the fact that the necessary condition and the sufficient condition combined cannot determine if an *arbitrary* class is the winner in general, one can always use the necessary condition to find *some* class that is not a winner. This is simply by the pigeonhole principle, there exists at least one class containing at most $\lfloor k/m \rfloor$ points, and this class is not the winner.

This algorithm is inspired by the procedure used by the International Olympic Committee [33] to select the host city for summer Olympic games (which also explains its name). In the procedure, instead of having a single round of ballots and selecting the favorite city as the winner (which would correspond to the "standard" $k$-NN algorithm), multiple rounds of ballots are cast. In each round, if a city gets a majority of the votes, then it is declared the winner and the procedure finishes. Otherwise, the city that gets the least votes is eliminated and a new round of ballots is cast. This continues until either a city gets a majority vote, or only one city is left, and this city is declared the winner.

We now describe the IOC algorithm at a high level. IOC starts by building a metric-tree for each class respectively, and then proceeds in *rounds*. In each round, either a winner is selected, or some "losers" are eliminated. More precisely, in each round, if a class $i$ contains more than $\lfloor k/2 \rfloor$ points in the $k$-nearest-neighbors of $\mathsf{q}$, this class is declared a winner and the algorithm terminates, labeling $\mathsf{q}$ with class $i$. Otherwise, the algorithm finds all the classes that contains at most $\lfloor k/m \rfloor$ points in the $k$-nearest-neighbors of $\mathsf{q}$, and declare these classes the "losers", all the loser classes will be removed from consideration. The number of classes, $m$, is reduced accordingly. This process continues until a winner is selected or there is only one class remaining, in which case the only remaining class is declared a winner. Please refer to [40] for detailed description of the IOC algorithm. [1] We also did a theoretical analysis of IOC and experimental comparison between conventional $k$-NN and IOC. In this way, we can justify that IOC is a reasonable variation of conventional $k$-NN classification.

### 4.2 Experimental Results

In this section, we tested the IOC algorithm on a variety of real-world datasets (listed in Table 3) and compared the results with three other algorithms: Naïve, SR-tree (an implementation by Katayama and Satoh [35]) and MT-DFS. We estimate two performance measures:

[1] In actual implementation, a more efficient algorithm was designed for practical performance.

1. **Speed**: this is the primary concern of this paper. We considered accelerations both in terms of number of distance computations and CPU time. For all the experiments below, we first show the computational cost of naïve $k$-NN. We then examine the speedups of SR-tree, MT-DFS and IOC over naïve $k$-NN. (Notice that for SR-tree, we omit the distance computations speedup, since the SR-tree implementation does not report this term.)

2. **Accuracy**: we compare the (empirical) classification accuracy between $k$-NN and IOC. We emphasize that since our goal is to accelerate many-class classification in high dimensions, we do not try to improve accuracy of $k$-NN (though we should expect no decline). We consider it acceptable to have both $k$-NN and IOC perform badly on some datasets as long as their performance is comparable.

| **Dataset** | **Train size** | **Test size** | **Num. Dimensions** | **Num. classes** |
|---|---|---|---|---|
| Letter | 16000 | 4000 | 16 | 26 |
| Isolet | 6238 | 1555 | 617 | 26 |
| Cov_type | 58101 | 522911 | 54 | 7 |
| Video | 35049 | 3894 | 62 | 3 |
| Internet_ads | 2952 | 327 | 1555 | 2 |

Table 3: The real-world datasets

For each dataset, we manualy partitioned them into a training set and a test set, and we ran our experiments with $k = 1$, 5, and 9. We report the average predict time per query (see Table 4), as well as the error rates (see Table 5) for all algorithms over these datasets.

| | | **NAIVE** | | **SR-tree** | | MT-DFS | | **IOC** | |
|---|---|---|---|---|---|---|---|---|---|
| | | dists | time (secs) | dists speedup | time speedup | dists speedup | time speedup | dists speedup | time speedup |
| Letter | k=1 | 16000 | 26.79 | n/a | 1.3 | 14.3 | 6.47 | 47.4 | 16.4 |
| | k=5 | — | — | — | 0.88 | 8.9 | 3.97 | 37.9 | 10.4 |
| | k=9 | — | — | — | 0.79 | 7.4 | 3.29 | 31.6 | 7.4 |
| Isolet | k=1 | 6238 | 111.72 | n/a | n/a | 1.1 | 1.1 | 20.4 | 16.6 |
| | k=5 | — | — | — | — | 0.98 | 0.93 | 11.1 | 8.9 |
| | k=9 | — | — | — | — | 0.92 | 0.88 | 8.8 | 6.3 |
| Cov_type | k=1 | 58101 | 40776 | n/a | 7.31 | 36.04 | 26.05 | 79 | 38.43 |
| | k=5 | — | — | — | 4.46 | 19.82 | 14.56 | 33.69 | 25.07 |
| | k=9 | — | — | — | 3.62 | 15.68 | 11.57 | 40.68 | 14.01 |
| Video | k=1 | 35049 | 177.4 | n/a | 3.7 | 28.4 | 19.8 | 554.6 | 663.6 |
| | k=5 | — | — | — | 2.89 | 21.3 | 14.8 | 95.8 | 43.1 |
| | k=9 | — | — | — | 2.63 | 18.8 | 13.2 | 70.4 | 30.2 |
| Internet | k=1 | 2952 | 28.02 | n/a | n/a | 2.4 | 2.4 | 49.4 | 58.4 |
| | k=5 | — | — | — | — | 1.7 | 1.7 | 14.9 | 14.7 |
| | k=9 | — | — | — | — | 1.5 | 1.4 | 10.3 | 9.8 |

Table 4: Speedup for real-world dataset.

| | | **NAIVE** | **SR-tree** | MT-DFS | **IOC** |
|---|---|---|---|---|---|
| Letter | k=1 | 0.043 | 0.043 | 0.043 | 0.112 |
| | k=5 | 0.054 | 0.054 | 0.054 | 0.088 |
| | k=9 | 0.056 | 0.056 | 0.056 | 0.077 |
| Isolet | k=1 | 0.114 | n/a | 0.114 | 0.119 |
| | k=5 | 0.077 | — | 0.077 | 0.085 |
| | k=9 | 0.08 | — | 0.08 | 0.08 |
| Cov_type | k=1 | 0.136 | 0.136 | 0.136 | 0.117 |
| | k=5 | 0.165 | 0.165 | 0.165 | 0.165 |
| | k=9 | 0.176 | 0.176 | 0.176 | 0.172 |
| Video | k=1 | 0.15 | 0.15 | 0.15 | 0.16 |
| | k=5 | 1.127 | 0.127 | 0.127 | 0.127 |
| | k=9 | 0.125 | 0.125 | 0.125 | 0.126 |
| Internet_ads | k=1 | 0.040 | n/a | 0.040 | 0.049 |
| | k=5 | 0.052 | — | 0.052 | 0.052 |
| | k=9 | 0.062 | — | 0.062 | 0.064 |

Table 5: error rates

# 5 Fast $(1+\epsilon)$-NN algorithm

Now, let us step out of classification problems and come back to the general $k$-NN search problem. To our knowledge, exact $k$-NN searching in high-dimension is still a tough problem and no technique can solve it in sublinear time. Our fourth new algorithm is a $(1+\epsilon)$-NN algorithm. It is partially motivated by the locality sensitive hashing (LSH) algorithm. Before we describe our new algorithm, we brifely summarize LSH and discuss its advantages and drawbacks.

## 5.1 LSH

Roughly speaking, a locality sensitive hashing function has the property that if two points are "close," then they hash to same bucket with "high" probability; if they are "far apart," then they hash to same bucket with "low" probability. See [30, 32] for a formal definition and detailed discussion. We stress that there exists a large family of LSH algorithms, and here we only focus on a very simple hash function used in [25]. From a very high level, we interpret the algorithm from a geometric point of view: We call it the "bucketing view." Then we can use a $d$-dimensional cube with its side size $C$ (the maximum distance between points in all $d$-dimensions) to bound all the points. Then the hash function simply partitions the space $[0,C]^d$ into sub-rectangles using $K$ *randomly* generated partition planes. Figure 4 shows an example where $d=2$, and there are four partition planes: $T_1$ and $T_2$ are along the $y$-axis; $T_3$ and $T_4$ are along the $x$-axis. These planes split the space $[0,C]^d$ into $3\times 3 = 9$ "buckets." Since these partition planes are randomly chosen, if two points are "close" (in $L_1$ norm), then the probability that they are mapped into the same bucket is "large." As in the example, point $B$ is closer to point $C$ than to point $A$, and thus it is more likely that $B$ and $C$ are mapped into the same bucket than that $A$ and $B$ are mapped into the

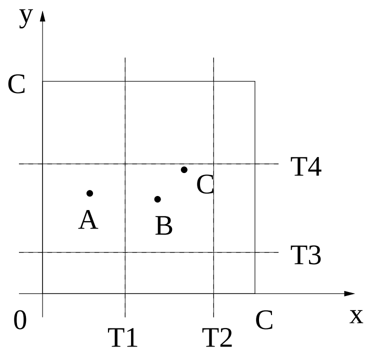


Figure 4: the "bucketing" view of LSH.

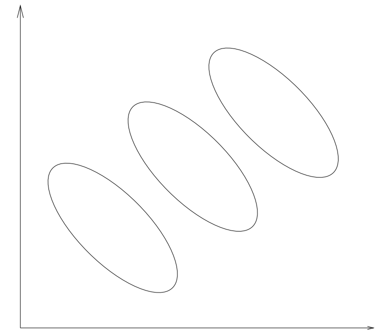

Figure 5: a tilted distribution.

same bucket. Equipped with such a hash function, one simply needs to search within the bucket a query point q falls into in order to find a neighbor that is "close enough" to q.

One very attractive feature of the LSH algorithm is that it enjoys a rigorous, theoretical performance guarantee. Indyk and Motwani [30] prove that even in the worst case, the LSH algorithm finds an $(1+\epsilon)$-NN of any query point with high probability in a reasonable amount of time. It is also demonstrated in [25, 31] that LSH can be useful in practice.

On the other hand, we observe that LSH has its own limitations. First, it was originally designed to work only in the $L_1$ norm, rather than the (more popularly used) $L_2$ norm. Although some later work has extended LSH to other norms, the hash function needs to be changed and become more complicated.

Second, since LSH is designed to guarantee the worst-case behavior, it might not be as efficient on real-world data, which normally exhibit a rather "benign" behavior. For example, the data points typically form clusters, rather than being uniformly distributed in the space. But since the LSH algorithm partitions the space uniformly, it does not exploit the clustering property of the data. In fact, the theoretical correctness of LSH stipulates that one must "guess" a correct asymptotic value of $d(\mathsf{q}, X)$, the nearest distance between q and points in $X$. Therefore, in the worst case, many instances of the LSH algorithm need to be run in order to guarantee correctness.

Finally, the partition planes used in LSH are all aligned with the coordinates. This limitation on the orientation of the partition can hurt efficiency. Imagine a "tilted" distribution where data points form clusters that are distributed along the line $x = y$ (see Figure 5) Naturally, partitions along the anti-diagonal($x = -y$) direction are more efficient than ones along $x$ or $y$ axis. By insisting on the partition direction, LSH may need to adopt a much higher resolution.

### 5.2 Spill-tree

A *spill-tree* is a variant of metric-tree in which the children of a node can "spill over" onto each other, and contain shared datapoints.

The partitioning procedure of a metric-tree (See section 2.2) implies that pointsets of v.lc and v.rc are disjoint: these two sets are separated by the decision boundary $L$. In spill-trees, we change the splitting criteria to allow overlaps between two children. In other words, some datapoints may belong to both v.lc and v.rc.

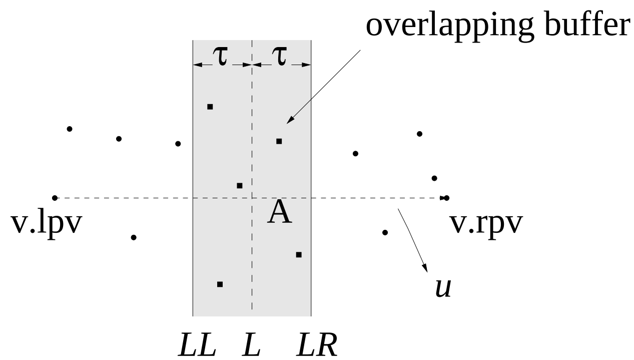


Figure 6: partitioning in a spill-tree.

We first explain how to split an internal node v. See Figure 6 as an example. As in a metric-tree, we first choose two pivots v.lpv and v.rpv, and find the decision boundary $L$ that passes through the mid-point $A$. Next, we define two new separating planes, $LL$ and $LR$, both of which are parallel to $L$ and at distance $\tau$ from $L$. Then, all the points to the *right* of plane $LL$ belong to the child v.rc, and all the points to the *left* of plane $LR$ belong to the child v.lc. Mathematically, we have

$$N(\mathsf{v.lc}) \;=\; \{x \mid x \in N(\mathsf{v}), d(x, LR) + 2\tau > d(x, LL)\} \tag{10}$$

$$N(\mathsf{v.rc}) \;=\; \{x \mid x \in N(\mathsf{v}), d(x, LL) + 2\tau > d(x, LR)\} \tag{11}$$

Notice that points which fall in the region between $LL$ and $LR$ are shared by v.lc and v.rc. We call this region the *overlapping buffer*, and we call $\tau$ the *overlapping size*. For v.lc and v.rc, we can repeat the splitting procedure, until the number of points within a node is less than a specific threshold, at which point we stop.

It may seem strange that we allow overlapping in spill-trees. The overlapping obviously makes both the construction and the MT-DFS less efficient than regular metric-tree. Nonetheless, the advantage of spill-trees over metric-trees becomes clear when we perform the *defeatist search*, an $(1+\epsilon)$-NN search algorithm based on spill-trees. Here we only briefly summarize the search algorithm. Please refer to [39] for more detailed discussion. For simplicity, we continue to use the example shown in Figure 6. As before, the *decision boundary* at node v is plane $L$. If a query q is to the left of $L$, we decide that its nearest neighbor is in v.lc. In this case, we only search points within $N(\mathsf{v.lc})$, i.e., the points to the left of $LR$. Conversely, if q is to the right of $L$, we only search node v.rc, i.e. points to the right of $LL$. Notice that in either case, points in the overlapping buffer are always searched. By introducing this buffer of size $\tau$, we can greatly reduce the probability of making a wrong decision. To see this, suppose that q is to the left of $L$, then the only points eliminated are the one to the right of plane $LR$, all of which are at least distance $\tau$ away from q. So immediately, if $\tau$ is greater than the distance between q and its nearest neighbor, then we *never* make a mistake. In practice, however, $\tau$ can be much smaller and the defeatist search still have a very high accuracy.

### 5.3 Experimental Results

We report our experimental results based on spill-trees search on a variety of real-world datasets, with the number of datapoints ranging from 20,000 to 275,465, and dimensions from 60 to 3,838. The first two datasets are same as the ones used in [25], where it is demonstrated that LSH can have a significant speedup over SR-trees.

Table 6: The datasets

| **Dataset** | **Num.Data** | **Num.Dim** |
|---|---|---|
| Aerial | 275,465 | 60 |
| Corel_small | 20,000 | 64 |
| Corel_uci | 68,040 | 64 |
| Disk | 40,000 | 1024 |
| Galaxy | 40,000 | 3838 |

We also summarize the datasets in Table 5.3.

We perform 10-fold cross-validation on all datasets. We measure the *CPU time* and *accuracy* of each algorithm. To measure accuracy, we use the *effective distance error* [25], which is defined as $E = \frac{1}{Q} \sum_{q \in Q} \left( \frac{d_{alg}}{d^*} - 1 \right)$, where $d_{alg}$ is the distance from a query $\mathtt{q}$ to the NN found by the algorithm, and $d^*$ is the distance from $\mathtt{q}$ to the true NN. The sum is taken over all queries. For the $k$-NN case where $(k > 1)$, we measure separately the distance ratios between the closest points found to the nearest neighbor, the 2nd closest one to the 2nd nearest neighbor and so on, and then take the average. Obviously, for all exact $k$-NN algorithms, $E = 0$, and for all approximate algorithms, $E \geq 0$. First, as a benchmark, we run the Naïve, SR-tree, and the MT-DFS. All of them find exact NN. The results are summarized in Table 7.

Table 7: the CPU time of exact SR-tree, MT-DFS, and Naïve search

| Algorithm (%) | Aerial | Corel_hist ($k=1$) | Corel_hist ($k=10$) | Corel_uci | Disk_trace | Galaxy |
|---|---|---|---|---|---|---|
| Naive | 43620 | 462 | 465 | 5460 | 27050 | 46760 |
| SR-tree | 23450 | 184 | 330 | 3230 | n/a | n/a |
| MT-DFS | 3650 | 58.4 | 91.2 | 791 | 19860 | 6600 |

Then, for approximate NN search, we compare spill-trees with three other algorithms: LSH, MT-DFS and SR-tree. For each algorithm, we measure the CPU time needed for the error $E$ to be 1%, 2%, 5%, 10% and 20%, respectively. Since metric-tree and SR-tree are both designed for exact NN search, we also run them on randomly chosen subsets of the whole dataset to produce approximate answers. We also examine the speedup of spill-trees over other algorithms. In particular, the CPU time and the speedup of spill-trees searches over LSH and metric-tree are summarized in Table 8 and 9 seperately.

Table 8: the CPU time (s) of Sp-tree and its speedup (in parentheses) over LSH

| Error (%) | Aerial | Corel_hist ($k=1$) | Corel_hist ($k=10$) | Corel_uci | Disk_trace | Galaxy |
|---|---|---|---|---|---|---|
| 20 | 33.5 (31) | 1.67 (8.3) | 3.27 (6.3) | 8.7 (8.0) | 13.2 (5.3) | 24.9 (5.5) |
| 10 | 73.2 (27) | 2.79 (9.8) | 5.83 (7.0) | 19.1 (4.9) | 43.1 (2.9) | 44.2 (7.8) |
| 5 | 138 (31) | 4.5 (11) | 9.58 (6.8) | 33.1 (4.8) | 123 (3.7) | 76.9 (11) |
| 2 | 286 (26) | 8 (9.5) | 20.6 (4.2) | 61.9 (4.4) | 502 (2.5) | 110 (14) |
| 1 | 426 (23) | 13.5 (6.4) | 27.9 (4.1) | 105 (4.1) | 1590 (3.2) | 170 (12) |

Table 9: the CPU time speedup of spill-trees over MT-DFS

| Error (%) | Aerial | Corel_hist ($k=1$) | Corel_hist ($k=10$) | Corel_uci | Disk_trace | Galaxy |
|---|---|---|---|---|---|---|
| 20 | 25 | 20 | 16 | 45 | 706 | 25 |
| 10 | 26 | 15.3 | 12 | 30 | 335 | 28 |
| 5 | 21 | 10 | 7.4 | 20 | 131 | 36 |
| 2 | 12 | 7.0 | 4.2 | 12 | 37.2 | 46 |
| 1 | 8.6 | 4.3 | 3.3 | 7.5 | 12.4 | 39 |

# 6 Future Work

In this section, I lay out a plan for the future work.

## 6.1 Self-tuned spill-trees

The performance of a spill-trees algorithm highly depends on the following three factors:

- **Random Projection $d'$:** One crucial pre-processing step of the spill-trees algorithm is called *random projection*. It is a well-known simple technique where one simply projects all the $d$ dimentional datapoints to a random $d'$ dimentional subspace, where $d' << d$. The Johnson-Lindenstrauss Lemma [11] states that one can embed a dataset of $n$ points in a subspace of dimension $O(\log n)$ with little distortion on the pair-wise distances. In fact, a practically useful $d'$ can vary a lot depending on different datasets. One needs to test a sequence of different $d'$ and use cross-validation to determine a best value.

- **Loops $L$:** The random projection used in spill-tree algorithms performs a similar function to the random partition used in LSH. For LSH, it repeats the process $L$ times. Each time, a random hash function is independently generated and all the points within the bucket $\mathsf{q}$ falls into are searched. Similarly, spill-tree algorithms perform independent random projections $L$ times and then do the search on $L$ different projected datasets. By increasing $L$, we can easily boost the probability that the algorithm finds a $(1+\epsilon)$-NN successfully. On the other hand, it makes the algorithm $L$ times slower. So we need to choose a good $L$ for the accuracy and efficiency trade-off.

- **Overlapping Size $\tau$:** As we have described in the previous section, the major difference between spill-tree and metric-tree is that spill-trees contain overlapping buffers. At each level, all the points in the overlap buffer belong to both v.lc and v.rc. By decreasing $\tau$ (the overlap buffer size), one can speedup the spill-trees search. In the extreme case, when $\tau = 0$, the search for each query takes exactly $\log n$ steps, but this gives us very low accuracy. On the other hand, increasing $\tau$ boosts the probability of finding a $(1 + \epsilon)$-NN succesfully, but many more points will be searched.

Currently, we do not have a very good strategy for choosing the set of parameters. We have to tune them manually for each dataset. This is time consuming and, without much experience, it is hard for a new user to use the algorithm. So a self-tuned spill-tree algorithm is needed.

We picked one dataset "Aerial" from section 5.3, and did some initial tests for $\{d', L, \tau\}$. This aims at understanding the behavior of spill-trees.

**Random Projection** The first batch of experiments are aimed at understanding the effectiveness of the random projection, used as a pre-processing step of our algorithm. We fix the overlapping size $\tau$ to be $\infty$, effectively turning the algorithm into the traditional MT-DFS algorithm. We fix the number of loops $L$ to be 1. We vary the *projected dimension* $d'$ from 5 to 60, and then measure the distance error $E$, the CPU time, and the number of distance computations (which accounts for most of the CPU time). The results are shown in Figure 7.

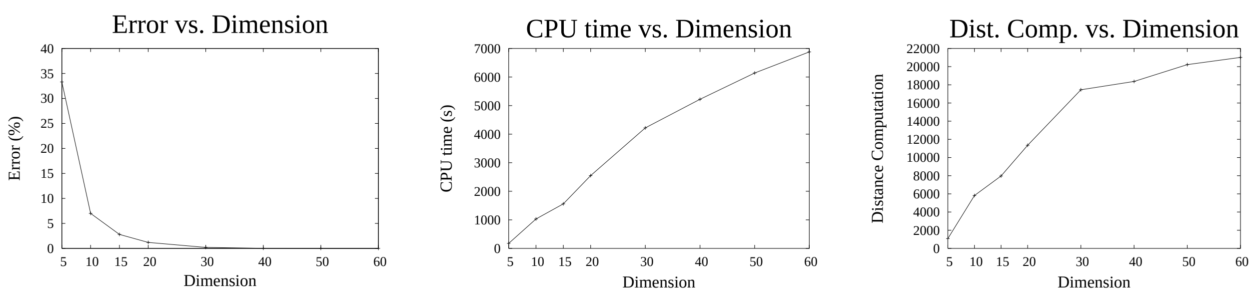


Figure 7: influence of the random projection (Aerial, $d = 60$, $n = 275,476$, $k = 1$).

As expected, the error $E$ is large when the dimension is low: it exceeds 30% when $d = 5$. However, $E$ decreases very fast as $d'$ increases: when $d' = 30$, $E$ is less than 0.2%, which is more than acceptable for most applications. This also suggests that the *intrinsic dimension* of Aerial can be quite small (at least smaller than 30).

The CPU time decreases almost linearly with $d'$. However, the slope becomes slightly steeper when $d < 30$, indicating sublinear dependence of CPU time on $d'$. This is more clearly illustrated in the number of distance computations: a sharp drop after $d < 30$. Interestingly, we observe the same behavior (that the CPU time becomes sublinear and the number of distance computation drops dramatically when $d < 30$) in all five datasets used in section 5.3. This observation suggests that for metric-trees, the "curse of dimensionality" occurs when $d'$ exceeds 30. As a consequence, searching algorithms based on metric-trees (including the spill-trees search) performs much better when $d < 30$. Empirically, for all five datasets we tested, our algorithm achieves optimal performance almost always when $d < 30$.

**Loops** We investigate the influence of the number of loops $L$. We fix $\tau = \infty$ and we test three projected dimensions: $d = 5, 10, 15$. See Figure 8. As expected, in all cases, the CPU time scales

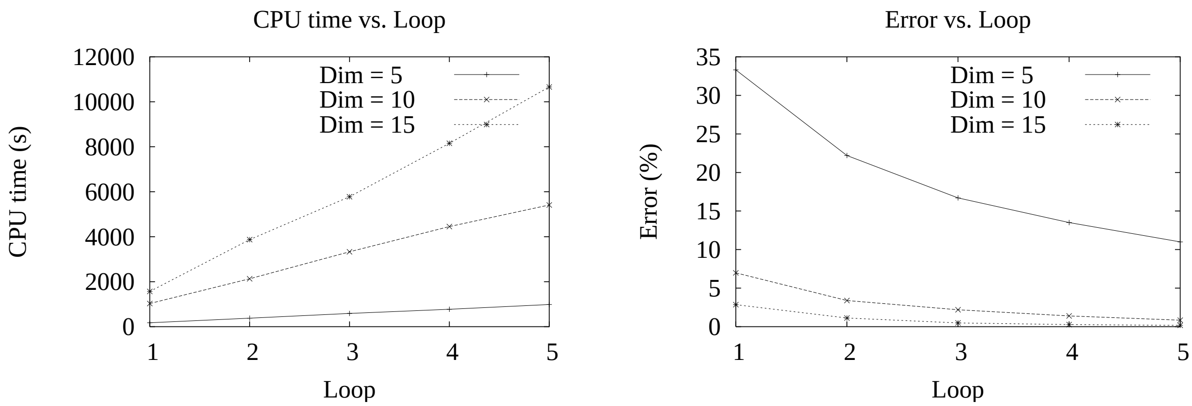


Figure 8: influence of the loop $L$ (Aerial, $d = 60$, $n = 275,476$, $k = 1$).

linearly with $L$, and the distance error $E$ decreases with $L$. However, the rate that $E$ decreases slows down as $L$ increases. Therefore, for the trade-off between $L$ and $d'$, it is typically more economical to have a single loop with a large $d'$ than having a larger $L$ and a smaller $d'$, unless $d'$ is very small. Very roughly speaking, one would expect about the same accuracy if we keep $L \cdot d$ a constant — if we project to a random $d'$-dimensional subspace for $L$ times, it is roughly equivalent to projecting to a random $(L \cdot d)$-dimensional subspace, when $L \cdot d$ is much smaller than the dimension of the original space. Notice that when $d > 30$, the CPU time scales linearly with both $L$ and $d'$, but $E$ decreases much faster with $d'$ increasing than with $L$. However, for very low dimensions ($d < 10$), since CPU time scales sub-linearly with $d'$, it might be worthwhile to choose a small $d'$ and $L > 1$.

**Overlapping Size $\tau$** We test the influence of overlapping size $\tau$. Again, we fix $L = 1$ and test three projected dimension: $d' = 5, 10, 15$. The results are shown in Figure 9.

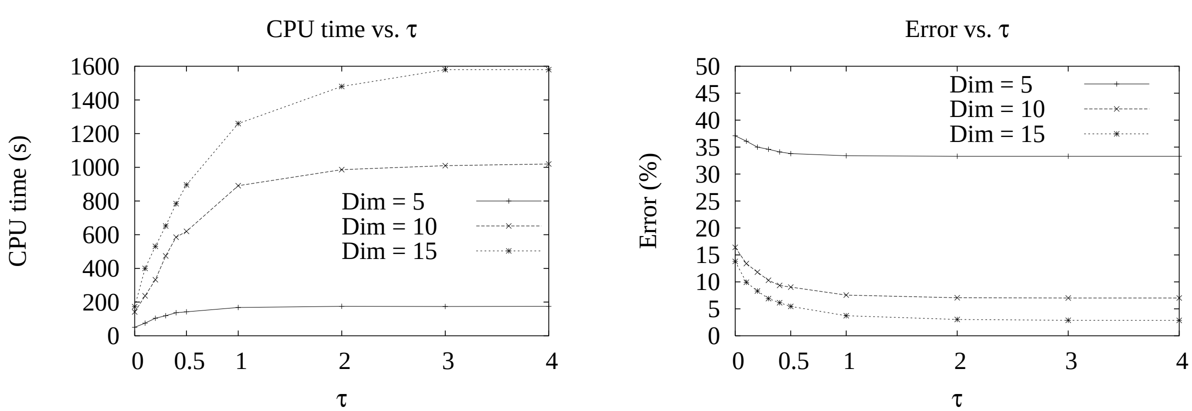


Figure 9: influence of overlapping size $\tau$ (Aerial, $d = 60$, $n = 275,476$, $k = 1$).

As we have mentioned before, both the accuracy and the CPU time increases with $\tau$. When $\tau = 0$, we have pure defeatist search; when $\tau$ is large enough (in this case, when $\tau > 3$), the search becomes MT-DFS, which is slow but perfectly accurate.[2] In the "interesting" range where $0 < \tau < 1$, we see dramatic decrease in CPU time and modest increase in $E$.

These observations tell us there exist some intrinsic patterns of the parameters. By fully exploring them, we should be able to come up with an automatic or semi-automatic parameter tuning system. And moreover, going though this whole procedure may help us gain more insight into the approximate-nearest-neighbor problem.

## 6.2 Theoretical Analysis

As we stated in section 5.1, one very attractive feature of the LSH algorithm is that it enjoys a rigorous, theoretical performance guarantee. Although the spill-tree algorithm has better performance over LSH in many real-world settings, there is as yet no theory about spill-trees that can guarantee its good performance. Unlike LSH, a spill-tree is a more sophisticated data structure, and this makes a theoretical analysis more challenging. There are theoretical analyses on tree-based $(1+\epsilon)$-NN algorithms [2, 3, 37], but these results are not applicable to our case. Above all, it is a challenge to develop a provable statistical guarantee for the spill-tree algorithm, but it will be very desirable to show results of the following flavor.

We have started working on this problem, and we plan to prove the following theorems:

**Conjecture 6.1** *Let $\mathcal{D}$ denote the uniform distribution over the unit ball in $\mathbb{R}^d$. Let $X_1$, $X_2$, ..., $X_n$ be points independently drawn from $\mathcal{D}$. Let $\mathcal{T}$ be the metric-tree built upon $X_1$, $X_2$, ..., $X_n$ using median-point partition, and $\mathsf{q}$ be a query point, also drawn from $\mathcal{D}$. There exist constants $\lambda$, such that for any real number $\delta > 0$, if $n > \frac{2}{\lambda}\left(\frac{3}{\delta}\sqrt{\frac{2d}{\pi}}\right)^d \cdot \log\left(\frac{3}{\delta}\right)$, then with probability at least $1-\delta$, the defeatist search algorithm finds the correct nearest neighbor of $\mathsf{q}$ using $\mathcal{T}$.*

**Conjecture 6.2** *Let $\mathcal{D}$ denote the normal distribution in $\mathbb{R}^d$ with mean $0$ and covariance $I$. Let $X_1$, $X_2$, ..., $X_n$ be points independently drawn from $\mathcal{D}$. Let $\mathcal{T}$ be the spill-trees built upon $X_1$, $X_2$, ..., $X_n$ using median-point partition and buffer size $\tau$. Let $\mathsf{q}$ be a query point, also drawn from $\mathcal{D}$. There exists constants $\lambda$ and $\mu$, such that for any real number $\delta > 0$, if $n > \lambda\left(\frac{1}{\delta}\right)^{\mu d}$, then with probability at least $1-\delta$, the spill-trees algorithm finds the correct nearest neighbor of $\mathsf{q}$ using $\mathcal{T}$.*

Both of the theorems assume certain distributions within the dataset. The first theorem says when the number of points in the datset is large enough, with high probability we can use defeatist search [39] on a metric-tree to find the correct nearest neighbor. In our second theorem, we hope to prove that for spill-trees, by introducing the overlapping buffer, we increase the probability of succesfully finding the true nearest neighbor. In other words, we only need $\lambda\left(\frac{1}{\delta}\right)^{\mu d}$ points to secure the high probability, removing $d$ from the base.

[2] The "residue" errors in Figure 9 in the case $\tau = 4$ come from random projection (c.f. Figure 7).

### 6.3 Dual-Tree

So far, we have been focusing on improving our previous algorithms. Now we try to explore a new direction for $k$-NN algorithms, namely the *dual-tree* algorithm [26].

In our previous algorithms we have concentrated on the training data to speedup $k$-NN search, while leaving the testset intact. Suppose we have $n_q$ query points, and the expected query time for each $\mathsf{q}$ is $c$, then the complexity for querying on the whole testset is $O(cn_q)$. Working on training data alone can only make $c$ as small as possible, but has nothing to do with $n_q$. When $n_q$ is *huge*, even if $c$ can be reduced to $O(\log n)$, the overall prediction time is still $O(n_q \log n)$, which is linear to $n_q$. In fact, we observed that in many $k$-NN classification problems, the test datapoints also form clusters. Points close to each other tend to be from the same class. This property might give us a chance to predict test data in bulk, and our goal is to design an algorithm that can achieve $O(n_q + n)$ time complexity.

From very high-level, we want to build two sets of metric-trees. One set for the training data, and the other for the test data. And then, instead of making predictions on one single query point each time, we want to do predictions on a *node* (i.e. a set of points) of the tree generated from the test data, Now, the first question that occured to us is how to estimate the distance between two nodes (one node from the training set, and one node from the test set). One easy solution is as follows:

Let $\mathsf{v1}$ and $\mathsf{v2}$ be two nodes, and we use $D_{\mathsf{min}}^{(\mathsf{v1},\mathsf{v2})}$ and $D_{\mathsf{max}}^{(\mathsf{v1},\mathsf{v2})}$ to denote the minimum and maximum distances between $\mathsf{v1}$ and $\mathsf{v2}$ respectively. Then we have

$$\begin{aligned} D_{\mathsf{min}}^{(\mathsf{v1},\mathsf{v2})} &= \max(||\mathsf{v1.center} - \mathsf{v2.center}|| - \mathsf{v1.r} - \mathsf{v2.r},\ 0) \\ D_{\mathsf{max}}^{(\mathsf{v1},\mathsf{v2})} &= ||\mathsf{v1.center} - \mathsf{v2.center}|| + \mathsf{v1.r} + \mathsf{v2.r} \end{aligned} \tag{12}$$

As soon as we are able to estimate the distance between two nodes, we can use it to speedup the search. In the example shown in figure 10(a), we have two nodes $\mathsf{v1}$ and $\mathsf{v2}$ from the training set, and one node $\mathsf{vq}$ from the test set. Since $D_{\mathsf{min}}^{(\mathsf{vq},\mathsf{v1})} > D_{\mathsf{max}}^{(\mathsf{vq},\mathsf{v2})}$, we know that none of the points from node $\mathsf{v1}$ could possibly become closer to any query in $\mathsf{vq}$ than points from $\mathsf{v2}$, in other words, the nearest neighbors of all the queries from node $\mathsf{vq}$ come from node $\mathsf{v2}$, so we only need to search node $\mathsf{v2}$, and $\mathsf{v1}$ can be pruned. Imagine if we have 1000 points in $\mathsf{vq}$, this estimation can save us 1000 distance computations.

Although $D_{\mathsf{min}}^{(\mathsf{v1},\mathsf{v2})}$ and $D_{\mathsf{max}}^{(\mathsf{v1},\mathsf{v2})}$ are easy to compute for any two given nodes $\mathsf{v1}$ and $\mathsf{v2}$, unfortunately, the estimation in general is not very good. In the above example, it is easy to notice that $D_{\mathsf{min}}^{(\mathsf{vq},\mathsf{v1})}$ is achieved only when the query point is at $\mathsf{q1}$; and $D_{\mathsf{max}}^{(\mathsf{vq},\mathsf{v2})}$ is achieved only when the query point is at $\mathsf{q2}$. But $\mathsf{q1}$ and $\mathsf{q2}$ are not at the same position, which means these two bounds are impossible to be reached at the same time. Furthermore, it is easy to see that only when the centers of $\mathsf{v1}$ $\mathsf{v2}$ and $\mathsf{vq}$ are in the same line are we able to achieve the two bounds at the same time. Experimental results show that in real-world datasets, this loose estimation generally does not give us many chances to prune nodes.

However, we can further tighten our estimations. We continue to use the same example. To simplify our description, we use $O1 = \mathsf{v1.center}$, $O2 = \mathsf{v2.center}$, $r1 = \mathsf{v1.r}$, and $r2 = \mathsf{v2.r}$. Now for any query $\mathsf{q}$ in node $\mathsf{vq}$, $D_{min}^{(\mathsf{vq},\mathsf{v1})} > D_{max}^{(\mathsf{vq},\mathsf{v2})}$ if and only if

$$\min_{\mathsf{q}\in\mathsf{vq}}(\ (||\mathsf{q} - O1|| - r1)\ -\ (||\mathsf{q} - O2|| + r2)\ ) > 0 \tag{13}$$

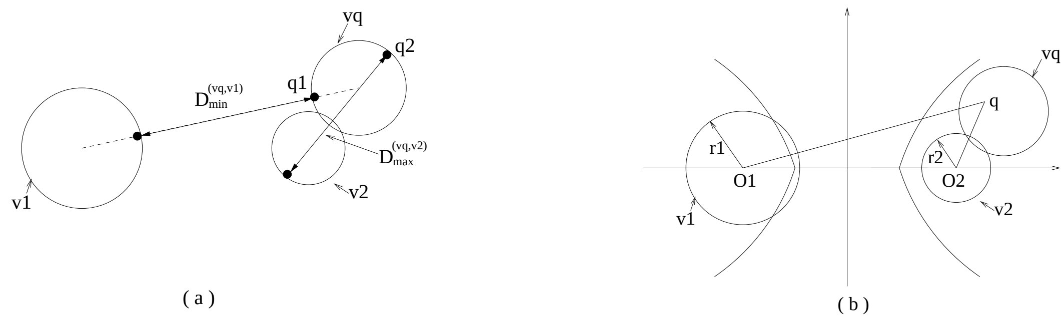


Figure 10: Estimate distance between balls.

Equation 13 can be rewritten as the follows:

$$\min_{\mathsf{q}\in\mathsf{vq}}(||\mathsf{q} - O1|| - ||\mathsf{q} - O2||) > r1 + r2 \tag{14}$$

Notice that all points satisfying $(||\mathsf{q} - O1|| - ||\mathsf{q} - O2||) = r1 + r2$ correspond to one branch of a hyperbola $\mathcal{H}$ determined by $O1$, $O2$ and $r1 + r2$. As in the above example, the points correspond to the right branch of $\mathcal{H}$. Then the problem can be simplified to test weather node $\mathsf{vq}$ intersects with $\mathcal{H}$ or not.

The second problem for the dual-tree algorithm is how to search it. One important problem is when we cannot reach a conclusion according to the bounds estimated from the current nodes, what should we do? There are at least two possible reasons: First, it might be because the estimation so far is not accurate enough (the nodes from the training data are too big). Second, because the points inside the test node are not all from the same class (the size of the test node is too big). So a desicion has to be made whether to split a training node or a testing node, and if we split a training node, which one of all the candidate nodes should be split.

## 6.4 Applications

When $k$-NN algorithms in high-dimension become tractable, we can use them to solve more real-world problems.

### 6.4.1 Video Segmentation

*Video segmentation* is dividing the video stream into a set of meaningful and manageable segments. It is an increasingly important problem. Numerous video retrieval and management tasks rely on accurate segmentation of scene boundaries. Many existing systems compute frame-indexed scores quantifying local novelty within the media stream. The novelty scores are calculated in two steps. First, an affinity or similarity matrix is generated, as in Figure 11. Next, the frame-indexed score is computed by correlating a small kernel function along the main diagonal of the similarity matrix. Typically, detected local maxima in the novelty score are labelled as segment boundaries.

In this proposal, we use a supervised classifier to identify segment boundaries calculated across multiple temporal scales. For this, kernel functions of varying width are used to compute a family

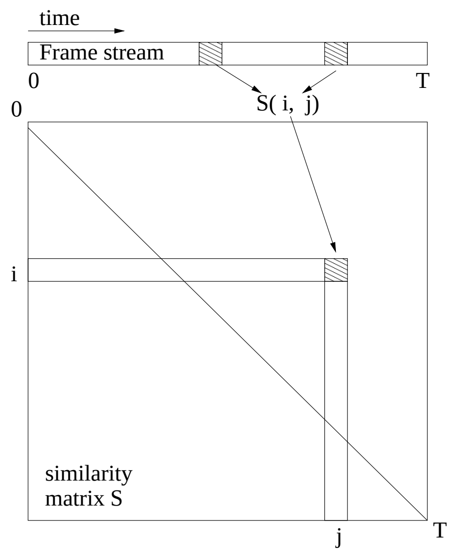


Figure 11: Diagram of the similarity matrix embedding.

of time-indexed novelty scores. Each time sample is then represented by a feature vector comprised of the corresponding novelty scores. These features are in turn classified as either being a boundary or non-boundary. TRECVID is a large-scale metric-based evaluation providing standard data used to compare various systems in video analysis. Next, we directly use the pairwise similarity data as input to train and test the classifier. For this experiment, we vary the specific local set of similarity data again according to proposed kernels. Throughout we will use our fast $k$-NN classifier to label video frames as a cut or gradual transition or non-boundary. The TRECVID dataset involves $10^6$ datapoints, and the feature space easily goes beyond 40, therefore a naïve $k$-NN classifier will be too slow. On the other hand, in order to achieve good performance, feature selection and feature weighting is also very important. Equipped with the new $k$-NN algorithm, we should be able to perform these explorations within a reasonable time. Our goal is to build a general framework using fast $k$-NN algorithm to solve the video segmentation problem.

### 6.4.2 Image retrieval

Nearest-neighbor search is an important component in computer vision applications such as image retrieval. Consider the problem of detecting whether an image is a near duplicate or a sub image of a database of images. One solution is to apply interest points and local descriptors to the images. Conceptually, we break each database image into many patches, and we index each patch independently. We represent the patches using PCA-SIFT, which is a 36-dimensional feature vector. The smaller the distance between two different feature vectors, the more likely that the two corresponding patches came from near duplicate images. On retrieval, we do nearest-neighbor search on patches extracted from interest points in the query image against the database of patches. If enough patches match in appearance and are geometrically consistent, then we can ascertain that the query image is a near duplicate of the database image.

Efficient nearest neighbor search is a critical component of this system. A typical 640x480 natural image contains on the order of 1000 interest points (and patches). If there are 10,000 images in the database, then we will need to index 10,000,000 interest points. Further, each (image) query is really 1000 independent queries on patches.

### 6.4.3 3-D models for computer vision

Approximate-nearest-neighbor algorithms can play an important role in 3D computer vision algorithms for object recognition and object modeling. One popular approach to recognizing objects in 3D data is to use semi-local shape signatures to find similarly-shaped regions between a scene and objects from a model database. Semi-local shape signatures, such as spin-images and 3D shape contexts, encode the local shape of a scene or model region in a feature vector that is invariant to rigid body transforms and robust to clutter and occlusion. Recognition is accomplished by sampling surface points in a query scene and computing the shape signature for each sample location. For each scene signature, a pre-computed model signature database is searched, and the best matching model signature is found. The recognized model is the one with the most matching signatures.

The process of matching scene signatures in the model database is a nearest-neighbor search. Traditional exact nearest-neighbor algorithms are limited by the dimensionality of the signatures. Recently, approximate nearest-neighbor algorithms, such as LSH, have been applied to this problem. These algorithms have enabled near-realtime recognition for databases containing hundreds of models, whereas previous work was limited to tens of models. We are investigating the application of spill-trees to further enhance recognition speed.

Object modeling presents similar challenges to object recognition. With object modeling, the goal is to create a digital model of a real-world object or scene. Three-dimensional views of the object are obtained using, for example, a laser scanner. The views can be from unknown viewpoints, and the task is to assemble the views like a 3D jigsaw puzzle. The problem can be solved by matching pairs of views to find similarly shaped overlapping regions, aligning the pairs with each other in the process. Pairs of views can be connected to form triplets, and so forth, eventually forming a complete 3D model. We are investigating the use of approximate nearest-neighbor methods to efficiently identify potentially overlapping pairs of views. The concept is similar to that of object recognition except that one view serves as the "scene" and the remaining views are the "models".

## 7 Time Table

We set out a fifteen month time table for future work:

- Jan - April, 2005: Working on the dual-tree algorithm, including design and implementation; testing the algorithm on real-world datasets.
- May - August, 2005: Working on improving the spill-tree algorithm and theoretical analysis.
- Sept - Nov, 2005: Working on applications of new $k$-NN algorithms.
- Dec, 2005 - Feb, 2006: Write up the final thesis.

## References


[1] S. Arya and H. A. Fu. Expected-case complexity of approximate nearest neighbor searching. In *Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*, pages 379–388, 2003.

[2] S. Arya, T. Malamatos, and D. M. Mount. Space-efficient approximate voronoi diagrams, 2002.

[3] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *Journal of the ACM*, 45(6):891–923, 1998.

[4] J. Buhler. Efficient large-scale sequence comparison by locality-sensitive hashing. In *Bioinformatics*, volume 17, pages 419–428, 2001.

[5] Claire Cardie and Nicholas Howe. Improving minority class prediction using case-specific feature weights. In *Proc. 14th International Conference on Machine Learning*, pages 57–65. Morgan Kaufmann, 1997.

[6] S. Chaudhuri, R. Motwani, and V. Narasayya. Random sampling for histogram construction: how much is enough? In *Proceedings of ACM SIGMOD*, pages 436–447, 1998.

[7] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An efficient access method for similarity search in metric spaces. In *Proceedings of the 23rd VLDB International Conference*, September 1997.

[8] K. Clarkson. Nearest Neighbor Searching in Metric Spaces: Experimental Results for sb(S). , 2002.

[9] S. Cost and S. Salzberg. A Weighted Nearest Neighbour Algorithm for Learning with Symbolic Features. *Machine Learning*, 10:57–67, 1993.

[10] T. M. Cover and P. E. Hart. Nearest neighbor pattern classification. *IEEE Trans. Information Theory*, IT-13,no.1:21–27, 1967.

[11] S. Dasgupta and A. Gupta. An elementary proof of the Johnson-Lindenstrauss lemma, 1999.

[12] Scott C. Deerwester, Susan T. Dumais, Thomas K. Landauer, George W. Furnas, and Richard A. Harshman. Indexing by latent semantic analysis. *Journal of the American Society of Information Science*, 41(6):391–407, 1990.

[13] K. Deng and A. W. Moore. Multiresolution Instance-based Learning. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence*, pages 1233–1239, San Francisco, 1995. Morgan Kaufmann.

[14] L. Devroye and T. J. Wagner. *Nearest neighbor methods in discrimination*, volume 2. P.R. Krishnaiah and L. N. Kanal, eds., North-Holland, 1982.

[15] A. Djouadi and E. Bouktache. A fast algorithm for the nearest-neighbor classifier. In *IEEE Trans.Pattern Analysis and Machine Intelligence*, volume 19, pages 277–282, 1997.

[16] N. R. Draper and H. Smith. *Applied Regression Analysis, 2nd ed.* John Wiley, New York, 1981.

[17] R. O. Duda and P. E. Hart. *Pattern Classification and Scene Analysis.* John Wiley & Sons, 1973.

[18] Christos Faloutsos, Ron Barber, Myron Flickner, Jim Hafner, Wayne Niblack, Dragutin Petkovic, and William Equitz. Efficient and effective querying by image content. *Journal of Intelligent Information Systems*, 3(3/4):231–262, 1994.

[19] Christos Faloutsos and Douglas W. Oard. A survey of information retrieval and filtering methods. Technical Report CS-TR-3514, 1995.

[20] T. Fawcett and F. J. Provost. Adaptive fraud detection. *Data Mining and Knowledge Discovery*, 1(3):291–316, 1997.

[21] E. Fix and J. L. Hodges. Discriminatory analysis, Nonparametric discrimination: consistency properties. Technical report, USAF School of Aviation Medicine, Randolph Field, Texas, 1951.

[22] E. Fix and J. L. Hodges. Discriminatory analysis: small sample performance. Technical report, USAF School of Aviation Medicine, Randolph Field, Texas, 1952.

[23] M. Flickner, H. Sawhney, W. Niblack, J. Ashley, Q. Huang, B. Dom, M. Gorkani, J. Hafner, D. Lee, D. Petkovic, D. Steele, and P. Yanker. Query by image and video content: the qbic system. *IEEE Computer*, 28:23–32, 1995.

[24] J. H. Friedman, J. L. Bentley, and R. A. Finkel. An algorithm for finding best matches in logarithmic expected time. *ACM Transactions on Mathematical Software*, 3(3):209–226, September 1977.

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

[26] A. Gray and A. W. Moore. N-Body Problems in Statistical Learning. In Todd K. Leen, Thomas G. Dietterich, and Volker Tresp, editors, *Advances in Neural Information Processing Systems 13*. MIT Press, 2001.

[27] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the Third ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*. Assn for Computing Machinery, April 1984.

[28] T. Hastie and R. Tibshirani. Discriminant adaptive nearest neighbor classification. In *IEEE Trans. Pattern Analysis and Machine Intelligence*, volume 18, pages 607–615, June 1996.

[29] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of 30th STOC, 1998*, pages 604–613, 1998.

[30] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *STOC*, pages 604–613, 1998.

[31] P. Indyk and N. Thaper. Fast image retrieval via embeddings. In *the 3rd International Workshop on Statistical and Computational Theories of Vision (SCTV 2003)*, 2003.

[32] Piotr Indyk. *High Dimensional Computational Geometry.* PhD. Thesis, 2000.

[33] IOC. International olympic committee: Candidature acceptance procedure, 1999.

[34] K. V. Ravi Kanth, D. Agrawal, and A. Singh. Dimensionality reduction for similarity searching in dynamic databases. In *Proceedings of SIGMOD*, pages 166–176, 1998.

[35] N. Katayama and S. Satoh. The SR-tree: an index structure for high-dimensional nearest neighbor queries. pages 369–380, 1997.

[36] V. Koivune and S. Kassam. Nearest neighbor filters for multivariate data. In *IEEE Workshop on Nonlinear Signal and Image Processing*, 1995.

[37] E. Kushilevitz, R. Ostrovsky, and Y. Rabani. Efficient search for approximate nearest neighbor in high dimensional spaces. In *Proceedings of the 30th Annual ACM Symposium on Theory of Computing*, pages 614–623, 1998.

[38] T. Liu, A. W. Moore, and A. Gray. Efficient exact k-nn and nonparametric classification in high dimensions. In *Proceedings of Neural Information Processing Systems*, 2003.

[39] T. Liu, A. W. Moore, A. Gray, and K. Yang. An investigation of practical approximate nearest neighbor algorithms. In *Proceedings of Neural Information Processing Systems*, 2004.

[40] T. Liu, K. Yang, and A. Moore. The ioc algorithm: Efficient many-class non-parametric classification for high-dimensional data. In *Proceedings of the conference on Knowledge Discovery in Databases (KDD)*, 2004.

[41] G. S. Manku, S. Rajagopalan, and B. G. Lindsay. Approximate medians and other quantiles in one pass and with limited memory. In *Proceedings of SIGMOD*, pages 426–435, 1998.

[42] Y. Matias, J. S. Vitter, and M. Wang. Wavelet-based histograms for selectivity estimation. In *Proceedings of SIGMOD*, pages 448–459, 1998.

[43] M. C. Monard and G. E. A. P. A. Batista. Learning with skewed class distribution. In J. M. Abe and J. I. da Silva Filho, editors, *Advances in Logic, Artificial Intelligence and Robotics*, pages 173–180, São Paulo, SP, 2002. IOS Press.

[44] A. W. Moore. The Anchors Hierarchy: Using the Triangle Inequality to Survive High-Dimensional Data. In *Twelfth Conference on Uncertainty in Artificial Intelligence*. AAAI Press, 2000.

[45] S. Omachi and H. Aso. A fast algorithm for a k-nn classifier based on branch and bound method and computational quantity estimation. In *In Systems and Computers in Japan, vol.31, no.6, pp.1-9*, 2000.

[46] S. M. Omohundro. Efficient Algorithms with Neural Network Behaviour. *Journal of Complex Systems*, 1(2):273–347, 1987.

[47] S. M. Omohundro. Bumptrees for Efficient Function, Constraint, and Classification Learning. In R. P. Lippmann, J. E. Moody, and D. S. Touretzky, editors, *Advances in Neural Information Processing Systems 3*. Morgan Kaufmann, 1991.

[48] E. P. D. Pednault, B. K. Rosen, and C. Apte. Handling imbalanced data sets in insurance risk modeling, 2000.

[49] D. Pelleg and A. W. Moore. Accelerating Exact $k$-means Algorithms with Geometric Reasoning. In *Proceedings of the Fifth International Conference on Knowledge Discovery and Data Mining*. ACM, 1999.

[50] A. Pentland, R. Picard, and S. Sclaroff. Photobook: Content-based manipulation of image databases, 1994.

[51] F. P. Preparata and M. Shamos. *Computational Geometry*. Springer-Verlag, 1985.

[52] G. Salton and M. McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill Book Company, New York, NY, 1983.

[53] A.W.M. Smeulders, R. Jain, and editors. Image databases and multi-media search. In *Proceedings of the First International Workshop, IDB-MMS'96*, 1996.

[54] S. Stolfo, W. Fan, W. Lee, A. Prodromidis, and P. Chan. Credit card fraud detection using meta-learning: Issues and initial results, 1997.

[55] Y. Hamamoto S. Uchimura and S. Tomita. A bootstrap technique for nearest neighbor classifier design. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 19(1):73–79, 1997.

[56] J. K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 40:175–179, 1991.

[57] K. Woods, K. Bowyer, and W. P. Kegelmeyer Jr. Combination of multiple classifiers using local accuracy estimates. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 19(4):405–410, 1997.

[58] T. Zhang, R. Ramakrishnan, and M. Livny. BIRCH: An Efficient Data Clustering Method for Very Large Databases. In *Proceedings of the Fifteenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems : PODS 1996*. ACM, 1996.