Fast Nonparametric Machine Learning Algorithms for High-dimensional Massive Data and Applications

Ting Liu

CMU-CS-06-124

March 2006

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was sponsored by DARPA EAGLE 7775.1.1150023 and NSF High Dimensional Data Mining 10868.1.1040295.

Keywords: nearest-neighbor, k-nearest-neighbor, approximate-nearest-neighbor, metric-trees, KNS2, KNS3, IOC, spill-tree, LSH

Abstract

Nonparametric methods have become increasingly popular in statistics and probabilistic AI communities. One well-known nonparametric method is "nearest-neighbor". It uses the observations in the training set $\mathcal{T}$ closest in input space to a query $\mathtt{q}$ to form the prediction of $\mathtt{q}$. Specifically, when k of the observations in $\mathcal{T}$ are considered, it is called **k-nearest-neighbor** (or **k-NN**).

Despite its simplicity, k-NN and its variants have been successful in many machine learning problems, including pattern recognition, text categorization, information retrieval, computational statistics, database and data mining. It is also used for estimating sample distributions and Bayes error.

However, k-NN and many related nonparametric methods remain hampered by their computational complexity. Many spatial methods, such as metric-trees, have been proposed to alleviate the computational cost, 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* answers. The premise of this research is that in many cases it is not necessary to insist on the exact answers; instead, determining an approximate answer should be sufficient. In fact, some approximate methods show good performance in a number of applications, and some methods enjoy very good theoretical soundness. However, when facing hundreds or thousands dimensions, many algorithms do not work well in reality.

I propose four new spatial methods for fast k-NN and its variants, namely **KNS2, KNS3, IOC** and **spill-tree**. The first three algorithms are designed to speed up k-NN classification problems, and they all share the same insight that finding the majority class among the k-NN of $\mathtt{q}$ need not to explicitly find those k-nearest-neighbors. Spill-tree is designed for approximate k-NN search. By adapting metric-trees to a more flexible data structure, spill-tree is able to adapt to the distribution of data and it scales well even for huge high-dimensional data sets. Significant efficiency improvement has been observed comparing to LSH (localify sensitive hashing), the state of art approximate k-NN algorithm. We applied spill-tree to three real-world applications: shot video segmentation, drug activity detection and image clustering, which I will explain in the thesis.

Acknowledgments

I owe a lot to my advisor, Andrew Moore, who first introduced me to the fascinating area of Machine Learning, which I enjoy greatly ever since. Andrew taught me how to do research: how to extract a problem from the real-world, how to understand a problem from a fundamental level, how to refine the solutions relentlessly, and how to strike a balance between being intuitive and being rigorous. An excellent speaker himself, Andrew also taught me how to give good talks, which I found extremely useful. Andrew is more than just an academic advisor to me, his nice personality, optimistic attitude and his encouragement to me whenever I met obstacles all become important factors that lead me to my finishing of my Ph.D. degree.

I started to be interested in working on nearest-neighbor algorithms when I worked on a phamathutical project, but the later applications I worked on are all computer vision related problems, which become a heavy part of my thesis. My knowledge of computer vision would be impossible without Martial Hebert, who taught a course at CMU. Martial is a very good teacher and very dedicated to the students. I got many ideas and suggestions from him. It is an hornor to have him in my thesis committee. I met Trevor Darrel in NIPS 2003. It was the first time I went to a conference. Trevor organized the workshop on k-nearest-neighbor methods with its applications on Computer Vision. I learned a lot from that workshop, and it broadened my eyes in this area. It is only appropriate to have Trevor in my thesis committee, and I am very happy that he agrees. Jeff Schneider is an important member of Auton lab, ever since I came to the group, Jeff has been offering a great many help to me, from coding to research ideas, thanks to Jeff for agreeing to become a commeettee member.

Paul Komarek is the first person in my group who I collaborated with. He helped me in every aspect, and whenever I asked him a question, he always stopped his own work and helped me. Alex Gray is unique in his own way. He has very broad knowledge and experience of machine learning and very good vision to look through a seemingly messy and complicated problem. Every time I talked to him, I can always got a lot of inspirations, and he helped me smoothed out the way towards conquering hard problems.

I thank all my collaborators, both inside CMU and out, for the advices and ideas they contributed selflessly: Ke Yang, Yanjun Qi, Alex Hauptman, Daniel Huber, Andrea Frome (Berkeley), Ke Yan and Chuang Lin (Tsinghua).

I remember that in year 2001, I spent a lot of time deciding which graduate school to attend, and I finally chose CMU. Now, five years after, I feel only how lucky I was in making the right choice. Indeed, CMU has the ideal atmosphere for me — nice and friendly people, open and relaxing environment, tons of free food... People say CMU students are so happy that they don't want to graduate — well, they are right to some extent. People do graduate, but yes, most of us are very, very happy here. I will surely miss this place, as well as its people: Sharon Burks, Jeanette Wing, Catherine Copetas, Kristen Schrauder, Jean Harpley, Minglong Shao, Changhao Jiang (at UIUC), Jeanie Komarek, Owen Cheng, Muralidhar Talupur, Weng-Keen Wong (at OSU), Daniel Neill, Anna Goldenberg, Purnamrita Sarkar, Jiayu Pan, Kristen Stubbs, Kang Li, Alina Oprea, Xinghua An (at Google), Yan Liu, Rong Yan, Yiheng Li, Bo Pang (at Cornell), Zhong Xiu and Hua Zhong, to name a few. They are the ones responsible for my personal happiness at CMU.

I spent one summer at FXPAL and one summer at Google Inc. as an intern. These experiences are eye-openers to me, offering to me a unique opportunity to see how people outside universities do "real" work and do research. I enjoy all my intern experiences and I thank my co-workers there that make it possible: Matthew Cooper, Eleanor Rieffle, Henry Rowley and Chuck Rosenberg.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

In this chapter, we introduce nonparametric methods and their problems. In particular, we focus on one important nonparametric method — the k-NN algorithm and its related work.

1.1 Nonparametric Methods

A *statistical model* Φ can be viewed as a set of distributions. A *parametric model* (Wasserman [2004]) refers to a set Φ that can be described using a finite number of parameters. For instance, if we draw data from a Normal distribution, we get a two-parameter model, which can be formalized as

$$\Phi = \{f(x;\mu,\sigma) = \frac{1}{\sigma\sqrt{2\pi}}\exp\{-\frac{1}{2\sigma^2}(x-\mu)^2\},\ \mu \in \mathcal{R},\ \sigma > 0\}. \tag{1.1}$$

We have written the density as $f(x;\mu,\sigma)$ to show that x is a value of the random variable whereas μ and σ are parameters. In general, a parametric model takes the form

$$\Phi = \{f(x,\theta) : \theta \in \Theta\} \tag{1.2}$$

where θ is the parameter (or a set of parameters) in the parameter space Θ.

A *nonparametric model* (Wasserman [2004]) is a set Φ that cannot be parameterized by a finite number of parameters. For instance, $\Phi = \{all\ CDF's\}$ is nonparametric.

When we know the underlying distribution of the data set, a *parametric method* can be used to estimate the parameters of the model, and further inference (prediction, regression, classification, etc.) can be done when the model is built. The advantage of a parametric method is obvious: after the training process during which the parameters are learned, further computations become very efficient, and since all future computations are made based on the learned model, there is no need to store previous data, so a parametric method is also storage efficient. However, utilizing a parametric model has its own risk. Sometimes a bad model (due to lack of data or over-fitting) could cost huge mistakes in future prediction. Furthermore, for a large fraction of the real world problems, it is usually very hard to use a simple parametric model to represent the data. To name a few reasons: the data could be very noisy; the underlying distribution could be too complicated to represent; there are not enough well studied parametric models at hand to fit the data, etc. A parametric method may fail for these cases.

A *nonparametric method*, on the other hand does not depend on knowing the form of the distribution from which the data are drawing. The inference can be made directly from the observed data. These properties alleviate the complexity for building the models, and make nonparametric methods very attractive. In fact, they have become more and more popular in numerous problems in a variety of areas. Despite their good performance, nonparametric methods, however, have not been employed as widely in applications where very large sets of high-dimensional data are involved. One main reason is the computational complexity of distance computation in high-dimensional spaces, often seen as prohibitive. The main focus of this work is on advances in computational geometry and machine learning that may alleviate these problems. We also show experimental results of a variety of applications ranging from computer vision to pharmaceutical drug discovery in which dramatic efficiency improvements have been observed. To be more specific, throughout this work, we only focus on a particular nonparametric model, namely k-NN model and its variants. The reason will become clear in section 1.2.

1.2 Nearest-neighbor Problem

The first formulation of a rule of the *nearest-neighbor* type was proposed in 1951 by Fix and Hodges (Fix and Hodges [1951, 1952]), 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 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 (Cover and Hart [1967]). This result makes k-NN methods very appealing and widely accepted. Some examples of the applications of k-NN are: pattern recognition (Duda and Hart [1973], Draper and Smith [1981]), text categorization (Hamamoto et al. [1997]), database and data mining (Guttman

[1984], Hastie and Tibshirani [1996]), information retrieval (Deerwester et al. [1990], Faloutsos and Oard [1995], Salton and McGill [1983]), image and multimedia search (Faloutsos et al. [1994], Pentland et al. [1994], Flickner et al. [1995], Smeulders and Jain [eds]), machine learning (Cost and Salzberg [1993a]), and statistics and data analysis (Devroye and Wagner [1982], Koivune and Kassam [1995]).

The *nearest-neighbor method* is depicted as follows. Assume the data set 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 q be a query. The nearest-neighbor algorithm is to find the closest point in $\mathcal{T}$ to q. It can be extended to the k-nearest-neighbor (k-NN) case, in which k closest points $\{w_1, w_2, \ldots, w_k\}$ 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 label $y_i \in \mathcal{L}$, where we define $m = |\mathcal{L}|$ to be the total number of classes. In particular, when $m = 2$, we call the problem *binary* k-NN classification. For k-NN classification, one first finds the k-NN of q from $\mathcal{T}$, denoted by $\mathcal{N}$, and then labels 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 q.

Many researchers have done work to make variations of k-NN algorithms improve their classification accuracy for different applications (Hamamoto et al. [1997], Hastie and Tibshirani [1996]) or to combine it with other methods (Woods et al. [1997]). We will not go into detail on all these variants of k-NN methods, since it is not the focus of this work.

The k-NN model has many attractive properties. First of all, it is a useful sanity check or baseline against which to check more sophisticated algorithms *provided* k-NN is tractable. It is often the first line of attack in a new complex problem due to its simplicity and flexibility. The user need only provide a sensible distance metric. The method is easy to interpret once this distance metric is understood. Secondly, it has theoretical soundness. A famous result of Cover and Hart (1967) show that asymptotically, the error rate of the 1-nearest-neighbor classifier is never more than twice the Bayes rate. This compelling property explains its surprisingly good performance in practice in many cases. For these reasons and others, k-NN is still very popular and we have mentioned many of its application areas. Furthermore, we believe making k-NN tractable is the first step towards solving the computational efficiency problem for other more complicated nonparametric models, and similar insights can be shared for other methods, such as nonparametric kernel density estimation and the prediction phase of support vector machine.

Here we list some major successes for k-NN over the years.

- k-NN has been widely used as a basic algorithm for text categorization (Aas and Eikvil [1999]) (Han et al. [2001]). In (Bergo [2001]), it shows that although not perfect, k-NN is the best overall performing system on diverse sets (Yang [1999]).

- In bioinformatics and drug discovery area, k-NN is one of the favorate approaches. In (Y et al. [2001]), it is used to classify tumor types for cancer diagnosis. In (Yao and Ruzzo [2006]), k-NN is used as a general framework for gene function prediction. (Komarek [2004]) used k-NN as a comparison approach in drug activity discovery.

- In computer vision area, k-NN turns out to be quite successful, and it has numerous applications, such as pose estimation (Shakhnarovich et al. [2006]), contour matching and scene recognition (Grauman and Darrell [2006]), image clustering (Shimshoni et al. [2006]) and object recognition (Frome and Malik [2006]).

- In learning and robotics, k-NN plays an important role. In (Aha et al. [1994]), k-NN is used for dynamic control tasks. In (Cost and Salzberg [1993b]), a weighted k-NN algorithm is used for learning with symbolic features. Further more, k-NN can be used to create adaptive on-line learning system (Shih and Lee) as well.

1.3 Speeding up Nearest-neighbor

As described earlier, k-NN is expensive. 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 q, and thus the computational time is $O(dn)$ per query. When both n and d are large, 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 (Faloutsos et al. [1994], Pentland et al. [1994]). In information retrieval for text documents, vector-space representations involve several thousand 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 sub-linear running time is needed.

Several effective methods exist for this problem when the dimension d is small, such as Voronoi diagrams (Preparata and Shamos [1985]), which work for 1 or 2 dimensions. Djouadi and Bouktache (Djouadi and Bouktache [1997]) 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.e., up to the 10's), such as $kd-trees$ (Friedman et al. [1977], Preparata and Shamos [1985]), R-tree (Guttman [1984]), and metric-trees (Omohundro [1991], Uhlmann [1991], Ciaccia et al. [1997]). Among these tree structures, metric-trees, or ball-trees (Uhlmann [1991]) so far represent the practical state of the art for achieving efficiency in the largest dimensionalities possible (Moore [2000], Clarkson [To appear]) 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 Omohundro [1987], Deng and Moore [1995], Zhang et al. [1996], Pelleg and Moore [1999], Gray and Moore [2001]. 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.

So far, we have considered the generic k-NN problem, not that of k-NN classification specifically. In fact, many algorithms designed specifically for k-NN classification have been proposed, virtually most of them pursuing the idea of reducing the number of training points. A few training set reduction methods have the capability of yielding exact classifications, while others yielding only approximate classifications (Fisher and Patrick [1970], Gates [1972], Chang [1974], Ritter et al. [1975], Sethi [1981], Palau and Snapp [1998]). Lee and Chae [1998] achieves exact classifications, but only obtained a speedup over naive search of about 1.7. Both approximate and exact methods are described in Djouadi and Bouktache [1997], however a speedup of only about a factor of two over naive search was reported for the exact case. It is in fact common among the results reported for training set reduction methods that only 40-60% of the training points can be discarded, *i.e.* no important speedups are possible with these approaches. For the approximate classifications, such as in (Hart [1968]), although the run time is reduced, so is the classification accuracy. Zhang and Srihari [2004] pursued a combination of training set reduction and a tree data structure, but its speedup over naïve search is limited.

1.4 Approximate Nearest-neighbor Searching

Since the k-NN problem is very difficult to solve exactly in high dimensions, another avenue of research focuses on investigating the approximate-nearest-neighbor problem. The premise of this research 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 (Chaudhuri et al. [1998]) median approximation (Manku et al. [1998]), using wavelets for selectivity estimation (Matias et al. [1998]) and approximate SVD (Kanth et al. [1998]).

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 $(1+\varepsilon)$-NN of q if $||w^\epsilon - \mathsf{q}|| \leq (1+\epsilon)||w - \mathsf{q}||$, where w is the true nearest-neighbor of q. For k-NN 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 (Arya et al. [2002, 1998], Kushilevitz et al. [1998]). For instance, Arya and Fu [2003] 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 [1998] proposed a new $(1+\varepsilon)$-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 5.1 for a detailed description). LSH turns out to be very successful both theoretically (Indyk and Motwani [1998]) and practically (Gionis et al. [1999], Buhler [2001]). 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.

1.5 A Brief Summary of Results

This work shows 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 (Omohundro [1991], Uhlmann [1991], Ciaccia et al. [1997]). We observe that metric-trees have the advantage in their flexibility, and thus it is easy to capture the intrinsic distribution of the data set. Also, the triangle inequality can be used for metric-trees search to prune away nodes which are far away from the query. The problem with metric-trees is that, in general, like all other spatial tree structures, it is hurt by the "curse of dimensionality". In other words, with d increasing, the speedup achieved by metric-trees deminishes, and in the worst case, metric-trees based k-NN search can be even slower than linear scan.

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

1.5.1 KNS2 and KNS3

We call the first two algorithms KNS2 and KNS3 (Liu et al. [2003]). 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 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 q?
(b) How many of the k-nearest-neighbors of q are from the positive class?
(c) Are at least 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 t is set, it is sufficient to just answer the much simpler question (c). The triangle inequality of metric-trees 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 [2000] 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 3.1, we address Omachi's method in more detail. We test our algorithms on 17 synthetic and real-world data sets, with dimensions ranging from 2 to 1.1×10^6 and number of data points ranging from 10^4 to 4.9×10^5. We observed up to a 100-fold speedup as compared to highly optimized traditional metric-trees-based k-NN.

1.5.2 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 (Liu et al. [2004b]). IOC assumes a slightly different processing of the data points 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, but which 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 neighbors – not the neighbors themselves. In section 4.1, 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 4.1) 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 data sets of up to 5.8×10^3 records and 1.5×10^3 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-trees search.

1.5.3 Spill-tree

In this algorithm (Liu et al. [2004a]), we introduce a new data structure, namely spill-tree. It is a variant of metric-trees. For metric-trees, the point sets contained by two children nodes must be disjoint, while a spill-tree does not have this restriction. Furthermore, we deliberately allow certain data points in a spill-tree to be shared between children nodes. We show a new approximate k-NN search algorithm based on spill-tree, the algorithm exploits a random-projection-based approximation, and it achieves very good efficiency and scales very well. In section 5.2, we provide a detailed empirical evaluation on five large, high-dimensional data sets and compare spill-tree with three other popular k-NN algorithms. We observe consistent speedup of spill-tree over all the other algorithms.

1.5.4 Applications

We use our new algorithms as tools to conquer hard real-world problems. Our first target is video segmentation, which is a basic problem in multimedia analysis. k-NN can not be used directly to this problem, and the data set is usually very large, on which k-NN is usually out of question. We first transformed the video segmentation problem to a classification problem, and then we applied KNS2 to it. We observed great performance and efficiency in the same time. Our second problem is on classification for drug screening. Although k-NN is not the best classifier for this problem, but without much tuning, its performance is comparable with many more sophisticated algorithms. Our last application solves a more complicated problem. We extended the spill-tree algorithm to a parallel version, and it is applied to an image clustering problem. The algorithm succesfully classified 1.5 billion images. All detailed descriptions of the applications can be found in section 6.

1.5.5 Summary

One key ingredient of k-NN is the distance metric. In fact, different distance metric could affect the searching result and classification accuracy quite a lot. However, since this is not the main focus of this work, for the rest of the paper, we use the most common distance metric, i.e., $L2$ distance. Notice that this does not mean the algorithms can only be used for $L2$ distance. In fact, all four algorithms can work on distance metric that satisfies triangle inequality, and the implementation can be easily adapted from $L2$. The detailed descriptions of our new algorithms can be found in the following three papers: KNS2 and KNS3 is introduced in Liu et al. [2003], IOC comes from Liu et al. [2004b], and spill-tree is from Liu et al. [2004a].

Chapter 2

Metric-trees

We review the metric-tree data structure, which is the basic data structure our algorithms rely on. For simplicity, throughout this work, we use $L2$ as the distance metric. In fact, the metric-tree structure is general enough to support any metrics that satisfy the triangle inequality.

2.1 Properties

A *metric-tree* (Omohundro [1991], Uhlmann [1991], Ciaccia et al. [1997], Moore [2000]) is a data structure that supports efficient k-NN 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 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 v, and use $\mathsf{v.lc}$ and $\mathsf{v.rc}$ to denote the left child and the right child of node v, then we have

$$\begin{aligned} N(\mathsf{v}) &= N(\mathsf{v.lc}) \cup N(\mathsf{v.rc}) && (2.1) \\ \emptyset &= N(\mathsf{v.lc}) \cap N(\mathsf{v.rc}) && (2.2) \end{aligned}$$

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 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 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 v.

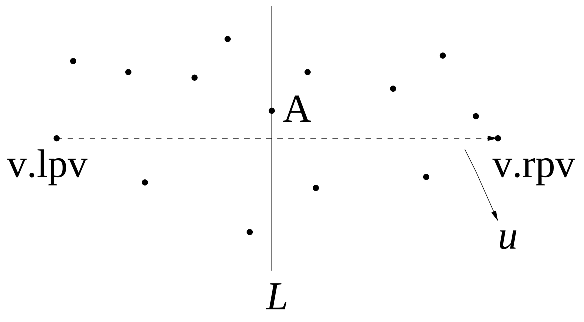


Figure 2.1: Partitioning in a metric-tree.

There are many ways to partition v, and we focus on a particular strategy. We first project all the points down to the vector $\vec{u} = \mathsf{v}.\vec{\mathsf{rpv}} - \mathsf{v}.\vec{\mathsf{lpv}}$, 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 Fig. (2.1)). By using a median point to split the data points, 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 for the following reasons. First of all, it is more efficient to compute, and in practice, we can still have a metric-tree of depth $O(\log n)$; also it can lead to smaller nodes. For instance, suppose there are two well-separated clusters where one cluster has 5 times as many points as the other. A median partition gives us one large child node and one small child node, whereas a mid-point partition returns two well seperated small children nodes.
Each node v also has a hyper-sphere $\mathbb{B}$, such that all points represented by v fall in the ball centered at $\mathsf{v.center}$ with radius $\mathsf{v.r}$, *i.e.* we have

$$N(\mathsf{v}) \subseteq \mathbb{B}(\mathsf{v.center}, \mathsf{v.r}). \tag{2.3}$$

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

Depending on the implementation, v.center may be one of the data points in $N(\mathsf{v})$, or it may be the centroid of $N(\mathsf{v})$. v.r is chosen to be the minimal value satisfying (2.3). 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. Fig. (2.2) shows an example of a 2-dimensional data set and the first few levels of a metric-tree.

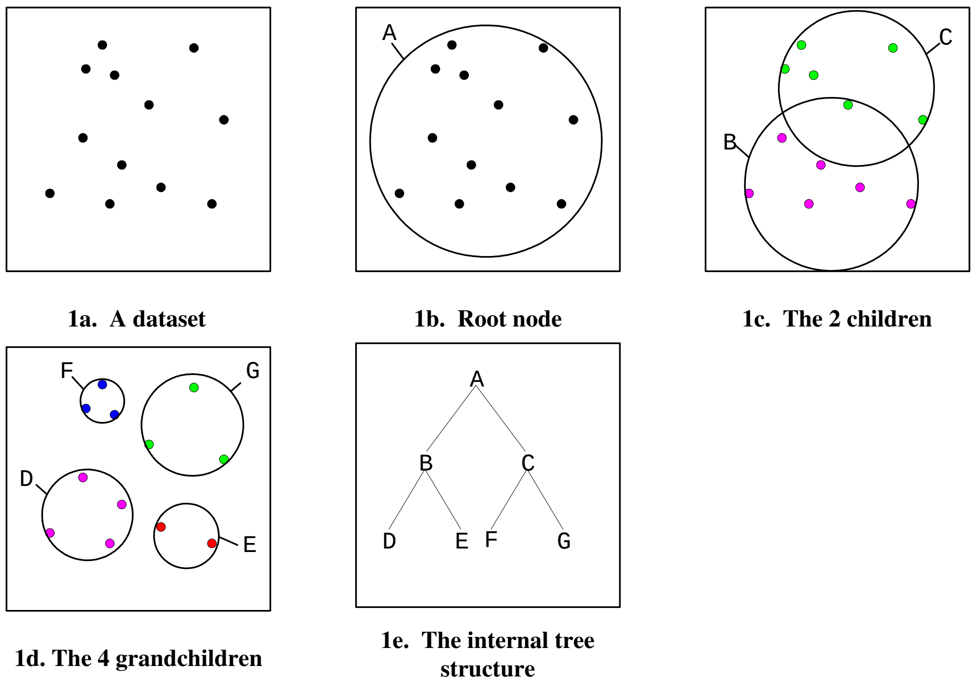


Figure 2.2: An example of metric-tree structure

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 q is on the left of L, then v.lc is searched first, otherwise, 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 q and x by r. If DFS is about to exploit a node v, but discovers that no member of v can be within distance r of 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 data set 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 data points. This flexibility makes metric-trees more efficient than *kd*-trees.

In practice, the MT-DFS algorithm is very efficient for k-NN search, and particularly when the dimensionality of a data set 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 data sets 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.

Chapter 3

Fast K-Nearest-Neighbor Classification

In this section, we describe our first two new algorithms KNS2 and KNS3. Both algorithms are based on the 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 data sets, it is far more common for the result of an experiment to be negative than positive. In detection of fraud telephone calls (Fawcett and Provost [1997]) or credit card transactions (Stolfo et al. [1997]), the number of legitimate transactions is far more common than fraudulent ones. In insurance risk modeling (Pednault et al. [2000]), a very small percentage of the policy holders 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 (Cardie and Howe [1997], Monard and Batista [2002]). 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 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 at 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 follows:

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

- Step 1 — **" Find positive"**: Find the k nearest positive class neighbors of q (and their distances to q) using MT-DFS.
- Step 2 — **"Insert negative"**: Do sufficient search on the negative tree to prove that the number of positive data points among the k-NN 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.**

  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 q, sorted in increasing order of distance. For notational convenience we will also write $\mathsf{Dists}_0 = 0$ and $\mathsf{Dists}_{k+1} = \infty$.

- $N(\mathsf{v})$**.**

  $N(\mathsf{v})$ is the set of points in a negative node v visited so far in the search.

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

  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 $N(\mathsf{v})$ if and only if

  1. $\forall i = 0, 1, \ldots, n$,

     $$C_i = \mid \mathsf{v} \cap \{x : \mid x - q \mid < \mathsf{Dists}_i\} \mid \tag{3.1}$$

     Intuitively C_i is the number of points in v whose distances to q are closer than Dists_i. In other words, C_i is the number of negative points in v closer than the i^{th} positive neighbor to q.

2.
$$\begin{cases} C_i + i \le k & \forall i = 0, 1, \ldots, n-1 \\ C_i + i > k & i = n \end{cases} \tag{3.2}$$

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 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 q to a current node v.

$$\text{Let } D^{\mathsf{v}}_{\mathsf{minp}} = \begin{cases} \max(||\mathsf{q} - \mathsf{v.center}|| - \mathsf{v.r}, D^{\mathsf{v.parent}}_{\mathsf{minp}}) & \text{if } \mathsf{v} \neq root \\ ||\mathsf{q} - \mathsf{v.center}|| - \mathsf{v.r} & \text{if } \mathsf{v} == root \end{cases} \tag{3.3}$$

$$\text{Let } D^{\mathsf{v}}_{\mathsf{maxp}} = \begin{cases} \min(||\mathsf{q} - \mathsf{v.center}|| + \mathsf{v.r}, D^{\mathsf{v.parent}}_{\mathsf{maxp}}) & \text{if } \mathsf{v} \neq root \\ ||\mathsf{q} - \mathsf{v.center}|| + \mathsf{v.r} & \text{if } \mathsf{v} == root \end{cases} \tag{3.4}$$

$D^{\mathsf{v}}_{\mathsf{minp}}$ 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.

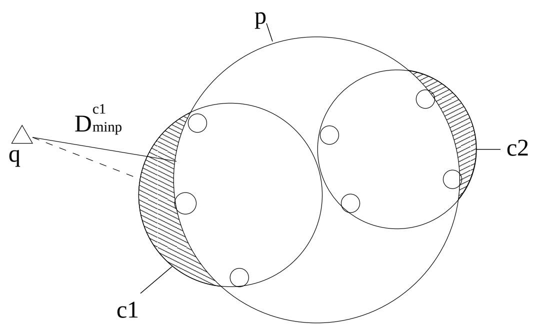


Figure 3.1: An example to illustrate how to compute $D^{\mathsf{v}}_{\mathsf{minp}}$

Fig. (3.1) gives a good example. There are 3 nodes p, $c1$ and $c2$, where $c1$ and $c2$ are p's children, and 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 v, if there exists an i ($i \in [1, n]$), such that $Dists_{i-1} \leq D^{\mathsf{v}}_{\mathsf{maxp}} < Dists_i$, then for $\forall x \in N(\mathsf{v})$, $Dists_{i-1} \leq | x - \mathsf{q} | < Dists_i$. According to the definition of C, we can add $| N(\mathsf{v}) |$ 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.

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 $N(\mathsf{v})$, 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 $\mathsf{v} \cup N(\mathsf{v})$. 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** $NegCount$ ($n^{in}, C^{in}, \mathsf{v}, j_{parent}$, 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 $C^{out}_{index} := C^{out}_{index} + | N(\mathsf{v}) |$     /* We only update the count less than $j_{parent}$
  Update $n^{out}$, such that                              to avoid counting twice for each point*/
    $C^{out}_{n^{out}-1} + (n^{out} - 1) \leq k$, $C^{out}_{n^{out}} + n^{out} > k$

  Set $Dists_{n^{out}}$ := $\infty$
  (1) if ($n^{out} == 1$) return(1, $C^{out}$)        /* At least $k$ negative points closer to q
                                                          than the closest positive one */
  (2) if ($i == j$) return($n^{out}, C^{out}$)         /* v is located between two adjacent
                                                          positive points, no need to split */
  (3) if (v is a leaf)
        Forall $x \in N(\mathsf{v})$, compute $| x - \mathsf{q} |$
        Update and return $(n^{out}, C^{out})$
  (4) else
        $\mathsf{v}_1$  :=  child of v closest to q
        $\mathsf{v}_2$  :=  child of v furthest from q
        $(n^{temp}, C^{temp})$  :=  NegCount($n^{in}, C^{in}, \mathsf{v}_1, j$, Dists)
        if ($n^{temp}$ == 1) return (1, $C^{out}$)
        else
          $(n^{out}, C^{out})$  :=  $NegCount(n^{temp}, C^{temp}, \mathsf{v}_2, j,$ Dists)
end
```

We can stop the procedure when n^{out} becomes 1 (which means all the k-nearest-neighbors of 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 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 q, we can stop. Notice that the k negative points we found are not necessarily the exact k-nearest-neighbors to 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 data sets. 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.

Omachi and Aso [2000] proposed a k-NN method based on branch and bound. For simplicity, we call their algorithm KNSV. KNSV shares a similar idea of KNS2. For the binary class case, it also builds two trees, one for each class. For consistency, let's still call them *Postree* and *Negtree*. KNSV first searches the tree whose center of gravity is closer to q. Without lose of generality, we assume *Negtree* is closer, so KNSV will search *Negtree* first. Instead of fully exploring the tree, it does a greedy depth first search only to find k candidate points. Then KNSV moves on to search *Postree*. The search is the same as conventional metric-tree search (KNS1), except that it uses the k^{th} candidate negative point to bound the distance. After the search of *Postree* is done. KNSV counts how many of the k-nearest-neighbors so far are from the negative class. If the number is more than $k/2$, the algorithm stops. Otherwise, KNSV will go back to search *Negtree* for the second time, this time fully search the tree. KNSV has advantages and disadvantages. The first advantage is that it is simple, and thus it is easy to extend to many-class case. Also if the first guess of KNSV is correct and the k candidate points are good enough to prune away many nodes, it will be faster than conventional metric-tree search. But there are some obvious drawbacks of the algorithm. First, the guess of the winner class is only based on which classes' center of gravity is the closest to q. Notice that this is a pure heuristic, and the probability of making a mistake is high. Second, using a greedy search to find the k candidate nearest neighbors has a high risk, since these candidates might not even be close to the true nearest neighbors. In that case, the chance for pruning away nodes from other class becomes much smaller. We can imagine that in many situations, KNSV will end up searching the first tree for yet another time. Finally, we want to point out that KNSV claims it can perform well for many-class nearest neighbors, but this is based on the assumption that the winner class contains at least $k/2$ points within the nearest neighbors, which is often not true for the many-class case. Comparing to KNSV, KNS2's advantages are (i) it uses the skewedness property of a data set, which can be robustly detected before the search. And (ii) more careful design gives KNS2 more chance to speedup the search.

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:

$$D_{\mathsf{t}}^{\mathsf{pos}} = distance\ of\ the\ t^{th}\ nearest\ positive\ neighbor\ of\ \mathsf{q} \tag{3.5}$$

$$D_{\mathsf{t'}}^{\mathsf{neg}} = distance\ of\ the\ (t')^{th}\ nearest\ negative\ neighbor\ of\ \mathsf{q} \tag{3.6}$$

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

Before introducing the algorithm, we state and prove an important proposition, which relates the two quantities $D_{\mathsf{t}}^{\mathsf{pos}}$ and $D_{\mathsf{t'}}^{\mathsf{neg}}$ with the answer to KNS3.

Proposition 1 $D_{\mathsf{t}}^{\mathsf{pos}} \leq D_{\mathsf{t'}}^{\mathsf{neg}}$ *if and only if at least t of the k nearest neighbors of* q *are from the positive class.*

Proof:

If $D_{\mathsf{t}}^{\mathsf{pos}} \leq D_{\mathsf{t'}}^{\mathsf{neg}}$, then there are at least t positive points closer than the t'^{th} negative point to q. This also implies that if we draw a ball centered at q, and with its radius equal to $D_{\mathsf{t'}}^{\mathsf{neg}}$, then there are exactly t' negative points and at least t positive points within the ball. Since $t + t' = k + 1$, if we use D_k to denote the distance of the kth nearest neighbor, we get $D_k \leq D_{\mathsf{t'}}^{\mathsf{neg}}$, which means that there are at most $t' - 1$ of the k-NNs of q from the negative class. It is equivalent to say that there are at least t of the k nearest neighbors of q are from the positive class. On the other hand, if there are at least t of the k-NNs from the positive class, then $D_{\mathsf{t}}^{\mathsf{pos}} \leq D_k$, the number of negative points is at most $k - t < t'$, so $D_k \leq D_{\mathsf{t'}}^{\mathsf{neg}}$. This implies that $D_{\mathsf{t}}^{\mathsf{pos}} \leq D_{\mathsf{t'}}^{\mathsf{neg}}$ is true. ▌

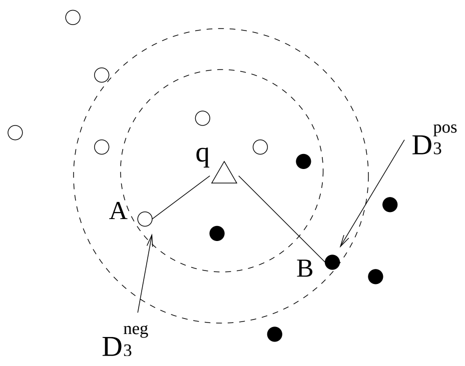


Figure 3.2: An example of $D_{\mathsf{t}}^{\mathsf{pos}}$ and $D_{\mathsf{t'}}^{\mathsf{neg}}$

Fig. (3.2) 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 $D_{\mathsf{t}}^{\mathsf{pos}}$ and $D_{\mathsf{t'}}^{\mathsf{neg}}$. In the given example, $D_{\mathsf{t}}^{\mathsf{pos}} = ||\mathsf{q} - A||$, $D_{\mathsf{t'}}^{\mathsf{neg}} = ||\mathsf{q} - B||$, and it is obvious that $D_{\mathsf{t}}^{\mathsf{pos}} \leq D_{\mathsf{t'}}^{\mathsf{neg}}$. So within the circle centered at 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 q are from the positive class.

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 $D_{\mathsf{t}}^{\mathsf{pos}}$ and $D_{\mathsf{t'}}^{\mathsf{neg}}$, instead, we can estimate them. We define $Lo(D_t^{\mathsf{pos}})$ and $Up(D_t^{\mathsf{pos}})$ as the lower and upper bounds of $D_{\mathsf{t}}^{\mathsf{pos}}$; and similarly we define $Lo(D_{t'}^{\mathsf{neg}})$ and $Up(D_{t'}^{\mathsf{neg}})$ as the lower and upper bounds of $D_{\mathsf{t'}}^{\mathsf{neg}}$. If at any point, we know $Up(D_t^{\mathsf{pos}}) \leq Lo(D_{t'}^{\mathsf{neg}}))$, then we can conclude that $D_{\mathsf{t}}^{\mathsf{pos}} \leq D_{\mathsf{t'}}^{\mathsf{neg}}$. On the other hand, if $Up(D_{t'}^{\mathsf{neg}}) \leq Lo(D_t^{\mathsf{pos}})$, we know $D_{\mathsf{t'}}^{\mathsf{neg}} \leq D_{\mathsf{t}}^{\mathsf{pos}}$. Now our computational task is to efficiently estimate $Lo(D_t^{\mathsf{pos}})$, $Up(D_t^{\mathsf{pos}})$, $Lo(D_{t'}^{\mathsf{neg}})$ and $Up(D_{t'}^{\mathsf{neg}})$. And it is very convenient for a metric-tree to do so. Below is the detailed description:

At each stage of KNS3 we have two sets of nodes in use called P and N, where P is a set of nodes from *Postree* built from positive data points, and N consists of nodes from *Negtree* built from negative data points.

Both sets have the property that if a node is in the set, then neither its metric-tree ancestors nor descendants are in the set, so that each point in the training set is a member of one or zero nodes in $P \cup N$. Initially, $P = Root_{pos}$ and $N = Root_{neg}$. Each stage of KNS3 analyzes P to estimate $Lo(D_t^{\mathsf{pos}})$, $Up(D_t^{\mathsf{pos}})$, and analyzes N to estimate $Lo(D_{t'}^{\mathsf{neg}})$, $Up(D_{t'}^{\mathsf{neg}})$. If possible, KNS3 terminates with the answer, else it chooses an appropriate node from P or N, and replaces that node with its two children, and repeats the iteration. Fig. (3.3) shows one stage of KNS3. The nodes involved are labeled *a* through *g* and we have

$$P = \{a, b, c, d\}$$

$$N = \{e, f, g\}$$

Notice that although c and d are inside b, they are not descendants of b. This is possible because when a node v is splitted, we only require $N(\mathsf{v.lc})$ and $N(\mathsf{v.rc})$ be disjoint, but $\mathbb{B}(\mathsf{v.lc.center}, \mathsf{v.lc.r})$ and $\mathbb{B}(\mathsf{v.rc.center}, \mathsf{v.rc.r})$ may be overlapped.

In order to compute $Lo(D_t^{\mathsf{pos}})$, we need to sort the nodes $u \in P$, such that

$$\forall u_i, u_j \in P, i < j \Rightarrow D_{\mathsf{minp}}^{\mathsf{i}} \leq D_{\mathsf{minp}}^{\mathsf{j}}$$

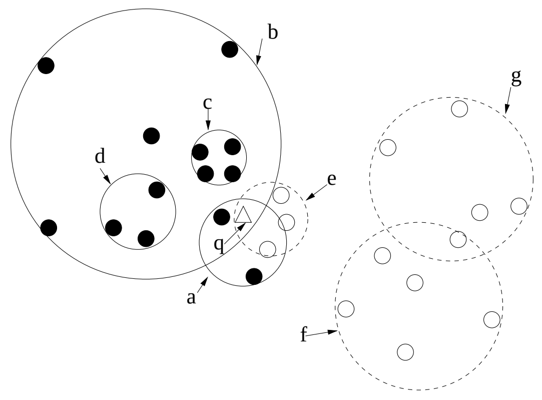


Figure 3.3: A configuration at the start of a stage.

Then

$$Lo(D_t^{\mathsf{pos}}) = D_{\mathsf{minp}}^{\mathsf{u_j}}, \text{ where } \sum_{i=1}^{j-1} \mid N(()u_i) \mid < t \text{ and } \sum_{i=1}^{j} \mid N(()u_i) \mid \geq t$$

Symmetrically, in order to compute $Up(D_t^{\mathsf{pos}})$, we sort $u \in P$, such that

$$\forall u_i, u_j \in P, i < j \Rightarrow D_{\mathsf{maxp}}^{\mathsf{i}} \leq D_{\mathsf{maxp}}^{\mathsf{j}}.$$

Then

$$Up(D_t^{\mathsf{pos}}) = D_{\mathsf{maxp}}^{\mathsf{u_j}}, \text{ where } \sum_{i=1}^{j-1} \mid N(()u_i) \mid < t \text{ and } \sum_{i=1}^{j} \mid N(()u_i) \mid \geq t$$

Similarly, we can compute $Lo(D_{t'}^{\mathsf{neg}})$ and $Up(D_{t'}^{\mathsf{neg}})$.

To illustrate this, it is useful to depict a node as an interval, where the two ends of the interval denote the minimum and maximum possible distances of a point owned by the node to the query. Fig. (3.4)(a) shows an example. Notice, we also mark "+5" above the interval to denote the number of points owned by the node B. After we have this representation, both P and N can be represented as a set of intervals, each interval corresponds to a node. This is shown in Fig (3.4)(b). For example, the second horizontal line denotes the fact that node b contains four positive points, and that the distance from any location in b to q lies in the range $[0, 5]$. The value of $Lo(D_t^{\mathsf{pos}})$ can be understood as the answer to the following question: what if we tried to slide all the positive points within their bounds as far to the

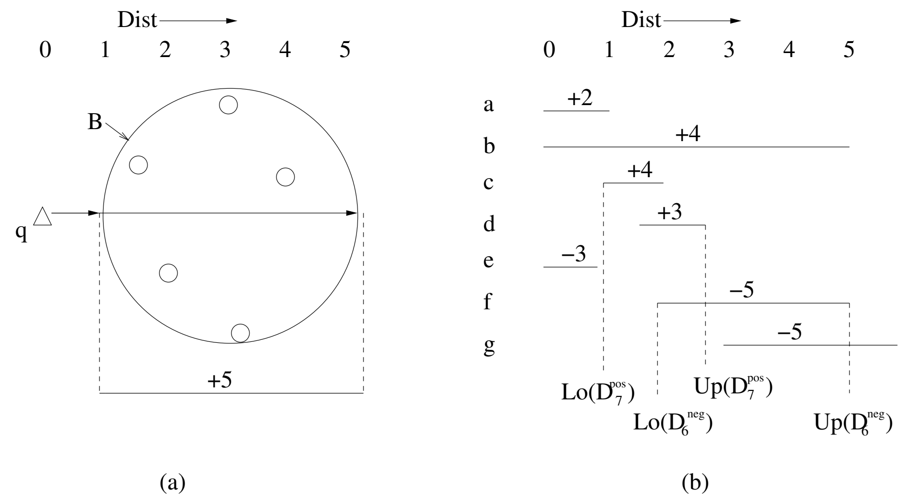


Figure 3.4: (a) The interval representation of a metric-tree node (b) The interval representation of a set of metric-tree nodes

left as possible, where would the t^{th} closest positive point lie? Similarly, we can estimate $Up(D_t^{\text{pos}})$ by sliding all the positive points to the right ends within their bounds.

For example, in Fig. (3.3), let $k = 12$ and $t = 7$. Then $t' = 12 - 7 + 1 = 6$. We can estimate ($Lo(D_7^{\text{pos}})$, $Up(D_7^{\text{pos}})$) and ($Lo(D_6^{neg})$, $Up(D_6^{neg})$), and the results are shown in Fig. (3.4). Since the two intervals ($Lo(D_7^{\text{pos}})$, $Up(D_7^{\text{pos}})$) and ($Lo(D_6^{\text{neg}})$,$Up(D_6^{\text{neg}})$) have overlap now, no conclusion can be made at this stage. Further splitting needs to be done to refine the estimation.

Below is the pseudo code of KNS3 algorithm: We define a loop procedure called *PREDICT* with the following input and output.

$$Answer = PREDICT(P, N, t, t')$$

The *Answer*, a boolean value, is TRUE, if there are at least t of the k-NNs from the positive class; and False otherwise. Initially, P = $Root_{pos}$ and N = $Root_{neg}$. The threshold t is given, and $t' = k - t + 1$.

Before we describe the algorithm, we first introduce two definitions.
Define:

$$(Lo(D_i^S), Up(D_i^S)) = Estimate\_bound(S, i) \tag{3.7}$$

Here S is either set P or N, and we are interested in the i^{th} nearest neighbor of q from set S. The output is the lower and upper bounds on the distance of this i'th nearest neighbor from the query. The concrete procedure for estimating the bounds was just described.

Notice that the estimation of the upper and lower bounds could be very loose in the beginning, and will not give us enough information to answer the question. In this case, we will need to split a metric-tree node and re-estimate the bounds. With more and more nodes being split, our estimation becomes more and more precise, and the procedure can be stopped as soon as $Up(D_t^{\text{pos}}) \leq Lo(D_{t'}^{\text{neg}})$ or $Up(D_{t'}^{\text{neg}}) \leq Lo(D_t^{\text{pos}})$. The function of $Pick(P, N)$ below is to choose one node either from P or N to split. There are different strategies for picking a node, for simplicity, our implementation only randomly picks a node to split.

Define:

$$split\_node = Pick(P, N) \tag{3.8}$$

Here split_node is the node chosen to be split. See Fig. (3.5).

```
Procedure PREDICT (P, N, t, m)
begin
  Repeat
    ($Lo(D_t^{\text{pos}}), Up(D_t^{\text{pos}})$) = Estimate_bound(P, t)          /* See Definition 3.7. */
    ($Lo(D_{t'}^{\text{neg}}), Up(D_{t'}^{\text{neg}})$) = Estimate_bound(N, t')
    if ($Up(D_t^{\text{pos}}) \leq Lo(D_{t'}^{\text{neg}})$) then
      Return TRUE
    if ($Up(D_{t'}^{\text{neg}}) \leq Lo(D_{t'}^{\text{neg}})$) then
      Return FALSE

    split_node = Pick(P, N)
    remove split_node from P or N
    insert $split\_node$.lc and $split\_node$.rc to P or N
end
```

Figure 3.5: Procedure PREDICT.

Our explanation of KNS3 was simplified for clarity. In order to avoid frequent searches over the full lengths of sets N and P, they are represented as priority queues. Each set in fact uses two queues: one prioritized by D^u_{maxp} and the other by D^u_{minp}.This ensures that the costs of all argmins, deletions and splits are logarithmic in the queue size.

Some people may ask the question: “It seems that KNS3 has more strengths than KNS2: it removes the assumption of skewedness of the dataset. In general, it has more chances to prune away nodes, etc. Why do we still need KNS2?” The answer is KNS2 does have its own advantages. It answers a more difficult question than KNS3. To know exactly how many of the nearest neighbors are from the positive class can be especially useful when the threshold for deciding a class is not known. In that case, KNS3 doesn’t work at all since we can not provide a static t for answering the question (c). But KNS2 can still work very well. On the other hand, the implementation of KNS2 is much simpler than KNS3. For instance, it does not need the priority queues we just described. So there does exist some cases where KNS2 is faster than KNS3.

3.3 Experimental Results

To evaluate our algorithms, we used both real data sets (from UCI and KDD repositories) and also synthetic data sets designed to exercise the algorithms in various ways.

3.3.1 Synthetic Data Sets

We have six synthetic data sets. The first synthetic data set we have is called *Ideal*, as illustrated in Fig. (3.3.1)(a). All the data in the left upper area are assigned to the positive class, and all the data in the right lower area are assigned to the negative class. The second data set we have is called *Diag2d*, as illustrated in Fig. (3.3.1)(b). The data are uniformly distributed in a 10 by 10 square. The data above the diagonal are assigned to the positive class, below diagonal are assigned to the negative class. We made several variants of Diag2d to test the robustness of KNS3. *Diag2d(10%)* has 10% data of *Diag2d*. *Diag3d* is a cube with uniformly distributed data and classified by a diagonal-plane. *Diag10d* is a 10 dimensional hypercube with uniformly distributed data and classified by a hyper-diagonal-plane. *Noise-diag2d* has the same data as *Diag2d(10%)*, but 1% of the data was assigned to the wrong class. Table (3.1) is a summary of the data sets in the empirical analysis.

3.3.2 Real-world Data Sets

We used UCI & KDD data (listed in Table (3.2), but we also experimented with data sets of particular current interest within our laboratory.

Life Sciences. These were proprietary data sets (*ds1* and *ds2*) similar to the publicly available Open Compound Database provided by the National Cancer Institute (NCI Open

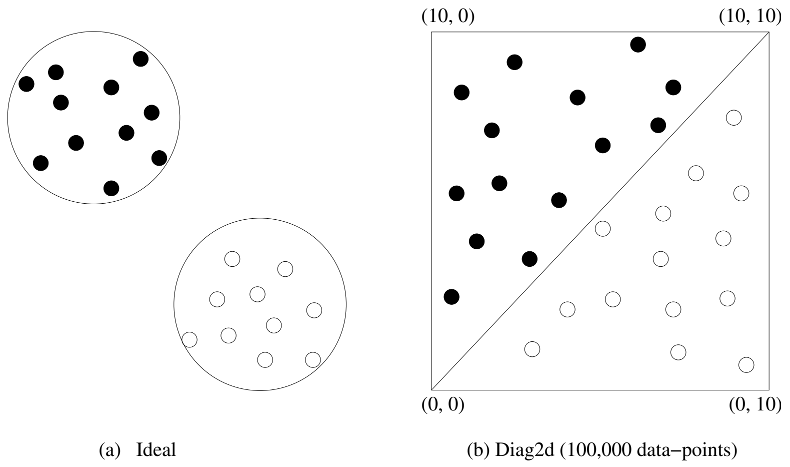


Figure 3.6: Synthetic data sets

Compound Database, 2000). The two data sets are sparse. We also present results on data sets derived from *ds1*, denoted *ds1.10pca*, *ds1.100pca* and *ds2.100anchor* by linear projection using principal component analysis (PCA).

Link Detection. The first, Citeseer, is derived from the Citeseer web site (Citeseer,2002) and lists the names of collaborators on published materials. The goal is to predict whether J_Lee (the most common name) was a collaborator for each work based on who else is listed for that work. We use *J_Lee.100pca* to represent the linear projection of the data to 100 dimensions using PCA. The second link detection data set is derived from the Internet Movie Database (IMDB, 2002) and is denoted *imdb* using a similar approach, but to predict the participation of Mel Blanc (again the most common participant).

UCI/KDD data. We use four large data sets from KDD/UCI repository. The data sets can be identified from their names. They were converted to binary classification problems. Each categorical input attribute was converted into n binary attributes by a 1-of-n encoding (where n is the number of possible values of the attribute).

1. *Letter* originally had 26 classes: A-Z. We performed binary classification using the letter A as the positive class and "Not A" as negative.

2. *Ipums* (from ipums.la.97). We predict *farm status,* which is binary.

3. *Movie* is a data set from informedia digital video library project [2001]. The TREC-2001 Video Track organized by NIST shot boundary Task. 4 hours of video or 13

Table 3.1: Synthetic data sets

Dataset	**Num. of records**	**Num. of Dimensions**	**Num. of positive**	**Num.pos/Num.neg**
Ideal	10000	2	5000	1
Diag2d(10%)	10000	2	5000	1
Diag2d	100000	2	50000	1
Diag3d	100000	3	50000	1
Diag10d	100000	10	50000	1
Noise2d	10000	2	5000	1

MPEG-1 video files at slightly over 2GB of data.

4. *Kdd99(10%)* has a binary prediction: Normal vs. Attack.

Table 3.2: Real-world data sets

Dataset	**Num. of records**	**Num. of Dimensions**	**Num.of positive**	**Num.pos/Num.neg**
ds1	26733	6348	804	0.03
ds1.10pca	26733	10	804	0.03
ds1.100pca	26733	100	804	0.03
ds2	88358	1.1×10^6	211	0.002
ds2.100anchor	88358	100	211	0.002
J_Lee.100pca	181395	100	299	0.0017
Blanc__Mel	186414	10	824	0.004
Dataset	**Num. records**	**Num. of Dimensions**	**Num.of positive**	**Num.pos/Num.neg**
Letter	20000	16	790	0.04
Ipums	70187	60	119	0.0017
Movie	38943	62	7620	0.24
Kdd99(10%)	494021	176	97278	0.24

3.3.3 Methodology and Results

For each data set, 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-NNs 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 data set) 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 (3.3) 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 data sets, 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.

3.4 Comments and Related Work

Applicability of other proximity query work. For the problem of "find the k nearest data points" (as opposed to our question of "perform k-NN or Kernel classification") in high dimensions, the frequent failure of traditional metric-tree to beat naïve has lead to some very ingenious and innovative alternatives, based on random projections, hashing discretized cubes, and acceptance of approximate answers. We will discuss these approaches in detail in chapter (5.1). However, these approaches are based on the notion that any points falling within a factor of $(1+\epsilon)$ times the true nearest neighbor distance are acceptable substitutes for the true nearest neighbor. Noting in particular that distances in high-dimensional spaces tend to occupy a decreasing range of continuous values (Hammersley [1950]), it remains unclear whether schemes based upon the absolute values of the distances rather than their *ranks* are relevant to the classification task. Our approach (KNS2 and KNS3), because it needs not find the k-NN to answer the relevant statistical question, finds an answer without approximation. The fact that our methods are easily modified to allow $(1+\epsilon)$ approximation in the manner of (Arya et al. [1998]) suggests an obvious avenue for future research.

No free lunch. For uniform high dimensional data no amount of trickery can save us without persuing approximation. The explanation for the promising empirical results is that all the interdependencies in the data mean we are working in a space of much lower intrinsic dimensionality (Maneewongvatana and Mount [2001]). Note though, that in ex-

Table 3.3: Number of distance computations and CPU time for Naïve k-NN classification (2nd column). Speed-ups of MT-DFS, KNS2 and KNS3 over Naïve.

		Naïve		MT-DFS		**KNS2**		**KNS3**	
		dists	time (secs)	dists speedup	time speedup	dists speedup	time speedup	dists speedup	time speedup
ideal	k=9	9.0×10^7	30	96.7	56.5	112.9	78.5	4500	**486**
	k=101			23.0	10.2	24.7	14.7	4500	**432**
Diag2d(10%)	k=9	9.0×10^7	30	91	51.1	88.2	**52.4**	282	27.1
	k=101			22.3	8.7	21.3	9.3	167.9	**15.9**
Diag2d	k=9	9.0×10^9	3440	738	366	664	**372**	2593	287
	k=101			202.9	104	191	107.5	2062	**287**
Diag3d	k=9	9.0×10^9	4060	361	**184.5**	296	**184.5**	1049	176.5
	k=101			111	56.4	95.6	48.9	585	**78.1**
Diag10d	k=9	9.0×10^9	6080	7.1	**5.3**	7.3	5.2	12.7	2.2
	k=101			3.3	**2.5**	3.1	1.9	6.1	0.7
Noise2d	k=9	9.0×10^7	40	91.8	20.1	79.6	30.1	142	**42.7**
	k=101			22.3	4	16.7	4.5	94.7	**43.5**
ds1	k=9	6.4×10^8	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	6.4×10^8	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	6.4×10^8	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	8.5×10^9	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	7.0×10^9	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	3.6×10^{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	3.8×10^{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	3.6×10^8	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**
Ipums	k=9	4.4×10^9	9520	195	136	665	501	1003	**515**
	k=101			69.1	50.4	144.6	121	5264	**544**
Movie	k=9	1.4×10^9	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**
Kdd99	k=9	2.7×10^{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

periments not reported here, k-NN classifiers give better performance on the original data than on PCA-projected low dimensional data, indicating that some of these dependencies are non-linear.

Chapter 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 q and its k-nearest-neighbor set $\mathcal{N}$. For binary classification, 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 The IOC Algorithm

In this section we discuss the IOC (standing for "International Olympic Committee," explained later) algorithm for approximating the k-NN classification for many-class setting. We first describe the problem with existing solutions using a metric-tree.

4.1.1 Previous Solutions and Problems

In chapter 3, we discussed KNS2 and KNS3 to speed up k-NN classification for binary classification problem, the techniques rely on the insight that in k-NN classification, one does not need to find the actual k-NNs. Rather, it is often sufficient to answer simpler, counting-related problems. As demonstrated in chapter 3, these questions can often be answered much more efficiently. To illustrate this point more clearly, we introduce a new concept, namely the "threshold nearest neighbor" function.

Definition 4.1.1 *The* threshold nearest neighbor *function, denoted by* tNN, *is defined as follows.*

$$\mathsf{tNN}(\mathsf{q}, \mathcal{T}, T, k, \ell) := \begin{cases} 1 & \text{if } |T \cap \mathsf{kNN}(\mathsf{q}, k, \mathcal{T})| \leq \ell \\ 0 & \text{otherwise} \end{cases} \tag{4.1}$$

Here, $\mathsf{kNN}(\mathsf{q}, k, \mathcal{T})$ denotes the k-NN set of q in training data $\mathcal{T}$ which is defined in chapter (1). Intuitively, $\mathsf{tNN}(\mathsf{q}, \mathcal{T}, T, k, \ell)$ checks whether of the k-NNs of q in $\mathcal{T}$, the subset T contains at most ℓ points. In fact, this function answers exactly the second type of the counting question discussed in KNS3.

Roughly speaking, the evaluation of $\mathsf{tNN}(\mathsf{q}, \mathcal{T}, T, k, \ell)$ is done by finding a "threshold bound" t, such that either

1. $\mathbb{B}(\mathsf{q}, t)$ contains at most ℓ points in T and at least $(k - \ell)$ points in $\mathcal{T} \setminus T$, or
2. $\mathbb{B}(\mathsf{q}, t)$ contains more than ℓ points in T and less than $(k - \ell)$ points in $\mathcal{T} \setminus T$.

In the first case, we have $\mathsf{tNN}(\mathsf{q}, \mathcal{T}, T, k, \ell) = 1$; in the second case, we have $\mathsf{tNN}(\mathsf{q}, \mathcal{T}, T, k, \ell) = 0$. In Section 4.1.3, we review the evaluation of tNN in more details.

Unfortunately, the insight in chapter 3 does not work in the many-class case. Consider a query point q and its k-NN set $\mathcal{N}$. For binary classification, q is classified as class i, if and only if $\mathcal{N}$ contains more than $\lfloor k/2 \rfloor$ points of class i.[1] Thus the task of finding the winner is reduced to a counting problem, or more specifically, evaluating the function $\mathsf{tNN}(\mathsf{q}, \mathcal{T}, T_i, k, \lfloor k/2 \rfloor)$. In the case of many-classes, 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 $\mathcal{N}$,[2] 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 techniques in chapter 3 do not extend to the many-class case.

4.1.2 IOC: High-level Descriptions

The IOC algorithm is a variant to the k-NN algorithm that allows speed-up using a metric-tree. The motivation behind IOC is to modify k-NN in such a way that it can be reduced

[1]This is assuming that k is odd.
[2]This is assuming that k is not a multiple of m.

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 because 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 (IOC [1999]) to select the host city for summer Olympic games (which also explains its name). In the procedure, instead of having a single round of ballot 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 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-NNs of q, this class is declared a winner and the algorithm terminates, labeling q with class i. Otherwise, the algorithm finds all the classes that contains at most $\lfloor k/m \rfloor$ points in the k-NNs of q, and declares 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.

To highlight the mathematics behind the algorithm, we present an "ideal" version of IOC, called the "IOC-ideal," in Fig. (4.1). As a convention, we use $T_1, T_2, ..., T_m$ to denote the collection of training data, partitioned according to their labels, i.e., T_i contains all training data labeled as class i. In this version of the IOC algorithm, we assume that there exists a very efficient procedure to evaluate the tNN function. In Section 4.1.3, we show a more practical (and more efficient) version of IOC that does not need this assumption.

There is a clear resemblance between the IOC city-picking procedure and the IOC classification algorithm: one can simply view each city as a "class," and each ballot as a point in the nearest neighbors of the query point. However, there are also differences. We point out the main ones for clarification.

Voting vs. classification. In the IOC city-picking procedure, the ballots across rounds can

```
Procedure IOC-ideal($k$, q, $\mathcal{T}$, $T_1, T_2, ..., T_m$)
/*  q is the query point  */
/*  $\mathcal{T}$ contains all training data  */
/*  $m$ is the number of classes  */
/*  $T_i \subseteq \mathcal{T}$ contains training data of class $i$ */
begin
  $A \leftarrow \{1, 2, ..., m\}$
  begin repeat
  /* check for winners  */
    if $\exists i \in A$, s.t., tNN$(\mathsf{q}, \mathcal{T}, T_i, k, \lfloor k/2 \rfloor) = 0$, then
      return $i$;
    else
      /* check for losers  */
      $\forall i \in A$,
        if tNN$(\mathsf{q}, \mathcal{T}, T_i, k, \lfloor k/m \rfloor) = 1$, then
          /* found a loser, remove it  */
          $A \leftarrow A \backslash \{i\}$, $\mathcal{T} \leftarrow \mathcal{T} \backslash T_i$, $m \leftarrow m - 1$;
        end if
      /* terminate if only one class remaining  */
      if $m = 1$ and $A = \{i\}$, return $i$
    end if
  end repeat
end
```

Figure 4.1: The IOC-ideal algorithm.

be completely unrelated, i.e., a voting member is free to change his/her mind in different rounds, and thus the number of votes a city receives in different rounds can vary significantly. In the IOC classification algorithm, however, the labels of the points are fixed for all rounds. On the other hand, since classes are eliminated in rounds, the nearest neighbors of q may change from round to round, and this implies that the number of points each class has in the k nearest neighbors will differ from round to round as well.

Eliminating losers. In the IOC city-picking procedure, exactly one loser is identified and eliminated. In the IOC classification algorithm, multiple losers can be eliminated in one round.

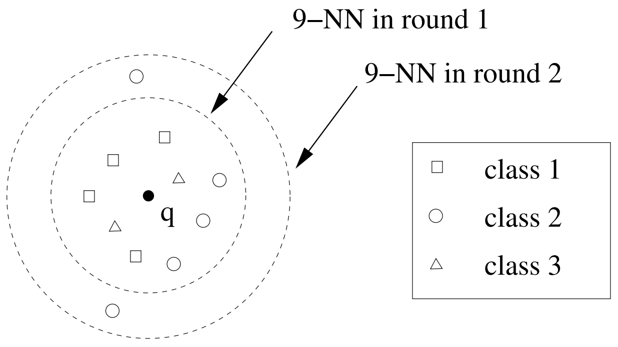


Figure 4.2: Different predictions by IOC and k-NN.

We notice that the IOC algorithm does not always behave identically to the standard k-NN algorithms, and in particular, the prediction made by the IOC algorithm may differ from that by the standard k-NN. See Fig. (4.2) for an example. In this example, there are 3 classes and $k = 9$. The 9 nearest neighbors of the query point q contains 4 points of class 1, 3 points of class 2, and 2 points of class 3. Therefore, standard k-NN algorithm would select class 1 as the winner. However, in the IOC algorithm, class 3 would be identified as a loser and removed in the first round. In the second round, the 9 nearest neighbors of q includes two additional points of class 2. Now we have 4 points of class 1 and 5 points of class 2 in this round, and IOC will choose class 2 as the winner.

Incidentally, a similar example occurred in the procedure for picking the host city for the 2000 Olympics game by IOC. The process proceeded in multiple rounds, and Beijing was the favorite city in all but the last round, but never won more than half of the votes. In the last round, Beijing lost to Sydney, and the IOC chose Sydney as the winner. If the standard k-NN algorithm had been used, Beijing would have been chosen.

4.1.3 The Actual Algorithm

We describe the actual IOC algorithm, which differs slightly from the IOC-ideal algorithm described in Section 4.1.2.

Evaluating the tNN Function

Recall that the IOC-ideal algorithm assumes that there exists an efficient procedure to evaluate the threshold nearest neighbor function tNN. Here, we describe the algorithm, denoted as MTtNN, that use metric-trees to evaluate tNN. This algorithm is adapted from (3.5).

To begin with, MTtNN builds one metric-tree for T_i, the set of training points of class i. Then, to evaluate function $\mathsf{tNN}(\mathsf{q}, \mathcal{T}, T_i, k, \ell)$, MTtNN needs to:

1. Find an appropriate threshold t, and
2. Prove that either:
   (a) $\mathbb{B}(\mathsf{q}, t)$ contains at most ℓ points in T_i and at least $(k - \ell)$ points in $\mathcal{T} \backslash T_i$ (so that $\mathsf{tNN}(\mathsf{q}, \mathcal{T}, T_i, k, \ell) = 1$), or
   (b) $\mathbb{B}(\mathsf{q}, t)$ contains more than ℓ points in T_i and less than $(k - \ell)$ points in $\mathcal{T} \backslash T_i$ (so that $\mathsf{tNN}(\mathsf{q}, \mathcal{T}, T_i, k, \ell) = 0$).

First, let us assume that t is known. We see how one can prove statement (2.a) or (2.b) using the metric-tree. Consider a node v in the metric-tree for class j. Suppose v represents s points, and the distance between $\mathsf{v.center}$ and q is x. By the triangle inequality, we know that if $t < x - \mathsf{v.r}$, we know none of the s points represented by v is in $\mathbb{B}(\mathsf{q}, t)$; if $t > x + \mathsf{v.r}$, then all the s points are in $\mathbb{B}(\mathsf{q}, t)$. In both cases, node v contributes information about the number of points in $\mathbb{B}(\mathsf{q}, t)$ and we say v is "useful." However, if $t \in [x - \mathsf{v.r}, x + \mathsf{v.r}]$, node v does not tell us anything, and we say node v is "useless." Then MTtNN sums up all the information from the useful nodes and checks if this information can be used to prove (2.a) or (2.b).

In case there is not sufficient information, MTtNN selects a useless node v to *split*, i.e., to replace node v by its two children $\mathsf{v.lc}$ and $\mathsf{v.rc}$. Since child nodes have smaller radii, they provide more "refined" information that might be useful. Ultimately, the leaf nodes provide very accurate information since they have very small radii.[3] However, splitting a node is an expensive operation, as one needs to compute the distance between q and the centers of the children nodes, and distance computations are the dominant operations in terms of time complexity. Therefore, to achieve optimal efficiency, one needs to minimize the number of splits.

Next, if we drop the assumption that t is known, MTtNN needs to search for t as well.

[3]As a matter of fact, it is typical to enforce each leaf node to contain a single point, in which case a leaf node has radius 0.

To do so, it maintains a list of "known" nodes from the metric-tree, i.e., the nodes where the distance between q and their pivots are computed and known, and searches for an appropriate t. If no such t is found due to insufficient information, the algorithm selects a node to split according to a certain splitting policy and tries again. As demonstrated in Fig. (3.5), with a carefully designed policy, one can indeed minimize the number of splits and make the algorithm very efficient.

Implementing IOC with Partial Functions

One could plug in the MTtNN algorithm directly into IOC-ideal, and we have an implementation of the IOC algorithm. However, this is not very efficient, since MTtNN may need to do a lot of splits in order to find the answer. In fact, observe that in each round, many instances of the tNN functions are evaluated — for each class i, we need to evaluate both $\mathsf{tNN}(\mathsf{q}, \mathcal{T}, T_i, k, \lfloor k/2 \rfloor)$ and $\mathsf{tNN}(\mathsf{q}, \mathcal{T}, T_i, k, \lfloor k/m \rfloor)$. We can make progress whenever we find one winner or one loser. This observation allows us to improve the efficiency by *dove-tailing*, i.e., evaluating all the tNN functions simultaneously, and terminating whenever a winner or a loser is found. More precisely, we modify the MTtNN algorithm so that it may also output $\perp$, standing for "unknown." Then we only do a split if all evaluations return $\perp$.

The actual algorithm, IOC, is described in Fig. (4.3). We denote by MTtNN$'$ the algorithm that partially computes tNN. In other words, MTtNN may return $\perp$ when it does not have sufficient information. On the other hand, MTtNN does not do any split. The splitting of the trees is now handled explicitly by the procedure do_split, which picks a particular class i and performs one split on the metric-tree of T_i. Effectively, the IOC algorithm minimizes the number of splits by aggressively attempting to evaluate all the tNN functions after each split.

We emphasize that the IOC algorithm is presented in a way to maximize clarity. In particular, we omit all optimizations, some of which are obvious. For example, after splitting class j, one only needs to update the information related to class j and there is no need to re-compute all x_i and y_i for all i's. Furthermore, many invocations of the MTtNN can be merged to improve efficiency. We choose not to mention these techniques in Fig. (4.3) since they are straightforward and are unnecessary to our discussion.

Picking the Right k

In k-NN classification, the choice of k has always been rather heuristic. Typically, one fixes k to be a constant (e.g., $k = 1$ or $k = 9$), or adaptively finds an optimal k using cross

```
Procedure IOC($k$, q, $\mathcal{T}$, $T_1, T_2, ..., T_m$)
/*  q is the query point */
/*  $\mathcal{T}$ contains all training data */
/*  $m$ is the number of classes */
/*  $T_i \subseteq \mathcal{T}$ contains training data of class $i$ */

begin
  $A \leftarrow \{1, 2, ..., m\}$
  begin repeat
  /* partially evaluate the tNN functions */
    foreach $i \in A$ do
      $x_i \leftarrow$ MTtNN'$(q, \mathcal{T}, T_i, k, \lfloor k/2 \rfloor)$
      $y_i \leftarrow$ MTtNN'$(q, \mathcal{T}, T_i, k, \lfloor k/m \rfloor)$
    end foreach
    progress $\leftarrow 0$
    /* check for winners and losers */
    if $\exists i \in A$, s.t., $x_i = 0$, then return $i$
    else
      $\forall i \in A$,
        if $y_i = 1$ then
          /* found a loser, remove it */
          $A \leftarrow A \backslash \{i\}$, $\mathcal{T} \leftarrow \mathcal{T} \backslash T_i$, $m \leftarrow m - 1$;
          progress $\leftarrow 1$;
        end if
        /* terminate if only one class remaining */
        if $m = 1$ and $A = \{i\}$, then return $i$
    end if
    /* need to split if no winner/loser can be found */
    if progress $= 0$ then
      do_split($\mathcal{T}$, $T_1, T_2, ..., T_m$)
    end if
  end repeat
end
```

Figure 4.3: The IOC algorithm.

validation. The situation for the IOC algorithm is more complicated, in that it proceeds in multiple rounds, and one could use a different k for different rounds. The IOC algorithms in Fig. (4.1) and Fig. (4.3) use the same k for all rounds, but one can easily modify this without compromising the results in the theoretical analysis. Obviously there can exist many different policies for changing k.

Intuitively, if the data set is dense so that the k-NNs of q remain sufficiently "local" to q, a larger k tends to yield more accurate prediction, as implied by Theorem (4.1.5). However, if the data is sparse, a large k will lead to a "non-local" neighbor set, which will decrease the prediction accuracy. On the other hand, in terms of efficiency, a small k typically implies faster prediction.

We use a different k in each round in the IOC algorithm, and our policy for picking k is as follows. Each time a loser class i is identified and removed, we estimate the number of points of this class in $\mathcal{N}$, the k-NNs of q, and we denote this by t_i. Then we change k to $k - t_i$ in the next round (if multiple classes are removed in one round, we reduce k for each of the loser class). The intuition behind this policy is that if we assume the training data points are well-clustered, then the points of a loser class i tend to be at the "outskirts" of $\mathcal{N}$. Therefore, the set consisting of the $(k - t_i)$ nearest neighbors of q excluding class i is about the same as the set consisting of the points that are the k-NNs of q but not in class i. More mathematically, we have

$$\mathsf{kNN}(\mathsf{q}, (k - t_i), \mathcal{T} \setminus T_i) \approx \mathsf{kNN}(\mathsf{q}, k, \mathcal{T}) \setminus T_i.$$

See Fig. (4.4).

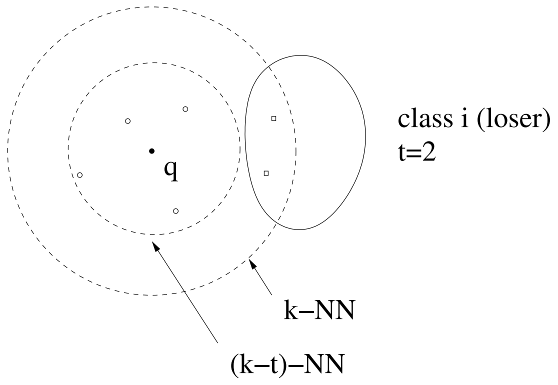


Figure 4.4: Change k between rounds: intuitions.

4.1.4 Theoretical Analysis

We analyze the behavior of the IOC algorithm from the theoretical perspective. First, we note that IOC always terminates and it is efficient even in the worst case.

Theorem 4.1.2 *IOC terminates in at most* $(m-1)$ *rounds.*

Proof: If during a round IOC finds a winner, then it terminates. Otherwise, notice that by the pigeonhole principle, there exists at least one class T_i that contains less than $\lfloor k/m \rfloor$ points in the k-NNs. Therefore, at least one class is eliminated in each round, and IOC runs for at most $m-1$ rounds. ▮

Next, we show that in many cases, IOC behaves identically to standard k-NN.

Theorem 4.1.3 *IOC behaves identically to the standard k-NN algorithm when $k = 1$ and when $m = 2$.*

Proof: For $k = 1$, we have $\lfloor k/m \rfloor = 1$ for any m. In this case, $\mathsf{tNN}(\mathsf{q}, \mathcal{T}, T, k, 1) = 0$ if and only if the nearest neighbor of q in $\mathcal{T}$ is in T. In other words, IOC returns the same class as the standard k-NN algorithm.

When $m = 2$, a class is a winner for IOC, if and only if it contains a majority of the points in the k-NNs. The standard k-NN algorithm uses exactly the same criterion. ▮

In the case of many-class classification and $k > 1$, IOC can behave differently from the standard k-NN algorithm. However, the next theorem shows that if a class is not chosen as the winner by IOC, then it will be a loser in k'-NN, for a properly chosen k'.

Theorem 4.1.4 *If class i is not chosen by the IOC algorithm as the winner, then there exists a k' such that class i is not the majority class in* $\mathsf{kNN}(\mathsf{q}, k', \mathcal{T})$.

Proof: If class i is not chosen as the winner, then either it is eliminated as a loser at some round, or another class is chosen as a winner at some round. In both cases, there exists a class j, such that there are more points of class j than that of class i in the k-NNs of q in some round ℓ. We focus on this round ℓ.

We denote the k-NNs of q by $\mathcal{N}$, and we use r to denote the maximum distance between q and points in $\mathcal{N}$. We assume that there are x_i points of class i and x_j points of class j in $\mathcal{N}$. Naturally we have $x_i < x_j$.

Now, consider all points in $\mathcal{T}$ that are within distance r of q — we assume that there are a total of k' points. Notice that k' can be greater than k since they may include points of the loser classes eliminated in rounds prior to round ℓ. Nevertheless, in these k' points, there are still x_i points from class i and x_j points from class j, since none of these two classes are eliminated before. Therefore, class i is not the favorite in $\mathsf{kNN}(\mathsf{q}, k', \mathcal{T})$. ∎

Intuitively, if the training data are well-clustered so that the standard k-NN algorithm is accurate, then we expect the algorithm to be "stable" in terms of k. In other words, the k-NN algorithm should output the same prediction for a range of k's. If this is indeed the case, then IOC will behave identically to the standard k-NN.

Remark: We stress here that IOC aims at solving the *exact* k-NN problem for *some* k — the only relaxation is that it might use a different k. This is in contrast to the *approximate* k-NN approach that finds points that are within $(1+\epsilon)$ of the nearest distance to q.

Finally, we show that asymptotically (i.e. when the size of the training set increases to infinity), IOC behaves identically to k-NN, provided that k is chosen appropriately. The proof is along the line of Cover and Hart (Cover and Hart [1967]), who proved that asymptotic error rate for 1-NN is at most twice of the optimal Bayes error rate. For larger values of k, the error rate of k-NN can be further reduced to approach $1 - p_1$ (see Duda and Hart [1973]).

We follow the convention in Duda and Hart [1973]. We denote the Bayes conditional probability for class i at q by p_i, and we assume that $p_1 \geq p_2 \geq \cdots \geq p_m$. Thus we assume (without loss of generality) that class 1 is the winner (as chosen by the optimal Bayes predictor). The Bayes error rate is thus $1 - p_1$. Next, we define $\delta = p_1 - p_2$, and we call it the "error margin." Intuitively, for the Bayes prediction (as well as the k-NN algorithm) to perform well, we need to have a *dominant class*, which implies a large δ. In fact, if δ is small, then class 1 and class 2 are equally likely in conditional probability, and one cannot have an accurate prediction. From now on, we assume that δ is "reasonably large." Further notice that as the IOC algorithm eliminates the loser classes, the error margin increases monotonically, since both p_1 and p_2 increases as conditional probabilities with the same rate.

Theorem 4.1.5 *Let q be the query point, m be the number of classes, n be the size of training set. Let $p_1 \geq p_2 \geq \cdots \geq p_m$ be the Bayes conditional probability for class i at q, and let $\delta = p_1 - p_2$. Then, with probability at least $1 - \epsilon$, the behavior of the IOC algorithm with*

$$k \geq \frac{12}{\delta^2} \log\left(\frac{2m}{\epsilon}\right)$$

is identical to the behavior of k-NN as $n \to \infty$.

Proof: First, We prove two useful inequalities:

$$p_1 - \frac{1}{m} \geq \frac{\delta}{2} \tag{4.2}$$

$$\frac{1}{2} - p_2 \geq \frac{\delta}{2} \tag{4.3}$$

The proof of (4.2) is straightforward: we have $p_1 - p_i \geq \delta$ for $i = 2, 3, ..., m$. Summing these $(m-1)$ inequalities together and we have

$$(m-1)p_1 - \sum_{i=2}^{m} p_i \geq (m-1)\delta,$$

or $m \cdot p_1 - 1 \geq (m-1)\delta$. Thus we have $p_1 - \frac{1}{m} \geq \frac{m-1}{m}\delta \geq \frac{\delta}{2}$.
To prove (4.3), we define $x = \frac{1}{2} - p_2$. Then we have $x = \frac{1}{2} - p_1 + \delta$ as well. Thus we have $2x = 1 - p_1 - p_2 + \delta \geq \delta$, or $x \geq \frac{\delta}{2}$.

Now we focus on a specific round in IOC, and we compute the probability that class 1 is chosen as a loser (we call it a"type I unlucky event") or a class other than 1 is chosen as a winner (we call it a "type II unlucky event"). Obviously, if no unlucky event ever happens in any of the rounds, class 1 will not be eliminated and will become the winner.

As $n \to \infty$, when the k-NNs of q are chosen, the expected number of points of class i contained in them is $k \cdot p_i$.

If a type I event happens, that means that the number of class 1 points is less than k/m, whereas the expected number of $k \cdot p_1$. By the Chernoff bound, we know that the probability of this event is at most

$$q_1 \leq e^{-(p_1 - \frac{1}{m})^2 \frac{k}{2p_1}} \leq e^{-\frac{\delta^2}{8}k}.$$

If a type II event happens, then a class j, which would have expected $k \cdot p_j$ number of points, has at least $k/2$ points. Again by the Chernoff bound, we know that the probability of this event is at most

$$q_2 \leq e^{-(\frac{1}{2} - p_j)^2 \frac{k}{3p_j}} \leq e^{-\frac{\delta^2}{12}k}.$$

Therefore, if we set $k \geq \frac{12}{\delta^2} \log\left(\frac{2m}{\epsilon}\right)$, then the probability that any of the unlucky events happens in any of the rounds is bounded by

$$2m \cdot e^{-\frac{\delta^2}{12}k} \leq \epsilon.$$

In this case, the IOC algorithm behaves exactly the same as the standard k-NN algorithm. ▮

Remarks

1. **IOC vs. k-NN vs. Bayes**
   Theorem 4.1.5 relates the behavior of IOC to that of k-NN. However, since asymptotically, k-NN has accuracy comparable to the (optimal) Bayes prediction (following the result by Cover and Hart [1967]), we can relate IOC and the optimal Bayes prediction by simply combining these two results. For example, by setting $\epsilon = (1-p_1)/100$, we will have that the error rate of IOC is at most $2.01(1-p_1)$.

2. **Asymptotic results.**
   We stress that the result of Theorem 4.1.5 is about the *asymptotic* behaviors of IOC and k-NN. In other words, the resultant behavior is guaranteed only as the size of the training set approaches infinity. In real-world experiments, we would naturally expect deviations of IOC from k-NN. Nevertheless, the theorem still provides insight into how IOC is related to k-NN.

3. **Worst-case guarantee.**
   One should view Theorem 4.1.5 as a *worst-case* analysis of the IOC algorithm. It guarantees that with appropriately chosen parameters, IOC performs *at least* almost as well as standard k-NN. One should not, however, interpret this result as evidence that IOC is inferior to k-NN in actual performance. From the perspective of Bayes inference, both k-NN and IOC are approximations of the optimal Bayes prediction, and their performances are not directly comparable. In fact, as we show in Section 4.3, empirical results suggest that the IOC algorithm typically has the same accuracy as the standard k-NN algorithm, and sometimes even outperforms k-NN.

As before, let q be a query point and let m be the number of classes. We denote the conditional probability for class i at q by p_i, and we assume that $p_1 \geq p_2 \geq \cdots \geq p_m$. Thus we assume (without loss of generality) that class 1 is the winner (as chosen by the standard k-NN algorithm). The Bayes error rate is thus $1 - p_1$. A result by Cover and Hart [1967] shows that the asymptotic error rate for 1-NN is at most $2(1-p_1)$, namely twice of the Bayes error rate. For larger values of k, the error rate of k-NN can be further reduced to approach $1 - p_1$ (see Duda and Hart [1973]).

Here we show that the IOC algorithm can do at least almost as well as k-NN, given k is reasonably large. We stress that one should view Theorem 4.1.5 as a *worst-case* analysis of the IOC algorithm. It guarantees that the appropriately chosen parameter, IOC performs *at least* almost as well as standard k-NN. One should not, however, interpret this result as an evidence that IOC is inferior to k-NN in actual performance. From the perspective of Bayes inference, both k-NN and IOC are approximations of the optimal Bayes prediction, and their performances are not directly comparable. In fact, as we show in Section 4.3, empirical results suggests that the IOC algorithm typically has the same accuracy as the standard k-NN algorithm, and often outperforms k-NN in terms of both accuracy and efficiency.

4.2 Making IOC Robust

We implemented the IOC algorithm described in the previous section and tested it on both artificially generated data and real-world data. It performs very well on simple artificial data, exhibiting significant speed up, both in number of distance computations and CPU time, over the naïve k-NN algorithm, as well as over ones using SR-trees and metric-trees.

Unfortunately, the speed-up of the IOC algorithm degrades when the complexity of the train set increases. For both complex artificial data and real-world data, and in particular, ones with large numbers of attributes, we typically only observe about 2–3 fold speed-up. In this section, we investigate this problem and propose a solution known as RIOC, standing for "Robust IOC." Our solution is validated by experiments.

4.2.1 The Simple IOC is Sensitive to Noise

The efficiency of the IOC algorithm in the previous section, which we call the "simple IOC" thereafter, is sensitive to noise in the training set. Consider an (simplified) example shown in Fig. (4.5).

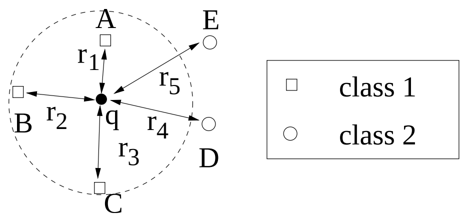


Figure 4.5: IOC with noise ($k = 3$).

In the figure, q is the query point and also the center of the circle. A, B, C, D, E are the training points, with distances $r_1, r_2, ..., r_5$ from q, respectively, where $r_1 < r_2 < r_3 < r_4 < r_5$. Of the five points, A, B, C are all of class 1, while D and E are of class 2. Notice that the 3 nearest neighbors of q, namely, $\mathcal{N} = \{A, B, C\}$ consist solely of points from class 1. For well-clustered training data, it is in fact a typical scenario to have one class dominating the nearest neighbor set. However, for real-world data, and especially when the number of classes is large, it is also common to have points from other classes in the vicinity of q — in the example, there are D and E. We call these points the "noisy points."

Since the 3-NN set of q consists solely of points from class 1, k-NN would classify q as class 1, and naturally we would expect IOC to do the same. However, if the noisy points are close to q, IOC would need to do many splits. To see this, recall that to prove that class 1 is the winner, IOC needs to prove that $\mathsf{tNN}(\mathsf{q}, \mathcal{T}, T_1, 3, 1) = 0$. To do so, IOC needs to find a threshold bound t such that $\mathbb{B}(\mathsf{q}, t)$ contains at least 2 points of class 1 and at most 1 point of all the other classes. Therefore, we must have $r_2 < t < r_5$. If r_2 and r_5 are close (which is also typical, especially in high dimensions), then the "margin" for picking the appropriate threshold bound t is very small. This means that the IOC algorithm has to split many nodes to get to the lower level of the metric-tree, so that the nodes have small radii. As a result, the IOC algorithm is not very efficient because of these splits.

4.2.2 Pre-pruning: Filtering the Noise

We use a technique called "pre-pruning" to filter out the noisy points and speed up the simple IOC algorithm. The technique is based on the observation that typically the winner class is dominant in the nearest neighbors and that the greedy search on the metric-tree has reasonably good accuracy. Therefore, we can use the greedy search (which is very efficient) to do a "pre-pruning" to "filter out" many of the loser classes. This works because the winner class is robust and is unlikely to be filtered out.

We describe our new algorithm, which we call RIOC standing for "Robust IOC." RIOC starts by building a metric-tree T for all the points. As a pre-processing step, RIOC finds the exact k-NNs for each point in the training set $\mathcal{T}$. We say leaf node v_1 is *related* another leaf node v_2, if there exist $x_1 \in N(\mathsf{v}_1)$ and $x_2 \in N(\mathsf{v}_2)$, such that $x_1 \in \mathsf{kNN}(x_2, k, \mathcal{T})$. RIOC records all of the related nodes for each leaf node.

Then, upon a query point q, RIOC performs a greedy NN search for q on T to find a leaf node v. Then v and all the leaf nodes related to v are searched to find the k-NNs of q within them. We denote the resulting set by $\mathcal{N}'$. Notice that since the greedy search does not backtrack, $\mathcal{N}'$ might not be the actual k-NNs, but is close. Next, all classes that do not have points in $\mathcal{N}'$ are removed. This finishes the pre-pruning procedure. After this, RIOC switches to IOC and runs with the remaining classes.

We analyze the pre-pruning process. First, we note that it will effectively remove many classes — consider a typical setting where $k = 9$. After the pre-pruning, at most 9 classes will remain. In practice, for well-clustered data, we often observe that only 1 or 2 classes remain after this process. Therefore, the pre-pruning process improves efficiency of IOC substantially (the pre-pruning itself is very efficient, since the greedy search does not backtrack).

Next, we show that the pre-pruning has a very low error rate, i.e., the probability that the winner class is removed by pre-pruning process is very small. Intuitively, this is because a winner class is typically very robust, in that it contains many points among the nearest neighbors of q, and it is unlikely that none of these points are selected by the greedy search algorithm. As an illustration, consider the following simple analysis. We assume that for each "true" nearest neighbor $x \in \mathcal{N}$, the greedy search algorithm finds it with probability at least p. Empirically p is quite large (at least 0.7). For a winner class i, it typically has many points in $\mathcal{N}$, and we assume that at least $k/2$ points in $\mathcal{N}$ are labeled as class i. Then, the probability that class i is not included in $\mathcal{N}'$ is at most $q = (1-p)^{k/2}$ — this is the probability that class i is removed. Substituting in $p = 0.7$ and $k = 9$, we have $q = 0.008$, which is quite small. Therefore, the pre-pruning process has a very limited negative effects on the accuracy, and thus RIOC has very good performance.

4.3 Experimental Results

In this section, we tested the IOC algorithm on both artificial and real-world data sets and compared the results with three other algorithms:

1. Naïve: a conventional k-NN algorithm, using linear scan to find the k-NN.
2. SR-tree: an implementation by (Katayama and Satoh [1997]).
3. MT-DFS: a highly optimized k-NN search based on metric trees (Uhlmann [1991]).

We estimate two performance measures:

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 speed-ups of SR-tree, MT-DFS and RIOC. (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 RIOC. We emphasis that since our goal is to accelerate multi-class classification in high dimensions, we do not try to improve accuracy (though we should expect no decline). We consider it acceptable to have both k-NN and RIOC perform badly on some data sets as long as their performance is comparable.

4.3.1 Artificial Data Sets

Before we look at real-world examples, we first test our algorithms on artificial data sets. The results will help us get a clear idea of how the IOC and the RIOC algorithms perform. Table 4.1 is a summary of the synthetic data sets: `Gauss_5c` contains 5 classes, each class has 5,000 2-dimensional points generated from Gaussian distribution. We chose different mean for each class so that the data are well separated. Fig. (4.6) shows the distribution of `Gauss_5c`. similarly we generated `Gauss_10c` and `Gauss_50c` with varying dimensions and number of classes.

Dataset	**Num. Data**	**Num. Dimensions**	**Num. classes**
`Gauss_5c`	25,000	2	5
`Gauss_10c`	50,000	5	10
`Gauss_50c`	250,000	10	50

Table 4.1: The artificial data sets.

For each of the data sets, we randomly select 10% of the data as the training set and use the remaining 90% as the test set. We ran our experiments with $k = 1$, 5, and 9.

The efficiency of various algorithms over these data sets are summarized in Table 4.3. We also plot the speed-up of various algorithm over naïve k-NN (CPU time) for the case $k = 5$ in Fig. (4.7). We do not report the classification accuracy, since nearly all algorithms have near perfect performance.

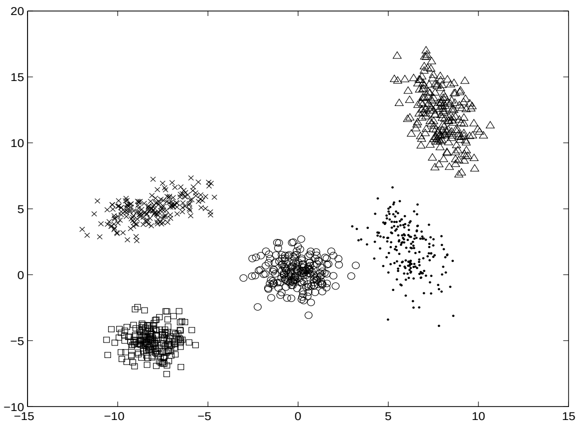


Figure 4.6: The `Gauss_5c` data set.

Notice that the IOC algorithm exhibits significant speed-up over all other algorithms (up to $3,900$-fold for naïve, and 17-fold for MT-DFS in terms of distance operation; up to 43-fold for naïve, 150-fold for SR-tree, and 7.8-fold for MT-DFS in terms of CPU time). But the speed-up degrades as the number of classes and the size of the data set increases. On the other hand, the RIOC algorithm scales much better than IOC with the number of classes and the data set size. For small data sets (e.g. `Gauss_5c`), IOC is more efficient than RIOC, but when the complexity of the data set increase, RIOC outperforms IOC. This serves as an evidence that RIOC is more robust than IOC.

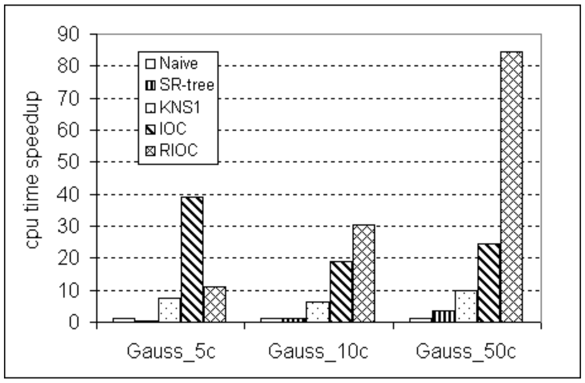


Figure 4.7: CPU time speedup over naïve k-NN for artificial data sets $(k = 5)$.

4.3.2 Real-world Data with RIOC

We tested our algorithm on a variety of real-world data sets (listed in Table 4.2) with multi-class classification tasks. The data sets are all publicly available.

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 4.2: The real-world data sets

1. *Letter* (Letter Recognition Database (Slate)). This data set is from UCI Machine Learning repository, and contains 20,000 instances with 26 classes. Each instance represents a bitmap image of a character as one of the 26 capital letters in the English alphabet. The objective is to identify the letter category from the images.

2. *Isolet* (Isolet Spoken Letter Recognition Database (Cole and Fanty)) This data set contains 6238+1559 instances with 26 classes. The data set was derived from 150 people spoke the name of each letter of the alphabet twice, 3 examples are missing. Each instance has 617 attributes. The goal is to predict which letter-name was spoken.

3. *Cov_type* (Forest CoverType Database) This data set is originally from UCI/KDD Archive. The data set contains 581012 datapoints with 7 classes. Detailed description can be found at (Blackard).

4. *Video* (TREC-2001 Video Data set (informedia digital video library project [2001])) It contains a 5.8 hours of MPEG-1 video files. The task is to detect the shot boundaries within the video files. The corpus contain 2 types of transition frames: cuts and gradual transitions, so we can see this problem as a 3 class classification problem, no transition, cut and gradual transition. After preprocessing, the final data set contains 38943 frames, each frame has 62 attributes (Qi et al. [2003]).

5. *Internet_ads* (Internet Advertisements Kushmerick) This data set represents a set of possible advertisements on Internet pages. The task is to predict whether an image

is an advertisement ("ad") or not ("non-ad"). After we remove the three continuous attributes, the final data set contains 3279 instances, and 1555 attributes for each instance.

For each data set, we manual 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 4.4), as well as the pre-processing time and the error rates (see Table 4.5) for all algorithms over these data sets. We also plot the speed-up of various algorithms over naïve k-NN (CPU time) for the case $k = 5$ in Fig. (4.8). Furthermore, we report how the CPU time of various algorithms scales with the size of the training data (see Fig. (4.9) for the case $k = 1$ and Fig. (4.10) for the case $k = 9$). We do not report the results for the simple IOC algorithm since RIOC essentially dominates it.

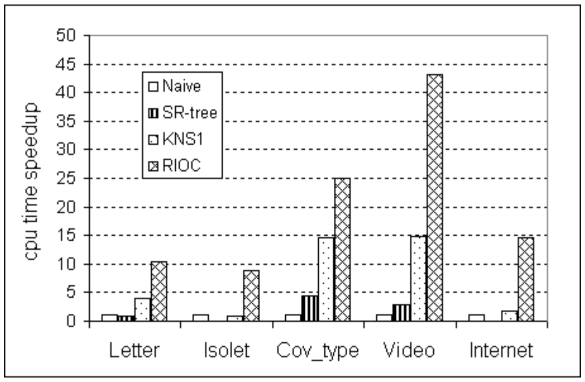


Figure 4.8: CPU time speed up over naïve k-NN for real-world data sets $(k = 5)$.

Error rate: The error rate of the naïve k-NN, the SR-tree, and the MT-DFS algorithms are the same, since they are all exact k-NN algorithms. For the RIOC algorithm, the error rate is slightly different. For *Letter* with k=1, the accuracy for RIOC is worse than k-NN, while for all the other data sets and other settings of k, the error rates are comparable. In some cases RIOC has even better accuracy than k-NN. This validates our claim that both k-NN and IOC (and hence RIOC) are approximate versions of the optimal Bayes prediction, and none generally outperforms the other. We also want to mention that, the error rate for the simple IOC algorithm is often slightly smaller than RIOC, but as we have stated before, this is at a cost of mediocre speed up. Again, this matches our intuition that the aggressive pre-pruning done by RIOC does not affect its performance significantly.

The data set *Internet_ads* is particularly interesting and merits special mention. Notice that it is a two-class data set, and KNS3 can be directly applied here. In particular, one can use the KNS3 algorithm to speed up the k-NN prediction. However, our RIOC algorithm, which is designed for many-class prediction, shows about 2-fold speed-up over KNS3, and has about the same accuracy. This fact suggests that the pre-pruning technique used by RIOC might have much wider applicability.

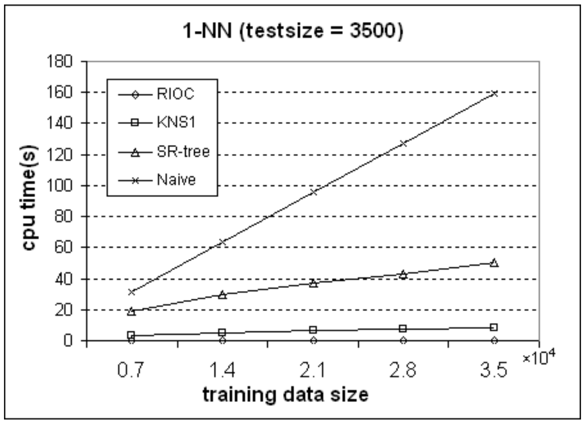


Figure 4.9: CPU time vs. train data size (data set=Video, $k = 1$).

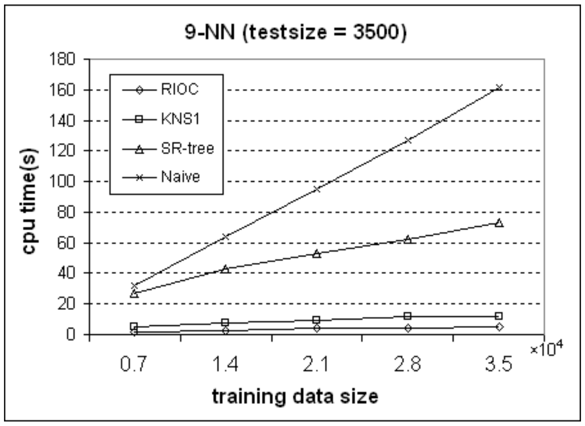


Figure 4.10: CPU time vs. train data size (data set=Video, $k = 9$).

Scaling: We performed the simulations for scaling over data set *Video*. We fixed 3500 points as a test set and trained on 5 training sets with sizes 7000, 14000, 21000 and 28000. To achieve better understanding of the scalability of our algorithms, we ran the experiments for both $k = 1$ and $k = 9$.The results are presented on Fig. (4.9) and Fig. (4.10). Notice that RIOC scales much better than all other algorithms.

Pre-processing time: One might notice that RIOC takes much longer preprocessing time compared with MT-DFS. For the purposes of this paper, we are not concerned with this, for two reasons. First, the pre-processing time is a one-time effort which can be amortized *if* there is a stream of many queries. The large pre-processing time can be misleading since in the experiments, because we chose to use a very large training set (compared to the testing set) in order to exercise the algorithms with large amounts of training data. The reason to do so is that we only have limited training data. We believe that the majority of real-world prediction situations involve very large numbers of predictions in comparison with the amount of training data (this is best exemplified by examples such as credit card fraud detection, vision-based security-screening, handwritten digit scanning and so on). The test set is thus usually much greater than the training set. In such a scenario, RIOC shall have very good efficiency as compared to other algorithms, even if we factor in the pre-processing time.

The second reason that we are prepared to tolerate RIOC's increased preprocessing time is that we have developed and tested another version of RIOC that eliminates almost all the pre-processing time at the cost of slightly worse accuracy (but still comparable to other algorithms as naïve k-NN and MT-DFS). This version of the RIOC would be appropriate when we indeed have a limited testing set. We will not discuss the new variation of the RIOC algorithm due to space limitations.

Discussions on pre-pruning: With the success of the pre-pruning technique in RIOC, it is tempting to speculate how aggressively one can use it. In particular, one may be tempted to try to use the pre-pruning *only* and completely skip the IOC process. It would yield extremely efficient classification algorithms. Unfortunately, this does not work since the pre-pruning is still imperfect in that the "nearest neighbor set" $\mathcal{N}'$ returned by the greedy search (which does not backtrack) can in fact bias significantly from the true nearest neighbor set $\mathcal{N}$. Notice that our analysis in Section 4.2.2 only shows that the winner class (one that dominates $\mathcal{N}$) is very likely to be *in* $\mathcal{N}'$, but not necessarily *dominate* $\mathcal{N}'$. Thus, if one bases the classification solely on $\mathcal{N}'$, the accuracy is unsatisfactory. This is validated by our empirical results.

		Naïve(s)	**SR-tree**	MT-DFS	**IOC**	**RIOC**
Gauss_5c	k=1	4.77	0.31	17.7	43.36	24.46
	k=5	4.78	0.29	7.68	39.10	10.9
	k=9	4.77	0.27	5.18	40.42	8.22
Gauss_10c	k=1	66.87	1.46	11.69	25.72	77.76
	k=5	66.90	1.0	6.48	19.01	30.5
	k=9	66.92	0.85	5.1	14.02	22.2
Gauss_50c	k=1	2430	4.39	13.45	30.08	154.85
	k=5	2430	3.54	9.78	24.65	84.38
	k=9	2430	3.34	8.65	19.19	59.94

Table 4.3: CPU time(s) of naïve k-NN and the Speed-up of 4 others methods over it.

		Naïve	**SR-tree**	MT-DFS	**RIOC**
Letter	k=1	26.79	1.3	6.47	16.4
	k=5	—	0.88	3.97	10.4
	k=9	—	0.79	3.29	7.4
Isolet	k=1	111.72	n/a	1.1	16.6
	k=5	—	—	0.93	8.9
	k=9	—	—	0.88	6.3
Cov_type	k=1	40776	7.31	26.05	38.43
	k=5	—	4.46	14.56	25.07
	k=9	—	3.62	11.57	14.01
Video	k=1	177.4	3.7	19.8	663.6
	k=5	—	2.89	14.8	43.1
	k=9	—	2.63	13.2	30.2
Internet	k=1	28.02	n/a	2.4	58.4
	k=5	—	—	1.7	14.7
	k=9	—	—	1.4	9.8

Table 4.4: Speed-up for real-world data set.

		Naïve	**SR-tree**		MT-DFS		**RIOC**	
		error	pre-pro (secs)	error	pre-pro (secs)	error	pre-pro (secs)	error
Letter	k=1	0.043	53.77	0.043	0.46	0.043	6.68	0.112
	k=5	0.054	53.77	0.054	0.46	0.054	6.68	0.088
	k=9	0.056	53.77	0.056	0.46	0.056	6.68	0.077
Isolet	k=1	0.114	n/a	n/a	4.43	0.114	68.8	0.119
	k=5	0.077	—	—	4.43	0.077	68.8	0.085
	k=9	0.08	—	—	4.43	0.08	68.8	0.08
Cov_type	k=1	0.136	311.32	0.136	5.10	0.136	101	0.117
	k=5	0.165	311.32	0.165	5.10	0.165	101	0.165
	k=9	0.176	311.32	0.176	5.10	0.176	101	0.172
Video	k=1	0.15	239.63	0.15	3.49	0.15	52.3	0.16
	k=5	1.127	239.63	0.127	3.49	0.127	52.3	0.127
	k=9	0.125	239.63	0.125	3.49	0.125	52.3	0.126
Internet	k=1	0.040	n/a	n/a	6.42	0.040	79.9	0.049
	k=5	0.052	—	—	6.42	0.052	79.9	0.052
	k=9	0.062	—	—	6.42	0.062	79.9	0.064

Table 4.5: Pre-processing time and error rates.

Chapter 5

Fast $(1+\varepsilon)$-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 an approximate-nearest-neighbor k-NN algorithm, i.e., $(1+\varepsilon)$-NN. Formally: given an error bound $\epsilon > 0$, we say that a point $w^{\epsilon} \in \mathcal{T}$ is a $(1+\varepsilon)$-NN of q if $||w^{\epsilon} - \mathsf{q}|| \leq (1+\epsilon)||w - \mathsf{q}||$, where w is the true nearest-neighbor of q. For k-NN the k^{th} point returned by the algorithm is no more than $(1+\epsilon)$ times the distance of the true k^{th} nearest-neighbor. Since our new $(1+\varepsilon)$-NN algorithm is partially motivated by the locality sensitive hashing (LSH) algorithm. Before we describe the 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 Indyk and Motwani [1998], Indyk [2000] 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 Gionis et al. [1999]. 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. Fig. (5.1) 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

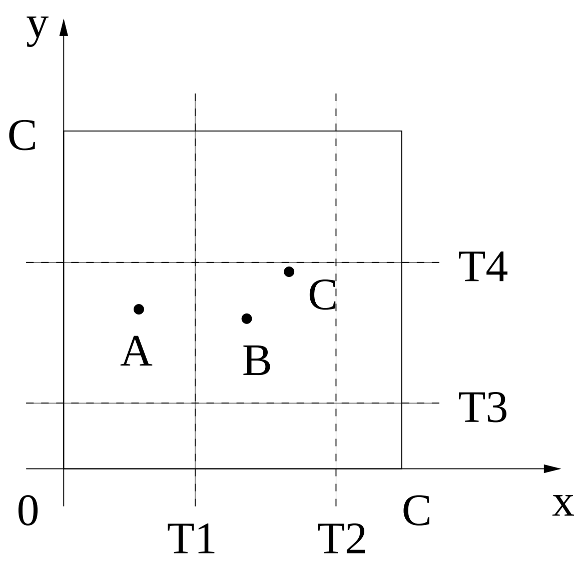


Figure 5.1: The "bucketing" view of LSH.

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 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 [1998] 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 Gionis et al. [1999], Indyk and Thaper [2003] that LSH can be useful in practice.

On the other hand, we observe that LSH has its own limitations. 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. Another issue with LSH is if you cannot find enough nearest-neighbors in the bins examined, there is no way to expand the set.

5.2 Spill-tree

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

The partitioning procedure of a metric-tree (See section 2.2) implies that $N(\mathsf{v.lc})$ and $N(\mathsf{v.rc})$ 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 data points may belong to both v.lc and v.rc.

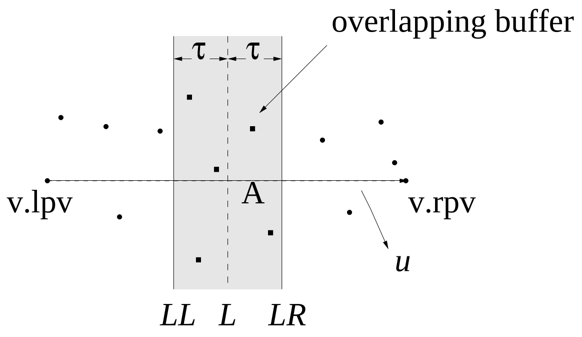


Figure 5.2: Partitioning in a spill-tree.

We first explain how to split an internal node v. See Fig. (5.2) as an example. As in 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 τ 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

$$N(\mathsf{v.lc}) = \{x \mid x \in N(\mathsf{v}), d(x, LR) + 2\tau > d(x, LL)\} \tag{5.1}$$

$$N(\mathsf{v.rc}) = \{x \mid x \in N(\mathsf{v}), d(x, LL) + 2\tau > d(x, LR)\} \tag{5.2}$$

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 τ 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.

5.3 Spill-tree-based k-NN Search

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-trees, since the points in the overlapping buffer may be searched twice. Nonetheless, the advantage of spill-trees over metric-trees becomes clear when we perform the *defeatist search*, an $(1+\varepsilon)$-NN search algorithm based on spill-trees.

5.3.1 Defeatist Search

As we have stated, the MT-DFS algorithm typically spends a large fraction of time backtracking to prove a candidate point is the true NN. Based on this observation, a quick revision would be to descend the metric-tree using the decision boundaries at each level without backtracking, and then output the point x in the first leaf node it visits as the k-NN of query q. We call this the *defeatist* search on a metric-tree. Since the depth of a metric-tree is $O(\log n)$, the complexity of defeatist search is $O(\log n)$ per query.

The problem with this approach is very low accuracy. Consider the case where q is very close to a decision boundary L, then it is almost equally likely that the NN of q is on the same side of L as on the opposite side of L, and the defeatist search can make a mistake with probability close to $1/2$. In practice, we observe that there exists a non-negligible fraction of the query points that are close to one of the decision boundaries. Thus the average accuracy of the defeatist search algorithm is typically unacceptably low, even for approximate k-NN search.

This is precisely the place where a spill-tree can help: the defeatist search on a spill-tree has much higher accuracy and remains very fast. We first describe the algorithm. For simplicity, we continue to use the example shown in Figure (5.2). 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 $\mathsf{v.lc}$. 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 $\mathsf{v.rc}$, 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 τ, 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 τ away from q.

So immediately, if τ is greater than the distance between q and its nearest neighbor, then we *never* make a mistake. In practice, however, τ can be much smaller and the defeatist search still have a very high accuracy.

In a nutshell, spill-tree gains more accuracy by introducing redundancy to a metric-tree. Naturally, an alternative of building a spill-tree is to explicitly scheduling a limited amount of backtracking on a metric-tree. For instance, we can say that we only allow to do 10 backtrackings. The advantage of this approach is we don't need to change the tree structure, the problem is the algorithm is not very flexible, since we might make mistakes at any level of the tree, it is really hard to use a single threshold to bound the number of backtrackings to get reasonable results.

5.3.2 Hybrid Spill-Tree Search

One problem with spill-tree is that their depth varies considerably depending on the overlapping size τ. When τ is relativelly small, there are not many duplicate data points between children nodes, so the depth of the spill-tree is close to $O(\log n)$. With τ increasing, there are more and more shared data points between children nodes, and the depth of the spill-tree increasing. In two extreme cases, if $\tau = 0$, a spill-tree turns back to a metric-tree with depth $O(\log n)$. On the other hand, if $\tau \geq ||\mathsf{v.rpv} - \mathsf{v.lpv}||/2$, then $N(\mathsf{v.lc}) = N(\mathsf{v.rc}) = N(\mathsf{v})$. In other words, both children of node v contain *all* points of v. In this case, the construction of a spill-tree does not even terminate and the depth of the spill-tree is ∞.

To solve this problem, we introduce *hybrid spill-trees* and actually use them in practice. First we define a *balance threshold* $\rho < 1$, which is usually set to 70%. The constructions of a hybrid spill-tree is similar to that of a spill-tree except the following. For each node v, we first split the points using the overlapping buffer. However, if either of its children contains more than ρ fraction of the total points in v, we undo the overlapping splitting. Instead, a conventional metric-tree partition (without overlapping) is used, and we mark v as a *non-overlapping* node. In contrast, all other nodes are marked as *overlapping nodes*. In this way, we can ensure that each split reduces the number of points of a node by at least a constant factor and thus we can maintain the logarithmic depth of the tree.

The k-NN search on a hybrid spill-tree also becomes a hybrid of the MT-DFS search and the defeatist search. We only do defeatist search on overlapping nodes, for non-overlapping nodes, we still do backtracking as MT-DFS search.

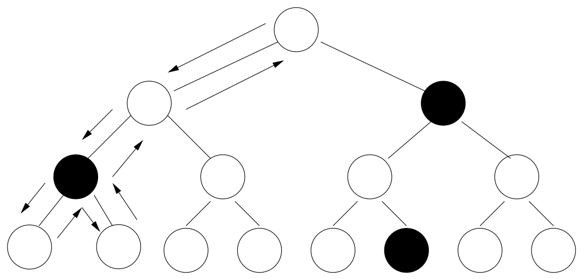


Figure 5.3: Search on a hybrid spill-tree.

An example of the hybrid search is shown in Fig. (5.3). The white nodes represent *overlapping* nodes, while black nodes are *non-overlapping* nodes. When the hybrid search algorithm hits a white node, it will either go left or right, depending on which side the query point is at, but will not backtrack. On the other hand, when the algorithm hits a black node, no matter which child node it searches first, it will also backtrack to search the other child. In Figure 5.3, the arrows indicates one possible route the hybrid search algorithm takes. As we can see, if the nodes at the top-levels are overlapping nodes, the hybrid search remains very efficient compared to MT-DFS.

Notice that we can control the hybrid by varying τ. If $\tau = 0$, we have a pure spill-tree with defeatist search — very efficient but not accurate enough; if $\tau \geq ||\mathsf{v.rpv} - \mathsf{v.lpv}||/2$, then every node is a non-overlapping node (due to the balance threshold mechanism) — in this way we get back to the traditional metric-tree with MT-DFS, which is perfectly accurate but inefficient. By setting τ to be somewhere in between, we can achieve a balance of efficiency and accuracy. As a general rule, the greater τ is, the more accurate and the slower the search algorithm becomes.

5.3.3 Further Efficiency Improvement Using Random Projection

By doing *Hybrid* search on a spill-tree, we can achieve very good efficiency as well as accuracy. In the best case, all nodes in a spill-tree are *non-overlapping* nodes, thus we can do *defeatist* search all the way down to a leaf node for any query, and the complexity per query is $O(\log n)$. With the increasing of the number of *non-overlapping* nodes, the speed of a spill-tree decreases. In the worst case, most of the nodes are *non-overlapping* nodes, then we need to do lots of backtracking, and the speed-up becomes marginal. Empirically, spill-tree performs the best when the dimension of the data points is relatively low (say less than 30), when the dimension increases, the advantage of spill-trees becomes less pronounced.

The hybrid spill-tree search algorithm is much more efficient than the traditional MT-DFS algorithm. However, this speed-up becomes less pronounced when the dimension of a data set becomes high (say, over 30). In some sense, the hybrid spill-tree search algorithm also suffer from the curse of dimensionality, only much less severely than MT-DFS.

However, a well-known technique, namely, *random projection* is readily available to deal with the high-dimensional datasets. In particular, the Johnson-Lindenstrauss Lemma (Dasgupta and Gupta [1999]) 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. Furthermore, the embedding is extremely simple: one simply picks a *random subspace* S and project all points to S. More mathematically, we state the Johnson-Lindenstrauss Lemma as the following. We use the version from Dasgupta and Gupta [1999].

Theorem 5.3.1 (Johnson-Lindenstrauss Lemma) *For any $0 < \tau < 1$ and any integer n, let k be a positive integer such that*

$$k \geq 4 \left(\frac{\tau^2}{2} - \frac{\tau^3}{3} \right)^{-1} \log n. \tag{5.3}$$

For any set V of n points in R^d, there is a map $f : R^d \rightarrow R^k$ such that for all $u, v \in V$,

$$(1-\tau)\|u-v\|^2 \leq \|f(u) - f(v)\|^2 \leq (1+\tau)\|u-v\|^2. \tag{5.4}$$

Further this map can be found in randomized polynomial time.

In fact, the proof in (Dasgupta and Gupta [1999] gives a slightly stronger (and more useful) result:

Given a point x and a point set Y, we use $\mathsf{dist}(x, Y)$ to denote the minimal distance between x and elements in Y. In other words, $\mathsf{dist}(x, Y) = \min_{y \in Y} \|x - y\|$.

Theorem 5.3.2 (Random Projection) *Let $\mathcal{T} = \{x_1, x_2, ..., x_n\}$ be a set of points in R^d and let q be a query point in R^d. Let f be a random projection of these points to a k-dimensional space, where $k \geq \frac{12}{\tau^2} \log \left(\frac{2n}{\delta}\right)$.*

We denote $\tilde{x}_i = f(x_i)$, $\widetilde{\mathcal{T}} = \{\tilde{x}_1, ..., \tilde{x}_n\}$, and $\tilde{\mathsf{q}} = f(\mathsf{q})$. Let x_i be the nearest neighbor of q in $\mathcal{T}$ and $\tilde{x}_j$ be the nearest neighbor of $\tilde{\mathsf{q}}$ in $\widetilde{\mathcal{T}}$. Then with probability at least $1 - \delta$, we have

$$\|\mathsf{q} - x_j\| \leq (1+\tau)\|\mathsf{q} - x_i\| \tag{5.5}$$

Proof: We apply the stronger version of the Johnson-Lindenstrauss lemma to each pair (q, x_i), for $i = 1, 2, ..., n$. We know that for every i, the probability that $\|\tilde{q}, \tilde{x}_i\|$ is within $(1 \pm \tau)$ of $\left(\frac{k}{d}\right) \|\mathsf{q}, \tilde{x}_i\|$ is at least $1 - \frac{\delta}{n}$. Using the union bound, we know that with probability at least $1 - \delta$, we have

$$\frac{1}{1+\tau}\left(\frac{k}{d}\right)\|\mathsf{q} - x_i\|^2 \leq \|\tilde{\mathsf{q}} - \tilde{x}_i\|^2 \leq (1+\tau)\left(\frac{k}{d}\right)\|\mathsf{q} - x_i\|^2.$$

We assume that this is the case. But then, we have

$$\|\mathsf{q} - x_j\| \leq \sqrt{1+\tau}\|\tilde{\mathsf{q}} - \tilde{x}_j\| \leq \sqrt{1+\tau}\|\tilde{\mathsf{q}} - \tilde{x}_i\| \leq (1+\tau)\|\mathsf{q} - x_i\|.$$

∎

In our $(1 + \varepsilon)$-NN search algorithm, we use random projection as a *pre-processing* step: project the data points to a subspace of lower dimension, and then do the hybrid spill-tree search. Both the construction of sp-tree and the search are conducted in the low-dimensional subspace. Naturally, by doing random projection, we will lose some accuracy. But we can easily fix this problem by doing *multiple rounds* of random projections and doing one hybrid sp-tree search for each round. Assume the failure probability of each round is δ, then by doing L rounds, we drive down this probability to δ^L.

5.4 Experimental Results

We report our experimental results based on spill-trees search on a variety of real-world data sets, with the number of data points ranging from 20,000 to 275,465, and dimensions from 60 to 3,838. The first two data sets are same as the ones used in Gionis et al. [1999], where it is demonstrated that LSH can have a significant speed-up over SR-trees.

Table 5.1: Five real-world data sets.

Data Set	**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 data sets in Table (5.4).

We perform 10-fold cross-validation on all data sets. We measure the *CPU time* and *accuracy* of each algorithm. To measure accuracy, we use the *effective distance error* (Gionis et al. [1999]), 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 q to the NN found by the algorithm, and d^* is the distance from 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 (5.2).

Table 5.2: 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 k-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 data set to produce approximate answers. We also examine the speed-up of spill-trees over other algorithms. In particular, the CPU time and the speed-up of spill-trees searches over LSH and metric-tree are summarized in Table (5.3) and (5.4) separately.

Table 5.3: CPU time(s) of Spill-tree and its speed-up (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 5.4: CPU time speed-up of a spill-tree 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

5.5 Parameter Estimations

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

- **Random Projection d':** As stated earlier, one crucial pre-processing step of the spill-trees algorithm is *random projection*. Although d' has a theoretical value, a practically useful d' can vary a lot depending on different data sets. 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 q falls into are searched. Similarly, spill-tree algorithms perform independent random projections L times and then do the search on L different projected data sets. By increasing L, we can easily boost the probability that the algorithm finds a $(1 + \varepsilon)$-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 τ:** 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 τ (the overlap buffer size), one can speed up 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 τ boosts the probability of finding a $(1 + \varepsilon)$-NN successfully, but many more points will be searched.

Now we pick one data set "Aerial" from section 5.4, and we test $\{d',\ L,\ \tau\}$ separately to understand the behavior of spill-trees in more detail.

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 τ to be ∞, 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 Fig. (5.4).

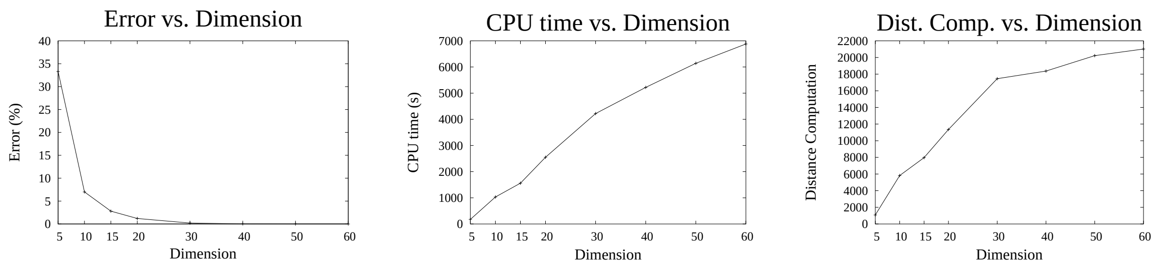


Figure 5.4: Influence of the random projection (Aerial, $d = 60$, $n = 275,476$, $k = 1$).

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 sub-linear 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 sub-linear and the number of distance computation drops dramatically when $d < 30$) in all five data sets used in section 5.3. This observation suggests that for metric-tree, the "curse of dimensionality" occurs when d' exceeds 30. As a consequence, searching algorithms based on metric-tree (including the spill-trees search) performs much better when $d < 30$. Empirically, for all five data sets 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 Fig. (5.5). As expected, in all cases, the CPU time scales 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

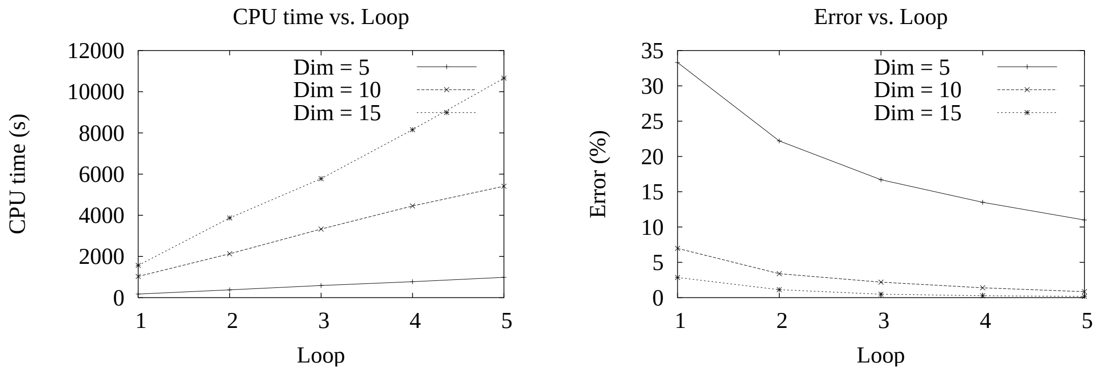


Figure 5.5: Influence of the loop L (Aerial, $d = 60$, $n = 275,476$, $k = 1$).

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 τ We test the influence of overlapping size τ. Again, we fix $L = 1$ and test three projected dimension: $d' = 5, 10, 15$. The results are shown in Fig. (5.6).

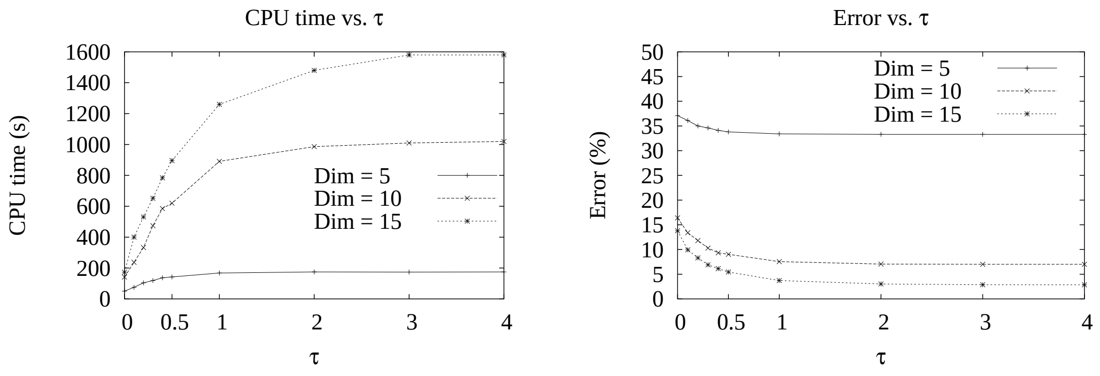


Figure 5.6: Influence of overlapping size τ (Aerial, $d = 60$, $n = 275,476$, $k = 1$).

As we have mentioned before, both the accuracy and the CPU time increases with τ. When $\tau = 0$, we have pure defeatist search; when τ is large enough (in this case, when $\tau > 3$), the search becomes MT-DFS, which is slow but perfectly accurate.[1] 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.

Parameter estimation from data set

Below we describe an algorithm to estimate the ideal random projection dimension d' and overlap buffer size τ, which is then relaxed in practice (by making the buffer smaller) to improve speed. For the rest of this section, we fix the number of loop L to 1.

Now let's focus on d' and τ. Ideally, d' should be close to the effective(intrinsic) dimension of the data set, and it is usually much smaller than the original dimension of the data set. Let R_S denote the average distance (averaged over the objects in set S) to their nearest neighbors. Following the heuristic described in (Clarkson [To appear]), if we make the approximation that points are uniformly distributed, we expect that the number of objects falling within a certain radius of a given object is proportional to the density of the objects (which is in turn proportional to the number of samples N_S), raised to the power of the dimensionality of the manifold d' on which the points are distributed. In particular, if we fix the expected number of points to 1, then the radius of interest is R_S, giving the following equation:

$$1 \propto N_S \cdot {R_S}^{d'} \tag{5.6}$$

Isolating R_S gives the following relationship, introducing a proportionality constant c:

$$R_S = \frac{c}{{N_S}^{1/d'}} \tag{5.7}$$

Now we can estimate the constant c and the effective dimensionality d' via random sampling. First we generate a number of different sized subsets of the data. In our experiments, our sample size ranges from 10 to 1000. For each of these sets, we can find the nearest neighbor of each point by computing all ${N_S}^2$ distances, and recording the average distance to the nearest neighbor of each point. By taking the log of both sides of Eq. (5.7), we get

$$\log R_S = \log c - \frac{1}{d} \log N_s \tag{5.8}$$

[1]The "residue" errors in Fig. (5.6) in the case $\tau = 4$ come from random projection (c.f. Fig. (5.4)).

From Eq. (5.8), it becomes obvious that c and d' can be estimated through standard linear regression methods.

Plugging these values along with the full sample set size into Eq. (5.7), we arrive at an estimate of the average nearest neighbor distance over the whole set.

At first it might appear that we should set τ to R_S. However, we need to take into account that the partition hyper planes are unlikely to be perpendicular to the vector between objects which are R_S apart. According to the Johnson-Lindenstrauss lemma (Johnson and Lindenstrauss. [1984]), after randomly projecting a point from the effective dimensionality d' of the samples down to the one dimensional space normal to the partitioning hyper plane, the expected distance will be as follows:

$$2\tau = \frac{R_S}{\sqrt{d'}} \tag{5.9}$$

This yields an estimate of τ. As mentioned earlier, we usually use a smaller value for τ than what is computed above for greater efficiency, but the above procedure provides an efficient method to get close to the right value.

Eq. (5.9) also implies that τ only depends on R_S and d', so it should be a fixed value for a given data set, the idea of changing τ for each partition does not work very well for this reason.

5.6 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 $(1+\varepsilon)$-NN algorithms (Arya et al. [2002, 1998], Kushilevitz et al. [1998]), 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.

Here we give some theoretical justification on the defeatist (i-am-feeling-lucky) search. In particular, towards the end of this section, we prove the following theorem:

Theorem 1 *Assuming that the training set contains N points drawn independently from the normal distribution $N(0, \sigma^2)$. Let $0 < \epsilon < 1/2$ be a real number. Let q be a query point, also drawn randomly from the same normal distribution. If $N > 192 \cdot \Gamma\left(\frac{d-1}{2}\right) e^{8d} \cdot \log\left(\frac{4}{\epsilon}\right)$, then the probability that i-am-feeling-lucky search algorithm makes a mistake in the first round is at most ϵ.*

Some comments about the result.

1. The choice of the normal distribution is mostly for the convenience of analysis. In a high dimensional Euclidean space, A normal distribution isn't too different from a uniform distribution over a ball of appropriate size, but makes the analysis easier since it is continuous everywhere.
2. The coefficients are chosen in such a way to simplify the analysis as much as possible, and no attempts are made to optimize any of them.
3. The theorem only analyzes one step of the i-am-feeling-lucky search, for simplicity. However, due to the recursive nature of the search algorithm, one can easily extend the theorem to multiple steps, although at the expense of more complicated statements.

5.6.1 Backgrounds, Definitions, and Notations

We say X has a Normal (or Gaussian) distribution with parameters μ and σ, denoted by $X \sim N(\mu, \sigma^2)$, if

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} e^{-\frac{1}{2\sigma^2}(x-\mu)^2} \tag{5.10}$$

The distribution of a sum of two normally distributed independent variables X and Y with means and variance (μ_x, σ_x^2) and (μ_y, σ_y^2), respectively is another normal distribution

$$f_{X+Y}(u) = \frac{1}{\sqrt{2\pi(\sigma_x^2 + \sigma_y^2)}} e^{\frac{-(u-(\mu_x+\mu_y))^2}{2(\sigma_x^2+\sigma_y^2)}} \tag{5.11}$$

The χ^2 distribution. X has a χ^2 distribution with d degrees of freedom – written $X \sim \chi_d^2$, if

$$f(x) = \frac{1}{\Gamma(d/2)2^{d/2}} x^{(d/2)-1} e^{-x/2}, \;\; x > 0 \tag{5.12}$$

If $Z_1, \ldots, Z_d$ are independent standard Normal random variables then $\sum_{i=1}^{d} Z_i^2 \sim \chi_d^2$.

Multivariate Normal. The univariate Normal has two parameters, μ and σ. In the multivariate version, μ is a vector and σ is replaced by a positive matrix $\sum$. Here, we only consider a very simple case. Let $Z = (Z_1, \ldots, Z_d)^T$, where $Z_1, \ldots, Z_d \sim N(0, 1)$ are independent. The density of Z is

$$\begin{aligned} f(z) &= \prod_{i=1}^{d} f(z_i) = \frac{1}{(2\pi)^{d/2}} \exp\{-\frac{1}{2}\sum_{j=1}^{d} z_j^2\} && (5.13) \\ &= \frac{1}{(2\pi)^{d/2}} \exp\{-\frac{1}{2} z^T z\} && (5.14) \\ & && (5.15) \end{aligned}$$

We say that Z has a standard multivariate Normal distribution, written as $Z \sim N(0, I)$ where it is understood that 0 represents a vector of d zeroes and I is the $d \times d$ identity matrix.
Further more, we have $X = (Z - \mu)^T \sum^{-1}(Z - \mu) \sim \chi_d^2$. In our case, $X = Z^T \cdot Z \sim \chi_d^2$.

5.6.2 A Simple Probabilistic Problem

Here we consider a simple probabilistic problem that will help us prove the theorem.
Say there are N d-dimensional vectors $\mathcal{V} = \{V_1, V_2, ..., V_N\}$, each independently chosen from normal distribution $N(0, \sigma^2 \cdot I)$. For computational convenience, we further assume the variance $\sigma^2 = 0.5$. Let q be the query point, independently drawn from the same distribution as well. We are interested in the probability P of $\mathsf{q}'s$ nearest neighbor having a distance advantage of at least ξ (ξ is a positive constant number) over all other data points to q, and the reason will become clear in the next section. Generally speaking, P is hard to compute since it is difficult to compute the distribution of $\mathsf{q}'s$ nearest neighbor directly, but we can approximate P via computing the probability of each $V_i \in \mathcal{V}$ having a distance advantage ξ over other data points in $\mathcal{V}$, and the union of all P_i provides a pretty good upper bound of P. Furthermore, since all $V_i \in V$ has equal probability to become the nearest neighbor or q, we only need to focus on any one data point in V, say V_1, and P_1 can be formalized as the following probability.

$$\begin{aligned} & \|V_1 - \mathsf{q}\|^2 < \|V_2 - \mathsf{q}\|^2 - \xi \\ \wedge \quad & \|V_1 - \mathsf{q}\|^2 < \|V_3 - \mathsf{q}\|^2 - \xi \\ \wedge \quad & \cdots \\ \wedge \quad & \|V_1 - \mathsf{q}\|^2 < \|V_N - \mathsf{q}\|^2 - \xi \end{aligned}$$

ξ is a positive constant number.

We can rewrite P_1 as $P_{d,\mathsf{q}}(N,\xi)$, then our task is to estimate $P_{d,\mathsf{q}}(N,\xi)$. We define a new set of random variable $\mathcal{Z} = \{Z_1,\ldots,Z_N\}$, where $Z_i = V_i - \mathsf{q}$, $0 \le i \le N$. It is easy to know from Eq. (5.11) that $Z_1,\ldots,Z_N \sim N(0,I)$, so $X_i = Z_i^T \cdot Z_i \sim \chi_d^2$, $1 \le i \le N$. And we know the p.d.f. $f(x)$ of the chi-square distribution from Eq. (5.12).
Further more, we use $F(x)$ to denote the c.d.f. of χ_d^2.

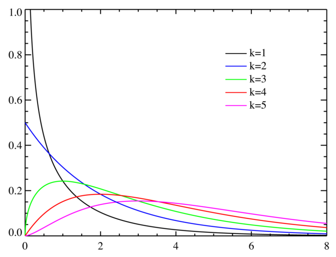


Figure 5.7: Probability density function.

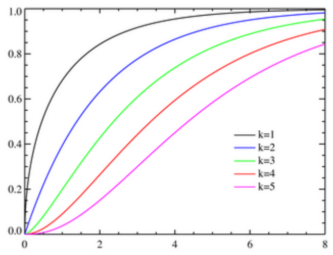


Figure 5.8: Cumulative distribution function.

Now we can write the probability $P_{d,\mathsf{q}}(V_1,N,\xi)$ as

$$P_{d,\mathsf{q}}(N,\xi) = \int_0^\infty (1 - F(x+\xi))^{N-1} f(x)dx \tag{5.16}$$

The intuition is that if the square distance of V_1 and q is x, where $x \ge 0$, then the square distance of V_1,..., V_N to q all have to be at least $x+\xi$, which happens with probability $(1 - F(x+\xi))^{N-1}$.

Now, let's estimate $P_{d,\mathsf{q}}(N,\xi)$ in Eq. (5.16). First, it is easy to see that

$$P_{d,\mathsf{q}}(N,0) = \int_0^\infty (1 - F(x))^{N-1} f(x)dx = \frac{1}{N} \tag{5.17}$$

Next, we assume d is large, and we estimate

$$R(x,\xi) = F(x+\xi) - F(x) = \int_x^{x+\xi} f(u)du \quad 0 \le x \le d$$

Lemma 1 *When $d > 10$, and $0 \le x \le d$, we have*

$$R_{d,\mathsf{q}}(x,\xi) \ge \frac{e^{-\frac{\xi}{4}} \cdot \xi^{\frac{d}{2}}}{2^d \Gamma(\frac{d}{2})} \tag{5.18}$$

Proof: If we take the derivative of $f(x)$, we can see that

$$f'(x) = \frac{-\frac{1}{2}e^{-\frac{x}{2}} \cdot x^{\frac{d}{2}-1} + (\frac{d}{2}-1)e^{-\frac{x}{2}} \cdot x^{\frac{d}{2}-2}}{2^{\frac{d}{2}}\Gamma(\frac{d}{2})}$$

setting $f'(x) = 0$, we know that $f(x)$ achieves its maximum at $x = d-2$. So, it is easy to see that $R(x,\xi)$ is minimized at $x = 0$. Now the problem boils down to bounding $R(0,\xi)$ from below. By standard calculus, we have

$$\begin{aligned} R(0,\xi) &= \int_0^{\xi} f_(u)du \\ &\geq \frac{\xi}{2} \cdot f(\frac{\xi}{2}), \quad for\ 0 \leq x < d \\ &= \frac{\xi}{2} \cdot \frac{e^{\frac{-\xi}{4}}\ (\frac{\xi}{2})^{\frac{d}{2}-1}}{2^{\frac{d}{2}}\ \Gamma(\frac{d}{2})} \\ &= \frac{e^{-\frac{\xi}{4}} \cdot \xi^{\frac{d}{2}}}{2^d\Gamma(\frac{d}{2})} \end{aligned}$$

The lemma is proved. ▌

Now we are ready to bound the probability $P_{d,\mathsf{q}}(N,\xi)$.

Lemma 2 *Assuming that d is large, $\epsilon < 1/2$, and $N > \frac{2^d e^{\frac{\xi}{4}}\Gamma(\frac{d}{2})}{\xi^{\frac{d}{2}}} \cdot \log(\frac{d}{\epsilon})$*

$$P_{d,\mathsf{q}}(V_1, N, \xi) < \frac{2\epsilon}{N}. \tag{5.19}$$

Proof: Notice that

$$\begin{aligned} P_{d,\mathsf{q}}(N,\xi) &= \int_0^{\infty} (1 - F(x+\xi))^{N-1} f(x)dx \\ &= \int_0^{d} (1 - F(x+\xi))^{N-1} f(x)dx + \int_d^{\infty} (1 - F(x+\xi))^{N-1} f(x)dx \end{aligned}$$

Notice that we split the sum into two parts with d as a threshold. Let's focus on the first part

$$\begin{aligned}
\int_0^d (1-F(x+\xi))^{N-1} f(x)dx &= \int_0^d (1-F(x))^{N-1} \cdot \left(\frac{1-F(x+\xi)}{1-F(x)}\right)^{N-1} \cdot f(x)dx \\
&= \int_0^d (1-F(x))^{N-1} \cdot \left(1-\frac{F(x+\xi)-F_{d,\mathsf{q}}(x)}{1-F(x)}\right)^{N-1} \cdot f(x)dx \\
&\leq \int_0^d (1-F(x))^{N-1} \cdot (1-R(x,\xi))^{N-1} \cdot f(x)dx
\end{aligned}$$

By Lemma 1, we have

$$\begin{aligned}
& \int_0^d (1-F(x))^{N-1} \cdot (1-R(x,\xi))^{N-1} \cdot f(x)dx \\
\leq\ & \int_0^d \left(1-\frac{e^{-\frac{\xi}{4}} \cdot \xi^{\frac{d}{2}}}{2^d\Gamma(\frac{d}{2})}\right)^{N-1} \cdot (1-F(x))^{N-1} \cdot f(x)dx \\
\leq\ & \left(1-\frac{e^{-\frac{\xi}{4}} \cdot \xi^{\frac{d}{2}}}{2^d\Gamma(\frac{d}{2})}\right)^{N-1} \cdot \int_0^d (1-F(x))^{N-1} \cdot f(x)dx \\
\leq\ & \frac{1}{N} \cdot \left(1-\frac{e^{-\frac{\xi}{4}} \cdot \xi^{\frac{d}{2}}}{2^d\Gamma(\frac{d}{2})}\right)^{N-1}
\end{aligned}$$

So if we let $N > \frac{2^d e^{\frac{\xi}{4}} \Gamma(\frac{d}{2})}{\xi^{\frac{d}{2}}} \cdot \log(\frac{d}{\epsilon})$, we have

$$\int_0^d (1-F(x))^{N-1} \cdot (1-R(x,\xi))^{N-1} \cdot f(x)dx < \frac{\epsilon}{N}. \tag{5.20}$$

Then, for the second part, we use a property of χ^2 distribution, namely the median of $F(x)$ is approximately $d - \frac{2}{3}$ when $d > 0$. So we have $1 - F(x) < 1/2$, for $x > d$. Now we have

$$\begin{aligned}
& \int_d^\infty (1 - F(x+\xi))^{N-1} \cdot f(x)dx \\
\leq & \int_d^\infty (\frac{1}{2})^{N-1} \cdot f(x)dx \\
= & (\frac{1}{2})^{N-1}) \int_d^\infty f(x)dx \\
\leq & (\frac{1}{2})^N
\end{aligned}$$

So we have

$$\int_d^\infty (1 - F(x+\xi))^{N-1} \cdot f(x)dx < \frac{1}{2^N} < \frac{\epsilon}{N} \tag{5.21}$$

For $N > 4\log(1/\epsilon)$ and $\epsilon < 1/2$.

Combining Eq.(5.20) and Eq.(5.21) gives us the proof. ▌

5.6.3 The Proof

Now we are ready to prove Theorem 1.

Proof: (to Theorem 1).

Suppose we use the spill-tree to do the i-am-feeling-lucky search. Let's just focus on one step of the decision.

See Fig. (5.6.3). It shows the situation for one step of the search. Suppose the algorithm is investigating node v and the hyper plane L is the decision boundary. Without loss of generosity, we assume that it is perpendicular to the x_1 axis[2]; the hyper planes LL and LR are parallel to L and distance τ from L. WLOG we assume that q is to the right of L. We say that the algorithm makes a mistake, if it prunes the true NN of q in this step. Notice that this can only happen if the true NN of q (denote it by Y) is to the left of plane LL. Now consider the area between L and LR that are of distance at most $\tau/2$ to L (the shaded area in Fig. (5.6.3)).

Let's assume that there are M points in this region, denoted by X_1, X_2, ..., X_M. An easy application of the Chernoff bound Motwani and Raghavan [1995] shows that when

[2]Notice that L_2 norm is invariant under rotation.

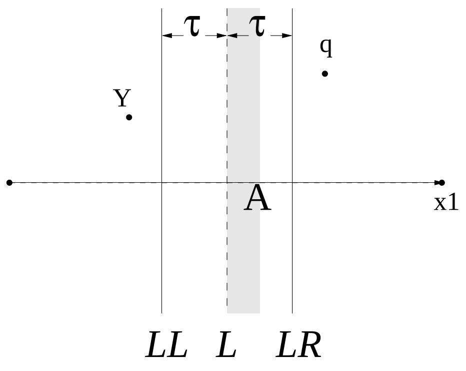


Figure 5.9: I-am-feeling-lucky search in a spill-tree.

$N > 192 \cdot \Gamma\left(\frac{d-1}{2}\right) e^{8d} \cdot \log\left(\frac{4}{\epsilon}\right)$, then $M > N/8 > 24 \cdot \Gamma\left(\frac{d-1}{2}\right) e^{8d} \cdot \log\left(\frac{4}{\epsilon}\right)$ with probability at least $1 - e^{-\frac{\tau N}{9}} > 1 - \frac{\epsilon}{4}$. From now on we assume that this is indeed the case.

Let's compare the distances between q and Y and q and the X_i's, we see that

$$\|\mathsf{q} - Y\|^2 - \|\mathsf{q} - X_k\|^2 \;=\; \sum_{i=1}^{d} \left((q^i - y^i)^2 - (q^i - x_k^i)^2\right)$$

Notice that $x_i^1 - y^1 > \tau$, $q^1 - y^1 > \tau$, and $q^1 - x_i^1 > -\tau/2$. Thus, we have

$$(q^1 - y^1)^2 - (q^1 - x_i^1)^2 \;=\; (x_i^1 - y^1)(q^1 - y^1 + q^1 - x_i^1) \geq \tau \cdot (\tau - \tau/2) = \tau^2/2.$$

In other words, Y has a "disadvantage" of $\tau^2/2$ against points X_1, X_2, ..., X_M on the dimension x_1.

In all other dimensions, however, Y and X_1, X_2, ..., X_M are independently and identically distributed. This is because that the distribution is invariant under rotation and also perfectly separable along different coordinates.

Therefore, for Y to be the true NN of q, it must have an "advantage" of at least $\tau^2/2$ in the other $d-1$ dimensions over points X_1, X_2, ..., X_M. But we know the probability of that from Lemma 2. Using the union bound, we know that for any query q, the probability that i-am-feeling-lucky search algorithm makes a mistake in the first round is at most $\epsilon/2$. Putting everything together and the theorem is proved. ▮

Chapter 6

Applications

In this section we discuss some real-world applications of our efficient algorithms. The applications expand in distinct areas from multimedia and image clustering to pharmaceutical drug discovery.

6.1 Video Segmentation

Video segmentation is an extensively studied problem in multimedia analysis, as shots are the most natural organizational unit for video content above the frame. Managing large-scale video repositories often necessitates processing at the shot-level to reduce computation and storage requirements. Hence, shot segmentation is a common first step in automatic and semi-automatic video management tools. The need for effective and efficient multimedia management tools has been exacerbated by recent trends. The confluence of decreasing storage costs, increasing processing power, and the growing availability of broadband data connections is producing rapid growth in both the size and number of personal and institutional video repositories.

A great deal of current video analysis research focuses on information retrieval within video databases. Web search companies are extending their text-based search capabilities to video assets. Video retrieval has also been the focus of the highly successful TRECVID workshops (Smeaton and Over [2002], Smeaton et al. [2003], Kraaij et al. [2004]). Within TRECVID, shot boundary detection is the most basic task in the evaluation, and shots serve as the units for both higher-level semantic annotation and retrieval tasks.

Good overviews of existing techniques in video segmentation operating on both uncompressed and compressed video streams can be found in (Koprinska and Carrato [2001], Lienhart [2001], Hanjalic [2002]). For uncompressed data, basically, most algorithms are based on frame differences for pixel, block-based or histogram comparisons. Most existing methods rely on suitable thresholding of differences between successive frames. However, these thresholds are typically highly sensitive to the specific type of video. There have only been a few machine learning approaches that tried to overcome this drawback. Gunsel et al. [1998a] views temporal video segmentation as a 2-class clustering problem ("scene change" and "no scene change") and uses K-means to cluster frame differences. Boreczky and Wilcox [1998] applies HMMs with separate states to model shot cuts, fades, dissolves, pans and zooms. Lienhart [2001] proposes a reliable dissolve detector. Hanjalic [2002] provides a statistical detector based on minimization of the average detection-error probability for cuts and dissolves.

As pioneered by the above methods, classification methods appear promising for video segmentation. However, most existing shot detection algorithms just use ad hoc frame classification with arbitrary thresholding rules. In our work, we employ an analytical framework encompassing a number of techniques that can perform reliable and efficient shot boundary detection. The key components to our algorithms can be summarized as following:

- Intermediate features extraction: transforms a time series problem to a supervised classification problem.
- KNS2: an efficient k-NN classification algorithm.
- Information-theoretic feature selection: exploits the discriminative power of different features, further improve algorithm performance.

In the following section, we will discuss the three key components in detail and show experiment results after that.

6.1.1 Feature Extraction

A. Low-level feature extraction There are numerous choices of low-level features that can be extracted to represent each time sample. One major distinction is between methods that model global (entire frame) pixel intensities directly, or those that operate on image sub-blocks. Most often, in either case, statistical measures or histograms are used to summarize the pixel values. Many color spaces have been used. Motion compensated features have been proposed, as well as more specialized features from the computer vision literature including edge or texture features, and estimates of object or camera motion. Finally, specialized features may include detectors of specific objects or phenomena such as faces or camera flashes.

B. Similarity analysis and kernel correlation After low-level features are extracted to represent each frame, shot boundary detection systems can be built based on quantifying local novelty with a longer source stream. Neighboring frames that are sufficiently different are declared to form a shot boundary. More generally, various pairs of frames within a local temporal neighborhood may be compared. This processing is readily visualized using matrices. First, an affinity or similarity matrix is generated, as in Fig. (6.1). We represent each frame n with a low-level feature vector V_n. Given a similarity measure d quantifying the similarity between pairs of feature vectors, we embed the similarity between every possible pair of frames features in the similarity matrix: $\mathbf{S}(i,j) = d(V_i, V_j)$. Thus, the number of rows and columns of $\mathbf{S}$ is the total number of frames, N, in the source video. Abrupt shot boundaries exhibit a distinct pattern in the similarity matrix. Frames in visually coherent shots have high (low) intra-shot (dis)similarity. Frames from two such shots that are adjacent in time generally show low (high) inter-shot (dis)similarity. This produces a checkerboard along the main diagonal of the similarity matrix whose crux is the diagonal element corresponding to the boundary frame.

This observation has motivated matched filter approaches to boundary detection. We refer to this method as kernel correlation. The matched filter is a square kernel matrix, $\mathbf{K}$, that represents the appearance of an ideal boundary in $\mathbf{S}$. To produce a quantitative frame-indexed novelty score, correlate $\mathbf{K}$ along the main diagonal of $\mathbf{S}$:

$$\nu(n) = \sum_{l=-L}^{L-1} \sum_{m=-L}^{L-1} \mathbf{K}(l,m)\mathbf{S}(n+l, n+m) \ . \tag{6.1}$$

Here, $\mathbf{K}$ is $2L \times 2L$. By varying the maximal lag L, the novelty score can be tuned to detect boundaries between segments of a specific minimum length. The correlation ν can be processed to detect segment boundaries. Maxima in this score correspond to locally novel frames, and are good candidate shot boundaries.

For the sake of brevity, we limit our review to systems that share elements in common with our approach. Here, we review several algorithms that are each characterized by a specific kernel used to generate a novelty score per $\nu(n)$. We emphasize the differences between the kernels in terms of their relative weighting of the elements of $\mathbf{S}$. Fig. (6.3) graphically depicts the kernels considered as square matrices. In each panel, a blank element does not contribute to the corresponding novelty score (i.e. $\mathbf{K}(l,m) = 0$ in Eq. (6.1). The elements

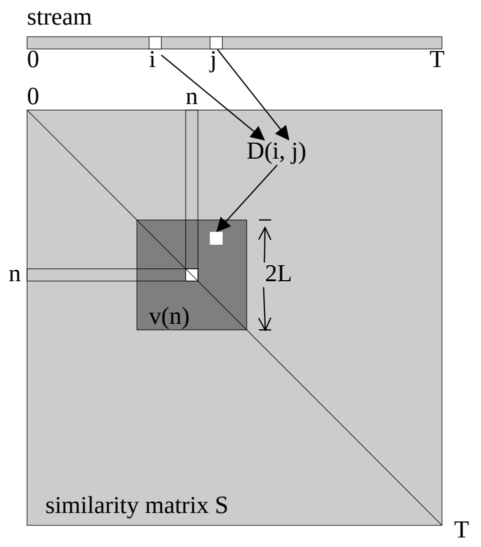


Figure 6.1: Diagram of the similarity matrix embedding.

containing solid circles contribute positively to the novelty score ($\mathbf{K}(l,m) > 0$). The elements containing unfilled circles contribute negatively to the novelty score ($\mathbf{K}(l,m) < 0$). Notice that the elements along the main diagonal of $\mathbf{K}$ align with the main diagonal elements of $\mathbf{S}$ in the correlation, where $\mathbf{S}(n,n) = d(V_n, V_n) = 0$.

The results of comparing adjacent video frames appear in the first diagonal above (and below) the main diagonal, i.e. the elements $\mathbf{S}(n, n \pm 1)$. Scale space analysis (Witkin [1981]) is based on applying a kernel of the form shown in Fig. (6.3)(a). It uses a family of Gaussian kernels of varying standard deviation to calculate a corresponding family of novelty scores. Define the scale space (SS) kernel as:

$$\mathbf{K}_{SS}^{(\sigma)}(l,m) = \begin{cases} \frac{1}{Z(\sigma)} \exp\left(-\frac{l^2}{2\sigma^2}\right) & |l-m| = 1 \\ 0 & \text{otherwise} \end{cases} . \tag{6.2}$$

where $Z(\sigma)$ is a normalizing factor[1]. Scale-space kernel was used in Slaney et al. [2001] for video segmentation.

Pye et al. [1998] presented an alternative approach using kernels of the form of Fig. (6.3)(b).

[1]The SS and DCS kernels are easily defined using a single variable, i.e. l, but we use the two variables (l,m) for consistency.

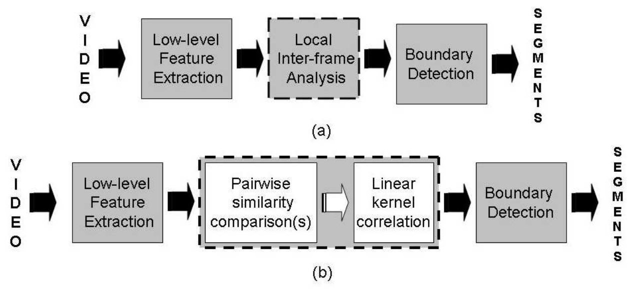

VIDEO
Low-level Feature Extraction
Local Inter-frame Analysis
Boundary Detection
SEGMENTS
(a)
VIDEO
Low-level Feature Extraction
Pairwise similarity comparison(s)
Linear kernel correlation
Boundary Detection
SEGMENTS
(b)

Figure 6.2: (a) A block diagram of a typical video segmentation system. (b) Decomposes local inter-frame analysis step into two separate steps.

When centered on a segment boundary, this kernel weights only elements of $\mathbf{S}$ that compare frames from different segments. This kernel is defined:

$$\mathbf{K}_{DCS}^{(L)}(l,m) = \begin{cases} \frac{1}{2L} & |l-m| = L \\ 0 & \text{otherwise} \end{cases} . \tag{6.3}$$

We refer to this kernel as the diagonal cross-similarity (DCS) kernel. In the correlation calculation, the elements of $\mathbf{S}$ for which $\mathbf{K}_{DCS} > 0$ lie on the L^{th} diagonal above (and below) the main diagonal of $\mathbf{S}$. $\mathbf{K}_{DCS}$ has been used in the segmentation systems by Pickering et al. [2002].

Weighting *all* the inter-segment elements implies the kernel of Fig. (6.3)(c). This kernel "includes" the DCS kernel, and adds the remaining between-segment (cross-similarity) terms within the kernel's temporal extent. The cross-similarity (CS) kernel is defined:

$$\mathbf{K}_{CS}(l,m) = \begin{cases} \frac{1}{2L^2} & l \geq 0 \text{ and } m < 0 \\ \frac{1}{2L^2} & m \geq 0 \text{ and } l < 0 \\ 0 & \text{otherwise} \end{cases} . \tag{6.4}$$

This kernel is precisely the matched filter for an ideal cut boundary in $\mathbf{S}$. Ideally, the inter-segment terms are maximally dissimilar, while the intra-segment terms will exhibit zero dissimilarity.

The kernel in Fig. (6.3)(d) is the full similarity (FS) kernel used in Cooper and Foote [2001], and it includes both between-segment and within-segment terms. This kernel replaces the zero elements in $\mathbf{K}_{CS}$ with negative weights. The negative weights penalize

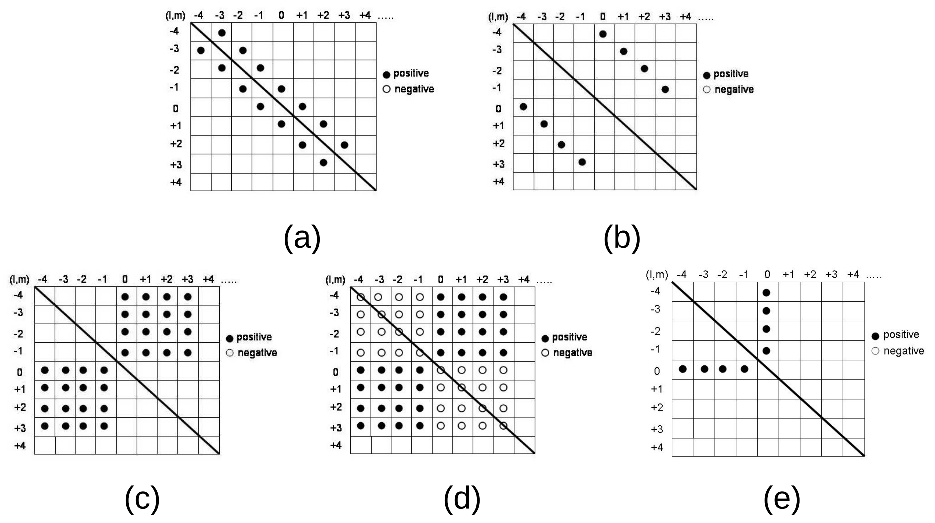


Figure 6.3: Different kernels for segment boundary detection via kernel correlation ($L = 4$). (a) scale-space kernel (b) diagonal cross-similarity (c) cross-similarity (d) full-similarity (e) row-similarity.

high within-segment dissimilarity:

$$\mathbf{K}_{FS}(l,m) = \begin{cases} \frac{1}{2L^2} & l \geq 0 \text{ and } m < 0 \\ \frac{1}{2L^2} & m \geq 0 \text{ and } l < 0 \\ -\frac{1}{2L^2} & \text{otherwise} \end{cases} . \tag{6.5}$$

Fig. (6.3)(3) shows the row (ROW) kernel used by (Qi et al. [2003]). This kernel weights only comparisons between the current frame and previous frames: $S(n, n-l)$:

$$\mathbf{K}_{ROW}(l,m) = \begin{cases} \frac{1}{2L} & l = 0 \text{ and } m < 0 \\ \frac{1}{2L} & m = 0 \text{ and } l < 0 \\ 0 & \text{otherwise} \end{cases} . \tag{6.6}$$

C. Using intermediate features In the previous section, we have seen numerous researches focused on designing different kernels, and relied on suitable thresholding of the novelty score for boundary detection. However, these thresholds are typically highly sensitive to the specific type of video. Now, instead of using a single score $\nu(n)$, we construct a frame-indexed intermediate feature vector X_n for each frame n, such that X_n effectively represents the local temporal structure around n. Corresponds to each kernel

K, we can construct a vector X_n that contains the elements of S which are multiplied by non-zero weights $K(l,m)$ in the calculation of $\nu(n)$ in Eq. (6.1). For example, corresponds to kernel K_{FS} and $L = 5$, we can create a $2L \times 2L$ dimensional feature vector X_n as:

$$X_n = \Big[\quad \mathbf{S}(n-5,n-5) \ \cdots \ \mathbf{S}(n-5,n+4) \\ \mathbf{S}(n-4,n-5) \ \cdots \ \mathbf{S}(n-4,n+4) \\ \cdots \\ \mathbf{S}(n+4,n-5) \ \cdots \ \mathbf{S}(n+4,n+4) \Big]^T$$

6.1.2 KNS2 Based Fast k-NN Classification

The second key component of our system is the use of supervised classification for boundary detection. As we have described in the previous paragraph, we used similarity matrix to generate intermediate features to reduce shot boundary detection to temporal pattern classification. In our experiments, we consider detection of two types of transitions: cut (abrupt) and gradual. So the combination of rich intermediate features and statistical classification averts the need for unduly complex low-level frame features or processing, and we only need to do a very simple task as to classify each frame as one of the three classes: “normal”, “cut” or “gradual”. Since there exist many powerful binary classification tools, we can fit this three classes classification problem to a two stage binary classification problem. In the first stage, we classify each frame as either “cut” or “non-cut”. In the second stage, we take all the classified “non-cut” frames, and further classify them as “normal” or “gradual”. This procedure is depicted in Fig. (6.4).

Among all classification algorithms, we chose k-NNṪhe advantage of k-NN has been stated in early chapters. More specifically, here we use KNS2, since it is designed for problems in which one class is more frequent than the rest. In our case, the number of frames that are not part of a transition is substantially greater than the number of transition frames. In experimental testing using video data, speedups between a factor of 20 and 30 in run time over the naïve implementation of k-NN have been observed. This acceleration is crucial in the present context.

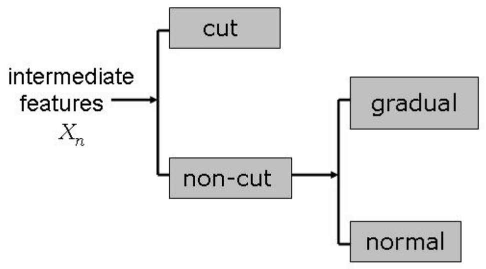


Figure 6.4: The classification process.

6.1.3 Information-theoretic Feature Selection

Since the two type of transitions (cut and gradual) appear to have different classification patterns, intuitively, we expect that specific inter-frame comparisons will be of differing relevance to detecting these two classes. Numerous existing systems that use solely adjacent frame comparisons of the form $\mathbf{S}(n, n \pm 1)$ achieve good cut transition detection. On the other hand, these features are not sufficient for robust gradual transition detection. Gradual detection requires analysis of frames over a greater temporal neighborhood, i.e. $\mathbf{S}(n, n \pm l), l > 1$. We greedily select the subsets of the elements of our intermediate feature vectors X that best discriminate among the transition classes. To determine these feature subsets, we calculate mutual information measures between the elements of the intermediate feature vectors and the corresponding class labels from our training sets.

The mutual information between two random variables quantifies the information that they share about one another. For discrete random variables, the mutual information is:

$$\begin{aligned} I(X(i);Y) &= \sum_{Y} P(Y) \sum_{X(i)} P(X(i)|Y) \log \left(\frac{P(X(i)|Y)}{P(X(i))} \right) \\ &= \langle D_{KL}(P(X(i)|Y) \| P(X(i))) \rangle_Y \quad . \end{aligned} \tag{6.7}$$

Here, $X(i)$ is the i^{th} element of the intermediate feature vectors $\mathbf{X}$, and Y denotes the class label as before.[2] The measure in Eq. (6.7) is referred to as marginal diversity and used for greedy feature selection in Vasconcelos [2003]. The latter form shows mutual information is the expected value of a Kullback-Leibler (KL) distance. The intuition is that informative

[2]To make the notation explicit, $X_n(i)$ is the i^{th} element of the feature vector associated with the n^{th} labeled frame in the training set. If $\mathbf{X}$ is a $P \times n$ matrix, then $1 \leq i \leq P$ and $X(i)$ appears in the i^{th} row of the matrix $\mathbf{X}$ of training data. We can also think of its observations as an $1 \times N$ vector.

features will exhibit high KL distance between the class-conditional distributions and the marginal distribution. The disadvantage of the greedy approach of Vasconcelos [2003] is that Eq. (6.7) does not account for inter-feature redundancies. Thus the resulting feature subset will not generally be maximally informative.

To account for inter-feature dependencies, more complicated mutual information forms must be calculated. Although mutual information is naturally extended to this case, its direct application can be computationally intractable. Denote the currently selected features by the set $\mathbf{X}^C$. Then, the next feature selected is

$$X^{(a)} = \underset{X(i) \notin \mathbf{X}^C}{\mathrm{ArgMax}}\, I(X(i); Y | \mathbf{X}^C) \tag{6.8}$$

Combinatorially, these forms of the mutual information are difficult to calculate as the cardinality of $\mathbf{X}^C$ increases. The main difficulty is estimation of quantities such as

$$P(X(i)|X^{(1)}, \cdots, X^{(a-1)})$$

Such density estimation quickly becomes computationally daunting, and is unreliable without massive amounts of training data. As a result, we seek approximations, and focus on second order terms of the form $I(X(i); Y|X^{(c)})$.

The approximation problem is studied in Vasconcelos and Vasconcelos [2004]. We approximate the relevant forms of the mutual information by neglecting higher order terms. To illustrate, consider the selection of the third feature given that $\mathbf{X}^C = \{X^{(1)}, X^{(2)}\}$ is known:

$$\begin{aligned} X^{(3)} &= \underset{X(a)}{\mathrm{ArgMax}}\, I(X(a), X^{(1)}, X^{(2)}; Y) & (6.9)\\ &= \underset{X(a)}{\mathrm{ArgMax}} \left(I(X^{(1)}; Y) + I(X^{(2)}; Y|X^{(1)}) + I(X(a); Y|X^{(1)}, X^{(2)}) \right) & (6.10) \end{aligned}$$

Already third order dependencies appear in $I(X(a); Y|X^{(1)}, X^{(2)})$. The complexity of the required mutual information forms grows with the size of $\mathbf{X}^C$. To simplify matters, we follow the assumption of *ℓ-decomposability* (Vasconcelos and Vasconcelos [2004]) for $\ell = 1$. This assumption states that:

$$I(X(a); Y|\mathbf{X}^C) = I(X(a); Y) + \sum_{X^{(c)} \in \mathbf{X}^C} \left[I(X(a); X^{(c)}|Y) - I(X(a); X^{(c)}) \right] \quad . \tag{6.11}$$

Neglecting terms with no dependence on $X(a)$ in Eq. (6.10), this implies that

$$X^{(3)} = \underset{X(a)\neq X^{(1)},X^{(2)}}{\text{ArgMax}} \Bigg(I(X(a);Y) + \left[I(X(a);X^{(1)}|Y) - I(X(a);X^{(1)}) \right] + \left[I(X(a);X^{(2)}|Y) - I(X(a);X^{(2)}) \right] \Bigg) \ . \tag{6.12}$$

More generally, we select a feature $X(a)$ to add to a previously selected set $\mathbf{X}^C$ such that,

$$X^{(a)} = \underset{X(i)\notin \mathbf{X}^C}{\text{ArgMax}} \Bigg(I(X(i);Y) + \sum_{X^{(c)}\in \mathbf{X}^C} \left[I(X(i);X^{(c)}|Y) - I(X(i);X^{(c)}) \right] \Bigg) \ . \tag{6.13}$$

The approximation states that the only critical feature interdependencies are pairwise. While this may not be wholly accurate in our context, it does significantly improve our resulting feature set by reducing inter-feature redundancy. The general framework of Vasconcelos and Vasconcelos [2004] can be used to systematically study the tradeoff between the computational complexity of assembling $\mathbf{X}^C$ and the corresponding performance gains in boundary detection. We focus here on the approximation of Eq. (6.13) which underlies the following greedy procedure for feature selection.

Algorithm 1 [Feature selection from Vasconcelos and Vasconcelos [2004]]

1. *Assume the availability of labeled training data* $\{\mathbf{X}, \mathbf{Y}\}$ *with binary labels* Y *and the desired size of the feature subset,* R.

2. *Select the first feature,* $X^{(1)}$ *by maximizing Eq. (6.7) over the elements of* $\mathbf{X}$*:*

$$X^{(1)} = \underset{X(a)}{\text{ArgMax}}\, I(X(a);Y) \ .$$

   Let $\mathbf{X}^C = \{X^{(1)}\}$.

3. *Select the second feature,* $X^{(2)}$ *as*

$$X^{(2)} = \underset{X(a)\neq X^{(1)}}{\text{ArgMax}} \Big(I(X(a);Y) + \left[I(X(a);X^{(1)}|Y) - I(X(a);X^{(1)}) \right] \Big) \ .$$

   Let $\mathbf{X}^C = \mathbf{X}^C \cup \{X^{(2)}\}$, *set* $r \leftarrow 3$.

4. *Select the* r^{th} *feature,* $X^{(r)}$ *as*

$$X^{(r)} = \underset{X(a)\notin \mathbf{X}^C}{\mathrm{ArgMax}}\left(I(X(a);Y) + \sum_{X^{(c)}\in\mathbf{X}^C} \left[I(X(a);X^{(c)}|Y) - I(X(a);X^{(c)}) \right] \right) .$$

   ] Let $\mathbf{X}^C = \mathbf{X}^C \cup \{X^{(r)}\}$, $r \leftarrow r+1$.

5. *Repeat step 4, terminating when* $\mathbf{X}^C$ *has the desired cardinality* R.

We apply KNS2 in two steps. In each step, we use different training sets. Thus, we apply Algorithm 1 separately to the two training sets, producing a feature subset optimized for each classification step. The training and test data sets are then projected to the appropriate lower-dimensional subspace prior to classification.

6.1.4 Experimental Results

In this section we present experimental results to validate the general approach and compare several specific system configurations.

Data Description

As noted in Bescos et al. [2005] and elsewhere, a longstanding obstacle to assessing progress in shot boundary detection has been the fact that most systems are validated on different data sets. Furthermore, many research groups do not wish to invest resources in generating ground truth segmentations for large test data sets. The TRECVID workshops (Kraaij et al. [2004]) represent a significant advance towards surmounting these issues. The 2004 workshop was the fourth annual open, metrics-based, large-scale evaluation of video analysis systems. Shot boundary detection is one of four tasks, and systems are evaluated on a common test set comprised of about six hours of broadcast news footage with manually labeled frame-level ground truth.

The TRECVID shot boundary detection task requires the detection of both abrupt and gradual transitions. Gradual transitions, such as fades, wipes, or dissolves, are exceedingly common in broadcast video, and substantially complicate shot boundary detection.

For testing, we use the data and evaluation protocol of TRECVID 2004 shot boundary determination task (Kraaij et al. [2004]). The test data is approximately 6 hours of broadcast news data produced by CNN and ABC from 1998. A manual ground truth segmentation is also provided in which shot boundaries are labeled as either cut or gradual transitions. The test data contained 618,409 total video frames with 2,774 cut transitions and 2,031 gradual transitions of various types. For the training data, we use the 2003 test set (Smeaton et al. [2003]) and ground truth segmentation. The 2003 data is news from CNN, ABC, and CSPAN from earlier in 1998. We removed 90% of the non-transition frames, and got a resulting training set with 63,822 labeled samples, among which, 2,489 frames are "cut" transitions, 22,074 frames are "gradual" transitions, and the rest are non-transition frames. The labeled training data is used to create two separate training sets corresponding to our two step classification process. In the first set, cuts are labeled positively and all other frames are labeled negatively. In the second training set, cuts are discarded, and gradual transition frames are labeled positively, while the non-transition frames are labeled negatively.

Feature Extraction

First, low-level features are computed from each frame. We use YUV color histograms, which are a simple and common feature parameterization (Gunsel et al. [1998b]). We compute 32-bin global frame histograms, and (sixteen) 8-bin block histograms using a 4×4 uniform spatial grid for each channel.

To construct intermediate features, we need to build the similarity matrix S, where each element $S(i, j)$ is the similarity between frame i and j. Denote the frame-indexed histogram feature data by $V = \{V_n : n = 1, \ldots, N\}$. We use the χ^2 distance as the similarity measures. Formally,

$$S(i, j) = d(V_i, V_j) = \frac{1}{2} \sum_k \frac{(V_i(k) - V_j(k))^2}{V_i(k) + V_j(k)}. \tag{6.14}$$

In practice, there is no need to calculate the whole similarity matrix. As shown in Eq. (6.1), when a kernel is applied to the matrix, for each frame n, we only care about its neighboring frames. A Lag parameter L determines the size of the kernel K, and $L << N$. Additionally, because both S and K are typically symmetric, many computations are redundant. For these reasons, we compute only a small portion of S near the main diagonal, and store the data in "lag domain" according to:

$$S_{lag}(n, l) = S(n, n + l) \quad n = 1, \ldots, N \; l = 1, \ldots, L. \tag{6.15}$$

The algorithmic complexity for calculating the required portion of S is $O(N)$. We generate two similarity matrices $S^{(G)}$ and $S^{(B)}$ corresponding to the global and block color histogram features respectively, and the final intermediate features for classification is formed by concatenating elements of $S^{(G)}$ and $S^{(B)}$.

Classification and Evaluation

We use KNS2 for k-NN classification. We control the sensitivity of the classifier using an integer parameter $\kappa : 1 \leq \kappa \leq K$. If at least κ out of the k nearest neighbors of the test vector X_n in the training data are from the "transition" class, we label frame n as a transition and otherwise label it as a non-transition. κ is varied to produce the performance curves trading off false-positive versus false-negative classification errors. Throughout, $k = 11$. The only post-processing is the application of simple temporal heuristics. We require detected transitions to be separated by at least 60 frames (2 seconds). In the event that multiple transitions are detected within a 60 frame interval, we retain the transition with the most positively labeled frames among its nearest neighbors breaking ties arbitrarily. We also require gradual transitions to have a minimum duration of 11 frames. For evaluation, we use the common figures of merit of precision and recall (Boreczky and Rowe [1996]):

$$\text{Precision} = \frac{\#(\text{Boundaries correctly detected})}{\#(\text{Total boundaries detected})} , \tag{6.16}$$

$$\text{Recall} = \frac{\#(\text{Boundaries correctly detected})}{\#(\text{Total ground truth boundaries})} . \tag{6.17}$$

These measures are computed over the test data sets using the TRECVID protocol and evaluation software (Smeaton and Over [2002]).

During the testing, we have also monitored the required computation time. The systems below all process 90 dimensional intermediate features ($L = 5$) for classification and training sets with cardinality between 55,000 and 65,000 frames, so that the run time of the various systems below are all similar. The bulk of the computation time is almost evenly divided between decoding the MPEG stream to extract individual frames and their corresponding histogram features, and the two k-NN classification steps. End to end, our system operates at about twice real-time. That is, the segmentation requires compute time equal to twice the duration of the input video [3].

[3]More details appear in Adcock et al. [2004], the machine used for testing has an Athlon 64 3500+ processor.

Similarity Feature Vector versus Novelty Scores

The goal of the first set of experiments is to compare choices for intermediate features based on different kernels, as described in Section 6.1.1. First, we set $L = 5$, and examine performance using the intermediate feature vector X_n as input to the k-NN classifier.

The results appear in the solid curves in Fig. (6.5)(a) shows FS SIM (X, "FS SIM"), the CS SIM ($\Box$, "CS SIM"), the SS SIM ($+$, "SS SIM"), the ROW SIM ($\triangle$, "ROW SIM"), and the DCS SIM ($\circ$, "DCS SIM"). The additional information in the FS features produce the best performance. Similarly, the CS features show the second best performance. The results exhibit a clear tradeoff between the dimensionality of the intermediate features and segmentation performance. The kernels with lower dimensionality (i.e. fewer non-zero terms in K, and hence lower dimensional X_n), perform worse.

For comparison, we produce novelty features by concatenating different novelty scores across a set of Lags $L = 2, 3, 4, 5$. For each L, we compute a frame-indexed kernel correlation score separately using $S^{(G)}$ and $S^{(B)}$, so that we have four scores for each frame for both the global and the block histogram features. We combine this data into a single 8×1 vector X_n to represent each frame n:

$$X_n = \left[\nu_2^{(G)}(n) \;\; \nu_3^{(G)}(n) \;\; \nu_4^{(G)}(n) \;\; \nu_5^{(G)}(n) \;\;\; \nu_2^{(B)}(n) \;\; \nu_3^{(B)}(n) \;\; \nu_4^{(B)}(n) \;\; \nu_5^{(B)}(n) \right]^T .$$

where $\nu_L^{(G)}$ denotes the novelty score computed using $\mathbf{S}^{(G)}$ with kernel width L, and $\nu_L^{(B)}$ denotes the novelty score computed using $\mathbf{S}^{(B)}$. In this case, the size of the intermediate vectors X_n is the same for all kernels (1×8, as in the equation above). However, varying numbers of inter-frame comparisons from S are used to compute the elements comprising X_n.

The results appear as dashed curves for the FS kernel (X, "FS KC"), the CS kernel($\Box$, "CS KC"), the SS kernel($+$, "SS KC"), the ROW kernel($\triangle$, "ROW KC"), and the DCS kernel ($\circ$, "DCS KC"). The best performance is achieved by the CS and the DCS kernels. As noted previously, the CS kernel is the matched filter for the expected pattern produced by cut segment boundaries in S. Both the CS and DCS kernels emphasize dissimilarity between segments at multiple time scales. The FS kernel performs somewhat worse, we believe due to the choice of the χ^2 dissimilarity measure. The FS kernel may be better suited to dissimilarity measures that take both positive and negative values such as the cosine similarity measure. Further details of system performance appear in Table (6.1), where we use P and R to denote precision and recall, respectively. The F1 is defined as

$F1 = (2 \cdot P \cdot R)/(P + R)$. Each row shows the results using the value of κ that maximizes the overall F-score for the corresponding system, the last row shows the mean results for all systems participating at TRECVID 2004 (TV MEAN).

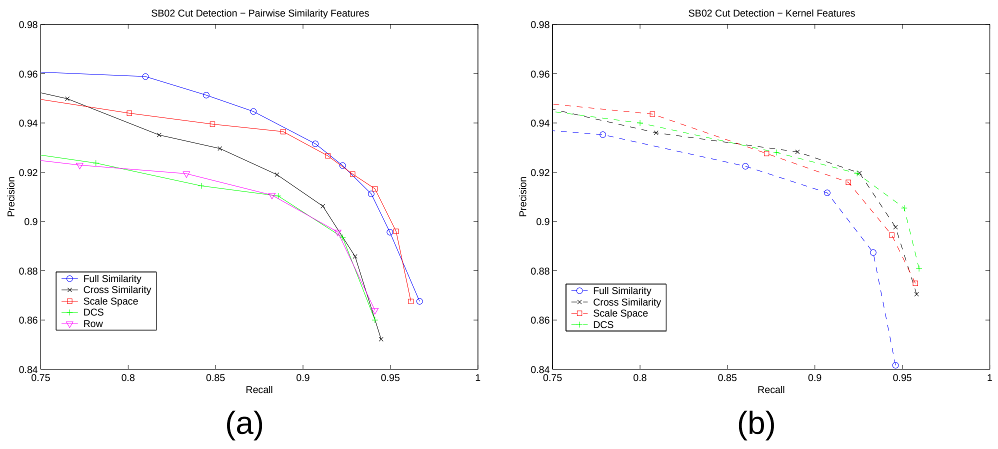
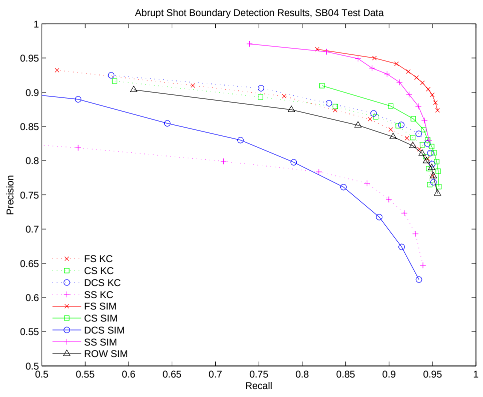
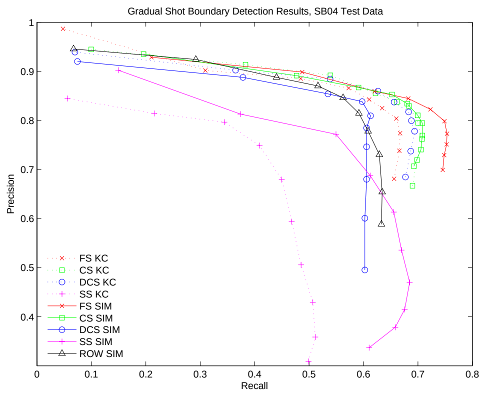


Figure 6.5: (a) Cut detection using raw similarity features. (b) kernel correlation features.

Fig. (6.6)(a) shows performance for abrupt and gradual boundary detection, providing further insights into the relative performance of these systems. We can see that systems using similarity feature performance better than using novelty scores in general. In particular, kernels that emphasize similarity comparisons between adjacent frames, $\mathbf{S}(n, n \pm 1)$, detect abrupt transitions well. For this reason, the SS SIM system outperforms the CS SIM system. However, the SS KC system performs poorly, because the critical comparisons between adjacent frames at a boundary are smoothed in the correlation. The DCS SIM system performs poorly as well, as it neglects terms comparing adjacent frames. Here, the FS SIM system performs best of all variations. Fig. (6.6)(b) shows gradual transition detection performance. Recall is generally lower than for abrupt transitions, which is a reflection of the relative difficulty of this task. In contrast to abrupt transitions, kernels which emphasize larger lag comparisons of the form $\mathbf{S}(n \pm l, n \pm m)$ for $l, m > 1$ perform best. These include the CS SIM and FS SIM systems, as well as the CS KC, DCS KC, and FS KC systems. The use of kernel correlation to smooth the pairwise similarity comparisons does not significantly hurt resolution as the transitions occur over a frame interval. Again, the FS SIM systems provides the best performance, due to its more complete representation of local temporal structure.



(a)



(b)

Figure 6.6: (a) Abrupt (cut) boundary detection (b)Gradual boundary detection.

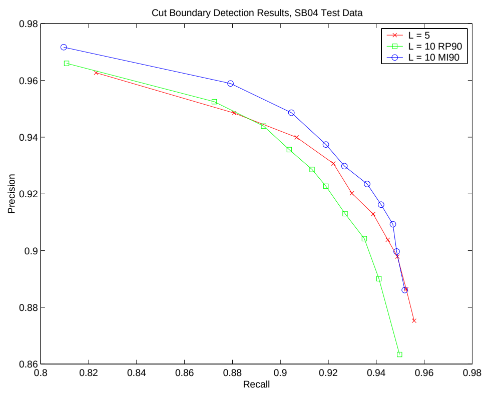
6.1.5 Feature Selection Experiments

We can conclude from the experiments in the previous section that building intermediate features for classification from the inter-frame comparisons corresponding to the FS kernel provides excellent performance. The relative performance of the various systems demonstrate that including additional comparisons in the intermediate features generally benefits performance. At the same time, increasing the dimensionality of the intermediate feature vectors increases the computation required for classification. The goal of this section is to explore this tradeoff between performance and complexity using feature selection. We focus on the FS SIM system with intermediate features $\mathbf{X}$ defined in Eq. (6.7). We first increase the lag parameter to $L = 10$, producing 380 inter-frame comparisons. While we expect that the performance would again improve with these additional intermediate features, the computational requirements are too great. We use feature selection to reduce dimensionality, while attempting to retain performance gains associated with the larger lag, L.

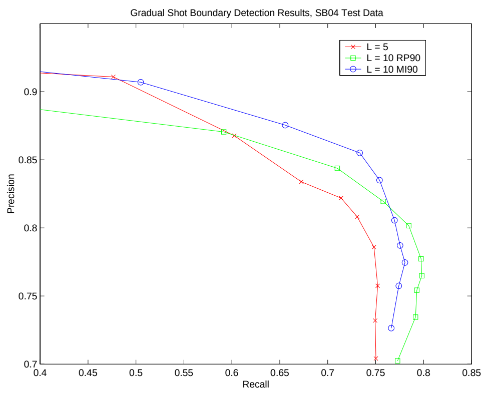
We compare two approaches to feature selection using the “FS SIM” system of the previous section as a baseline. For the baseline, the intermediate features $\mathbf{X}$ are generated using Eq. (6.7) with $L = 5$. Results appear in the curves of Fig. (6.7). The curves for this basic system are denoted (X “L=5”). Fig. (6.7)(a) and (b) shows performance in the detection of abrupt and gradual boundaries, respectively.

Table 6.1: Various systems tested for three class shot boundary detection.

Experimental results (best runs for each)									
	MEAN			ABRUPT			GRADUAL		
SYS	R	P	F1	R	P	F1	R	P	F1
ROW SIM	0.82	0.81	0.81	0.94	0.81	0.87	0.59	0.81	0.69
SS KC	0.74	0.72	0.73	0.87	0.77	0.82	0.47	0.59	0.52
SS SIM	0.76	0.90	0.83	0.86	0.95	0.91	0.55	0.77	0.64
DCS KC	0.85	0.83	0.84	0.93	0.84	0.88	0.6828	0.8172	0.7440
DCS SIM	0.731	0.7941	0.76	0.79	0.80	0.79	0.61	0.78	0.68
CS KC	0.85	0.83	0.84	0.93	0.83	0.88	0.68	0.83	0.75
CS SIM	0.85	0.85	0.85	0.94	0.85	0.89	0.65	0.85	0.74
FS KC	0.84	0.83	0.83	0.92	0.83	0.87	0.66	0.80	0.73
L=5	0.87	0.88	0.88	0.94	0.91	0.93	0.73	0.81	0.77
L=10 RP90	0.87	0.89	0.88	0.91	0.93	0.92	0.78	0.80	0.79
L=10 MD90	0.86	0.84	0.85	0.94	0.84	0.89	0.69	0.84	0.76
L=10 MI25	0.84	0.89	0.87	0.92	0.92	0.92	0.68	0.82	0.74
L=10 MI45	0.86	0.90	0.88	0.92	0.93	0.92	0.72	0.83	0.77
L=10 MI90	0.87	0.90	0.89	0.93	0.93	0.93	0.75	0.84	0.79
TV MEAN	0.73	0.73	0.71	0.83	0.76	0.78	0.50	0.58	0.57



(a)



(b)

Figure 6.7: (a) Cut boundary detection using feature selection. (b) Gradual boundary detection using feature selection.

We select feature subsets from FS similarity features $\mathbf{X}$ generated using Eq. (6.7) with $L = 10$, producing vectors of dimensionality of 380. The first feature selection method is random projection (RP) (Fradkin and Madigan [2003]). This approach generates a subspace for projection randomly with the constraint that it be orthogonal. The method is proven to preserve distances in the original high-dimensional space, and thus naturally complements nearest-neighbor methods. The second set of results with feature selection uses Algorithm 1 to select 90 dimensions from the same 380-dimensional intermediate features (○ "L=10 MI90"). Performance results also appear in Table (6.1).

Both the systems using feature selection do better than the original L=5 system overall, even though the dimensionality of the features used for classification is the same in all three cases (90). The system using information-theoretic feature selection performs best of the three. Fig. (6.7) reveals further differences among the systems. The L=5 system outperforms L=10 RP90 in cut boundary detection, while the opposite is true for gradual boundary detection. This reaffirms our intuitions. First, information critical to cut boundary detection is in the pairwise similarities closest to the boundary frame, while the extra information in the L=10 features is superfluous. Random projection mixes these features together which reduces the informativeness of the most valuable features for cut detection. In the gradual case, the additional features are crucial, and mixing them does not hurt accuracy, since gradual boundaries extend over a set of frames. Thus Fig. (6.7)(b) shows improved performance with random projection.

The L=10 MI90 systems exploits the discriminative power of the L=10 features. It selects the important features for cut detection by design, and demonstrates superior performance. For gradual boundary detection, it selects combinations of complementary features from the L=10 feature set. In this way, it is able to generally outperform the random projection system, although the random projection system performs best in the high recall and low precision region. We believe this is because random projection linearly combines all the features in the L=10 data, whereas Algorithm 1 only selects 90 of the features. Gradual transitions routinely extend over 20-40 frames, so it is likely that all the L=10 features contain useful information for gradual transition detection.

We further examine the results using information theoretic feature selection in Fig. (6.8). The plots compare the performance of L=10 MI90 and L=10 RP90 to three additional variations of the system. L=10 MD90 (x) shows performance using greedy feature selection based on the mutual information measure of Eq. (6.7), as in Vasconcelos [2003]. This approach ignores any redundancies in the feature data, and we see the resulting overall performance is poor. L=10 MI25 (□) and L=10 MI45 (*) show the results of applying Algorithm 1 to greedily build feature subsets of size 25 and 45, respectively. All the curves based on information-theoretic feature selection show gradual improvement with additional features. The L=10 MI90 system performs best. Table (6.1) also indicates that the L=10 MI45 system approximately matches the overall performance of the L=10 RP90 system using half as many features.

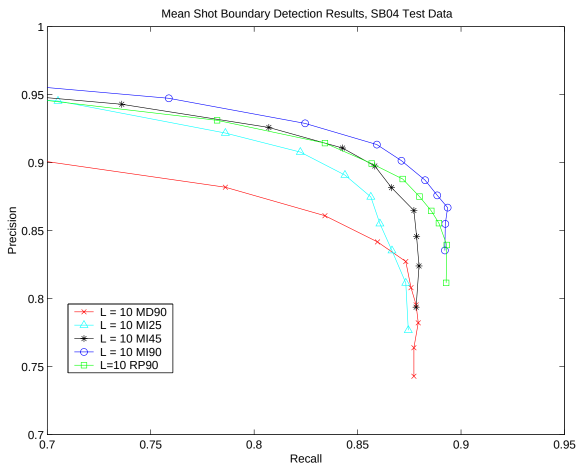


Figure 6.8: Mean performance for three-class shot boundary detection with feature selection.

6.2 Classification for Drug Screening

6.2.1 Problem description

Virtual screening refers to the use of statistical and computational methods for prioritizing candidate molecules for biological testing for their possible use as drugs. Because these assay are time-consuming and expensive, accurate “virtual” assay, or prioritization for molecules by computer, has direct impact in cost savings and more rapid drug development. Virtual screening, part of the more general enterprise of *high-throughput screening*, has thus become an increasingly pressing new component of modern drug development research.

The classification problem. We are concerned with the scenario of a large pharmaceutical research and development laboratory, which is as follows: We assume there is a single target molecule. There are multiple molecules which are known to interact in the desired fashion with the target molecule, i.e. are active with respect to the target, and a generally larger number of molecules known to be inactive with respect to the target. The task is to predict whether a previously unseen molecule will be active with respect to the target.

The features. The structure of a molecule determines its interaction with a target molecule - whether and how it will interlock, or “dock” with complete characterization remains an outstanding problem of science. Thus, thousands of binary (0/1) features, collecting all manner of both generic and target-specific properties which might be relevant to the classification task. Typical binary features record the absence or presence of a certain kind of atom or substructure, proximity relationship, and so on.

The goal. Our goal is to design a classifier with the best possible prediction performance based on a proprietary commercial training set of 26,733 molecules, 6,348 binary features, and one output variable (“active” or not).

Recent work in virtual screening. Most of the well-known classification methods have been proposed for the virtual screening problem, including logistic regression, naive Bayes classifiers, and support vector machines (SVM) (Vapnik [1995]), which are currently considered to be one of the most empirically successful in general.

Table 6.2: The ds1 data set and its variants.

Name	Attributes	Rows	Sparsity	Num Nonzero	Num Pos Rows
ds1	6,348	26,733	0.02199	3,732,607	804
ds1.100pca	100	26,733	1.00000	2,673,300	804
ds1.10.pca	10	26,733	1.00000	267,330	804

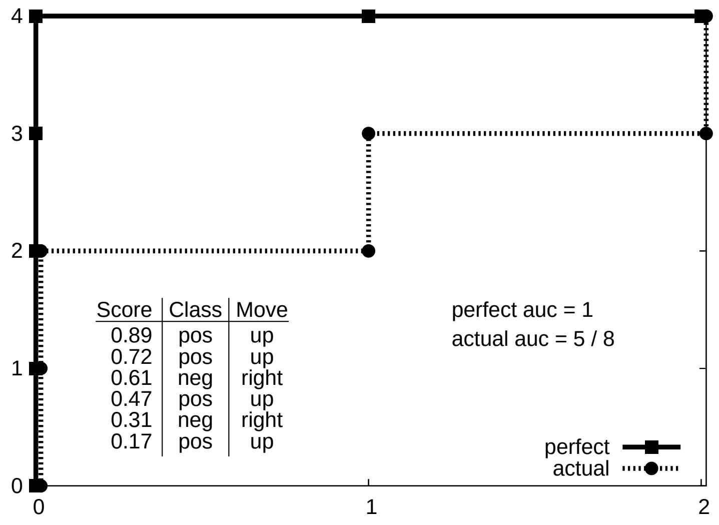
6.2.2 Experiments

We perform our experiments on a data set similar to the publicly available Open Compound Database and associated biological test data, as provided by the National Cancer Institute (Database [2000]). To make it consistent with Komarek [2004], we call the data set *ds1*, it contains 26,733 records and 6,348 attributes, and is sparse. It has 804 positive output values ("active" class). For comparison, we generate two variations of ds1. We perform PCA, keeping only 100 and 10 of these dimensions. In particular, the value 100 was chosen to correspond roughly to the inflection point of the eigenspectrum, as per common practice, and captured 97% of the variance in this case. We call the two variation *ds1.100pca* and *ds1.10pca* respectively. The dimensions and sparsity of the data sets are shown in Table (6.2).

All experiments were performed using 10-fold cross-validation, in which the data is performed on one of them while training is performed on the other 9 put together. The predictive performance of the experiments is measured using the Area Under Curve (AUC) metric, which is described below.

Before we describe AUC scores, we must first describe Receiver Operating Characteristic (ROC) curves (Duda and Hart [1973]). To construct an ROC curve, the data set rows are sorted according to the probability a row is in the positive class under the learned logistic model. Starting at the graph origin, we examine the most probable row. If that row is positive, we move up. If it is negative, we move right. In either case we move one unit. This is repeated for the remaining rows, in decreasing order of probability. Every point (x, y) on an ROC curve represents the learners "favorite" $x + y$ rows from the data set. Out of these favorite rows, x are actually positive, and y are negative.

Fig. (6.9) shows an example ROC curve. Six predictions are made, taking values between 0.89 down to 0.17, and are listed in the first column of the table in the lower-left of the graph. The actual outcomes are listed in the second column. The row with highest prediction, 0.89, belongs to the positive class. Therefore we move up from the origin, as written



Score	Class	Move
0.89	pos	up
0.72	pos	up
0.61	neg	right
0.47	pos	up
0.31	neg	right
0.17	pos	up

Figure 6.9: Example ROC curve.

in the third column and shown by the dotted line in the graph moving from (0,0) to (1,0). The second favorite row was positive, and the dotted line moves up again to (2,0). The third row, however, was negative and the dotted line moves to the right one unit to (2,1). This continues until all six predictions have been examined.

Suppose a data set had P positive rows and $R - P$ negative rows. A perfect learner on this data set would have an ROC curve starting at the origin, moving straight up to $(0, P)$, and then straight right to end at $(R - P, P)$. The solid line in Fig. (6.9) illustrates the path of a perfect learner in our example with six predictions. Random guessing would produce, on average, an ROC curve which started at the origin and moved directly to the termination point $(R - P, P)$. Note that all ROC curves will start at the origin and end at $(R - P, P)$ because R steps up or right must be taken, one for each row.

As a summary of an ROC curve, we measure the area under the curve relative to area under a perfect learners curve. The result is denoted AUC. A perfect learner has an AUC of 1.0, while random guessing produces an AUC of 0.5. In the example shown in Fig. (6.9), the dotted line representing the real learner encloses an area of 5. The solid line for the perfect learner has an area of 8. Therefore the AUC for the real learner in our example is 5/8.

Table 6.3: Classifier performance for each data set.

	ds1		ds1.100pca		ds1.10pca	
Classifier	Time	AUC	Time	AUC	Time	AUC
KNN K=1	424	0.790±0.029	74	0.785±0.024	9	0.753±0.028
KNN K=9	782	0.909±0.016	166	0.894±0.016	14	0.859±0.019
KNN K=129	2381	0.938±0.010	819	0.938±0.010	89	0.909±0.013
LR-CGEPS	86	0.949±0.009	44	0.918±0.011	8	0.846±0.013
LR-CGDEVEPS	59	0.948±0.009	35	0.913±0.011	9	0.842±0.015
CG-MLE	151	0.946±0.008	364	0.916±0.012	48	0.844±0.014
SVM LINEAR	188	0.918±0.012	130	0.874±0.012	68	0.582±0.048
SVM RBF	1850	0.924±0.012	1036	0.897±0.010	490	0.856±0.017
BC	4	0.884±0.011	8	0.890±0.012	2	0.863±0.015

Whereas metrics such as precision and recall measure true positives and negatives, the AUC measures the ability of the classifier to correctly rank test points. This is very important for data mining. We often want to discover the most interesting galaxies, or the most promising drugs, or the products most likely to fail. When presenting results between several classification algorithms, we will compute confidence intervals on AUC scores. For this we compute one AUC score for each fold of our 10-fold cross-validation, and report the mean and a 95% confidence interval using a T distribution.

We reuse the result from Paul Komarek's thesis (Komarek [2004]), where he compared four different algorithms: k-NN, logistic regressions, SVM and naive Bayes classifier. For k-NN, because of the high-dimensionality issue, and the obvious unbalanced classes, he used KNS2, and there are three different settings for the number of nearest-neighbors: k=1, k=9 and k=129. We also showed some brief observations of the ROC curves for these experiments. We have organized these curves by data set. Fig. (6.10), (6.11) and (6.12) correspond to ds1, ds1.100pca and ds1.10pca respectively. Each figure has two graphs. The graph to the left has a linear "False positives" axis, while the graph to the right has a logarithmic "False positives" axis. Each plot is built from as many data points as there are rows in the corresponding data set. For technical reasons these points are interpolated with cubic splines, and the splines are re-sampled where the bullets appear in the plots. One should not associate the bullets with the individual steps of the ROC curve.

The most interesting part of these figures is the left corner of the logarithmic plot. It is here that we see how accurate the classifiers are for the first few predictions. In the ds1 graphs, we see LR and SVM close together for their top five predictions, but SVM appears to have an edge over LR for initial accuracy. In the long run, we know that LR outperforms SVM as indicated by the AUC scores, and careful observation of the graphs confirms this. The ds1.100pca and ds1.10pca linear ROC curves show k-NN outperforming the other algorithms, though the initial segment does not show any clear leaders. The most important conclusion from the ROC curves is that the classifier with the best initial ranking performance might not be the most consistent classifier. In terms of running time, k-NN is not the fastest algorithm, however, its running time is acceptable comparing with other methods, except for the ds1 data set with k=129. Due to its simplicity and good performance for some cases, it can be used as a building block of a more sophisticated classifier, where a prediction is made based on the vote of a number of different classifiers.

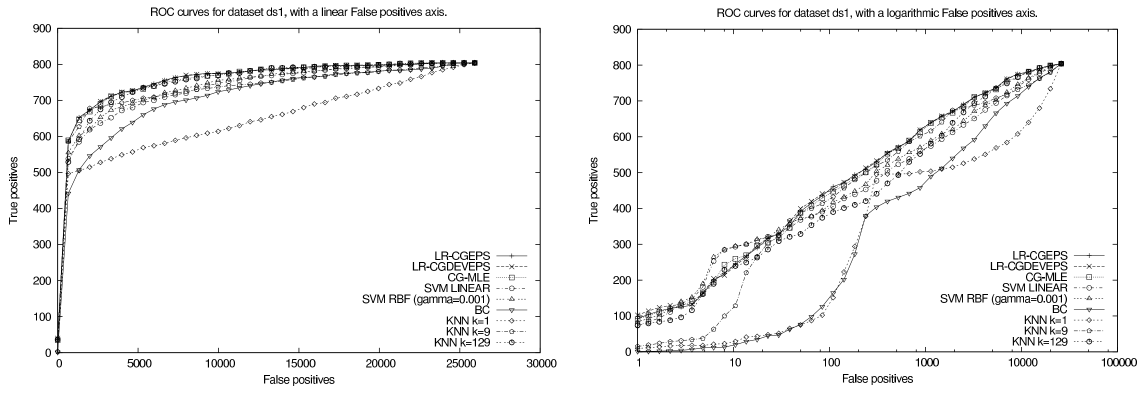


Figure 6.10: ROC curves for ds1.

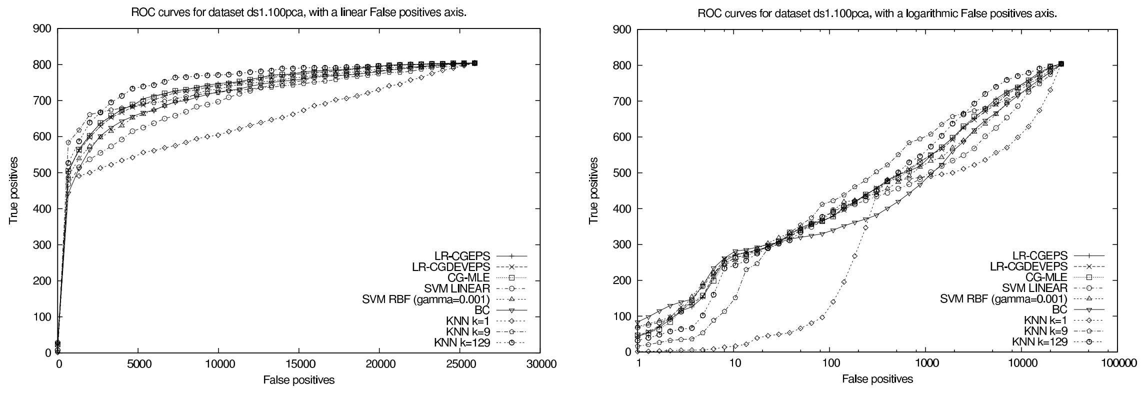


Figure 6.11: ROC curve for ds1.100pca.

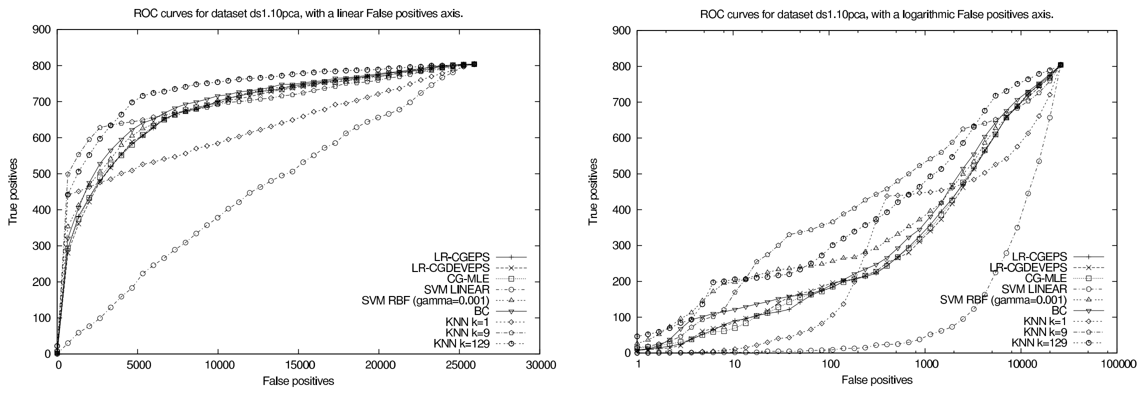


Figure 6.12: ROC curve for ds1.10pca.

More detailed discussion about Drug Screening and experiments can be found in (Gray et al. [2004]) and (Komarek [2004]).

6.3 Image Retrieval

The proliferation of the web and digital photography have made large scale image collections containing billions of images a reality. Image collections on this scale make performing even the most common and simple computer vision, image processing, and machine learning tasks non-trivial. An example is the k-NN search, which not only serves as a fundamental subproblem in many more sophisticated algorithms, but also has direct applications, such as image retrieval and image clustering. In this paper, we address the k-NN problem as the first step towards scalable image processing. We describe a parallel version of the spill-tree algorithm and discuss how it can be used to find near duplicates among over a billion images.

Very large scale image collections are difficult to organize and navigate. One operation which can facilitate this task is the identification of near duplicate images in the collection. Near duplicate images of popular items, such as book covers, CD covers, and movie posters, appear frequently on the web. This is because they are often scanned or photographed multiple times with varying resolutions and color balances. To tackle this problem at the scale of the whole web, one needs efficient, scalable, and parallelizable algorithms for locating nearest neighbors in the image feature space, and clustering them together.

We have showed that spill-tree is able to locate approximate nearest neighbors in high-dimensional spaces with high accuracy and speed. However the algorithms for this data structure are designed for a single machine. The current work attempts to extend the spill-tree algorithm by making the tree building parallel, ande performing efficient k-NN search in such trees. To evaluate this work in a practical setting, we generated feature vectors for over a billion[4] images, built an efficient search tree for these feature vectors, and performed clustering based on these results.

All algorithms, including our new algorithms, we discussed so far are serial algorithms, running in a single machine and requiring random access to the entire set of objects to be placed in the tree. This work focuses on two extensions to the spill-tree work: making the tree building algorithms work in parallel to handle large data sets which cannot fit into a single computer's memory, and doing a large number of queries efficiently in parallel.

[4]Throughout this paper we follow the convention that a billion = 10^9.

6.3.1 Image Features

Before building a search tree of images, we need to define how the images will be represented as feature vectors. We first normalize the image by scaling the maximum value of each color channel to cover the full range of intensities, and then scale the image to a fixed size of 64×64 pixels. From here, one obvious representation might be the image pixels themselves, however this would likely be quite sensitive to noise and other small image variations. Instead we used an adaptation of the technique presented by (Jacobs et al. [1995]), in which the image is converted to a Haar wavelet domain, the wavelet coefficients are quantized to ± 1, and all but the largest 60 magnitude coefficients are simply set to 0. The feature vector as described is quite large, $64 \times 64 \times 3$, so random projection is used to reduce the dimensionality of the feature vector to 100 dimensions (Kleinberg [1997]). The average of each color channel and the aspect ratio $w/(w+h)$ are appended to this feature vector for a total of 104 dimensions. The parallel spill-tree algorithm described later is designed to handle generic feature vectors, and is not restricted to this particular representation.

6.3.2 Parallel Computing Framework

All of the parallel algorithms described will be expressed in terms of the MapReduce operations (Dean and Ghemawat [2004]), which provide a convenient framework hiding many of the details necessary to coordinate processing on a large number of machines. An operation in the MapReduce framework takes as input a collection of items in the form of key-value pairs, and produces a collection of output in the same format. It has three basic phases, which are described in Fig. (6.13).

In essence, an operation in the MapReduce framework is completely described by the *map operation*, the *shuffle operation*, and the *reduce operation*. The algorithms below will be described in terms of these operations, however this should not be taken as the only way to implement these algorithms.

6.3.3 Building Hybrid Spill-tree in Parallel

The main challenge in scaling up the hybrid spill-tree generation algorithm is that it requires all the objects' feature vectors to be in memory, and random access to this data. When the number of objects becomes large enough, it is no longer possible to store everything in memory. For our domain, with 104 floating point numbers to represent each object, or around 416 bytes, this means we could typically fit two million points comfortably on a machine with 1GB of memory. In a collection of over a billion images, there are

Map in which a user-defined *Map Operation* is performed on each input key-value pair, optionally one or more key-value pairs can be generated. This phase works in parallel, with the input pairs being arbitrarily distributed across machines.

Shuffle in which each key-value pair generated by the Map phase is distributed to a collection of machines, based on a user-defined *Shuffle Operation* of their keys. In addition, within each machine the key-value pairs are grouped by their keys.

Reduce in which a user-defined *Reduce Operation* is applied to the collection of all key-value pairs having the same key, optionally producing one or more output key-value pairs.

Figure 6.13: The three phases which make up an operation in the MapReduce framework. All steps run in parallel on many machines.

nearly a thousand times as many as can fit on one machine.

The first question then is how to partition the data. One suggestion might be to randomly partition the data, building a separate hybrid spill-tree for each partition. However, at query time, this would require each query be run through all the trees. While this could be done in parallel, the overall query throughput would be limited.

Another alternative would be to make a more intelligent partition of the data. We propose to do this through the use of metric trees structure. We first create a random sample of the data small enough to fit on a single machine, say $1/M$ of the data, and build a metric-tree for this data. Each of the leaf nodes in this *top tree* then defines a partition, for which a hybrid spill-tree can be built on a separate machine. The overall tree consisting of the top tree along with all the *leaf subtrees* can be viewed conceptually as a single hybrid spill-tree, spanning a large number of machines. Here, we use random sampling to generate a top tree just for simplicity. A more sophisticated method can be applied to get better sampled data.

At first glance it might appear that the top tree should also be a spill-tree, because one of their benefits is removing the need to backtrack during search (which would mean having to search multiple leaf subtrees). However, a negative aspect of spill-trees is that objects appear in multiple leaf subtrees. In practice however we found this lead to an unacceptable increase in the total storage required by the system. The resolution was to force the top tree to be a metric-tree, and to make modifications to the search procedure which will be described in the next subsection. Here, we used an even simlified version of metric-tree. We eliminate the procedure for looking for centroid of each node and generating the balls. To estimate the minimum possible distance from a query q to a node, we use the distance from q to the partition plane plus $\tau/2$. This simplified version reduces the time for building metric-trees and provides even better speed for k-NN search in many high-dimensional data case. Another advantage of this approach is that it can be easily extended to other distance metric besides L2 distance.

The metric trees building procedure needs a stopping condition for its leaves. Typically the condition is an upper bound on the leaf size. In order for each partition to fit on a machine, we set the upper bound U such that the expected number of objects $U \cdot M$ which will fall into a single leaf subtree can fit on a single machine. We typically set $U \cdot M$ a factor of two or more smaller than the actual limit, to allow for variability in the actual number of objects going into each leaf. In addition we set a lower bound on the number of nodes. The lower bound L is set empirically to prevent individual partitions from being too small, typically we use a value of five.

The algorithm as described so far is implemented in a sequence of three MapReduce operations and one sequential operation in Fig. (6.14).

6.3.4 Efficient Queries of Parallel Hybrid Spill-tree

After the trees have been built, they can be queried. As mentioned earlier, the top tree together with the leaf subtrees can be viewed as one large hybrid spill-tree. The normal way to query such a tree allows for backtracking through non-overlapping nodes, such as those which appear in the top tree. However such an approach would be expensive to implement since the entire tree is not stored on a single machine. Instead, we speculatively send each query object to multiple leaf subtrees when the query appears to be too close to the boundary. This is effectively a run-time version of the overlap buffer which was previously only applied at tree building time. The benefit of this is that fewer machines are required to hold the leaf subtrees (because there is no duplication of objects across the subtrees), but with the expense that each query may be sent to several leaf trees during search. In practice we can adjust the overlap buffer size to control the amount of computation done at query time.

For clustering, we actually need the k-NN lists for every object, so we organize the search as a batch process, which takes as input a list of queries (which will be every image), and produces their k-NN lists. The process is described in Fig. (6.15), using two MapReduce

Sample Data Input is all the objects, output is a sampled subset for building the top tree.

- **Map Operation** For each input object, output it with probability $1/M$.
- **Shuffle Operation** All objects map to a single machine.
- **Reduce Operation** Copy all objects to the output.

Build Top Tree On a single machine, build the top tree using the standard metric tree building algorithm as described in chapter 2, with the additional restriction on the upper bound U and lower bound L on the number of objects in each leaf node.

Partition Data and Create Leaf Subtrees Input is all the objects, output is the set of leaf subtrees

- **Map Operation** For each object, find which leaf subtree number it falls into, and output this number as the key along with the object.
- **Shuffle Operation** Each distinct key is mapped to a different machine, to collect the data for each leaf subtree.
- **Reduce Operation** For all the objects in the leaf subtree, use the serial hybrid spill-tree algorithm to create the leaf subtrees.

Figure 6.14: The parallel steps to build a distributed hybrid spill-tree.

Find Neighbors in Each Leaf Subtree Input is the set of query objects, output is the k-NN lists for each query object for each subtree the query was routed to.

Map Operation For each input query, compute which leaf subtree numbers it falls into. At each node, the query may be sent down both sides of the tree if it falls within the overlap buffer width of the decision plane. Generate one key-value pair for each leaf subtree that should be searched.

Shuffle Operation Each distinct key is mapped to a different machine, to search the leaf subtrees in parallel.

Reduce Operation The standard hybrid spill-tree search procedure is used for each object that is routed to each leaf subtree, and the k-NN lists for each query object are generated.

Combine k-NN Lists Input is the k-NN lists for each object in each leaf subtree, output is a single k-NN list for each query object.

Map Operation Copy each query, k-NN list pair to the output.

Shuffle Operation The queries (image numbers) are partitioned randomly by their numerical value.

Reduce Operation The k-NN lists for each query are merged together, keeping only the k objects closest to the query.

Figure 6.15: Batch k-NN search in two MapReduce operations.

operations.

The overall parallel hybrid spill-tree is a batch system. We send the queries in batch, all queries go check the top tree first and be sent to corresponding subtrees, then all subtrees process their own queries simultaneously and generate the k-NN lists.

6.3.5 Experiments

Below we describe the experiments used to evaluate hybrid spill-trees by using them to assist with clustering a large collection of images. We begin with a description of the image sets, then outline the clustering procedure, and finally describe the results.

Data Sets

There were two main data sets used for these experiments, one in which the clusters were hand labeled for setting various algorithm parameters, and the second larger set which is our target for clustering.

The labeled set was generated by performing text-based image search queries on several large search engines and collecting the first 60 results for each query. The queries were chosen to provide a large number of near duplicate images. Queries for movie posters, CDs, and popular novels worked well for this. The duplicate sets within the results of each query were manually labeled. In addition, 1000 images were chosen at random to represent non-duplicate images. The full collection consisted of 3385 images, in which each pair of images is labeled as either a duplicate or non-duplicate.

The second much larger set of images consisted of nearly 1.5 billion images from the web (hereafter the 1.5B image set). This was our target for clustering. We have no way of knowing in advance how many of these images are duplicates of one another.

Clustering Procedure

Although most of the work described so far was on efficiently finding the k nearest neighbors of points, either for single points or in a batch mode. In order to adapt this for clustering, we compute the k-NN for all images in the set, applying a threshold to drop images which are considered too far apart. This can be done as a MapReduce operation as shown in Fig. (6.16).

The result of this algorithm is a set of prototype clusters, which further need to be combined. Once singleton images are dropped in the 1.5B image set, we are left with fewer than 200 million images, which is a small enough set to run the final union-find algorithm on a single machine.

Clustering Results

To evaluate the image features, we first performed clustering on the smaller labeled data set. For each pair of images, we compute the distance between their feature vectors (since this is a small enough data set this is practical). As the distance threshold is varied, we compute clusters by joining all pairs of images which are within the distance threshold of one another. Each image pair within these clusters is then checked against the manual labeling. The results are shown in Fig. (6.17). This figure plots the error rate on duplicate

Map Operation Input is the k-NN list for each image, along with the distances to each of those other images. We first apply a threshold to the distances, shortening the neighbor list. The list is then treated as a prototype cluster, and reordered so that the lowest image number is first. The generated output consists of this lowest number as the key, and value is the full set. Any images with no neighbors within the distance threshold are dropped.

Shuffle Operation The keys (image numbers) are partitioned randomly by their numerical value.

Reduce Operation Within a single set of results, the standard union-find algorithm (Cormen et al. [2001]) is used to combine the prototype clusters.

Figure 6.16: Algorithm for initial clustering of data.

image pairs vs. the error rate on non-duplicate image pairs. As can be seen, a distance threshold of 0.45 is a good compromise at reducing the number of false matches, while detecting as many duplicates as possible. We do admit here that 0.45 is an empirical value, and it depends on the image representation. However, for a different task, we can do similar test on a relative small data set and estimate a different threshold. Note that using the 10 nearest neighbors produced by spill-trees and hybrid spill-trees gives equivalent accuracy, but with 20 times less computation. The reason that manual examination of the clusters was necessary is because there were some labeling errors in this set, and manual examination revealed which groupings were in fact errors and which were correct. The graph also shows the result of using at most 10 nearest neighbors (instead of all within the distance threshold), and the approximate 10 nearest neighbor lists generated by the spill-tree algorithm and hybrid spill-tree algorithms. All of these results are quite close in accuracy, although the spill tree-based algorithms are almost 20 times faster for this smaller set. This difference in speed will grow as the size of the set grows.

We then applied the parallel nearest neighbor finder and clustering procedure to the 1.5B image set. The entire processing time from start to finish was less than 10 hours on the equivalent of 2000 CPU's. Much of that time was spent with just a few machines running, as the sizes of the leaf subtrees was not be controlled directly (but see Section 6.3.6 for ideas about how to improve this). Although not discussed here, the computation of the features themselves was also done using the MapReduce framework, and took roughly the same amount of time as the clustering (but with fewer machines). The resulting distribution of cluster sizes is shown in Fig. (6.18). Around fifty million clusters are found, con-

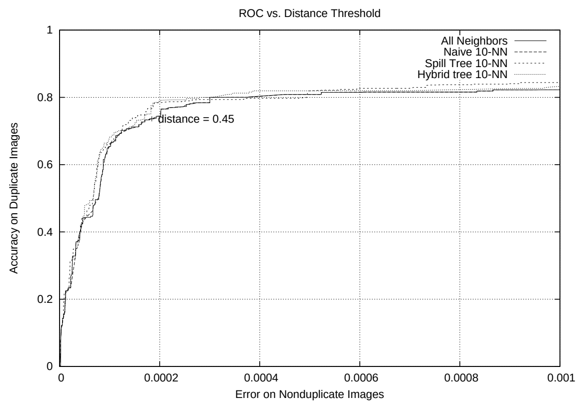


Figure 6.17: ROC curve for the small labeled test set.

taining nearly two hundred million images. The most common cluster size is two, which is perhaps not surprising given the number of thumbnail-full-size image pairs which exist on the web.

As there is no ground truth labeling for clusters in this larger set, it is hard to objectively evaluate the accuracy of the clustering. For a subjective evaluation, we show some of the actual clusters in Fig. (6.19).

As can be seen the images tend to be quite similar to one another, although in some cases images which are quite far apart are grouped together. It is expected that by combining these results with the results of a text query, it will be possible to get more precise clusters when displaying results to users. Another alternative will be to apply a post-processing step to cut clusters which are "long and thin" into smaller clusters.

6.3.6 Summary and Future Work

We have described an algorithm for the building of parallel distributed hybrid spill-trees which can be used for efficient online or batch searches for nearest neighbors of points in high dimensions spaces. This algorithm has enabled us to perform clustering on a set of over a billion images with the goal of finding near duplicates. To our knowledge, this is the largest image set that has been processed in this way.

The algorithm does not depend on the types of objects or the application; all it requires is that the objects be described by a feature vector. Because of this, we look forward to

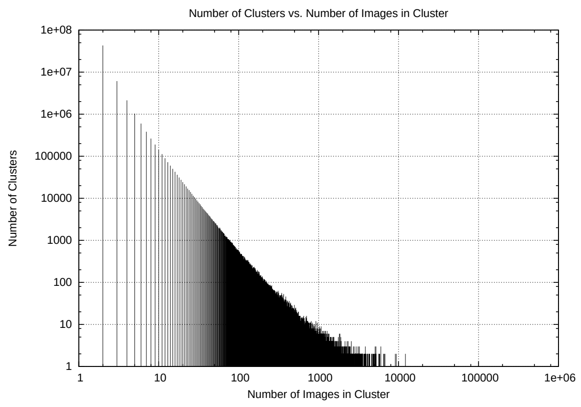


Figure 6.18: Histogram of cluster sizes for the 1.5B image set. Note the logarithmic scale on both axes.

seeing its application in a wide variety of domains, for instance face recognition, OCR, searching through SIFT descriptors (Lowe [2004]), and machine learning and classification problems. All of these applications could have online versions, in which a query object is presented and we want to find the nearest neighbor, or in an offline or batch setting in which we want to find the nearest neighbors of every point in our collection.

There are several directions for future work. One of the simplest is remove the arbitrary 10-nearest neighbor restriction, and attempt to find all neighbors within the distance threshold to create the clusters.

One of the main bottlenecks in the algorithm is caused by the fact that the number of images in each leaf subtree is not controlled precisely. Currently we approximately control this by setting lower bound L and upper bound U on the number of nodes in each leaf when building the top tree. However since the top tree is built with a small sample of the data, this approximation will have errors. An alternative procedure would be to build the top tree with a stopping condition of a single object per leaf node. After doing this, all objects can be run through the tree, and counts can be recorded for how many times an object reaches each leaf node. With this information, we can prune back the top tree until the nodes are within a desired size range, more precisely controlling the subproblem.

Since many machines are required to hold the entire search tree for efficient querying, the query system could be arranged as a “nearest neighbor service”, which accepts a query and returns the closest neighbors. One machine would contain the top tree, and direct the query to one or more leaf machines. The top tree may be located on a different machine

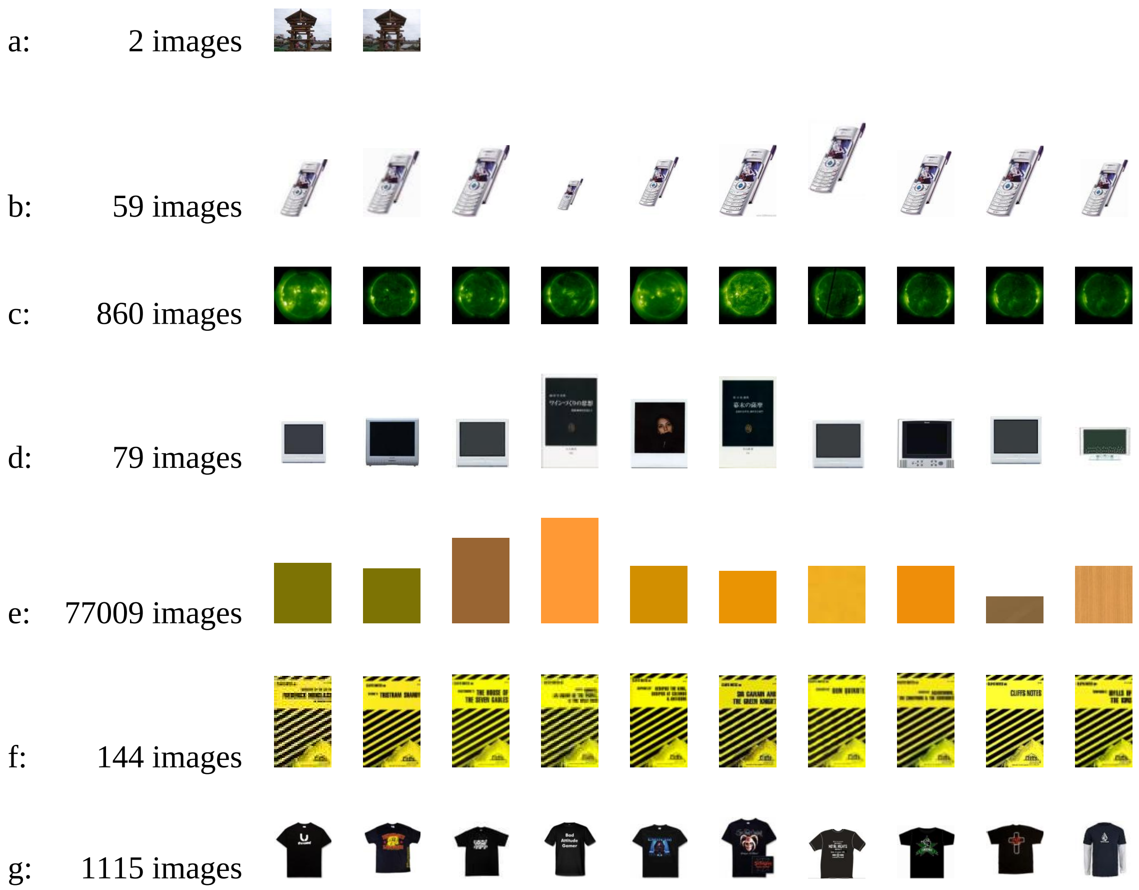


Figure 6.19: Selection of clusters found by the algorithm. Note the many different sizes of the object in B, and the different words on the same pattern in F and G.

from the client (as is typical of services), or may be located on the same machine (or even in the same process). One application of such a service might be an image retrieval application.

Finally, a basic step of our clustering method was to solve the all-nearest-neighbor problem by performing a large batch of individual nearest neighbor searches. Work on dual trees (Gray and Moore [2001]) provides an alternative to such a batch approach, which may be applicable to our parallel hybrid spill-trees.

Bibliography

K. Aas and L. Eikvil. Text categorisation: A survey, 1999. URL `citeseer.ist.psu.edu/aas99text.html`. 1.2

J. Adcock, A. Girgensohn, M. Cooper, T. Liu, L. Wilcox, and E. Rieffel. Fxpal experiments for trecvid 2004. In *Proceedings of the TREC Video Retrieval Evaluation (TRECVID)*. NIST, 2004. 3

D. W. Aha, S. L. Salzberg, and Ling. Learning to catch: Applying nearest neighbor algorithms to dynamic control tasks. In *Proceedings of the 11th International Conference on Machine Learning*, pages 12–18. Morgan Kaufmann, 1994. URL `citeseer.ist.psu.edu/article/aha94learning.html`. 1.2

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. 1.4

S. Arya, T. Malamatos, and D. M. Mount. Space-efficient approximate voronoi diagrams, 2002. URL `citeseer.ist.psu.edu/624368.html`. 1.4, 5.6

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. URL `citeseer.ist.psu.edu/arya94optimal.html`. 1.4, 3.4, 5.6

A. Bergo. Text categorization and prototypes, 2001. URL `citeseer.ifi.unizh.ch/bergo01text.html`. 1.2

J. Bescos, G. Cisneros, J. M. Martinez, J. Menendez, and J. Cabrera. A unified model for techniques on video-shot transition detection. *IEEE Trans. on Multimedia*, 7(2): 293–307, 2005. 6.1.4

J. A. Blackard. Forest covertype database. URL `http://kdd.ics.uci.edu/databases/covertype/covertype.data.html`. 3

J. S. Boreczky and L. A. Rowe. Comparison of video shot boundary detection techniques. In *Storage and Retrieval for Image and Video Databases (SPIE)*, pages 170–179, 1996. URL `citeseer.ist.psu.edu/boreczky96comparison.html`. 6.1.4

J. S. Boreczky and L. D. Wilcox. A hidden markov model frame work for video segmentation using audio and image features. In *Proceedings of ICASSP'98*, pages 3741–3744, 1998. 6.1

J. Buhler. Efficient large-scale sequence comparison by locality-sensitive hashing. *BIOINF: Bioinformatics*, 17:419–428, 2001. URL `citeseer.ist.psu.edu/buhler01efficient.html`. 1.4

C. Cardie and N. Howe. Improving minority class prediction using case-specific feature weights. In *Proceedings of the 14th International Conference on Machine Learning*, pages 57–65, 1997. URL `citeseer.nj.nec.com/cardie97improving.html`. 3.1

C. L. Chang. Finding prototypes for nearest neighbor classifiers. *IEEE Trans. Computers*, C-23(11):1179–1184, November 1974. 1.3

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. URL `citeseer.ist.psu.edu/chaudhuri98random.html`. 1.4

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*, 1997. 1.3, 1.5, 2.1

K. L. Clarkson. *Nearest-Neighbor Methods for Learning and Vision: Theory and Practice*. To appear. URL `http://cm.bell-labes.com/cm/cs/who/clarkson/nn_survey/b.pdf.` 1.3, 5.5

R. Cole and M. Fanty. Isolet spoken letter recognition database. URL `ftp://ftp.ics.uci.edu/pub/machine-learning-databases/isolet/`. 2

M. Cooper and J. Foote. Scene boundary detection via video self-similarity analysis. In *IEEE Intl. Conf. on Image Processing (3)*, pages 378–381, 2001. 6.1.1

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. stein. The MIT Press, 2001. 6.3.5

S. Cost and S. Salzberg. A Weighted Nearest Neighbour Algorithm for Learning with Symbolic Features. *Machine Learning*, 10:57–67, 1993a. 1.2

S. Cost and S. Salzberg. A weighted nearest neighbor algorithm for learning with symbolic features. *Machine Learning*, 10:57–78, 1993b. URL `citeseer.ist.psu.edu/cost93weighted.html`. 1.2

T. M. Cover and P. E. Hart. Nearest neighbor pattern classification. *IEEE Trans. Information Theory*, IT-13(1):21–27, 1967. 1.2, 4.1.4, 1, 4.1.4

S. Dasgupta and A. Gupta. An elementary proof of the johnson-lindenstrauss lemma. Technical Report ICSI Technical Report TR-99-006, MIT, 1999. 5.3.3, 5.3.3

NCI Open Compound Database. National cancer institute open compound database, 2000. URL `http://cactus.nci.nih.gov/ncidb2.` 6.2.2

J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In *Symposium on Operating System Design and Implementation*, 2004. 6.3.2

S. C. Deerwester, S. T. Dumais, T. K. Landauer, G. W. Furnas, and R. A. Harshman. Indexing by latent semantic analysis. *Journal of the American Society of Information Science*, 41(6):391–407, 1990. 1.2

K. Deng and A. W. Moore. Multiresolution Instance-based Learning. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence*, pages 1233–1239, 1995. 1.3

L. Devroye and T. J. Wagner. *Nearest neighbor methods in discrimination*, volume 2. P.R. Krishnaiah and L. N. Kanal, eds., North-Holland, 1982. 1.2

A. Djouadi and E. Bouktache. A fast algorithm for the nearest-neighbor classifier. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 19(3):277–282, 1997. 1.3

N. R. Draper and H. Smith. *Applied Regression Analysis, 2nd ed.* John Wiley, New York, 1981. 1.2

R. O. Duda and P. E. Hart. *Pattern Classification and Scene Analysis*. John Wiley & Sons, 1973. 1.2, 4.1.4, 4.1.4, 6.2.2

C. Faloutsos, R. Barber, M. Flickner, J. Hafner, W. Niblack, D. Petkovic, and W. Equitz. Efficient and effective querying by image content. *Journal of Intelligent Information Systems*, 3(3/4):231–262, 1994. 1.2, 1.3

C. Faloutsos and D. W. Oard. A survey of information retrieval and filtering methods. Technical Report CS-TR-3514, Carnegie Mellon University, 1995. 1.2

T. Fawcett and F. J. Provost. Adaptive fraud detection. *Data Mining and Knowledge Discovery*, 1(3):291–316, 1997. URL `citeseer.nj.nec.com/fawcett97adaptive.html`. 3.1

F. P. Fisher and E. A. Patrick. A preprocessing algorithm for nearest neighbor decision rules. *Proc. Nat'l Electronic Conf.*, 26:481–485, December 1970. 1.3

E. Fix and J. L. Hodges. Discriminatory analysis, Nonparametric discrimination: consistency properties. Technical report, USAF School of Aviation Medicine, Randolph Field, Texas, 1951. 1.2

E. Fix and J. L. Hodges. Discriminatory analysis: small sample performance. Technical report, USAF School of Aviation Medicine, Randolph Field, Texas, 1952. 1.2

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. 1.2

D. Fradkin and D. Madigan. Experiments with random projections for machine learning. In *Proc. 9th ACM international conference on Knowledge discovery and data mining*, pages 517–522, 2003. 6.1.5

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. 1.3

A. Frome and J. Malik. Object recognition using locality sensitive hashing of shape contexts. 2006. 1.2

G. W. Gates. The reduced nearest neighbor rule. *IEEE Trans. Information Theory*, IT-18 (5):431–433, May 1972. 1.3

A. Gionis, P. Indyk, and R. Motwani. Similarity Search in High Dimensions via Hashing. In *Proceedings of the 25th VLDB Conference*, 1999. 1.4, 5.1, 5.1, 5.4, 5.4

K. Grauman and T. Darrell. Contour matching and scene recognition using approximate earth mover's distance. 2006. 1.2

A. Gray, P. Komarek, T. Liu, and A. W. Moore. High-dimensional probabilistic classification for drug discovery. In *Proceedings of the Computational Statistics*, 2004. 6.2.2

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. 1.3, 6.3.6

B. Gunsel, A. M. Ferman, and A. M. Tekalp. Temporal video segmentation using unsupervised clustering and semantic object tracking. *Journal of Electronic Imaging*, 7(3): 592–604, 1998a. URL `citeseer.ist.psu.edu/gunsel98temporal.html`. 6.1

B. Gunsel, M. Ferman, and A. M. Tekalp. Temporal video segmentation using unsupervised clustering and semantic object tracking. *Journal of Electronic Imaging*, 7: 592–604, 1998b. 6.1.4

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. 1.2, 1.3

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. 1.2

J. M. Hammersley. The distribution of distances in a hypersphere. *Annals of Mathematical Statistics*, 2:447–452, 1950. 3.4

S. Han, G. Karypis, and V. Kumar. Text categorization using weight adjusted k -nearest neighbor classification. *Lecture Notes in Computer Science*, 2035:53–??, 2001. URL `citeseer.ist.psu.edu/han99text.html`. 1.2

A. Hanjalic. Shot boundary detection: Unraveled and resolved. *IEEE Trans. on Circuits and Systems for Video Technology*, 12(2):90–105, 2002. 6.1

P. E. Hart. The condensed nearest neighbor rule. *IEEE Trans. Information Theory*, IT-14 (5):515–516, May 1968. 1.3

T. Hastie and R. Tibshirani. Discriminant adaptive nearest neighbor classification. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 18(6):607–615, June 1996. 1.2

P. Indyk. *High Dimensional Computational Geometry*. Phd. thesis, Stanford University, Department of Computer Science, 2000. 5.1

P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *STOC*, pages 604–613, 1998. 1.4, 5.1, 5.1

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. 5.1

CMU informedia digital video library project. The trec-2001 video trackorganized by nist shot boundary task, 2001. 3, 4

IOC. International olympic committee: Candidature acceptance procedure, 1999. URL `http://multimedia.olympic.org/pdf/en_report_711.pdf`. 4.1.2

C. E. Jacobs, A. Finkelstein, and D. H. Salesin. Fast multiresolution image querying. In *Proceedings of SIGGRAPH*, pages 227–286, 1995. 6.3.1

W. Johnson and J. Lindenstrauss. Extensions of lipschitz maps into a hilbert space. *Contemporary Mathematics*, 26:189–206, 1984. 5.5

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. 1.4

N. Katayama and S. Satoh. The SR-tree: an index structure for high-dimensional nearest neighbor queries. In *Proceedings of ACM SIGMOD*, pages 369–380, 1997. URL `citeseer.ist.psu.edu/katayama97srtree.html`. 2

J. Kleinberg. Two algorithms for nearest neighbor search in high dimension. In *ACM Symposium on the Theory of Computing*, pages 599–608, 1997. 6.3.1

V. Koivune and S. Kassam. Nearest neighbor filters for multivariate data. In *IEEE Workshop on Nonlinear Signal and Image Processing*, 1995. 1.2

P. Komarek. *Logistic Regression for Data Mining and High-Dimensional Classification*. PhD. Thesis, Carnegie Mellon University, Department of Math Science, 2004. 1.2, 6.2.2, 6.2.2, 6.2.2

I. Koprinska and S. Carrato. Temporal video segmentation: a survey. In *Signal Processing: Image Communication*, volume 16, pages 477–500, 2001. 6.1

W. Kraaij, A. Smeaton, P. Over, and J. Arlandis. Trecvid 2004 - an introduction. In *Proceedings of the TREC Video Retrieval Evaluation (TRECVID)*, Washington D.C., 2004. NIST. 6.1, 6.1.4

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. 1.4, 5.6

N. Kushmerick. Internet advertisements. URL `ftp://ftp.ics.uci.edu/pub/machine-learning-databases/internet_ads/`. 5

E. Lee and S. Chae. Fast design of reduced-complexity nearest-neighbor classifiers using triangular inequality. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 20(5): 562–566, May 1998. 1.3

R. Lienhart. Reliable transition detection in videos: A survey and practitioner's guide. *International Journal of Image and Graphics (IJIG)*, 1(3):469–486, 2001. 6.1

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. 1.5.1, 1.5.5

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*, 2004a. 1.5.3, 1.5.5

T. Liu, K. Yang, and A. W. 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)*, 2004b. 1.5.2, 1.5.5

D. G. Lowe. Distinctive image features from scale-invariant keypoints,. *International Journal of Computer Vision*, 60:91–110, 2 2004. URL `http://cm.bell-labes.com/cm/cs/who/clarkson/nn_survey/b.pdf.` 6.3.6

S. Maneewongvatana and D. M. Mount. The analysis of a probabilistic approach to nearest neighbor searching. In *WADS*, 2001. 3.4

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. 1.4

Y. Matias, J. S. Vitter, and M. Wang. Wavelet-based histograms for selectivity estimation. In *Proceedings of SIGMOD*, pages 448–459, 1998. URL `citeseer.ist.psu.edu/matias98waveletbased.html`. 1.4

M. C. Monard and G. E. A. P. A. Batista. *Learning with Skewed Class Distribution*. IOS Press, 2002. URL `citeseer.nj.nec.com/monard02learning.html`. 3.1

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. 1.3, 2.1

R. Motwani and P. Raghavan. Cambridge University Press, 1995. 5.6.3

S. Omachi and H. Aso. A fast algorithm for a k-nn classifier based on branch and bound method and computational quantity estimation. *Systems and Computers in Japan*, 31 (6):1–9, 2000. 1.5.1, 3.1

S. M. Omohundro. Efficient Algorithms with Neural Network Behaviour. *Journal of Complex Systems*, 1(2):273–347, 1987. 1.3

S. M. Omohundro. Bumptrees for efficient function, constraint, and classification learning. In *Advances in Neural Information Processing Systems 3*, 1991. 1.3, 1.5, 2.1

A. M. Palau and R. R. Snapp. The labeled cell classifier: A fast approximation to k nearest neighbors. In *Proceedings of the 14th International Conference on Pattern Recognition*, 1998. 1.3

E. P. D. Pednault, B. K. Rosen, and C. Apte. Handling imbalanced data sets in insurance risk modeling, 2000. 3.1

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. 1.3

A. Pentland, R. Picard, and S. Sclaroff. Photobook: Content-based manipulation of image databases, 1994. URL `citeseer.ist.psu.edu/pentland95photobook.html`. 1.2, 1.3

M. J. Pickering, D. Heesch, R O'Callaghan, S Rger, and D Bull. Video retrieval using global features in keyframes. In *Proc. TREC Video Track*. NIST, 2002. 6.1.1

F. P. Preparata and M. Shamos. *Computational Geometry*. Springer-Verlag, 1985. 1.3

D. Pye, N. Hollinghurst, T. Mills, and K. Wood. Audio-visual segmentation for content-based retrieval. In *Proc. Intl. Conf on Spoken Language Processing*, 1998. 6.1.1

Y. Qi, A. G. Hauptmann, and T. Liu. Supervised classification for video shot segmentation. In *Proceedings of 2003 IEEE International Conference on Multimedia & Expo*, 2003. 4, 6.1.1

G. L. Ritter, H. B. Woodruff, S. R. Lowry, and T. L. Isenhour. An algorithm for a selective nearest neighbor decision rule. *IEEE Trans. Information Theory*, IT-21(11):665–669, November 1975. 1.3

G. Salton and M. McGill. *Introduction to Modern Information Retrieval.* McGraw-Hill Book Company, New York, NY, 1983. 1.2

I. K. Sethi. A fast algorithm for recognizing nearest neighbors. *IEEE Trans. Systems, Man, and Cybernetics*, SMC-11(3):245–248, March 1981. 1.3

G. Shakhnarovich, P. Viola, and T. Darrell. Parameter-sensitive hashing for fast pose estimation. 2006. 1.2

B. Y. Shih and W. I. Lee. The application of nearest neighbor algorithmon creating an adaptive on-line learning system. URL `citeseer.ist.psu.edu/507043.html`. 1.2

I. Shimshoni, B. georgescu, and P. Meer. Adaptive mean shift based clustering in high dimensions. 2006. 1.2

M. Slaney, D. Ponceleon, and J. Kaufman. Multimedia edges: finding hierarchy in all dimensions. In *MULTIMEDIA '01: Proceedings of the ninth ACM international conference on Multimedia*, pages 29–40. ACM Press, 2001. 6.1.1

D. J. Slate. Letter recognition database. URL `ftp://ftp.ics.uci.edu/pub/machine-learning-databases/letter-recognition/`. 1

A. Smeaton, W. Kraaij, and P. Over. The trec 2003 video track report. In *Proceedings of the TREC Video Retrieval Evaluation (TRECVID)*, Washington D.C., 2003. NIST. 6.1, 6.1.4

A. F. Smeaton and P. Over. The trec-2002 video track report. In *TREC*. NIST, 2002. 6.1, 6.1.4

A.W.M. Smeulders and R. Jain(eds). Image databases and multi-media search. In *Proc. 1st Workshop on Image Databases and Multi-Media Search*, 1996. 1.2

S. Stolfo, W. Fan, W. Lee, A. Prodromidis, and P. Chan. Credit card fraud detection using meta-learning: Issues and initial results, 1997. URL `citeseer.nj.nec.com/stolfo97credit.html`. 3.1

J. K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 40:175–179, 1991. 1.3, 1.5, 2.1, 3

V. N. Vapnik. *The Nature of Statistical Learning Theory*. Springer-Verlag, 1995. 6.2.1

N. Vasconcelos. Feature selection by maximum marginal diversity: optimality and implications for visual recognition. In *CVPR (1)*, pages 762–772, 2003. 6.1.3, 6.1.5

N. Vasconcelos and M. Vasconcelos. Scalable discriminant feature selection for image retrieval and recognition. In *Proc. IEEE Conf. on CVPR (2)*, pages 770–775, 2004. 6.1.3, 6.1.3, 6.1.3, 1

L. A. Wasserman. *All of Statistics: A Concise Course in Statistical Inference*. Springer, 2004. 1.1, 1.1

A. Witkin. Scale-space filtering: A new approach to multi-scale description. In *Proc. IEEE Intl. Conf. on Acoustics, Speech, and Signal Processing*, pages 39A.1.1–39A.1.4, March 1981. 6.1.1

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. 1.2

C-H Y, S. R, P. Tamayo, S. Mukherjee, R. M. Rifkin, M. Angelo, M. Reich, E. Lander, J. Mesirov, and T. Golub. In *Bioinformatics*, 2001. 1.2

Y. Yang. An evaluation of statistical approaches to text categorization. *Journal of Information Retrieval*, 1:67–88, 1999. 1.2

Z. Yao and W. L. Ruzzo. A regression-based k nearest neighbor algorithm for gene function prediction from heterogeneous data. In *BMC Bioinofrmatics*, 2006. 1.2

B. Zhang and S. Srihari. Fast k-nearest neighbor classification using cluster-based trees. *IEEE Trans. Pattern Analysis and Machine Intelligence*, 26(4):525–528, April 2004. 1.3

T. Zhang, R. Ramakrishnan, and M. Livny. BIRCH: An Efficient Data Clustering Method for Very Large Databases. In *Proc. 5th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, 1996. 1.3