NEW PARADIGMS FOR APPROXIMATE NEAREST-NEIGHBOR SEARCH

A Dissertation
Presented to
The Academic Faculty

by

Parikshit Ram

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in
Computer Science

School of Computational Science and Engineering
Georgia Institute of Technology
August 2013

NEW PARADIGMS FOR APPROXIMATE NEAREST-NEIGHBOR SEARCH

Approved by:

Professor Maria-Florina Balcan,
Committee Chair
School of Computer Science
Georgia Institute of Technology

Professor Alexander G. Gray, Advisor
School of Computational Science and
Engineering
Georgia Institute of Technology

Dr. Kenneth L. Clarkson
Principles and Methodologies Group
IBM Almaden Research Center

Professor Guy Lebanon
School of Computational Science and
Engineering
Georgia Institute of Technology

Professor Santosh S. Vempala
School of Computer Science
Georgia Institute of Technology

Date Approved: 1 July 2013

To Jennifer, for giving me perspective.

ACKNOWLEDGEMENTS

I would like to thank my parents for letting me choose this path, and I would like to thank my sister Arunima for her support. I would like to thank my future wife Jennifer for showing me a life outside of this. I would like to thank my advisor Alex for letting me explore and wander for so long. Finally, these acknowledgements would not be complete without me acknowledging my gratitude for my colleagues and labmates, past and present, who wasted so much time and energy tolerating and helping me.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Similarity search is everywhere, and so are its solutions. If exact search is the task at hand, there is no ambiguity in the task – we seek the objects *most similar* to our query. But the increasing size of data necessitates approximation. And once we start approximating, it is natural to think about and investigate the "right" way to approximate. Approximate search allows us to incur some error in our answers in exchange for efficiency. A lot of methods have been developed for approximate search with rigorous guarantees on their efficiency that present the precise relationship between the amount of error incurred and the resulting efficiency gain. In this scenario, approximate search has two key pieces: (i) the definition of search error that will be tolerated in approximate search, and (ii) the precise input to the search algorithm which indicates that we can tolerate error in exchange for fast results. The usual notion of error is *relative value error* which judges how similar the answer is to the query relative to the best possible answer to the query. The input to the search algorithm is usually the amount of search error that can be tolerated. Hence the search algorithm is required to return an answer which satisfies this condition. I call this the *error-constrained* setting.

I explore alternate forms of these two pieces in approximate search: (i) I present an alternate notion of search error, *rank error*, which corresponds to the number of objects that are better than our answer, and (ii) I consider an alternate form of approximate search algorithm where the input to the algorithm is the amount of time available for search instead of the amount of search error that can be tolerated (*time-constrained* setting). The notion of rank-error makes the choice of the tolerable search error easier in various applications. The choice of time as the input to the

search algorithm appears more intuitive to me because it is the lack of time that usually necessitates approximation of search. Choosing time as an input will ensure that the search is completed within a fixed amount of time.

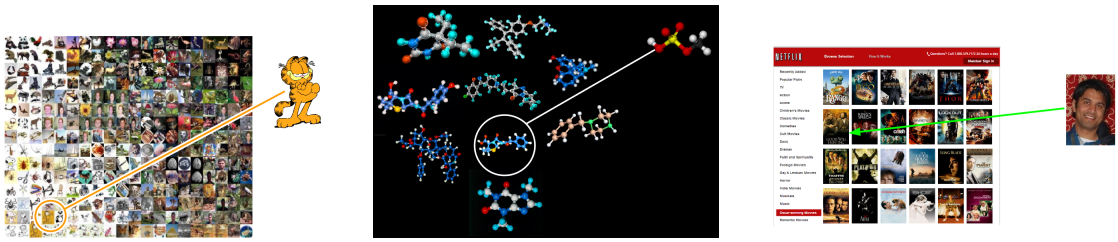
I present rank-approximate search algorithms in the error-constrained setting which utilize random sampling and tree data structures. I develop some very simple algorithms for time-constrained search which employ permutations of the data and space-partitioning tree data structures. The performance of these search algorithms in the time-constrained setting are improved upon by using machine learning techniques. Finally, I present an exact algorithm for similarity search in a more general setting where the objects do not need to have a fixed-length vector format and the similarity function does not need to be a metric.

CHAPTER I

NEW PARADIGMS FOR APPROXIMATE SIMILARITY SEARCH

In this thesis, we tackle the problem of similarity search. Similarity search has three main ingredients – (i) a set or collection of objects R, (ii) a query q whose similar objects we seek, and (iii) a similarity function that tells us, in terms of a real or natural number, how similar any object in question is to the query. Given these, the main task of similarity search is to find the object(s) in the collection **most similar** to the query. Finding similar images, a widely performed task in computer vision, is an example of similarity search (Figure 1(a)). Here we have a collection of images as the set of objects, the query is yet another image, and the goal is to find similar images to the query image with respect to some image-specific similarity function. Similarity search also appears in drug discovery in the form of molecule subgraph matching (Figure 1(b)). Here we have a set of molecules represented as labelled graphs and the query is a motif (a subgraph) and we want to find the molecule(s) in our collection that best matches the query. Similarity functions here are generally graph matchers which assign some scores to matchings. Even the task of movie recommendation (Figure 1(c)) is an instance of similarity search; the set of objects is the movie collection and the service user is the query. The similarity function is the user's (predicted) preference score for any movie. Then the task is to find the most preferred movie(s) for the user.

Beyond these specific applications, similarity search is ubiquitous. It has a big presence in machine learning and computer vision. Traditionally spawning from computer science theory and databases, it is currently a core task in information retrieval,



(a) Image matching (b) Subgraph matching (c) Movie recommendation

Figure 1: Examples of similarity search in practice.

various web applications and even collaborative filtering. Search is also a vital task in various scientific computing problems such as particle simulation and drug discovery. Since similarity search is fairly ubiquitous, a lot of research has focussed on similarity search. All the work has been to:

> *"efficiently find objects of interests in oceans of data."*

This is achieved by efficient exact similarity search. If approximation is necessary, we usually strive for efficient and accurate approximate search. By efficiency, I refer to query times sub-linear in the number of objects in the collection with storage requirements sub-quadratic in the number of objects in the collection. It would be preferable to have sub-quadratic preprocessing times as well. Given this goal, the main challenges in similarity search are (i) the sheer size of the data, and (ii) the numerous variety of data types.

1.1 Understanding the Challenges

The first challenge of size is best motivated by some examples. Netflix had over $100,000$ titles in 2009[1]. Pandora, the online music radio, has around a million songs[2],

[1] `http://techcrunch.com`, 2009
[2] `http://nysemagazine.com`

and Youtube have over 120 million videos in 2010 itself[3]. In 2011, Facebook announced that over 100 billion photographs were uploaded on their site[4], and Google indexed a trillion (that is 10^{12}) unique URLs almost 4 years ago[5]. These numbers necessitate sub-linear time algorithms – algorithms that are more efficient than simply comparing the query to the whole list of objects in our collection and returning the most similar object.

This simple algorithm, usually known as *linear search* or *brute-force search* or *exhaustive search*, does suffice when the collection size is less than 1000 and each object in the collection (as well as the query) does not have too many features. For large collections of objects with small number of features, tree-based methods are extremely efficient. This method involves indexing the set of objects into a hierarchical structure, and the search is performed with a branch-and-bound algorithm on this tree. When the objects themselves have large number of features, the hope for exact search is usually abandoned and approximate search is achieved with search on some tree, skip-lists or hash-based indices. These techniques will be discussed in Chapter 2. The high level idea in all cases is to preprocess the set of objects into an index and then search the index for an acceptable neighbor candidate. We will briefly discuss here what approximate search means.

1.1.1 Approximate Search

Search approximation allows us to accept objects other than the true best match as similar objects for a query from our collection. However, the restriction is that the similarity of these objects to the query is almost as much as that of the best match. This is a very intuitive notion of approximation and we will refer to it as *value approximation*. However, the appropriate level of approximation is not very

[3]The book *Youtube:* 120 *million videos and counting* came out in 2010.
[4]`http://allfacebook.com`, 2011
[5]`http://googleblog.blogspot.com`, 2009

easy to guess; a large enough approximation can trivialize the problem by accepting any object in the collection as a similar objects, while a small enough approximation will allow for no approximation, possibly allowing for no search efficiency gain.

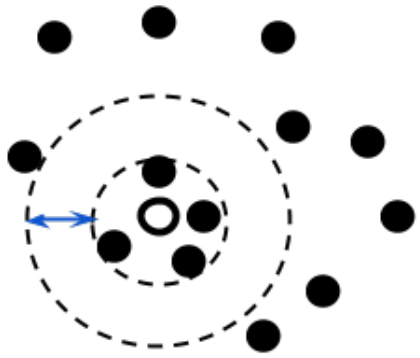


Figure 2: An example depicting the ideal situation where there is significant gap between the similar objects and the dissimilar objects (the solid circles) to the query (the hollow circle).

To choose the appropriate level for approximation, we need to understand the motivation for this form of approximation. Value approximation stems out of the intuition that for any query, the set of "true similar objects" (which includes the best match and possibly some other objects) are significantly more similar to the query than all other objects in the collection. This implies that there is a significant margin between the similarity values of the set of true similar objects to the query and the similarity values for the rest of the objects. Figure 2 depicts such a scenario for objects lying on a plane. The appropriate level of approximation accepts only objects from the small set of true similar objects.

Provided that the search problem at hand has this property, the appropriate level of approximation not only varies between collections, the appropriate level can and does vary among different queries for the same collection of objects. Hence, the task of guessing the appropriate level of approximation, when there exists one, is non-trivial. There are some domains where the approximation level does come out of the application. But in most cases, guess work guides the choice of approximation level.

I will discuss this further in Chapter 2.

The approximation level is generally an input to the search algorithm and the returned candidates are supposed to be within the specified approximation. The theoretical guarantees are on the search runtime based on the specified approximation level. This is very intuitive and in line with the general form of approximate algorithms in computer science. The high level goal of approximate search algorithm development can be stated as follows (exact search can be thought of as a special case of approximate search with zero approximation):

> **Error-constrained search:** Minimize search time subject to the restriction that the returned match is within the specified approximation level.

This implies that the search algorithm needs to guarantee the approximation level, that is, verify that the search error of the returned match is in fact within the desired approximation level. However, it is the lack of time that generally necessitates approximation of search. Yet it is the approximation level that is usually (explicitly) guaranteed by the search algorithm, not the search runtime. There are usually rigorous theoretical search times guarantees, but it is generally hard to deduce actual runtimes from these bounds.

As discussed earlier, appropriate approximation levels are hard to guess; different values are usually tried until some acceptable result is obtained or the search is completed within an acceptable amount of time. Moreover, some search algorithms spend time validating the approximation level; they seek a certificate that guarantees that their search result in fact satisfies the approximation constraint. Some applications have actual constraints on the search time – they require the answer within a hard time limit. In this setting, error-constrained search becomes hard and suboptimal – it is hard to guess the appropriate value of error for obtaining an answer within the given time limit, and it is suboptimal to spend part of the limited time on guaranteeing the error bound (the approximation level) instead of improving it. I will visit

this possible issue again in Chapter 2.

1.1.2 Beyond Fixed-Length Data

As mentioned earlier, depending on the number of features representing each object in our collection, there are many efficient methods to solve the search problem exactly and approximately. However, it is expected that the data has a fixed-length representation (that is, each object is represented by a fixed number of features) and some ℓ_p metric based similarity functions (such as Euclidean or Hamming distances) are used.

Various objects do not inherently have a fixed length representation. Some such objects with a standard structure are graphs, strings or time series. For example, our collection might contain graphs as objects, and the graphs can have the different number of vertices and/or different number of edges. In this scenario, it is hard to represent all the objects in a fixed number of features without losing a lot of structure in the data. Even strings and time series do not inherently have a fixed length form. Beyond these types of objects, there are much more complex objects out there which employ similarity search. For example, a movie can be represented as a time series of audio and images along with meta data concerning the cast and contributors to the movie. A website can be represented as a combination of text, images and various other meta data and link information.

Complex objects like these can and are coerced into some fixed length form before using Euclidean or Hamming distance for similarity search with existing methods. This might suffice in many scenarios. However, it might be desirable to retain the objects in their native non-fixed length form and to use a domain or application specific similarity function. In this case, efficient search algorithms need to be developed for non-fixed length data. Domain specific search techniques exist for certain applications such as biological sub-sequence search and time series search. However, a

general search framework would be useful for emerging domains as well.

1.2 Thesis Statement

In this thesis, we try to address the issues with nearest-neighbor search briefly discussed in the previous sections. Specifically,

> *"we develop new paradigms for nearest-neighbor search (along with new theory and algorithms in these paradigms) that make nearest-neighbor search more usable and accurate."*

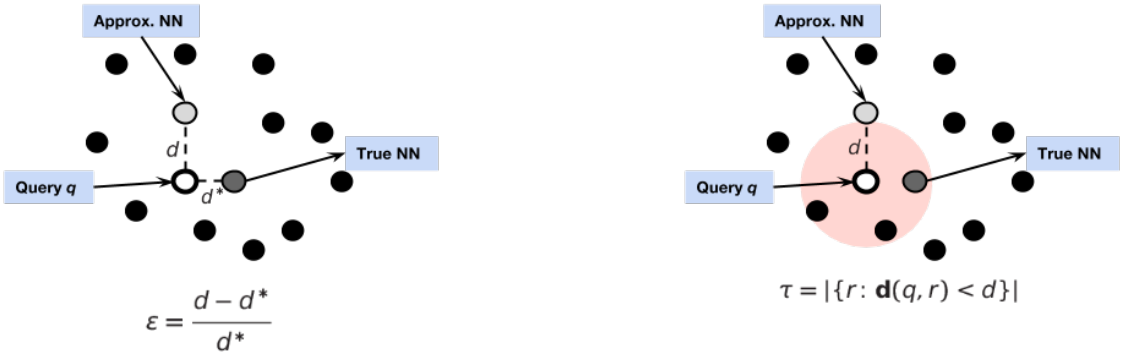
To this end, the following are my contributions:

- We consider a more robust and usable notion of approximation, *rank approximation*, and present algorithms for efficient rank approximate search.
- We consider approximate search in the scenario where time is a hard constraint (we call this *time-constrained search*) and present various algorithms for time-constrained search.
  - We present ways of *learning to search* where we utilize machine learning methods to improve search performance in the time-constrained setting.
- We focus on similarity search for a general class of (fixed-length or non-fixed-length) objects by considering the versatile class of kernel similarity functions applicable to a variety of data types to get the problem of *max-kernel search*, and present exact and approximate algorithms for max-kernel search.

1.2.1 Rank Approximation

Instead of approximating search in terms of the relative value error with respect to the similarity values (Figure 3(a)) (the precise form of value approximation), we consider approximating search in terms of the rank error, the number of objects in the collection that are more similar to the query than the returned match (Figure

3(b)). Equivalently, rank-error also corresponds to the fraction of the collection closer to the query than the returned match.



(a) Relative value error ε

(b) Rank error τ

Figure 3: Notions of search error. The traditional notion of *relative value error* and the proposed notion of *rank error* error is shown here for an approximate nearest-neighbor of the query.

This notion of error means the same across different queries in the same set and even across datasets. Moreover, the choice of approximation level is more straightforward and intuitive – you can chose to have any neighbor among the 10 nearest-neighbors or you can chose to have a neighbor among the closest 1%-tile of the objects to the query. The robustness of this notion of error makes it harder to work with; the relative value error of a neighbor candidate can be guessed with the query's similarity to this candidate and a guess of the query's similarity to its nearest-neighbor. However, it is not apparent how to guess the number of better candidates in the collection given a candidate without enumerating these better candidates. However, enumeration is computationally expensive.

Chapter 3 presents algorithms for *rank approximate nearest-neighbor search* that utilize random sampling and indexing schemes to return guaranteed rank-approximate neighbors. These algorithms have been shown to be efficient while returning meaningful answers (Ram et al., 2009b). These algorithms have also been used for efficient

k-nearest-neighbor classification of objects with large number of features (Ram and Gray, 2013). The specific contributions are :

- A new notion of approximation for nearest-neighbor search.
- A simple random sampling based algorithm for rank approximate search.
- A tree-based algorithm for rank approximate search with runtime guarantees.
- A detailed empirical comparison of the proposed rank approximate algorithms with existing exact and approximate nearest-neighbor search algorithms.

1.2.2 Time-Constrained Search

To overcome the inappropriateness of the error-constrained search for nearest-neighbor search on a time budget, we propose a new paradigm for approximate nearest-neighbor search – consider the nearest-neighbor error (value or rank error) incurred for answering a given time-limited query. To formalize, the nearest-neighbor search methods would be required to perform the following task:

> **Time-constrained search:** Minimize nearest-neighbor error subject to limited computation time.

This setting requires the query to be answered within a hard time constraint, but the goal is to obtain the best possible results within that time. If enough time is provided, the algorithm should return the exact nearest-neighbor. A lot of applications with limited response times fall in this framework – they can incur slight loss in the accuracy of the result, but they only have a small search time budget they need to strictly adhere to. For example, new users at an online music library want to listen to a playlist of songs as soon as they sign up and input their preferences, but they do not require the perfect songs to be played right away. However, users would prefer

the service provider with the better results. Similar situations exist in various other applications.

The notion of time-constrained search was proposed in Ram et al. (2012). Chapter 4 discusses the notion of time-constrained search and presents various algorithms that can return an approximate nearest-neighbor within a given time constraint. The specific contributions are:

- A new paradigm of time-constrained search for accurate search on a time budget.
- A permutation based algorithm that uses a permutation of the object set to answer time-constraint queries. This will serve as the baseline for time-constrained search.
- A binary space-partitioning-tree based algorithm that uses a greedy depth-first traversal of the tree to answer time-constraint queries.
- A binary space-partitioning-forest based algorithm that employs defeatist-search on a forest of random trees for very accurate solutions to time-constraint queries at the cost of extra storage.

1.2.3 Learning to Search

Even though we consider time-constrained search, we still want an accurate answer (preferably the exact answer in most cases). Since we might not have time to improve the results during time-constrained search, we would need data preprocessing which lead to better results. Indexing schemes that explicitly learn from the data are shown to be more effective for nearest-neighbor search (Cayton and Dasgupta, 2007; Clarkson, 1999; Maneewongvatana and Mount, 2002). These methods utilize "example queries" to learn indices that have optimal performance on this example queries. Machine learning techniques are also heavily used to create explicitly data-dependent hash functions that assign binary codes to points, assigning same code to similar

points and different codes to dissimilar points (Weiss et al., 2008; Wang et al., 2010; Norouzi and Fleet, 2011; Gong and Lazebnik, 2011). These techniques are heavily used in computer vision and are shown to empirically perform better than the widely popular mostly data-oblivious *locality sensitive hashing* (Indyk and Motwani, 1998).

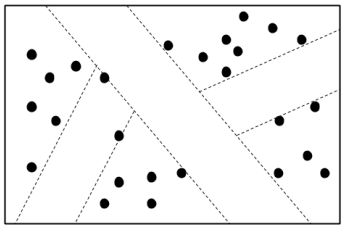


Figure 4: Max-margin tree.

Driven from the success of these "learned" searching techniques, we develop data-dependent preprocessing schemes for better search performance under a time-constraint in Chapter 5. We theoretically analyze the performance of the proposed time-constrained search algorithm to discover the factors affecting performance. Explicitly exploiting these factors, we propose a new binary space-partitioning tree. Specifically, my contributions are:

- A technique to learn permutations with example queries for better permutation-based time-constrained search performance.
- A theoretical analysis of the proposed binary tree based time-constrained search algorithm, exposing factors that affect search performance.
- A new binary space-partitioning tree, the *max-margin tree* (Figure 4), which utilizes a large margin split of the data at each level of the tree.
- Theoretical guarantees for the max-margin tree and its efficient variants.

1.2.4 Max-Kernel Search

We consider the following general task of max-kernel search: for a given set R of n objects (the *reference set*), a Mercer kernel function $\mathcal{K}(\cdot, \cdot)$, and a query q, find the object $p \in R$ that is most similar to q with respect to the similarity function $\mathcal{K}(\cdot, \cdot)$.

A Mercer kernel can define a measure of similarity for various classes of objects including points in $\mathbb{R}^{\mathcal{D}}$, and extending to objects which do not have natural fixed-length representations. Kernel similarity functions are ubiquitous and can be devised for any new class of objects, such as images and documents (which can be considered as points in $\mathbb{R}^{\mathcal{D}}$), to more abstract objects like strings (protein sequences (Leslie et al., 2002), text), graphs (molecules (Borgwardt et al., 2005), brain neuron activation paths), and time series (music, financial data) (Müller et al., 1997). The beauty of kernels is the renowned "kernel trick" – the ability to evaluate similarity between any pair of objects in some richer but hidden feature space without requiring the explicit representations of those objects in the richer feature space.

The problem of nearest-neighbor search in metric spaces is a special case of max-kernel search where the closest object to the query with respect to a distance metric is sought. However, the requirements of a distance metric and the explicit representation of the objects in some fixed-length form make many efficient methods for exact and approximate nearest-neighbor search inapplicable to the general problem of max-kernel search.

In Chapter 6, we present a method to accelerate max-kernel search *for any class of objects with a corresponding Mercer kernel* (Curtin et al., 2013; Ram and Gray, 2012). The contributions are:

- The first concept for characterizing the hardness of max-kernel search in terms of the concentration of the similarities values – the *directional concentration*.

- A way to utilize an existing $O(n \log n)$ indexing scheme to index any set of objects *directly in the Hilbert space defined by the kernel* without requiring explicit representations of the objects in this space.

- A novel branch-and-bound algorithm on the index in the Hilbert space, which can achieve orders of magnitude speedups over linear search.

- The first $O(\log n)$ runtime bound for *exact* max-kernel search with our proposed algorithm for *any* Mercer kernel.

- Value-approximate and rank-approximate extensions to the exact max-kernel search algorithm.

1.2.5 Unanswered Questions

While I have tried to validate this thesis with the aforementioned contributions, there are various other related questions that I have encountered during this process which I am yet to answer (and sometimes, yet to formulate properly as a problem). In Chapter 7, I will discuss some of these questions to conclude this thesis.

CHAPTER II

A BRIEF HISTORY OF SIMILARITY SEARCH

The problem of nearest-neighbor search is formally defined as:

Nearest-neighbor search. Given a dataset R of size n, a query q, and a similarity function $\mathcal{S}\colon \{q\} \times R \to \mathbb{R}$, efficiently find a point $p \in R$ such that

$$\mathcal{S}(q, p) = \max_{r \in R} \mathcal{S}(q, r). \tag{1}$$

However, nearest-neighbor search is usually considered in a metric space associated with a distance metric:

Nearest-neighbor search in a metric space. Given a dataset $R \subset X$ of size n in a metric space $(X, \mathbf{d})$ and a query $q \in X$, efficiently find a point $p \in R$ such that

$$\mathbf{d}(q, p) = \min_{r \in R} \mathbf{d}(q, r). \tag{2}$$

Here the similarity function is based on a distance metric, low distance implying high similarity. For the ensuing discussion, we will consider nearest-neighbor search in a metric space since that is the most extensively studied general form of nearest-neighbor search. We will return to general similarity search (Equation (1)) in Section 2.5. A common extension of the nearest-neighbor problem is to find the k closest points to q instead of only the closest point. This is formally defined as:

k-nearest-neighbor search. Given a dataset $S \subset X$ of size n in a metric space $(X, \mathbf{d})$ and a query $q \in X$, efficiently find a set $S_k(q) \subset S$ of k points such that

$$\sum_{p \in S_k(q)} \mathbf{d}(q, r) = \min_{R \subset S, |R| = k} \sum_{r \in R} \mathbf{d}(q, r). \tag{3}$$

This chapter will be a review of existing methods for search and related tasks. I will begin by discussing existing methods for efficient exact nearest-neighbor search and

the need for approximation in Section 2.1. Following that, I will present the traditional form of approximation, distance-approximation, and the existing techniques for fast approximate search in Section 2.2. I will discuss different kinds of space-partitioning trees widely used for search as well as the closely related task of vector quantization in Section 2.3. In Section 2.4, I will review the use of machine learning techniques to enhance nearest-neighbor search, while in Section 2.5 I will discuss the task of nearest neighbor search beyond usual vector data and distance metrics. After reviewing this existing literature on similarity search, I will motivate the goals of this thesis in Section 2.6. I will present the need for the new paradigms that will subsequently be presented in this thesis.

2.1 Exact Nearest-Neighbor Search

The simplest approach of exhaustive search over R for the nearest-neighbor is easy to implement, but requires n distance computations for a single query, making it computationally prohibitive for moderately large n. Tree data structures built on the dataset are used to answer queries efficiently. Binary space-partitioning trees, like *kd*-trees (Friedman et al., 1977), ball trees (Omohundro, 1989) and metric trees (Preparata and Shamos, 1985) utilize the triangle inequality of the metric $\mathbf{d}(\cdot,\cdot)$ (commonly the Euclidean metric) for a branch-and-bound algorithm that can potentially answer queries in empirically $O(\log n)$ distance computations.

The generic tree-based algorithm using a binary tree is presented as Algorithm 1 (for exact search, the error is zero[1]). The algorithm determines the appropriate side of the space partition (the splitting hyperplane) for the given query and traverses down to the leaf (corresponding to a region in the data space) containing the query. The algorithm then finds a neighbor candidate for the query from that leaf and subsequently uses the current neighbor candidate to prune away certain branches of

[1]Algorithm 1 refers to binary tree based error-constrained search. However, exact search is essentially error-constrained search with zero error.

the tree (corresponding to regions in data space) while backtracking along the depth-first traversal path in the tree. The neighbor candidate is updated if better candidates are encountered. Non-binary cover trees (Beygelzimer et al., 2006) answer queries in theoretically bounded $O(\log n)$ time using a similar branch-and-bound algorithm.

Finding nearest-neighbors for $O(n)$ queries would require at least $O(n \log n)$ computations using the trees. The dual-tree algorithm (Gray and Moore, 2000) for nearest-neighbor search also indexes the queries in a tree data structure to amortize the cost of search over the queries. The dual-tree algorithm utilizes the intuition that queries spatially "close by" follow the same path down the tree on R. Hence the dual-tree search amortizes the search cost by consolidating such tree traversals into a single one. This algorithm shows orders of magnitude improvement empirically and has been to be shown to take a total of $O(n)$ time for answering $O(n)$ queries using the cover trees (Ram et al., 2009a) (which is essentially the lower bound).

The runtimes of all these algorithms to find the k-nearest-neighbors increase with increasing k. If $k = n$, then all the proposed algorithms would take the same time as the exhaustive search algorithm. But in most applications $k \ll n$, hence most of these algorithms are much more efficient than exhaustive search.

Remark. For any decently sized dataset, the preprocessing time required for tree construction on the dataset is generally insignificant compared to the improvement in efficiency it provides over exhaustive search. The efficiency gained is generally measured by the amount of "speedup" obtained in the query time over exhaustive search.

2.1.1 Nearest-Neighbor Search in High Dimensions

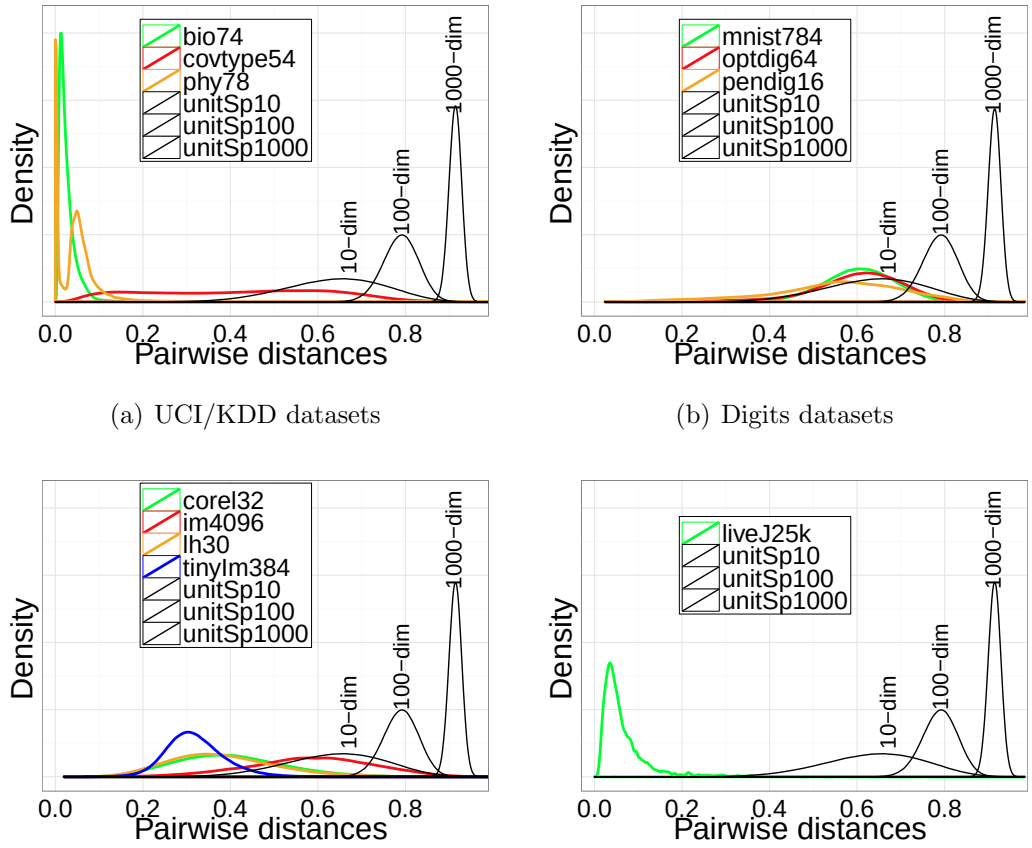
The frontier of research in nearest-neighbor methods is high dimensional problems where objects are defined by a large number of features. These problems stem from

common datasets like images and documents to micro-array data. But high dimensional data poses a possible problem for Euclidean nearest-neighbor search as implied by the following result:

Proposition 2.1.1. *(Hammersley, 1950) Let C be a $\mathcal{D}$-dimensional hypersphere with radius a. Let A and B be any two points chosen at random in C, the distributions of A and B being independent and uniform over the interior of C. Let r be the Euclidean distance between A and B ($r \in [0, 2a]$). Then the asymptotic distribution of r is $N(a\sqrt{2}, a^2/2\mathcal{D})$.*

This implies that in high dimensions, the Euclidean distances between uniformly distributed points lie in a small range of continuous values. This means that almost all pairs of points could be almost equally far away from each other and the nearest-neighbor distances approach the farthest-neighbor distances with increasing dimensionality. However, real high-dimensional datasets are generally assumed to have some structure which avoids this concentration of pairwise distances.

We present the density of the pairwise distances of 11 real datasets and 3 synthetic datasets in Figure 5. For all datasets, the pairwise distances are scaled to the range $[0, 1]$. The 3 synthetic datasets (presented in each plot as reference) correspond to uniformly distributed points in a 10, 100, 1000 dimensional unit hypersphere respectively. The description of the real datasets will be presented in Table 3 in Section 3.4 in Chapter 3 (the dimensionality of the real datasets vary from 16 to over 25000). It is apparent from Figure 5 that the pairwise distances of the synthetic datasets get more concentrated with increasing dimensions (as expected from Hammersley's theorem). However, the real datasets do not behave in any predictable manner. The pairwise distances of the digits and the image datasets have a distribution similar to a Gaussian. Some of the datasets have a peak in the pairwise-distance-distribution close to zero. This possibly implies the presence of some tightly packed clusters and some far away outliers.



(a) UCI/KDD datasets

(b) Digits datasets

(c) Image datasets

(d) Text datasets

Figure 5: Hammersley's theorem on real and synthetic datasets. The distribution of the pairwise distances (*please view in color*). The numbers at the end of the labels for each dataset corresponds to the dimensionality of the dataset.

Even in the presence of this structure in real datasets, *kd*-trees and metric trees can only scale to datasets of low and medium dimensionality respectively. Beyond that, the tree-based search performs no better than exhaustive search. This is usually attributed to the fact that the branch-and-bound algorithm with these trees are generally *unable to obtain sufficiently tight bounds in high dimensions*. Hence the algorithm is unable to efficiently remove (prune) large parts of the dataset R from the computation.

Another way of understanding this phenomenon is the following – in the worst-case, the number of leaves of the tree visited by the branch-and-bound algorithm

(Algorithm 1) is speculated to be exponential in the dimensionality of the data (the number of features representing every object). For high dimensional data, this dependence on the dimensionality of the data outweighs the dependence on the number of points, forcing the tree-based algorithm to perform comparably to exhaustive search.

There is empirical evidence that even in high dimensions, these tree-based algorithms are usually able to locate the nearest-neighbors quite early in the algorithm, and, in the absence of tight bounds, spend most of the search time validating the correctness of the neighbor candidate (Ram et al., 2012). Theoretically confirming this observation remains an open question. This inefficiency of exact search in high dimensions necessitates the approximation of nearest-neighbor search.

Algorithm 1 Error-constrained search with trees. Every non-leaf node N in the tree T is partitioned into a left (N.left) and a right (N.right) child with the hyperplane $(N.\mathbf{w}, N.b)$. The function 'cannot_prune_node(q, N, d, e)' decides if the node N can be safely ignored for query q with distance d to the current best neighbor candidate p and allowed approximation e.

Input: Data R, Tree T Query q, Allowed approximation e
Output: Neighbor candidate p
Initialize: Set $d \leftarrow \infty$, set stack S.push(T)
while S is non-empty **do**
  Node $N \leftarrow S$.pop
  if cannot_prune_node(q, N, d, e) **then**
    if is_a_leaf_node(N) **then**
      for $r \in N \cap R$ **do**
        if $\mathbf{d}(q, r) < d$ **then**
          $p \leftarrow r$; $d \leftarrow \mathbf{d}(q, r)$
        end if
      end for
    else
      if $\langle N.\mathbf{w}, q \rangle + N.b < 0$ **then**
        S.push(N.right); S.push(N.left)
      else
        S.push(N.left); S.push(N.right)
      end if
    end if
  end if
end while

2.2 Distance-Approximate Nearest-Neighbor Search

The traditional form of approximation views nearest-neighbor search as a discrete optimization problem $(\min_{r \in S} \mathbf{d}(q, r))$ and considers the following form of value approximation:

Distance-approximate nearest-neighbor search. Given a dataset $R \subset X$ of size n in a metric space $(X, \mathbf{d})$, $\epsilon > 0$ and a query $q \in X$, efficiently find a point $p' \in R$ such that

$$\mathbf{d}(p', q) \leq (1 + \epsilon) \min_{r \in R} \mathbf{d}(r, q). \tag{4}$$

The k-nearest-neighbor search is approximated in a similar manner:

Distance-approximate k-nearest-neighbor search. Given a dataset $R \subset X$ of size n in a metric space $(X, \mathbf{d})$, $\epsilon > 0$ and a query $q \in X$ with its k-nearest-neighbors $S_k(q)$ in R, efficiently find a set $S_k^{\epsilon}(q) \subset R$ of k points such that

$$\forall p \in S_k(q) \exists \text{ a distinct } p' \in S_k^{\epsilon}(q) \text{ such that } \mathbf{d}(q, p') \leq (1 + \epsilon)\mathbf{d}(q, p). \tag{5}$$

Many algorithms for distance-approximate nearest-neighbor search can be broadly grouped into three categories: (1) methods using hierarchical trees, (2) methods using skip-lists, and (3) methods utilizing random projections. The literature on distance-approximate nearest-neighbor search is huge and this is not a complete survey. However, we present the different flavors of existing methods for distance-approximate nearest-neighbor search.

Hierarchical structures. The distance approximation can be achieved with space-partitioning trees by using a relaxed branch-and-bound search algorithm to prune more aggressively. This relaxed algorithm is shown in Algorithm 1. A node is now pruned if the current upper bound to the nearest-neighbor distance (i.e. the distance to the current best nearest-neighbor candidate) for a query is less than $(1 + \epsilon)$ times

the distance of the query to the node in question. This returns a $(1+\epsilon)$-distance-approximate nearest-neighbor and provides some speedup over the exact search algorithm. Another approach modifies the tree to obtain the desired approximation with a root-to-leaf traversal of the tree instead of the branch-and-bound algorithm for significant speedups (Liu et al., 2005). This root-to-leaf traversal is usually referred to as *defeatist search*. This algorithm achieves desired approximation by allowing sibling nodes in binary space-partitioning trees to share points near their boundaries to get *spill trees*. These algorithms have been shown be empirically efficient, but lack any theoretical guarantees.

The idea of *approximately correct* nearest-neighbor (satisfying Equation (4)) is further extended to a formulation where the $(1+\epsilon)$ bound can be exceeded with a low probability δ, thus forming the PAC-nearest-neighbor search framework (Ciaccia and Patella, 2000). The distribution of the distances of the query from the points in R is estimated to predict an approximate upper-bound on the nearest-neighbor distance for a query. This estimate is used to improve the efficiency of the branch-and-bound algorithm on a metric tree by allowing the search algorithm to ignore nodes in the tree farther away than this predicted upper bound. This results in 1-2 orders of magnitude improvement in moderately large datasets.

Various tree based algorithms have rigorous theoretical bounds. A hierarchical decomposition of the input space containing n points in $\mathbb{R}^{\mathcal{D}}$ into a *balanced box-decomposition tree* in $O(\mathcal{D}n \log n)$ time and $O(\mathcal{D}n)$ space can return a $(1+\epsilon)$-distance-approximate nearest-neighbor of a query in $O\left(\lceil 1 + 6\mathcal{D}/\epsilon \rceil^{\mathcal{D}} \log n\right)$ time (Arya et al., 1998). In general, the distance-approximations to the k-nearest-neighbors of a query can be computed in additional $O(k\mathcal{D} \log n)$ time. Krauthgamer and Lee (2004) build a hierarchy of progressively finer ϵ-nets of size $O(n)$ which can answer a query approximately in $O(\log \phi + \epsilon^{-O(d)})$ where ϕ is the *aspect ratio* of the set R with respect

to the metric $\mathbf{d}(\cdot,\cdot)^2$, d is the intrinsic dimension of R, and ϵ is the distance error. A faster construction of these hierarchical nets proposed by Har-Peled and Mendel (2005) has an expected preprocessing time of $2^{O(d)} n \log n$ with $2^{O(d)} n$ space requirement and a similar query time of $O(2^{O(d)} \log n + \epsilon^{-O(d)})$. If the distance between points can be computed in constant time, this query time is essentially optimal since there are examples of point sets in which the query time is $2^{\Omega(d)} \log n$, and examples in which the query time is $\epsilon^{-\Omega(d)}$ (Krauthgamer and Lee, 2004).

Skip-list. Skip-lists (Pugh, 1990) have been used for the purpose of nearest-neighbor search. Any n element set $S \subset \mathbb{R}^{\mathcal{D}}$, using randomization, can be preprocessed into a "flattened" skip-list in higher dimensions of size $O(n \log n)$, and a query can be answered approximately in $O(\log^3 n)$ expected time (Arya and Mount, 1993). Clarkson (1994) provides an algorithm that builds a skip-list of size $O(m\eta)O(1/\epsilon)^{(\mathcal{D}-1)/2}$, where $\eta = \log(\phi/\epsilon)$ and ϕ is the aspect ratio of the set R, and approximately answers a nearest-neighbor query in $O(\log n)O(1/\epsilon)^{(\mathcal{D}-1)/2}$ with high probability.

Random projections. The Johnson-Lindenstrauss transform (Johnson and Lindenstrauss, 1984) allows embedding of data in $\ell_2^{\mathcal{D}}$ into $\ell_2^{O(\log n)}$ (effectively reducing the dimensionality of a high-dimensional dataset) while approximately preserving the pairwise distances of the points by projecting the points onto multiple random directions (this is also known as a low-distortion embedding). Kleinberg (1997) proposed two algorithms for distance-approximate search using projections onto random lines through the origin – one answers queries in $O((\mathcal{D} \log^2 \mathcal{D})(\mathcal{D} + \log n))$ time with storage requirement of $O(n \log \mathcal{D})^{2\mathcal{D}}$, other runs in $O(n + \mathcal{D} \log^3 n)$ time with

[2]The aspect ratio is defined as the ratio between the largest and smallest inter-point distances in R: $\phi = \frac{\max_{x,y \in R} \mathbf{d}(x,y)}{\min_{x,y \in R} \mathbf{d}(x,y)}$

$O(\mathcal{D}n \text{ poly} \log(\mathcal{D}n))$ space requirements. Kushilevitz et al. (1998) improved these results to obtain a query time of $O(\mathcal{D}\epsilon^{-2} \text{ poly} \log(\mathcal{D}n))$ in the $\mathcal{D}$-dimensional Hamming cube with the ℓ_1 metric, and $O(\mathcal{D}^2 \text{ poly}(1/\epsilon) \text{ poly} \log(\mathcal{D}n))$ in the Euclidean space with $(\mathcal{D}n)^{O(1)}$ storage .

Indyk and Motwani (1998) proposed the first algorithm to obtain $\text{poly}(\mathcal{D}, \log n, \epsilon^{-1})$ query time and polynomial storage for fixed ϵ by reducing a distance-approximate nearest-neighbor query to $O(\log^2 n)$ ϵ-PLEB (approximate point location in equal balls) queries (later improved to $O(\log(n/\epsilon))$ queries by Har-Peled (2001)). *Locality sensitive hashing* (LSH) (Gionis et al., 1999) hashes the data into buckets using hash functions which guarantee that "close" points are hashed into the same bucket with high probability and "farther apart" points are hashed into the same bucket with low probability. LSH answers approximate PLEB queries in $O(n^{1/1+\epsilon})$ time with near-quadratic storage (for small ϵ). Random projections are used to perform dimensionality reduction to provide $O(\epsilon^{-2}\mathcal{D} \log n)$ query times for ϵ-PLEB with $n^{O(\epsilon^{-2})}$ storage. This method has significant improvements in running times over traditional methods in high dimensional data and is shown to be highly scalable. Datar et al. (2004) extend the LSH method to the Euclidean space. LSH is widely used in practice, especially in the computer vision community for finding similar images.

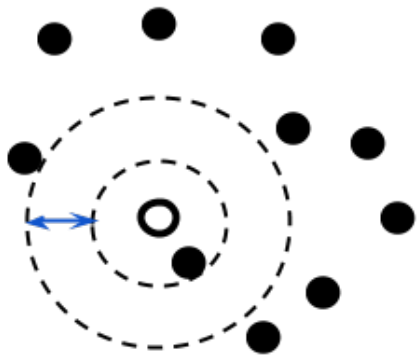


Figure 6: Planted nearest-neighbor model.

The authors suggest that LSH performs well when the dataset fits the "planted

nearest-neighbor model", "in which there are few points that are relatively close to the query point, and most of the database points lie quite far from the query point". Figure 6 provides an artificial example of the planted nearest-neighbor model. He et al. (2012) presented a quantification of this notion of planted nearest-neighbor known as the *relative contrast*. Relative contrast is defined as the ratio of the average distance of the query to the points in the set to the nearest-neighbor distance of the query. Higher the relative contrast for a problem instance, more planted the nearest-neighbor.

The random projections can be preconditioned by the Fourier transform (via fast Fourier transform) to obtain *Fast-Johnson-Lindenstrauss-Transform* (Ailon and Chazelle, 2006). This embeds $\ell_2^{\mathcal{D}}$ to $\ell_p^{O(\log n)}$, $p = 1, 2$ in $O\left(\mathcal{D}\log\mathcal{D} + \min\{\mathcal{D}\epsilon^{-2}\log n\,,\ \epsilon^{p-4}\log^{p+1} n\}\right)$ time, where ϵ controls the distortion in the embedding. This is used to provide two different algorithms for distance-approximate nearest-neighbor: (1) the first algorithm works in $\mathbb{R}^{\mathcal{D}}$ and answers a query approximately in $O(\mathcal{D}\log\mathcal{D} + \epsilon^{-3}\log^2 n)$ time with $n^{O(\epsilon^{-2})}$ storage, (2) the next algorithm works in the $\mathcal{D}$-dimensional Hamming cube and answers a query in $O((\mathcal{D} + \epsilon^{-2}\log n)\log n)$ time with $\mathcal{D}^2 n^{O(\epsilon^{-2})}$ storage.

Hybrid spill trees (Liu et al., 2005) combines two different techniques used in nearest-neighbor search by building hierarchical spill trees on randomly projected data to obtain significant speedups on high dimensional data.

2.3 Space-Partitioning Trees

Binary space-partitioning trees (or BSP-trees) are simple data structures and are heavily used in practice for search. Standard worst case analyses of BSP-trees in high dimensions usually lead to trivial worst case performance guarantees (such as, an $O(n)$ search time guarantee for a single nearest-neighbor search query in a set of n points). This can generally be attributed to the "curse of dimensionality" – the high

dimensionality can force the search algorithm to visit every node in the BSP tree. However, these BSP trees are still widely used in practice, even in the presence of more sophisticated methods.

There are several heuristics for the construction of BSP-trees. The most popular *kd*-tree uses axis-aligned splits (Algorithm 2), often employing a median split along the coordinate axis with the largest spread. The *principal-axis tree* (*PA*-tree) splits the data in each level at the median along the principal eigenvector of the covariance matrix of the data in that node (Sproull, 1991; McNames, 2001; Verma et al., 2009) (Algorithm 4). Another heuristic partitions the space based on a 2-means clustering solution for the data in the region to form the *two-means tree* (*2M*-tree) (Fukunaga and Nagendra, 1975; Nister and Stewenius, 2006) (Algorithm 5). The more recent *random-projection tree* (*RP*-tree) is built by projecting the data along a random unit direction and choosing an appropriate splitting threshold (Dasgupta and Freund, 2008) (Algorithm 3).

Algorithm 2 Axis-aligned *kd*-tree split.

Input: Set S, Region A
Output: Regions A_l and A_r

$d_s \leftarrow \arg\max_{i \in \{1:\mathcal{D}\}} range(S(i))$
$b \leftarrow \text{median}\{x(d_s) : x \in A \cap S\}$
$A_l \leftarrow \{x(d_s) \leq b\}$
$A_r \leftarrow \{x(d_s) > b\}$

Algorithm 3 A *RP*-tree split.

Input: Set S, Region A
Output: Regions A_l and A_r

$z \leftarrow$ a random standard normal vector
$b \leftarrow \text{median}\{\langle x, z\rangle : x \in A \cap S\}$
$A_l \leftarrow \{x : \langle z, x\rangle \leq b\}$
$A_r \leftarrow \{x : \langle z, x\rangle > b\}$

Algorithm 4 A *PA*-tree split.

Input: Set S, Region A
Output: Regions A_l and A_r

$R \leftarrow \text{centered}(A \cap S)$
$p \leftarrow$ the principal eigenvector of $R^\top R$
$b \leftarrow \text{median}\{\langle x, p\rangle : x \in A \cap S\}$
$A_l \leftarrow \{x : \langle p, x\rangle \leq b\}$
$A_r \leftarrow \{x : \langle p, x\rangle > b\}$

Algorithm 5 A *2M*-tree split.

Input: Set S, Region A
Output: Regions A_l and A_r

$M_1, M_2 \leftarrow$ 2-means solution for $A \cap S$
$A_l \leftarrow \{x : \|x - M_1\| \leq \|x - M_2\|\}$
$A_r \leftarrow \{x : \|x - M_1\| \leq \|x - M_2\|\}$

Figure 7: Splitting heuristics. Some hyperplane based splitting rules for binary space-partitioning trees.

The favorable performance of different BSP-trees in high dimensions is usually attributed to the low "intrinsic" dimensionality of real data. However, no clear relationship between search performance of BSP-trees and intrinsic dimensionality has been established. Moreover, the precise definition of the data-dependent intrinsic dimensionality varies between problems. Recent work by Verma et al. (2009) has established novel rigorous guarantees on the *vector quantization* performance of several of these BSP-trees. These guarantees present precise relationships between the quantization performance of some of these BSP-trees and intrinsic data-dependent properties.

Given a set of points (vectors) $S \subset \mathbb{R}^D$ of n points, the task of vector quantization is to generate a set of points $M \subset \mathbb{R}^D$ of size $k \ll n$ with low average quantization error. The optimal quantizer for any region A is given by the mean $\mu(A)$ of the data points lying in that region. The quantization error of the region A is then given by

$$\mathcal{V}_S(A) = \frac{1}{|A \cap S|} \sum_{x \in A \cap S} \|x - \mu(A)\|_2^2 , \tag{6}$$

and the average quantization error of a collection of regions $\{A_1, \ldots, A_k\}$ is given by:

$$\mathcal{V}_S(\{A_1, \ldots, A_k\}) = \frac{\sum_{i=1}^k |A_i \cap S| \mathcal{V}_S(A_i)}{\sum_{i=1}^k |A_i \cap S|}. \tag{7}$$

Note that we use the subscript S to denote the quantization error of the data in S lying within the region in question. This is because the quantization error depends on both the region and the set of points. We will use this subscript notation unless it is obvious from the context.

Tree-based *structured* vector quantization in widely used for efficient vector quantization – a BSP-tree of depth $\log_2 k$ on S partitions the region containing the set S into k disjoint regions, with the centroid of the points in each region as the quantizer. The theoretical results established for various tree-based vector quantization guarantee the improvement in average quantization error obtained by partitioning any single region (with a single quantizer) into two disjoints regions (subsequently with

two quantizers). The guarantee for various binary trees have this following general form (introduced in Freund et al. (2007)):

Definition 2.3.1 (Quantization error improvement)**.** *Given a set of points (vectors) $S \subset \mathbb{R}^D$, a region A split into two disjoint regions $\{A_l, A_r\}$, and a quantity β depending on the data in the region A, the quantization error improvement is characterized by:*

$$\mathcal{V}_S(\{A_l, A_r\}) < (1 - 1/\beta)\, \mathcal{V}_S(A). \tag{8}$$

Table 1: Quantization error decrease rates for various trees. $\lambda_1, \ldots, \lambda_D$ are the (sorted) eigenvalues of the covariance matrix of the points in $A \cap S$, and $d_c < D$ is the covariance dimension of data in region A. All the results are due to Verma et al. (2009) which also has the definition of d_c. No single split guarantees for *kd*-trees are known to us.

Tree	**Definition of β**
PA-tree	$O(\rho^2) - O(\rho)$: $\rho \doteq \left(\sum_{i=1}^{D} \lambda_i\right)/\lambda_1$
RP-tree	$O(d_c)$
kd-tree	$\times$
2M-tree	optimal (smallest possible)

This general definition characterizes the quantization performance of different trees, which depends inversely on the data dependent quantity β – lower β implies better quantization performance. We present the definition of β for different BSP-trees in Table 1. For the *PA*-tree, the improvement rate depends on the ratio of the sum of the eigenvalues of the covariance matrix of data $(A \cap S)$ to the principal eigenvalue. The improvement rate of the *RP*-tree depends on the covariance dimension of the data in the node A ($\beta = O(d_c)$) (Verma et al., 2009), which roughly corresponds to the affine plane of lowest dimensionality that can best approximate the data. The *2M*-tree does not have an explicit form for β but it has the optimal theoretical improvement rate for a single partition. This is because the 2-means clustering objective is equal to $|A_l|\mathcal{V}(A_l) + |A_r|\mathcal{V}(A_r)$ and minimizing this quantity maximizes

the improvement. Note that the 2-means problem is NP-hard and generally an approximate solution is used in practice. We will use these quantization performance guarantees to obtain guarantees on nearest-neighbor approximation.

Algorithm 6 A split by distance to remove outliers.

Input: Set S, Region A
Output: Regions A_l and A_r

$b \leftarrow \text{median}\{\|z - \mathit{mean}(A)\| : z \in A \cap S\}$
$A_l \leftarrow \{x \in A\colon\ \|x - \mathit{mean}(A)\| \leq b\}$
$A_r \leftarrow A \setminus A_l$

Algorithm 7 Construction of a binary space-partitioning tree.

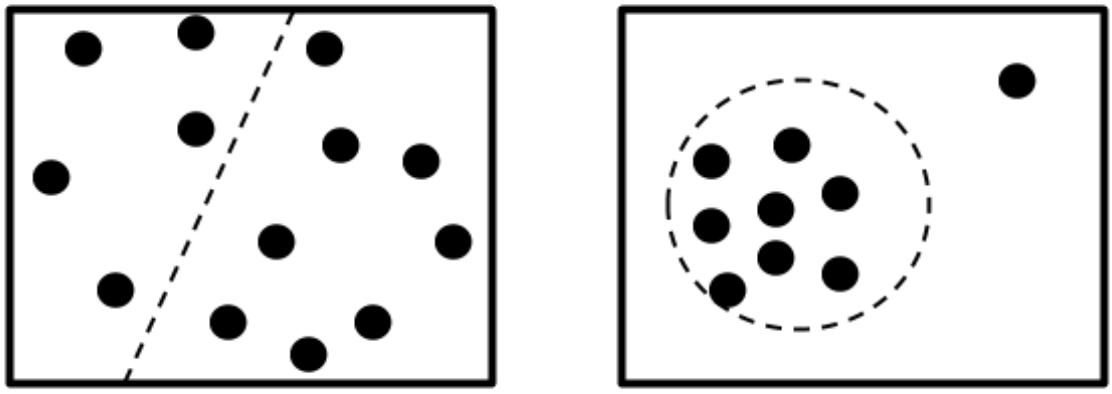
Input: Set S
Output: BSP-tree T

Initialize: $T \leftarrow$ region containing S, set queue Q.enqueue(T)
while Q is not empty **do**
  Current node $A \leftarrow Q$.dequeue
  if $|A \cap S| > \mathit{MinimumSize}$ **then**
    if $\Delta^2(A) < \eta V_S(A)$ **then**
      Split A with a hyperplane into A_l and A_r
    else
      Split A by distance into A_l and A_r
    end if
    A.left $\leftarrow A_l$; A.right $\leftarrow A_r$
    Q.enqueue(A_l); Q.enqueue(A_r)
  end if
end while

These theoretical results are valid under the condition that there are no outliers in the node A (containing the set of points $A \cap S$). This is characterized as

$$\Delta_S^2(A) \leq \eta \mathcal{V}_S(A),$$

where $\Delta_S^2(A) = \max_{x,y \in A \cap S} \|x - y\|^2$ for a fixed $\eta > 1$. This notion of the presence of outliers was first introduced by Dasgupta and Freund (2008) for the theoretical analysis of the *RP*-trees. Verma et al. (2009) describe outliers as "points that are much farther away from the mean than the typical distance-from-mean". In this



(a) Split by hyperplane (b) Split by distance

Figure 8: BSP-tree splits. The two kinds of splits used for the tree construction.

situation, an alternate type of split is used to remove these outliers (Algorithm 6) and this split is guaranteed to reduce the diameter of the nodes by a constant fraction (the original result is modified here to follow the notation used here):

Proposition 2.3.1 (Diameter decrease)**.** *(Dasgupta and Freund, 2008, Lemma 12) Suppose* $\Delta_S^2(A) > \eta \mathcal{V}_S(A)$ *so that* A *is split using Algorithm 6 into* A_l *and* A_r*. Then*

$$\frac{|A_l \cap S|\Delta^2(A_l) + |A_r \cap S|\Delta^2(A_r)}{|A \cap S|} \leq \left(\frac{1}{2} + \frac{4}{\eta}\right)\Delta^2(A). \tag{9}$$

The split by distance (Figure 8(b)) is used until a region does not violate the outlier condition, at which point, a hyperplane split (Figure 8(a)) is used. The implicit assumption is that the split by distance is not used very often in the tree construction.

These results on the quantization performance of BSP-trees indicate that different heuristics are adaptive to different properties of the data. Lower the quantization error, tighter is the index created by the BSP-tree. However, there is no existing result that relates this performance of BSP-trees to the search performance of these same trees. The precise connection between the quantization performance and the search performance of these BSP-trees is yet to be found.

2.4 Learning to Search

The BSP-tree construction hierarchically partitions the space based on certain heuristic. This split is a decision boundary that determines a query's candidate set. Instead of using a heuristic, the decision boundary can be learned to minimize runtimes and approximation error. This idea has been recently used by (Cayton and Dasgupta, 2007; Li et al., 2011). These methods use machine learning techniques for search that empirically minimize a combination of approximation error and search time. Cayton and Dasgupta (2007) theoretically demonstrate that the performance of the learned tree on the training set is close to the performance in general. The idea of learning hyperplane splits to reduce search time was initially proposed by Maneewongvatana and Mount (2002). Clarkson (1999) uses training queries to learn a data structure that effectively tries to estimate the Voronoi tessellation of the reference set.

Learning to search has garnered a lot of interest in the computer vision and representation learning community, leading to various data-dependent extensions to locality-sensitive hashing (LSH). The idea of LSH is to map data points to binary codes such that similar objects (based on some metric) have the same binary codes. Andoni and Indyk (2006) suggested calculating every bit in the code by a random linear projection followed by a random threshold. This leads to long code lengths for a low distortion Hamming cube embedding. The search for a new query involves generating a code for the query and then seeking the corresponding bucket in the hash lookup table to find the similar objects.

Long codes imply high precision, but lead to low recall in the hash lookup table. Hence, plain-vanilla LSH uses multiple hash lookup tables to get enough recall, leading to increased storage and longer search query times. Machine learning techniques have thus been used to learn shorter and shorter codes. A simple data-dependent code generation technique based on principal component analysis is as follows – for a binary code of length c, project the data to the space spanned by the top c eigenvectors of

the covariance matrix of the (centered and scaled) data and assign every point the binary code corresponding to the vertex in the c-dimensional binary Hamming cube closest to the point's projection. This was improved upon by Gong and Lazebnik (2011). They proposed an orthogonal rotation of the projected data to minimize the quantization error of mapping the projected data to the vertices of the c-dimensional binary hypercube.

Various "supervised" hashing methods exist which make use of some extra information. Cayton and Dasgupta (2007) still use random linear projections but learn data-adaptive thresholds using example queries to reduce the code length and increase recall. A nonlinear embedding to the Hamming cube is learned using a restricted Boltzmann machine by Salakhutdinov and Hinton (2007). If the similarity graph of the data is based on the Euclidean distance, spectral hashing (Weiss et al., 2008) generates bits by thresholding a subset of the eigenvectors of the Laplacian of the similarity graph. To generate code for out-of-sample queries, the graph Laplacian eigenvectors are extrapolated to eigenfunctions. Semi-supervised hashing (Wang et al., 2010) incorporates given pairwise semantic similarity and dissimilarity constraints from labelled data to learn hash functions that empirically minimize error on the labelled data (in terms of the code mismatch for similar objects and code match for dissimilar objects) while maximizing variance and independence of the bits on labelled and unlabelled data to minimize the code length. In most cases, the codes are learned one bit at a time. All codes can also be learned together at the same time by considering the code as a structured label of the point. The structural SVM framework has been used to learn such codes at the same time while minimizing the distortion in the Hamming space embedding (Norouzi and Fleet, 2011).

The widespread interest in the use of machine learning techniques to enhance search is awarded by better empirical performance in terms of precision and recall in the search for similar images. All these various data-dependent methods try to adapt

to some structure in the data. In the case of binary code generation of LSH, this generally amounts to capturing the variance in the data with the minimum number of bits. However, the precise (theoretical) relationship between these different data-dependent hashing techniques and their search performance still remains unknown.

2.5 General Similarity Search

Since similarity search is ubiquitous, a lot of research has been done for similarity search. Various domains have very domain specific but highly efficient search techniques. For example, BLAST (Altschul et al., 1990) is the method of choice for biological subsequence search. Rakthanmanon et al. (2012) present a search technique for time-series data that scales up to a trillion objects. However, to solve similarity search in a general sense, the search problem has to be first posed in a general form. To this end, the problem of max-kernel search has obtained some attention.

Max-kernel search finds the best match for a query with respect to a kernel similarity function. These kernel functions are widely used in machine learning. Not only do these kernel functions allow for a variety of similarity functions for a single class of objects, they are applicable to a wide variety of data types. Kernel functions provide a versatile class of similarity functions. In fact, there is a subfield in machine learning that focuses on engineering and developing appropriate kernel functions for data types and applications at hand (Table 2 presents some examples of kernel functions).

Table 2: Some examples of kernel similarity functions for different data types.

Objects	**Kernel functions**
Images	*linear, Gaussian, polynomial, pyramid match*
Documents	*cosine*
Sequences	*p-spectrum, matching and alignment kernels*
Trees	*subtree, syntactic, partial tree*
Graphs	*random walk*
Time-series	*cross-correlation, dynamic time-warping*
Natural language	*convolution, decomposition, lexical, semantic*

Although there are existing techniques for max-kernel search, *almost all of them solve the approximate problem under restricted settings*. The most common assumption is that the objects are points in some metric space and the kernel function is *shift-invariant* – a monotonic function of the distance between the two objects ($\mathcal{K}(p, q) = f(\|p - q\|)$), such as the Gaussian radial basis function (RBF) kernel. For shift-invariant kernels, a tree-based recursive algorithm has been shown to scale to large sets for maximum-a-posteriori inference (Klaas et al., 2005). However, a shift-invariant kernel is only applicable to objects already representable in some metric space. In fact, the max-kernel search with a shift-invariant kernel is equivalent to nearest-neighbor search in the input space itself, and can be solved directly using existing methods for nearest-neighbor search, an easier and better-studied problem.

For shift-invariant kernels, the points can also be explicitly embedded in some low-dimensional metric space using random Fourier features such that the inner product between the representations of any two points approximates their corresponding kernel value (Rahimi and Recht, 2007). This step takes $O(n\mathcal{D}^2)$ time for a set R of n points in $\mathbb{R}^{\mathcal{D}}$ and can be followed by nearest-neighbor search on this embedding to accomplish max-kernel search. Raginsky and Lazebnik (2009) build on these random features to obtain short binary codes that can then be used for efficient approximate similarity search with LSH. This method of utilizing random features to approximate the inner product is closely related to the low dimensional mapping for black-box kernel functions proposed by Balcan et al. (2006). This mapping projects points into a low dimensional space such that if the classes are separable and have a large margin between them in the kernel space induced by the black-box kernel function, then the classes are still separable with a large margin in the low dimensional projections. This method does not necessarily approximate the kernel value, but rather preserves separability. The low dimensional mappings are performed by accessing samples from the underlying distribution of the unlabelled points.

Locality-sensitive hashing (LSH) (Gionis et al., 1999) has been widely used for image matching, but only with explicitly representable kernel functions which admit a locality sensitive hashing function (Charikar, 2002)[3]. Kulis and Grauman (2009) apply LSH to solve max-kernel search *approximately* for normalized kernels without any explicit representation. Normalized kernels restrict the self-similarity value to a constant ($\mathcal{K}(x,x) = \mathcal{K}(y,y)\ \forall\ x,y \in S$). Hashing requires a random vector in the kernel space, but this is not obtainable since the explicit representation of the objects in the kernel space are not generally known. Kulis and Grauman (2009) accomplish this in a very smart way by creating a quasi-random-Gaussian vector in the kernel space by using p points in the dataset. The preprocessing time for this kernelized LSH is $O(p^3)$ and a single query requires $O(p)$ kernel evaluations. Here p controls the accuracy of the algorithm – larger p implies better approximation; the suggested value for p is $O(\sqrt{n})$ and the approximation guarantees are inherited from LSH.

Another technique to perform LSH in the kernel space expresses the problem as a kernel SVM problem (Joly and Buisson, 2011). p points are randomly selected from the dataset and are randomly divided up into two equally sized classes. Then the kernel SVM problem is solved for these p artificially labelled points and the separating large margin hyperplane is used to assign a bit for every point and the query. c such hyperplanes are learned to obtain a c length binary code. The preprocessing step requires the training of c instances of a kernel SVM, each requiring $O(p^2 \sim p^3)$ time. A single query requires $O(cp)$ kernel evaluations. Much like kernelized LSH, this random maximum margin hashing technique inherits the approximation guarantees from LSH.

[3]The Gaussian and cosine kernels admit locality sensitive hashing functions with some modifications.

A case for un-normalized kernels. While some of the kernels used in machine learning (for example, the Gaussian and the cosine kernel) are normalized, some common kernels like the linear kernel (and the polynomial kernels) are not normalized. Many applications require un-normalized kernels: (1) In the retrieval of recommendations, the normalized linear kernel will result in inaccurate user-item preference scores. (2) In biological sequence matching with the domain-specific matching functions, $\mathcal{K}(x, x)$ implicitly corresponds to the presence of (genetically) valuable letters (like W, H, P) or invaluable letters (like X)[4] in the sequence x. This crucial information is lost in kernel normalization. We wish to consider general un-normalized kernels as well as the special case of normalized kernels.

None of the existing techniques can be directly applied to every instance of max-kernel search with general Mercer kernels and any class of objects. Moreover, almost all existing techniques resort to approximate solutions. However, these approximate techniques do exhibit good empirical performance, especially for the problem of image matching. In Chapter 6, I present a method for exact max-kernel search with any Mercer kernel with sub-quadratic preprocessing time and sub-linear query time, as well as approximate extensions for further efficiency.

2.5.1 Combinatorial Framework for Similarity Search

Beyond Mercer kernels, a more general form of similarity search was proposed by Goyal et al. (2008) in which we do not have access to the similarity function itself; only a *comparison oracle* is available which answers questions of the form "For any object a, is object b more similar to a than c?". The goal is to efficiently answer similarity search queries in this setting. This combinatorial framework for similarity search is applicable to various problems and this framework does not require explicit representations of the objects or even a precise similarity function, making it even

[4]The score matrix for letter pairs in protein sequences can be found at `http://www.ncbi.nlm.nih.gov/Class/FieldGuide/BLOSUM62.txt`.

more general than max-kernel search. Quoting Yury Lifshits (Lifshits, 2009), the combinatorial framework allows for a *"model of computation for problems dealing with an informal concept of closeness"*.

In such a general setting, certain assumptions are needed for efficient (and exact) similarity search. Goyal et al. (2008) proposed a very natural assumption of bounded *disorder constant*. This assumption implies two things: (i) if the neighbor rank of b with respect to a is small, then the neighbor rank of a with respect to b cannot be very high (this assumption provides a form of approximate symmetry for the neighbor ranking function), and (ii) if the neighbor rank of a with respect to c is small and the neighbor rank of b with respect to c is small, then the neighbor rank of a with respect to b is small as well (this assumption provides a form of approximate triangle inequality for the neighbor ranking function).

Given this assumption, two randomized algorithms for exact similarity search in the combinatorial framework was proposed by Goyal et al. (2008). The first algorithm performs a random walk on a quadratic sized data structure while the second algorithm performs a walk on a $O(n \log n)$ size data structure known as the *navigation array*. A deterministic algorithm for exact search utilizes a combinatorial net (Lifshits and Zhang, 2009)(which is combinatorial analog of a navigating net proposed by Krauthgamer and Lee (2004)).

2.6 The Need for New Paradigms

In this section, we will motivate the need for new paradigms for approximate search. We will first present the possible issues with the traditional relative value approximation. Then we will motivate the need for the paradigm of time-constrained search by presenting the issues with error-constrained search.

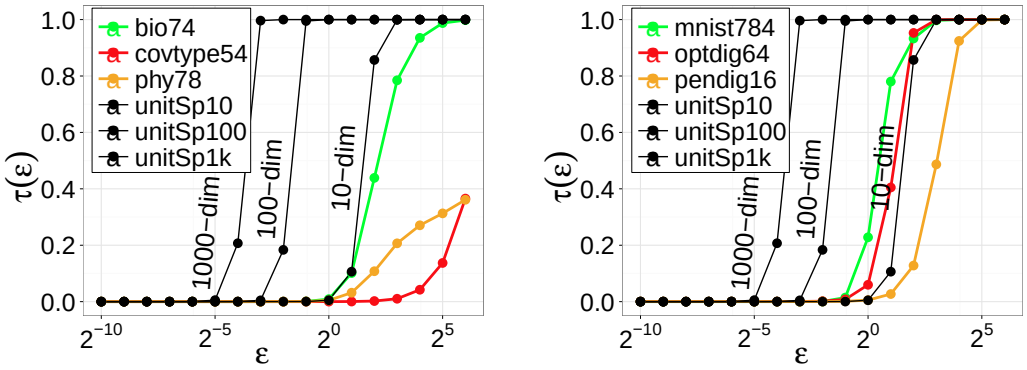
2.6.1 Possible Issues with Distance-Approximation

Exact nearest-neighbor search becomes hard in high dimensions mainly because of two reasons – (1) the possibility of concentrated pairwise distances, and (2) the inability to have "tight" indexing of the data. Multidimensional trees like *kd*-trees generally fail to perform well in high dimensions mainly because of the second reason, albeit there are moderately high dimensional sets on which the *kd*-trees provide over a magnitude of speedup over exhaustive search (for example, Figure 5(a)). Distance-approximation is common but only makes sense in the "planted nearest-neighbor" setting. This setting is intuitive since it is natural to assume that the boundary separating the similar objects (to a query) from the dissimilar ones has a significant margin in terms of their distances. However, given that this assumption holds, finding an appropriate ϵ for each dataset is not an easy task (and the appropriate value of ϵ might vary even between queries for the same dataset). If this assumption does not hold, it is not clear if distance-approximation is appropriate.

In this subsection, I will first discuss the appropriate choice for the approximation level ϵ, and then present two situations where distance-approximate nearest-neighbor search is not appropriate – (1) the pairwise distances are relatively concentrated, and (2) the pairwise distances are heuristic metrics developed to capture ordering information.

Choosing appropriate ϵ. Even if the similar points are significantly separated from the dissimilar points in a dataset, selecting the value of ϵ which matches this boundary is non-trivial – too low of an ϵ value allows for no approximation and too high of an ϵ trivializes the search since every point is now an approximate answer. Ideally, ϵ needs to be selected after an analysis of the distribution of the pairwise distances, where the latter task is computationally expensive. In most cases, the value of ϵ is chosen without the knowledge of the distribution of the distances and

hence lacks any meaningful interpretation. One usual technique of choosing ϵ is "guess and halve", selecting the value appropriate for the current need (based on required accuracy level or amount of time available). One can consider the (average) fraction of the points in dataset R within the $(1+\epsilon)$-nearest-neighbor ball of the query $\tau(\epsilon)$ – if $\tau(\epsilon)$ is large, then the value of ϵ is reduced, but only until ϵ is large enough to still be an approximation and allow for efficient search.



(a) UCI/KDD datasets

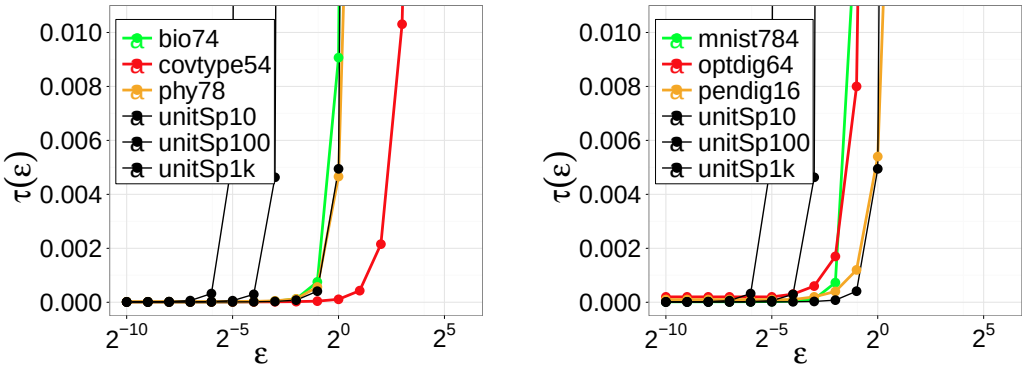
(b) Digits datasets

(c) Image datasets

(d) Text datasets

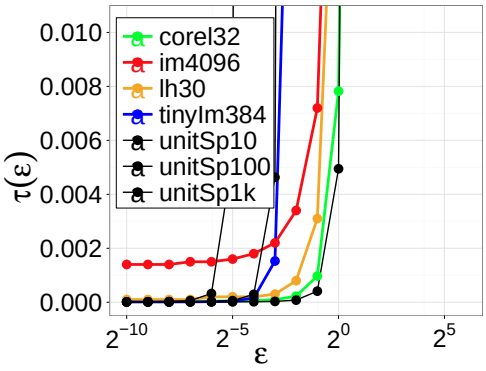
Figure 9: Effect of ϵ on neighborhood quality. This figure presents $\tau(\epsilon)$ with varying ϵ for different datasets (*please view in color*). The numbers at the end of the labels for each dataset corresponds to the dimensionality of the dataset.

We perform this task of choosing ϵ with respect to $\tau(\epsilon)$ for all the previously used datasets. The average $\tau(\epsilon)$ over all queries for varying values of ϵ is presented in Figure 9 (the results for the 3 synthetic datasets are again presented in each of the

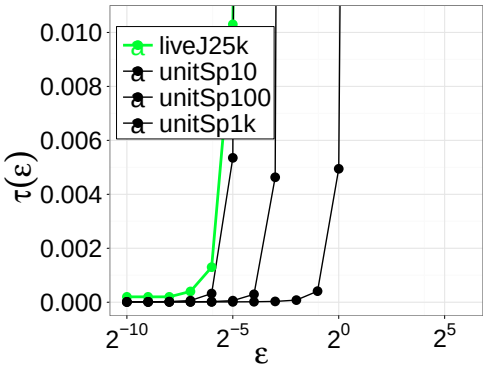


(a) UCI/KDD datasets (magnified)

(b) Digits datasets (magnified)



(c) Image datasets (magnified)



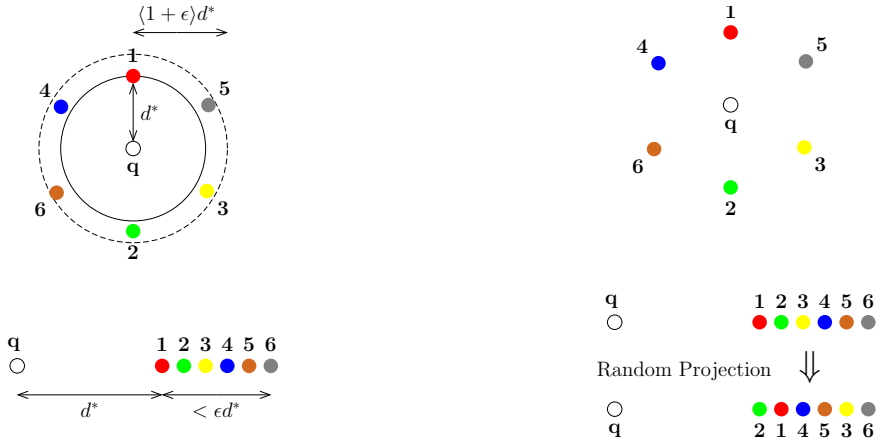
(d) Text datasets (magnified)

Figure 10: Effect of small ϵ on neighborhood quality. This figure presents $\tau(\epsilon)$ with varying ϵ for different datasets magnified for small ϵ (*please view in color*). The numbers at the end of the labels for each dataset corresponds to the dimensionality of the dataset.

plots for reference). It can be seen that for most datasets in Figure 9, after a point ($\epsilon \geq 0.25$), $\tau(\epsilon)$ is extremely sensitive to the value of ϵ – $\tau(\epsilon)$ varies from less than 0.1 to more than 0.8 in just 3 (or less) doubling of ϵ. For smaller values of ϵ (as shown in Figure 10 for the same datasets), $\tau(\epsilon)$ is really small and can be considered to obtain appropriate values of ϵ. Finding the "sweet spot" for ϵ with appropriate $\tau(\epsilon)$ requires this detailed analysis (something like Figure 9 & 10) for every dataset. Moreover, the appropriate value of ϵ also varies for each datasets, and it might even vary among queries for the same dataset.

A related issue is that many of the random-projection-based algorithms for distance-approximate nearest-neighbor search rely on the specified ϵ for the data preprocessing. This implies that the data preprocessing needs to be repeated for any new value of ϵ, making such algorithm less usable when the appropriate value of ϵ is not known beforehand. While this issue can be mitigated by performing the preprocessing at multiple values of ϵ, this creates a storage overhead.

Concentrated pairwise distances. If the pairwise distances are concentrated in a small interval, distance-approximation can trivialize the search. For example, if the all pairwise distances are within the range (100.0, 101.00), any distance approximation $\epsilon \geq 0.01$ is inappropriate. Another toy example demonstrating the above mentioned situation is shown in Figure 11(a). This is because the concentration of the pairwise distances removes the significant margin between the similar and dissimilar points (for a query). Moreover, in such a situation (where the distances lie in a small interval), random projection based dimensionality reduction can completely corrupt the ordering of the neighbors (Figure 11(b)), which could be critical.



(a) Bad approximation (b) Random Projection

Figure 11: Toy examples. Artificial situations where distance approximation is not the way to go (*please view in color*).

Moreover, the concentration of the pairwise distances is not limited to high-dimensional datasets. Datasets with low dimensional structure can still have their pairwise distances concentrated in a small interval. The pairwise distances can even get concentrated by the addition of uninformative dimensions to low dimensional data. For example, consider the two-dimensional dataset $(1,1),(2,2),(3,3),(4,4),\ldots$ with a query at the origin $(0,0)$. Appending non-informative dimensions to each of the points produces higher dimensional datasets of the form $(1,1,u_1,u_1',....)$, $(2,2,u_2,u_2',...)$, $(3,3,u_3,u_3',...)$, $(4,4,u_4,u_4',...)$, $\ldots$ where each of the u_i and u_i' are uniform noise. The pairwise distances of this dataset gets more concentrated with increasing dimensions.

Heuristic distance metrics. In many applications of nearest-neighbor search, the numerical values of the distance are not as important as the ordering of the distances of the query from the points – the ordering information defines similar and dissimilar points. Distance-approximate methods focus on the numerical values of the distances, and risk the loss of this ordering information. Moreover, in many datasets like documents, images and songs, the distance measures used are generally heuristics which try to preserve the relative ordering of the objects (Goyal et al., 2008). The numerical values of the distances themselves have no clear meaning. Moreover, methods using low distortion dimensionality reduction techniques (like random projections) run the risk of corrupting the ordering information of the pairwise distances which can be essential in some applications of nearest-neighbor search. This might be too large of a loss in accuracy for scalability.

These reasons imply that distance-approximation can be inappropriate in situations where the pairwise distances are concentrated and the orderings of the distances is crucial. Moreover, the unclear dependence between the user specified distance-approximation (ϵ) and the returned accuracy makes it hard to use. For these reason,

we propose a new form of approximation for nearest-neighbor search, rank approximation, in Chapter 3 which directly works with $\tau(\epsilon)$.

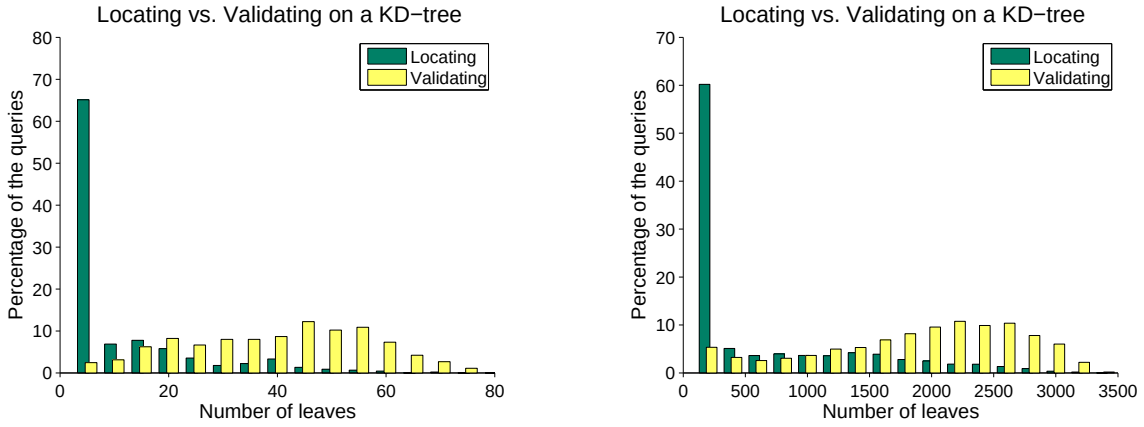
2.6.2 Possible Issues with Error-Constrained Search

A traditional theme in nearest-neighbor research is to develop algorithms that obtain the most efficient runtime for a fixed approximation (sometimes only with high probability). The exact search is a special case of this setting with zero error. While *error-constrained search* is widely studied, search approximation is necessitated by the lack of time. In error-constrained search, we usually have direct control on the allowed approximation level (or search error), but we lack any control on the amount of time the algorithm spends in answering the query. The search time can be implicitly controlled by the allowed search error. This poses the following issues – (1) for applications requiring search queries answered in a limited time, the choice of the appropriate search error (the parameter we have control over) is unclear, and (2) various error-constrained search algorithms spend time validating the search error bound, possibly wasting precious time in the time-constrained setting.

For the first point, the runtime guarantees for the error-constrained search algorithms can be used to guess the appropriate error that would guarantee a search result within the desired time budget. However, these runtime guarantees often have hidden constants and/or dependence on data dependent properties that cannot be easily computed. Nonetheless, it is still possible to obtain some appropriate guess of search error for our time constraint.

On the other hand, some error-constrained methods require the search to algorithmically validate that the error constraint is actually satisfied. This implies that just locating an acceptable candidate is not enough, a certificate is required for its acceptability. This validation phase is not computationally free. We do not have a precise grasp on how much of the search time is spent in the validation phase

for high dimensional data. We empirically verify this phenomenon by the following experiment:



(a) OPTDIGITS (b) MNIST

Figure 12: Location vs. validation in the depth-first search on a *kd*-tree. The histogram of the number of leaves of a *kd*-tree visited by the query to locate the true nearest neighbors are contrasted with the histogram of the total number of leaves eventually visited by the depth-first strategy on the *kd*-tree.

Experiment. We consider 2 digits image data sets - OPTDIGITS (64 dimensional) (Blake and Merz, 1998) and MNIST (784 dimensional) (LeCun, 2000). We perform the depth-first nearest-neighbor search for the exact nearest-neighbor of the each of the queries in the reference set (Algorithm 1) using *kd*-trees (with a leaf size of 30) to index the reference sets. We present the histogram of the number of leaves visited to locate the true nearest neighbor for each query and total number of leaves to find as well as guarantee its correctness in Figure 12. The maximum number of leaves visited by any query was 76 for OPTDIGITS and around 3500 for MNIST. In both the cases, around 60% of the queries located their true nearest neighbors in the first 5% of the leaves visited. But only less than 5% of queries finished validating the error bound within that time. This empirically exhibits the capability of the depth-first nearest-neighbor search on tree data structures to locate the correct answer early even in high dimensional data, but the failure to validate efficiently.

Applications requiring bounded errors require error-constrained search. However, for applications with limited time, it is suboptimal to spend the available time in guaranteeing the error bound instead of possibly improving upon the current result. However, the argument against validation is not necessarily that straightforward – part of the work done in the validation phase is to look for better candidates. Hence an easy fix to any error-constrained search algorithms is, if possible, to have a provision to stop the search algorithm at any time. This would allow the search algorithm to locate and validate as long as there is time available, while being able to return an answer within the time constraint by cutting the search algorithm short if required. And this is precisely the basis for some of the time-constrained search algorithms I will present in Chapter 4.

Remark. Note that there is a significant difference between the way LSH and the way tree-based error-constrained search algorithms deal with approximations. The usual tree-based search strategy (Algorithm 1) ensures the upper bound on the approximation by visiting all required parts of the tree, guaranteeing the approximation algorithmically, yielding more stable and accurate results (obviously, at the cost of some efficiency). With LSH, a query is answered by collecting the points in the buckets of the hash lookup tables which contain the query. The number of lookup tables used is chosen by making the failure probability small (here failure implies the inability to find an approximate nearest-neighbor). However, the actual search approximation is handled during the preprocessing where the buckets are created so as to answer $(1 + \epsilon)$-distance-approximate nearest-neighbor queries. Unlike the tree-based algorithm, the LSH search algorithm does not algorithmically verify the approximation guarantee. This makes LSH a highly efficient algorithm, albeit, at the cost of having inaccurate as well as no results with very low probability.

An interesting observation is that the requirement to consider multiple hash

lookup tables makes search with LSH easily suited for the time-constrained setting – the search can just go through the lookup tables one-by-one until time runs out. The only remaining issue is the choice of the approximation level that is generally required for the creation of the binary codes for the points – small ϵ indicates that only really close points get hashed into the same bucket, while a larger value of ϵ allows for less close points to also be hashed into the same bucket. This can be mitigated by creating hash tables for multiple values of approximations (multiple scales) at the cost of extra storage requirements. In this scenario, the time-constrained search algorithm can begin with hash lookup tables for the smallest scale and move up to larger scales as time permits. This is probably how LSH is usually applied in large scale limited time searches. This does imply that time-constrained search is not sometime entirely new and has possibly existed in practice for quite a while. However, we are not aware any formalization of time-constrained search in literature.

The goals of time-constrained search is very similar to the goals of the paradigm of *online aggregation* proposed by Hellerstein et al. (1997) for the task of computing aggregates in a database. The goal of this work is to compute aggregates (such as mean, counts, etcetera) for a chosen attribute(s) in a database table, but with an added desire to get good estimates early on, improving on the estimate with more time available. This is similar to our goals for time-constrained search where we desire a good neighbor candidate early on even if we cannot verify if our neighbor candidate is in fact good.

However, the focus in online aggregation is on aggregation tasks (as suggested by its name). Since early estimates are desired with confidence intervals, the authors suggest random access of the data that is being aggregated, emulating random samples from the data. Any form of sorting or indexing of the data, leading to non-random sequential access of the data, will lead to biased estimates. The goal in time-constrained search is a little different – we explicitly desire a non-random sequential access of the

data; we desire candidates closer to the search query as early as possible. This implies that some form of sorting or indexing is desirable in time-constrained search. This difference arises from the fact that nearest-neighbor search is more of an optimization task than an aggregation task, and the techniques used for aggregation are not directly applicable to the task of similarity search. However, the overall goals of good early solutions (possibly without validation) in online aggregation aligns with our goals in time-constrained search.

2.6.2.1 Doing Better in Time-Constrained Search

In the time-constrained setting, the goal is to locate the similar objects as early as possible. In LSH, this is being achieved by "learning" shorter and shorter binary codes that capture maximum information (implying high independence and variance among the bits) which are also easy to generate for a new out-of-sample query (Section 2.4). In BSP-trees, this can be achieved by "learning" trees on which the depth-first algorithm locate the similar objects early on, preferably in the first leaf visited. In terms of Figure 12, this implies a highly left-skewed green histogram.

In Chapter 5, we learn BSP-trees with improved time-constrained search performance. While there is a lot of research in learning hash functions, we focus on learning BSP-trees. This choice is primarily because BSP-trees present a hierarchical view of the data with increasing resolution, which allows any search algorithm to definitely find the best match if enough time is available (which is not necessary for LSH based methods). Moreover, hash based methods are prone to low recall (implying small number or zero returned neighbors), whereas the hierarchical tree can easily provide any desired amount of recall.

CHAPTER III

RANK APPROXIMATE SEARCH

When approximation is involved, a notion of error is necessary. The traditional form of approximation of optimization problems focus on the relative value error. The quality of an achieved solution depends on how close (relatively) the value of the objective function at this solution is to the optimal value of the objective function. Nearest-neighbor search is effectively an optimization problem, albeit a discrete optimization problem, where we seek the optimal solution from a large number of choices. Hence value approximation appears natural. However, discrete approximation presents a new form of error – the number of solutions better than our achieved solution. In this chapter, I explore this notion of error, which I call rank error.

The notion of rank approximation is introduced formally in Section 3.1 along with its application to k-nearest-neighbor search. I present a simple random sampling based algorithm for rank approximate search in Section 3.2, and combine this to a tree-index-based search scheme to have a possibly more efficient algorithm with better guarantees in Section 3.3. In Section 3.4, I evaluate the efficiency and accuracy of the proposed algorithms for rank approximate search and conclude with a discussion of the empirical results.

3.1 A New Notion of Error

The rank approximate nearest-neighbor search was recently proposed in Ram et al. (2009b). We slightly modify the definition for simplicity as follows (the following definition is for nearest-neighbor search in a metric space, but can easily be modified for general similarity search):

Rank approximate nearest-neighbor search. Given a dataset $R \subset X$ of size n in a metric space $(X, \mathbf{d})$, $\tau \in (0, 1)$ and a query $q \in X$, efficiently find a point $p' \in R$ such that

$$\frac{|\{r \in R : \mathbf{d}(q, r) < \mathbf{d}(q, p')\}|}{n} \leq \tau. \tag{10}$$

This formulation of approximate nearest-neighbor search introduces the approximation in the ranks of the nearest-neighbor (as opposed to the distance values). The true nearest-neighbor for a query is the rank 1 nearest-neighbor with a rank-error of zero; this new approximation accepts any neighbor with a true rank less than $\tau n + 1$ with respect to the query. Another way of looking at it is that this form of approximation allows the search to return any point within the closest $100\tau\%$-tile of the points for a query. For a dataset of size $n = 100$, a $\tau = 0.05$-rank-approximation implies that any point among the 6-nearest-neighbors of the query is acceptable. Similarly, for $n = 1000$, the same $\tau = 0.05$-rank-approximation accepts any point among the 51-nearest-neighbors of the query.

Rank approximation has the following advantages over distance approximation: (1) *This form of approximation is immune to the concentration of the pairwise distance* – for the same value of n and τ, this approximation is the same whether all the pairwise distances are in the interval $[100.0, 101.0]$ or the interval $[100.0, 10000.0]$. It is important to note that this robustness of rank approximation does not imply that the computational cost of achieving this approximation is independent of the distribution of the pairwise distances. (2) Given the size of the dataset, *the level of approximation, τ, can be intuitively selected as per the application.* For example, τ can be chosen as is – as the nearest $100\tau\%$-tile of the data for the query. τ can also be chosen such that any point among the t-nearest-neighbors of the query is desired (in this case, $\tau = (t-1)/n$). No analysis of the pairwise-distance distribution is required to select an appropriate value of τ. Obviously, a very low value of τ might imply no-approximation ($\tau < 1/n$). But this can be detected without any computation.

The k-nearest-neighbor search can be approximated in multiple ways in the rank approximation framework. We consider the approximation which allows the method to return any k points within the nearest τn points to the query. This is formally defined as:

Rank approximate k-nearest-neighbor search. Given a dataset $R \subset X$ of size n in a metric space $(X, \mathbf{d})$, $\tau \in (0, 1)$ and a query $q \in X$, efficiently find a set $S_k^\tau(q) \subset R$ of k points such that

$$\frac{\left|\left\{r \in R \colon \mathbf{d}(q, r) \leq \max_{p \in S_k^\tau(q)} \mathbf{d}(q, p)\right\}\right|}{n} \leq \tau. \tag{11}$$

3.2 Rank Approximate Search by Random Sampling

Given a query and its nearest-neighbor in a dataset R, it is easy to compute the distance error of any new point in R. However, the same is not true when dealing with ranks – given a query and its nearest-neighbor in R, it is not generally possible to guess the rank of any new point with respect to the query. The ordering information is required to obtain the rank of a new point and this information is much harder to obtain (possibly requiring computation of all pairwise distances of the query with points in R). This is the cost of working with a most robust form of approximation. This is possibly one of the reasons why distance approximate nearest-neighbor search is usually considered. Even though rank approximation appears hard, we present a really simple algorithm to achieve any user-specified rank approximation in this subsection.

We utilize random sampling to obtain this robust form of approximate nearest-neighbor search with high probability. Consider a τ-rank-approximate search problem. For a query q, if a single point p_1 is randomly sampled from R, then the probability that p_1 is not among the closest τ fraction of points to q is given by:

$$\Pr\left(\frac{|\{r \in R \colon \mathbf{d}(q, r) < \mathbf{d}(q, p_1)\}|}{n} > \tau\right) \leq 1 - \tau.$$

If m samples $p_1, p_2, \ldots, p_m$ are made from S with replacement, then the probability that none of them is among the closest τ fraction of points to q is given by:

$$\Pr\left(\frac{\left|\{r \in S : \mathbf{d}(q,r) < \min_{i=1,\ldots,m} \mathbf{d}(q,p_i)\}\right|}{n} > \tau\right) \leq (1-\tau)^m.$$

Hence, if the rank approximation is desired with probability greater than $(1-\delta)$, then the number of samples required can be determined by choosing the smallest m satisfying the inequality $(1-\tau)^m \leq \delta$. Algorithm 8 summarizes this method. Thus, the rank approximate search via random sampling has the following runtime guarantee:

Theorem 3.2.1. *For a given set $R \subset \mathbb{R}^{\mathcal{D}}$ of n points and a query q, and for $\tau \in (0,1)$ and $\delta > 0$, the τ-rank-approximate nearest-neighbor of q can be obtained via random-sampling with probability at least $(1-\delta)$ in time $O\left(\left\lceil \frac{\log 1/\delta}{\log 1/(1-\tau)} \right\rceil\right)$.*

Algorithm 8 Rank approximation by random sampling

Input: Dataset R, query q, rank-approximation τ, failure probability δ
Output: τ-rank-approximate nearest-neighbor of q

$m \leftarrow \left\lceil \frac{\log 1/\delta}{\log 1/(1-\tau)} \right\rceil$
$S' \leftarrow m$ random samples from R
return $\arg\min_{p \in S'} \mathbf{d}(q,p)$.

The runtime of Algorithm 8 depends on the number of samples $m = \lceil (\log \frac{1}{\delta} / \log \frac{1}{1-\tau}) \rceil$. The dependence on the dimension is solely through the computation of distances $\mathbf{d}(\cdot,\cdot)$ of the query to the samples. These distance computations can usually be done in $O(\mathcal{D})$ time. It is interesting to see that this runtime bound has no explicit dependence on the size n of the set S. This is because the rank approximation factor τ implicitly depends on n. The dependence on n can be made explicit in the following corollary by considering the rank approximation in a different manner:

Corollary 3.2.1. *For a given set $S \subset \mathbb{R}^d$ of n points and a query q, and for $1 < k < n$ and $\delta > 0$, a neighbor among the k-nearest-neighbors of q (implying the $(k - 1/n)$-rank-approximate nearest-neighbor of q) can be obtained via random-sampling with probability at least $(1 - \delta)$ in time $O\left(\left\lceil \frac{\log 1/\delta}{\log 1/(1-(k-1)/n)} \right\rceil\right)$.*

Caveat of Algorithm 8. For a fixed k and δ, the runtime of Algorithm 8 grows linearly in n. This can be seen as follows: For a fixed k , if $t(n_0)$ is the time taken by Algorithm 8 for n_0 points, then the time taken for cn_0 points $t(cn_0) > ct(n_0)$ for $c > 1$. This is because $|\log(1 - x)| > c\,|\log(1 - x/c)|$, by the Taylor-expansion of $\log(1-x)$ for $x \in (0, 1)$ and $c > 1$. This implies that, for a fixed k and δ, the runtime of Algorithm 8 grows at least multiplicatively with the size of the set R. Contrast this to an additive growth of an algorithm with only logarithmic dependence on the size of the set for a fixed k – if $t'(n_0) = C \log n_0$, then $t'(cn_0) = C \log c + t(n_0)$.

3.2.1 Random Sampling for $k > 1$

For rank approximate k-nearest-neighbor, if m samples $p_1, p_2, \ldots, p_m$ are made from R with replacement, and assume without loss of generality, that $p_1, \ldots, p_k$ are the k closest point to q among the m samples. Then the probability that at least k of those samples are among the closest τ fraction of points to q is given by:

$$\Pr\left(\frac{|r \in S\colon \mathbf{d}(q, r) \leq \mathbf{d}(q, p_k)|}{n} \leq \tau\right) = \sum_{j=k}^{m} \binom{m}{j} \tau^j (1 - \tau)^{m-j}.$$

Hence, finding the smallest m satisfying the following inequality

$$\sum_{j=k}^{m} \binom{m}{j} \tau^j (1 - \tau)^{m-j} \geq 1 - \delta$$

gives us the appropriate choice for m. Since the above equation does not have a closed form, it can be solved using binary search in the range $[k, n]$. Since we are considering sampling with replacement, it is possible that some points are sampled more than once, resulting in less than k distinct neighbors satisfying the rank approximation.

However, random samples with replacement from a large set are usually distinct. And the procedure to compute the minimum required number of samples can be easily modified to consider sampling without replacement for distinct samples.

3.2.2 Generality of the Random Sampling Technique

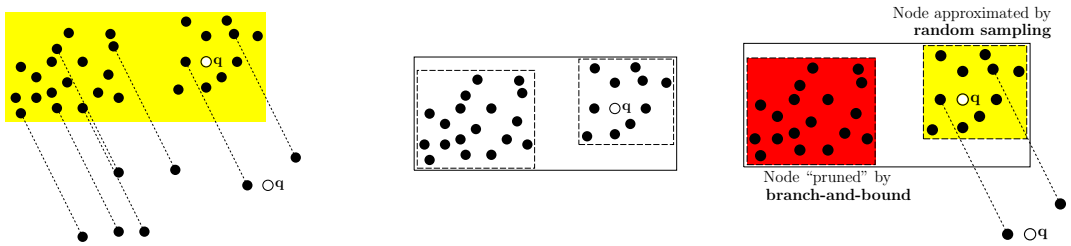
Algorithm 8 can be used for rank approximate nearest-neighbor search even if $\mathbf{d}(\cdot, \cdot)$ is not a metric. In fact there is no restriction for the dataset S to even be in any vector space if there is a way to compute the distance between any pair of points (or equivalently the similarity). The random sampling technique will work for any similarity function for the rank approximation of any maximum similarity search. Moreover, this same technique can be used to obtain the rank approximate version of the farthest-neighbor (or minimum similarity search). However, this generality obviously comes with the cost of the (previously discussed) multiplicative scaling of the runtime of the algorithm.

3.3 Rank Approximate Search via Stratified Sampling

As discussed in the previous section, τ-rank-approximate nearest-neighbor search with simple random sampling (Algorithm 8) does not scale well with increasing number of points n in the database if $\tau = \Omega(1/n)$ (that is, τ decreases with increasing n). The minimum required number of samples, m, grows at least multiplicatively with the factor of increase in the size of the dataset. However, if the random sampling is executed in a manner that allows the search to ignore the samples farther away from the query, the computation can be more efficient than $O(m)$.

A tree data structure can index the dataset. With a tree index on R, random samples from R are effectively indexed. If the random samples themselves are indexed then the random samples closer to the query q can be accessed before random samples from farther parts of the dataset. This indexing provides us with two potential advantages – (i) having obtained random samples closer to q, the farther parts of the

set R for q can be removed from computation with the branch-and-bound technique on the tree if possible (hence ignoring the required samples from the farther parts of the set R and saving computation), and (ii) stratified sampling usually implies lower variance in the quality of the results.



(a) Sampling the whole set (b) An indexing of the set (c) Sampling from the index

Figure 13: Rank approximate search with random sampling.

It is important to note that both these possible advantages depend on how well the tree (or any index) is capable of indexing the set. If the indexing is fairly tight, the branch-and-bound technique should be capable of saving more computation while picking high quality candidates (with low error) from the index. Figure 13 provides an example of how sampling from a tree might save computation: Figure 13(a) demonstrates the sampling by Algorithm 8 over the whole set of points. Figure 13(c) demonstrates that if the set is indexed into two tighter nodes of a tree (as shown in Figure 13(b)), samples can be obtained first from the set of points closer to q, while samples from the farther set of points can be subsequently removed from computation by branch-and-bound.

3.3.1 Random Sampling on a Binary Tree

If the dataset R is indexed in a (binary) tree T, and m random samples are required from the whole set, we require a way to randomly sample points from subsets of the data present in the nodes of T without violating the requirement of the sampling being random from the whole of the dataset. The following fact provides a way to sample

points from just the nodes of the tree and still provide the required randomness:

Fact 3.3.1. *Let m_l and m_r be the number of random samples from the subsets S_l and S_r of the dataset S respectively, where $S = S_l \cup S_r$, $|S_l| = n_l, |S_r| = n_r$ and $|S| = n = n_l + n_r$. Then, making m random samples from S is equivalent to making m_l and m_r random samples from S_l and S_r respectively if $m_l + m_r = m$ and $m_l/m_r = n_l/n_r$.*

This fact provides a way to decide how much to sample from a particular node without violating the τ-rank-approximation (with high probability). This can be used to obtain a lower bound on the number of samples required to approximate the unpruned and unexplored part of the tree. Since the ratio of the sample size to the population size is a constant $\beta = m/n$, Fact 3.3.1 is generalizable to any level of the tree. Using these tools, we present a tree-based algorithm combining random sampling with the usual branch-and-bound technique.

3.3.1.1 The Tree-based Rank Approximate Search Algorithm

The complete rank approximate search algorithm is presented in Algorithm 9. This algorithm is an extension of the tree-based depth-first branch-and-bound exact search algorithm using binary trees. The dataset R is indexed as a binary tree rooted at T_{root}. After obtaining the number of required samples m and the sampling ratio β, the recursive sub-routine **TreeBasedRANN**(q, T, β) computes a τ-rank-approximate nearest-neighbor of the query q. During the search, if a leaf node is reached (since the tree may not be balanced), the best candidate from the leaf is computed with exhaustive search. If a non-leaf node cannot be approximated, the child node closer to the query is always traversed first.

The branch-and-bound part of the algorithm works as follows – if the minimum possible distance of q to any tree node T (min_dist_to_node(q, T)) is less than q's distance to its current best candidate q.NN, the node T is considered for traversal, otherwise the node T is pruned from the computation. The branch-and-bound section

Algorithm 9 Rank Approximate search via Random Sampling on a Binary Tree

Input: Dataset R, query q, rank-approximation τ, failure probability δ
Output: τ-rank-approximate nearest-neighbor of q

$m \leftarrow \left\lceil \frac{\log \delta}{\log(1-\tau)} \right\rceil$, $\beta \leftarrow m/|R|$, $T_{\text{root}} \leftarrow \text{BinaryTree}(R)$
//$T.S$ contains the sets of points contained in the node T
//T.lchild & T.rchild denote the left and right child of T respectively
$q.\text{ub} \leftarrow \infty$, $q.\text{NN} \leftarrow \emptyset$.
//$q.$NN is the current best neighbor candidate for the query q & $q.\text{ub} = \mathbf{d}(q, q.\text{NN})$
TreeBasedRANN(q, T_{root}, β) *//The depth-first tree-search sub-routine*
return $q.$NN.

TreeBasedRANN(query q, Tree Node T, Sampling Ratio β)

```
if q.ub > min_dist_to_node(q, T) then
  // Cannot be pruned by branch-and-bound
  if is_a_leaf(T) then
    q.NN ← arg min_{r∈T.S∪{q.NN}} d(q, r); q.ub ← d(q, q.NN)
  else if ⌈β · |T.S|⌉ ≤ MaxSamples then
    // Can approximate by sampling
    S' ← ⌈β · |T.S|⌉ random samples from the tree node T
    q.NN ← arg min_{r∈S'∪{q.NN}} d(q, r); q.ub ← d(q, q.NN)
  else
    // Otherwise recurse in a depth-first manner
    if min_dist_to_node(q, T.lchild) ≤ min_dist_to_node(q, T.rchild) then
      TreeBasedRANN(q, T.lchild, β), TreeBasedRANN(q, T.rchild, β)
    else
      TreeBasedRANN(q, T.rchild, β), TreeBasedRANN(q, T.lchild, β)
    end if
  end if
end if
```

of the algorithm does not introduce any approximation to the search. The intended approximation occurs in the unpruned portion of the tree. The traversal down the tree is stopped at any node T which can be summarized within desired error bounds with a small number of samples (below a certain threshold *MaxSamples*). Required number of points, $(\lceil \beta \cdot |T.S| \rceil)$, are sampled from the set of the points contained in that node $(T.S)$, and the best candidate is selected via exhaustive search over these samples. The following theorem proves the correctness of the Algorithm 9:

Theorem 3.3.1. *For a query q and a dataset R, and specified values of δ and τ,*

Algorithm 9 computes a τ-rank-approximate nearest-neighbor of q in R with probability at least $(1-\delta)$.

Proof. A query requires at least $m = \left\lceil \frac{\log \delta}{\log(1-\tau)} \right\rceil$ random samples from the set R to have a τ-rank-approximate nearest-neighbor with probability at least $(1-\delta)$. Let $\beta = (m/|S|)$. Let $T.S$ denote the set of points contained in a node T. In Algorithm 9, sampling occurs when a base case of the recursion is reached. There are three base cases:

- Case 1 - Exact Pruning (**if** q.ub $\leq$ min_dist_to_node(q, T)) (Line 1): Then number of points required to be sampled from the node is at least $\lceil \beta \cdot |T.S| \rceil$. However, since this node is pruned, we ignore these points and nothing needs to be done in the algorithm.

- Case 2 - Exact Computation (**if** q.ub $>$ min_dist_to_node(q, T) **and** is_a_leaf(T)) (Line 3): In this subroutine, the best candidate is obtained by exhaustive search. Hence number of points actually sampled is $|T.S| \geq \lceil \beta \cdot |T.S| \rceil$.

- Case 3 - Approximate Computation (**if** q.ub $>$ min_dist_to_node(q, T) **and** $\lceil \beta \cdot |T.S| \rceil \leq$ *MaxSamples*) (Line 6): In this subroutine, exactly $\beta \cdot |R|$ samples are considered.

Let the total number of points effectively sampled from S be m'. The recursion implies that every subset of S is encountered in any one of the aforementioned three base cases of the algorithm. Hence, $m' \geq \lceil \beta \cdot n \rceil = m$, giving us the statement of the theorem. □

Algorithm 9 is not as general as Algorithm 8 and relies on the triangle-inequality of the distance metric $\mathbf{d}(\cdot, \cdot)$ to employ the branch-and-bound technique. Hence the runtime of Algorithm 9 will have some implicit dependence of the data dimensionality.

3.3.2 Random Sampling on a Cover Tree

The cover tree data structure (Beygelzimer et al., 2006) has been shown to have rigorous runtime bounds for a single NN query. The cover tree based rank-approximate search algorithm is presented in its entirety in Algorithm 10. The main goal of this is to leverage the existing cover tree analysis to obtain a runtime bound for rank-approximate search (Algorithm 10).

Analysis. The runtime analysis of nearest-neighbor search with cover trees makes use of a data-dependent quantity known as the expansion constant. While our analysis here mostly leverages the original cover tree analysis and does not explicitly make use of the definition of the expansion constant, we provide the precise definition here for completeness. We will discuss the expansion constant in more detail in Chapter 5. It is defined as following:

Definition 3.3.1. *For $p \in R$, let $\mathcal{B}(p, \Delta) = \{r \in R : \mathbf{d}(p, r) < \Delta\}$. Then the expansion constant of $(R, \mathbf{d})$ is defined as the smallest $c \geq 2$ such that $|\mathcal{B}(p, 2\Delta)| \leq c|\mathcal{B}(p, \Delta)|$ $\forall p \in R$ and $\forall \Delta > 0$.*

From Beygelzimer et al. (2006, Lemma 4.3), the maximum depth of any point in a cover tree is $O(c^2 \log n) \leq K c^2 \log n$ for some K. Now there are two ways in which subtrees are removed at each level. It is pruned either because the best possible candidate in the subtree is farther than the current best candidate for the query (subtree "pruned" by **branch-and-bound**) or because the subtree contains small enough number of points to be approximated by a single sample (subtree approximated by **random sampling**).

Consider an explicit node r whose subtree is not approximated by random sampling. Then $\beta|L(r)| > 1$. Let d be the minimum depth of any such subtree by itself. The minimum depth would be achieved when the branching factor is tight at each

level. By Beygelzimer et al. (2006, Lemma 4.1), this implies that for the smallest possible depth d, $(c^4)^d > (1/\beta)$, giving us

$$d > \frac{\log(1/\beta)}{\log c^4}.$$

This implies that the maximum depth of any explicit node visited by the algorithm is given by $K' = Kc^2 \log n - (\log(1/\beta)/\log c^4)$.

Algorithm 10 Rank Approximate search via Random Sampling on a Cover Tree

Input: Dataset R, query q, rank-approximation τ, failure probability δ
Output: A τ-rank-approximate nearest-neighbor of q

$m \leftarrow \left\lceil \frac{\log \delta}{\log(1-\tau)} \right\rceil$, $\beta \leftarrow m/|R|$, $T \leftarrow \text{CoverTree}(R)$
// C_∞ *corresponds to the tree root*; C_i *contains all nodes in the tree at level* i
// $L(r)$ *corresponds to the set of points in the subtree rooted at point* r
$q.\text{ub} \leftarrow \infty$, $q.\text{NN} \leftarrow \emptyset$.
//$q.\text{NN}$ *is the current best neighbor candidate for the query* q & $q.\text{ub} = \mathbf{d}(q, q.\text{NN})$
TreeBasedRANN(q, T_{root}, β)
return $q.\text{NN}$.

TreeBasedRANN(query q, Cover Tree T, Sampling Ratio β)

Initialize $S_\infty \leftarrow C_\infty$.
for $i = \infty$ **to** $-\infty$ **do**
  $S \leftarrow \{\text{Children}(r) : r \in S_i\}$ //*the set of children of* S_i.
  $q.\text{NN} \leftarrow \arg\min_{r \in S \cup \{q.\text{NN}\}} \mathbf{d}(q, r)$; $q.\text{ub} \leftarrow \mathbf{d}(q, q.\text{NN})$
  $S' \leftarrow \{r \in S : d(q, r) \leq q.\text{ub} + 2^i\}$
  // *all children of* R_i *whose subtree might contain the* q*'s* NN
  $S_{i-1} \leftarrow \{r \in S' : \beta|L(r)| > 1\}$
  //*the set of children of* S_i *which cannot be approximated by sampling at level* i
  $S'' \leftarrow \{$ some $r' \in L(r) \forall r \in S' \setminus S_{i-1}\}$
  //*sample from the subtrees rooted at nodes in* S' *which can be approximated*
  $q.\text{NN} \leftarrow \arg\min_{r \in S'' \cup \{q.\text{NN}\}} \mathbf{d}(q, r)$; $q.\text{ub} \leftarrow \mathbf{d}(q, q.\text{NN})$
end for
$q.\text{NN} \leftarrow \arg\min_{r \in S_{-\infty} \cup \{q.\text{NN}\}} \mathbf{d}(q, r)$

Following the original cover tree analysis, let S^* be the last explicit (completely unpruned) S_i considered by the algorithm. Then the maximum number of iterations is bounded by $K'|S^*| \leq K' \cdot \max_i |S_i|$. Total number of children encountered throughout the whole algorithm bounded by $K' \cdot c^4 \cdot \max_i |S_i|$. The total time required throughout

the algorithm to decide which nodes require explicit descend (nodes that cannot be pruned by branch-and-bound) is given by $K' \cdot \max_i |S_i|^2$.

The amount of work done to approximate small subtrees with samples never exceeds the work done in the previous decision phase, which in turn, is bounded by the product of the number of iterations $K' \cdot \max_i |S_i|^2$. Hence the total time taken by the algorithm is bounded by

$$2K' \max_i |S_i|^2 + K' \max_i |R_i| c^4.$$

Now $\max_i |S_i|$ is upper bounded by c^5 as in the original algorithm because the pruning based on distances is not changed, and the sampling only makes S_i smaller. Hence the final runtime is bounded by $3c^{10}K'$. This gives us the following result:

Theorem 3.3.2. *Given a cover tree T on a dataset R and query q, let the expansion constant of $(\{q\} \cup R)$ be c. Then Algorithm 10 returns a τ-rank-approximate neighbor of q with probability at least $1 - \delta$ in $O\left(c^{10}\left(c^2 - \frac{1}{\log c^4}\right)\log n + \frac{c^{10}}{\log c^4}\frac{\log 1/\delta}{\log 1/(1-\tau)}\right)$ time.*

This analysis is not very tight since it implies that it has effectively the same worse-case runtime guarantee as the exact search runtime guarantee of $O(c^{12} \log n)$ with cover trees (Beygelzimer et al., 2006). However, the implications are meaningful; the result implies that the algorithm is logarithmic in n, however, the approximation allows for a slight reduction in the dependence on the expansion constant c in the first term, while implying more work for smaller approximations in the second term of the bound.

3.4 Empirical Evaluation of Rank Approximate Search

We evaluate the performance of the proposed Algorithms 8 (RS) and 9 (TSS) on various metrics in this section. For the tree-based rank-approximate search, we use the *kd*-trees (Friedman et al., 1977) to demonstrate the performance of TSS with the simplest form of trees. The axis-aligned splits of the *kd*-tree are performed along the

mid-point of the dimension with the largest variation. The *kd*-tree is built recursively and the maximum leaf size is 20. The value of *MaxSamples* in TSS (Algorithm 9) is also chosen to be 20. While the best values for both the leaf-size and *MaxSamples* can be chosen by cross-validation, we just considered a single value for our experiments. The rank approximate search algorithms are implemented in MLPACK (Curtin et al., 2012)[1].

We begin by demonstrating their efficiency over tree-based exact search methods for different levels of rank-approximation and their scaling with respect to the size of the dataset. Following that, we compare our proposed algorithms to Locality-sensitive hashing (LSH) with p-stable distributions (with $p = 2$) (Datar et al., 2004) in terms of their error/search-time tradeoff. We consider both forms of nearest-neighbor error, distance error and rank error (we will provide the exact definition of the errors in the relevant subsection).

Table 3: Details of the datasets.

Dataset	**Abbrv.**	**# points**	**# queries**	**# dimensions**
Synthetic 10-dim ball	*unitSp10*	100000	100000	10
Synthetic 100-dim ball	*unitSp100*	100000	100000	100
Synthetic 1000-dim ball	*unitSp1k*	100000	100000	1000
Covertype	*covtype54*	380000	200000	54
Protein Homology	*bio74*	145000	139000	74
Quantum Physics	*phy78*	50000	100000	78
MNIST	*mnist784*	60000	10000	784
Optdigits	*optdig64*	3823	1797	64
Pendigits	*pendig16*	7494	3498	16
Corel	*corel32*	27750	10000	32
Layout Histograms	*lh30*	7500	2500	30
Images	*im4096*	523	175	4096
80 million tiny images	*tinyIm384*	100000	100000	384
Live Journal blog	*liveJ25k*	10000	1000	25327

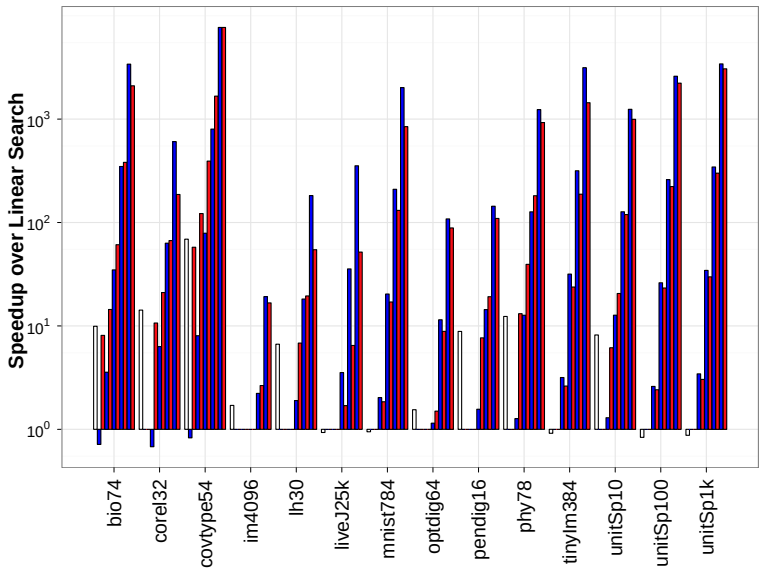
[1]The code is available at the following repository: http://svn.cc.gatech.edu/fastlab/mlpack/trunk. The source code for the rank-approximate search methods (RS & TSS) is in the following sub-directory: http://svn.cc.gatech.edu/fastlab/mlpack/trunk/src/mlpack/methods/rann.

Datasets. We use a variety of datasets. We use 3 synthetic datasets (generated by uniformly sampling from a unit hypersphere in 10, 100 and 1000 dimensions) and the following real datasets: (i) Science datasets – we use the Covertype set (Blake and Merz, 1998) and the Protein homology and the Quantum physics sets from the KDD cup 2004, (ii) Handwritten digits datasets – we use the MNIST (LeCun, 2000) and the Optdigits and Pendigits sets (Blake and Merz, 1998), (iii) Image datasets – we use the Corel, Layout Histogram, Images (Tenenbaum et al., 2000) sets and a subset of the 80 million tiny images (Torralba et al., 2008), (iv) Text dataset – we consider the LiveJournal blog dataset (Kim et al., 2011). The sizes of the datasets are provided in Table 3.

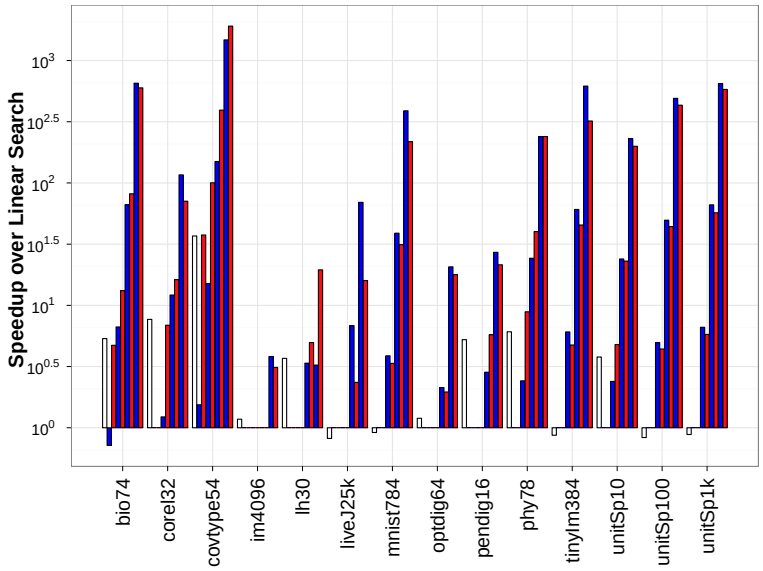
The datasets are chosen based on the applications of nearest-neighbor search. Nearest-neighbor search is heavily used in machine learning, hence justifying the choice of multitude of machine learning datasets from the UCI Machine Learning Repository (Blake and Merz, 1998). Image search is also a very common application of nearest-neighbor search. Hence I chose these commonly used image datasets in search literature (such as Corel and Layout Histogram) as well as datasets in computer vision literature (such as Tiny Images). Similarity search is also widely used in document search and scientific computing, hence we chose some widely used sets from data mining literature (such as the Protein Homology and Quantum Physics datasets from KDD cup 2004).

3.4.1 Efficiency and Scaling

We begin by evaluating the efficiency and scaling of the rank-approximate search algorithms RS and TSS. We compare the efficiency of the approximate algorithms to the tree-based exact search algorithm. We consider the speedup of the algorithm in question over the exhaustive search algorithm as the measure of efficiency. For the scaling experiments, we report the actual search times of the algorithms.



(a) $k = 1$, $\tau = 0.001\%, 0.01\%, 0.1\%, 1\%\&10\%$



(b) $k = 10$, $\tau = 0.01\%, 0.1\%, 1\%\&10\%$

Figure 14: Efficiency over exact search. The speedups of 3 algorithms for k-nearest-neighbor search over exhaustive search are presented here. For each dataset, the leading white bar represents the speedup of tree-based exact nearest-neighbor search, the blue bars indicate the speedup of RS and the red bars indicate the speedup of TSS for different levels of rank-approximation with $\delta = 0.05$. The missing bars for some datasets indicate that the approximation is too low ($\tau n \leq k$) (*please view in color*).

3.4.1.1 Efficiency over exact search

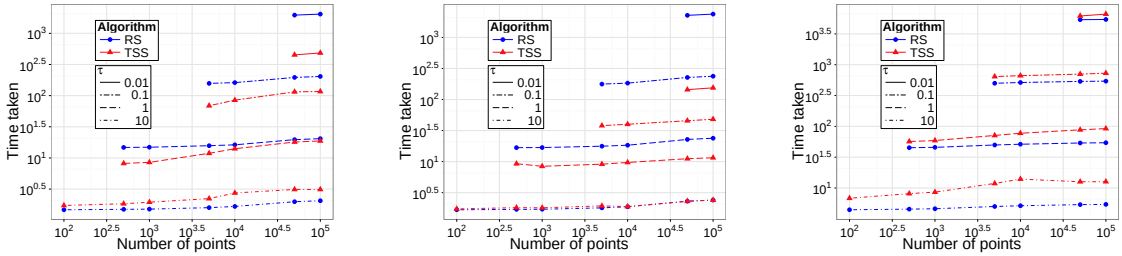
We present the speedups of the exact tree-based algorithm and our proposed rank approximate algorithms for k-nearest-neighbor search over the exhaustive search algorithm in Figure 14 with varying values of τ and $\delta = 0.05$. RS (blue bars) and TSS (red bars) exhibit increasing speedups with increasing levels of approximation. For appropriately high levels of approximation ($\tau \geq 1\%$), both TSS and RS outperform the tree-based exact algorithm. The exact search algorithm performs slightly better than the approximate algorithms for very low levels of approximation in some cases (for example, for $k = 1$ *bio74, corel32, covtype54, optdig64, unitSp10*).

For the datasets where the exact tree-based search algorithm does not exhibit any speedup over exhaustive search, the approximate algorithms perform better than the exact algorithms as expected, with RS performing better than TSS for all levels of approximation. For the datasets where the exact algorithm shows some improvement over exhaustive search (*bio74, corel32, covtype54, lh30, pendig16, phy78* & *unitSp10*), TSS outperforms RS for low levels of approximation ($\tau < 1\%$). For higher levels of approximation in almost all datasets, RS slightly outperforms TSS.

3.4.1.2 Scaling

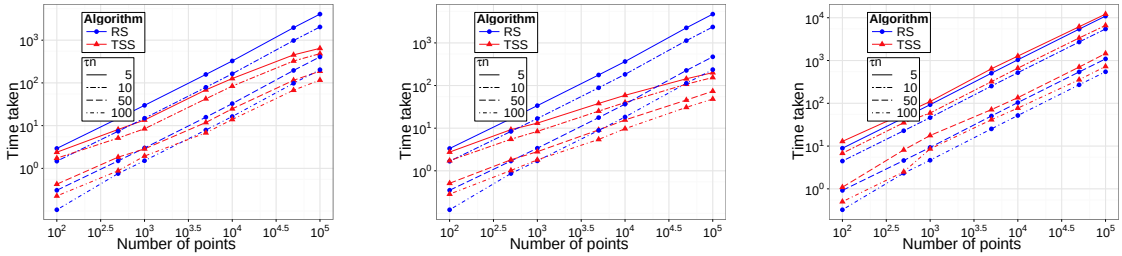
The scaling of the rank approximate algorithms (RS and TSS) with respect to the number of instances in the set is presented for 3 of the larger real datasets in Figures 15 & 16 The scaling is considered under two conditions – (i) fixed level of rank approximation τ with $\tau = 0.01\%, 0.1\%, 1\%\&10\%$ and $\delta = 0.05$ in all cases (only the cases where $\tau n > 1$ are reported), and (ii) fixed level of τn (hence finding a neighbor among the $\lceil \tau n \rceil$-nearest-neighbors) with $\lceil \tau n \rceil = 5, 10, 50, 100$ and $\delta = 0.05$ in all cases.

For a fixed value of τ (Figure 15), the runtimes of RS and TSS appear to be independent of the size of the dataset. While the scaling of RS and TSS appear to



(a) *bio74* (b) *covtype54* (c) *tinyIm384*

Figure 15: Scaling with fixed τ. Time taken is measured in seconds and values chosen for τ are $0.01\%, 0.1\%, 1\%$ & 10% (*please view in color*).



(a) *bio74* (b) *covtype54* (c) *tinyIm384*

Figure 16: Scaling with fixed τn. Time taken is measured in seconds and values chosen for $\lceil \tau n \rceil$ are $5, 10, 50$ & 100 (*please view in color*).

be similar for a fixed τ, the actual values of the runtimes are different for RS and TSS. TSS has significantly lower runtimes than RS for lower values of τ in the *bio74* and *covtype54* datasets, while RS performs slightly better than TSS for the *tinyIm384* set for low values of τ. For larger values of fixed τ, both TSS and RS have similar runtimes.

The scaling behavior of RS and TSS varies in the case where τn is fixed (Figure 16). The runtimes of RS increases linearly with the number of points in all datasets for all values of τn. Under the same conditions, TSS exhibits a sub-linear dependence on the number of points for the *bio74* and *covtype54* datasets, and a linear dependence on the number of points for the *tinyIm384* set. The improvement of TSS over RS (where there is any improvement) appears to decrease with increasing values of τn.

3.4.1.3 Discussion

In Figure 14, TSS performs better than RS on datasets for low levels of approximations ($\tau < 1\%$) where the exact branch-and-bound algorithm is already significantly more efficient than the exhaustive search. This is expected because TSS saves computation by pruning away parts of the dataset far away from the query (by branch-and-bound). However, for higher levels of approximation, the performance of TSS and RS become more comparable because the number of samples required for guaranteeing the approximation is very low and the computational overhead of the tree-traversal in TSS outweighs any gains obtained by reducing the number of samples to be considered. For datasets where the exact algorithm records very little speedups over the exhaustive search, TSS records lower speedups than RS for every level of approximation. This is expected because all these datasets have moderate to high dimensionality and the *kd*-tree is unable to obtain significant computational savings from the branch-and-bound technique and the tree-traversal overhead outweighs any gains obtained by branch-and-bound.

This effect is also exhibited in the scaling of RS and TSS. As presented in the Theorem 3.2.1, for a fixed τ, the runtimes of RS are seen to be independent of the number of points n in the set. For the *bio74* and *covtype54* sets, the runtimes of TSS are better than that of RS because the tree-based branch-and-bound algorithm is capable of obtaining significant computational gains. However, the difference between their runtimes reduce as the level of approximation τ increases because the required number of samples is very low. For the *tinyIm384* set, the runtimes of RS is better than that of TSS since the tree-traversal overhead outweighs any computational gain obtained by the branch-and-bound. As discussed in Corollary 3.2.1, for a fixed $k = \lceil \tau n \rceil$, the runtimes of RS are seen to increase linearly with n for all datasets, while TSS appears to have a sub-linear dependence on n for the *bio74* and the *covtype54* datasets since the branch-and-bound provides significant computational savings. However, for

the *tinyIm384* set, TSS appears to have a linear dependence on n similar to RS because the branch-and-bound is unable to provide any gain.

As discussed earlier, the performance of TSS appears to depend heavily on how successful the tree is in indexing the dataset and in allowing the branch-and-bound algorithm to obtain significant computational gains. A indexing scheme more sophisticated than *kd*-trees can possibly present much better performance of TSS. However, for high levels of rank approximation, RS is simpler and more efficient.

3.4.2 Error/Search-time Tradeoff for Approximate Search

Now we compare the error/search-time tradeoff of the two proposed algorithm RS and TSS to locality sensitive hashing (LSH) (Datar et al., 2004). The preprocessing of LSH requires three parameters for the first-level hash and two additional parameters for the second-level hash. The parameters for the first-level hash and our choices for them are as follows: (i) the number of projections in a hash table (we choose 10 projections per hash table), (ii) the number of hash tables (we use 5, 10, 15, 20, 25 and 30 hash tables[2] for the first-level hash to study the error/search-time tradeoff of LSH), (iii) the bin-width (this parameter should depend on the pairwise distances of the dataset, so we use the average pairwise distance in the dataset from 25 random pairs as the bin-width). The parameters of the second-level hash are the hash size (we choose 99901) and the bucket size in the hash (we choose 300). The 2-stable LSH is also implemented in MLPACK (Curtin et al., 2012)[3].

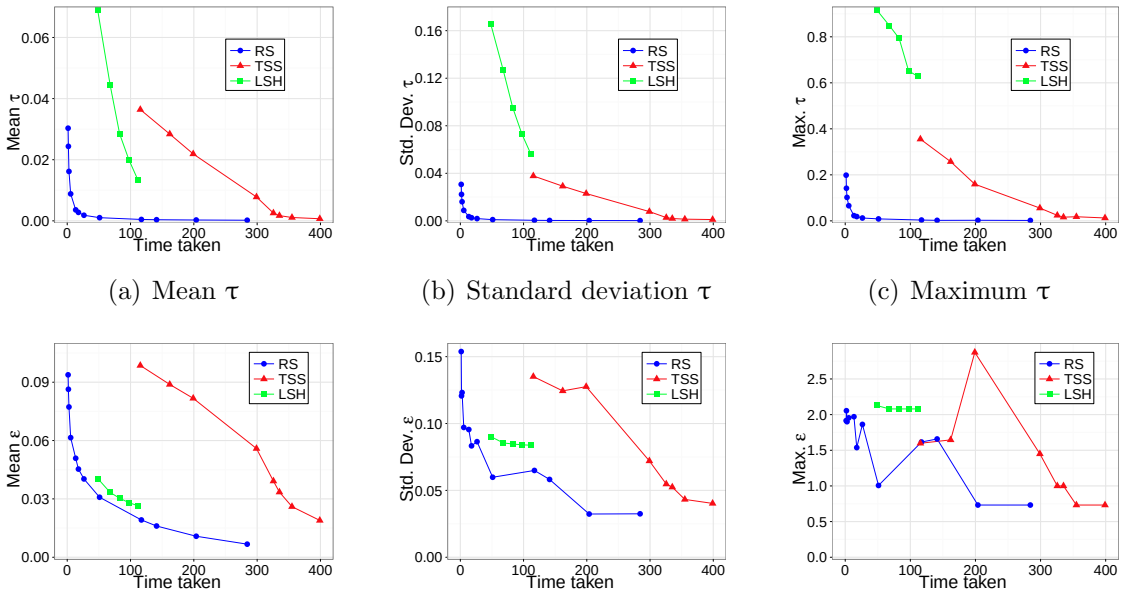
We present the error/search-time tradeoff of the three algorithms for two different forms of nearest-neighbor error, rank error τ and distance error ε , defined as follows: for the approximate nearest-neighbor $p' \in R$ for the query q,

$$\bullet\ \tau = \frac{|\{p \in R:\ \mathbf{d}(q,p) < \mathbf{d}(q,p')\}|}{|R|}, \qquad \bullet\ \varepsilon = \frac{\mathbf{d}(q,p') - \min_{p \in R} \mathbf{d}(q,p)}{\min_{p \in R} \mathbf{d}(q,p)}.$$

[2]We did not choose more than 30 hash tables because of memory constraints.

[3]The code is available at the following repository: http://svn.cc.gatech.edu/fastlab/mlpack/trunk. The source code for search with locality-sensitive hashing (LSH) is in the following sub-directory: http://svn.cc.gatech.edu/fastlab/mlpack/trunk/src/mlpack/methods/lsh.

For each form of error, we present the mean, standard deviation and the maximum of the error over all queries. We remove the instances where LSH does not return a single neighbor candidate for any one of the queries since it is not clear what the rank or distance error should be. We consider four sets – three sets, *liveJ25k*, *mnist784* and *tinyIm384*, in which the exact tree-based algorithm was unable to obtain any speedup over the exhaustive search in Figure 14, and one set, *phy78*, where the exact tree-based algorithm did get more than an order of magnitude speedup over the exhaustive search. The results are presented in Figures 17, 18, 19 & 20. Of these sets, *tinyIm384* and *phy78* do contain a lot of near duplicates and can qualify as instances of the "planted nearest-neighbor" model. Each plot contains the error/search-time tradeoff of RS (blue), TSS (red) and LSH (green).



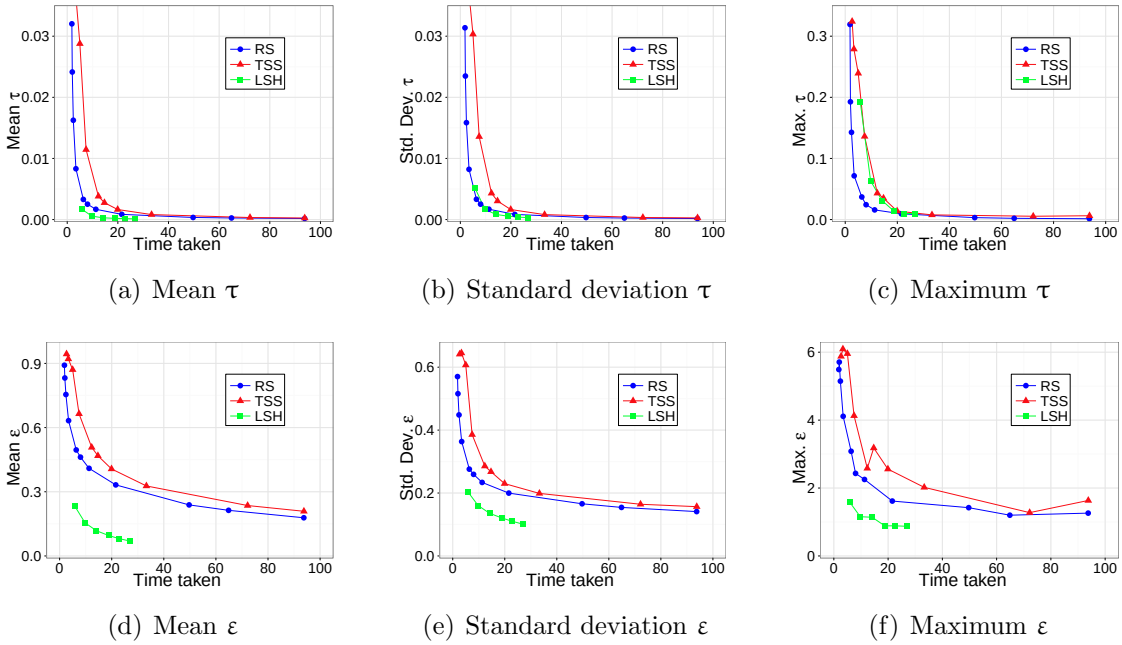
(a) Mean τ (b) Standard deviation τ (c) Maximum τ

(d) Mean ε (e) Standard deviation ε (f) Maximum ε

Figure 17: Error/search-time tradeoff for the *liveJ25k* set. Time taken is measured in seconds (*best viewed in color*).

For the *liveJ25k* dataset (Figure 17), RS performs the best in terms of τ. LSH has a better mean rank error tradeoff than TSS. However, in terms of standard deviation and maximum rank error, TSS performs significantly better than LSH,

while still performing worse than RS. In terms of the distance error ε, RS and LSH have comparable performance, and TSS performs the worst for each case.



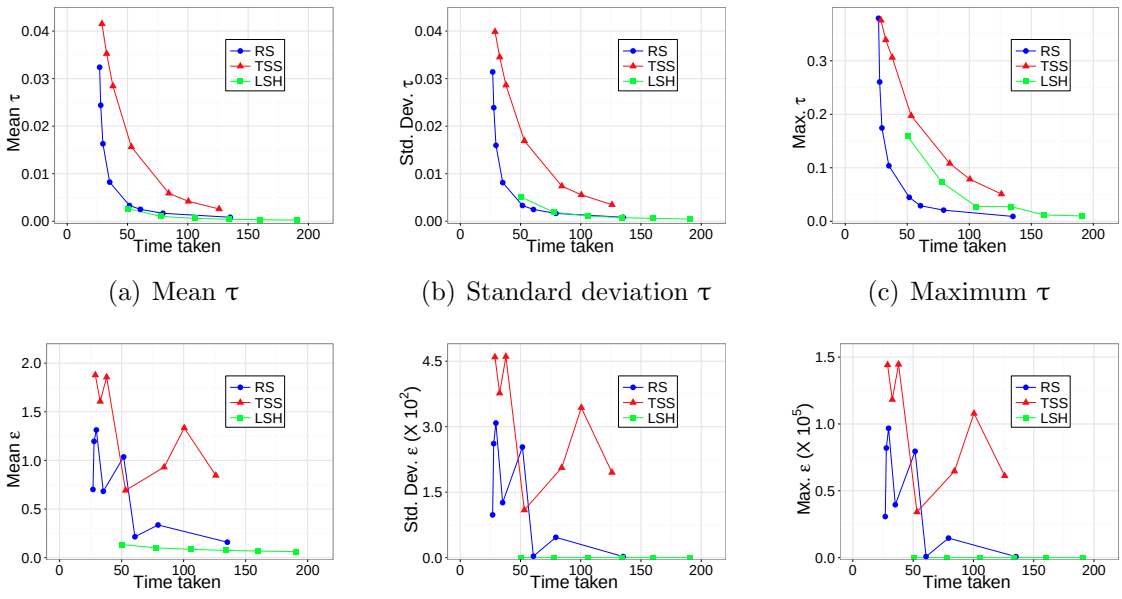
(a) Mean τ (b) Standard deviation τ (c) Maximum τ

(d) Mean ε (e) Standard deviation ε (f) Maximum ε

Figure 18: Error/search-time tradeoff for the *mnist784* set. Time taken is measured in seconds (*best viewed in color*).

In the *mnist784* dataset (Figure 18), RS and LSH appear to have similar behavior in terms of mean, standard deviation and maximum rank error with RS exhibiting better error/time tradeoff for the maximum rank error. However, in terms of distance error, LSH performs significantly better than RS. RS outperforms TSS significantly in all cases.

Similar to the *mnist784* set, RS and TSS have similar rank error time tradeoff for the *tinyIm384* set (Figure 19), with RS outperforming LSH in terms of the maximum rank error. RS performs better than TSS for all error statistics. However, in terms of the distance error, LSH outperforms RS by a significant margin. However notice that the distance error time tradeoff of the mean, standard deviation and maximum ε for RS and TSS have the same shape. This indicates that the mean and the standard deviation of the ε is skewed because a single query had really high distance error.



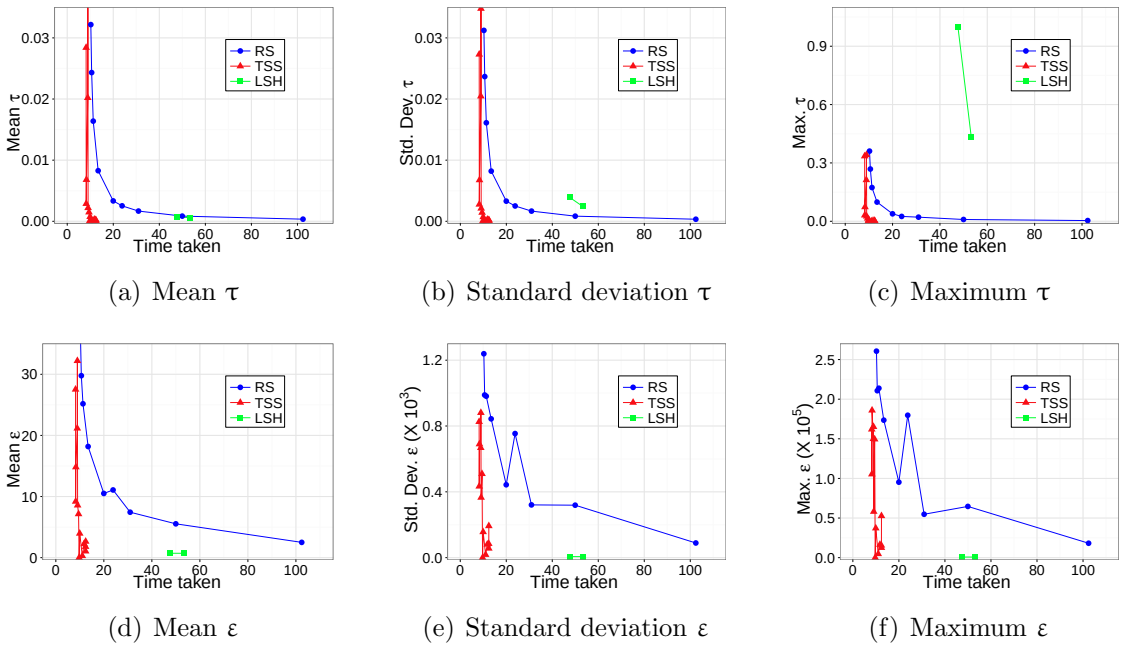
(a) Mean τ (b) Standard deviation τ (c) Maximum τ

(d) Mean ε (e) Standard deviation ε (f) Maximum ε

Figure 19: Error/search-time tradeoff for the *tinyIm384* set. Time taken is measured in seconds (*best viewed in color*).

This is an artifact of the random sampling used in the algorithms RS and TSS and will be further discussed in the discussion section (a similar situation occurs for LSH where the search returns no neighbor candidates). However, the maximum rank error of RS and TSS indicates that the neighbor candidate with really high distance error does not have a high rank error. This is possible when the nearest-neighbor of a query is a near-duplicate of the query, resulting in a really small nearest-neighbor distance.

As expected, TSS significantly outperforms RS and LSH in the *phy78* set (Figure 20) for both kinds of errors (distance and rank) and all error statistics. For the cases where LSH did return a neighbor candidate for every query, LSH demonstrated comparable mean rank error time tradeoff to RS, while performing significantly worse than RS (and TSS) for the standard deviation and maximum of the rank error (over all queries). However, LSH shows better performance than RS in terms of the distance error ε. Because of the presence of near-duplicates, the maximum ε of RS is really high, skewing the mean and standard deviation ε.



(a) Mean τ (b) Standard deviation τ (c) Maximum τ

(d) Mean ε (e) Standard deviation ε (f) Maximum ε

Figure 20: Error/search-time tradeoff for the *phy78* set. Time taken is measured in seconds (*best viewed in color*).

3.4.2.1 Discussion

From the results in Section 3.4.1, we expected TSS to outperform RS in low-dimensional and easier datasets (datasets where the exact tree-based search algorithm achieves significant speedups over exhaustive search). This is exhibited in the error/time tradeoff of the *phy78* set (Figure 20). In this set, LSH outperforms RS. However, for low-dimensional/easy datasets, there is generally no need for approximation because the tree-based exact search algorithm is already efficient.

For the higher dimensional datasets, RS performs better than TSS because the tree traversal costs outweigh the computational gains obtained by the branch-and-bound algorithm. This is because the tree is unable to obtain tight enough bounds in high dimensions. This is exhibited in the error/time tradeoff of the *mnist784* (Figure 18), the *liveJ25k* (Figure 17) and the *tinyIm384* (Figure 19) sets. In these sets, LSH does not appear to perform significantly better than the simple random sampling

algorithm RS in terms of the rank error. In fact, RS outperforms LSH in terms of the rank error τ. However, in terms of the distance error ε, LSH outperforms RS in the *mnist784* and *tinyIm384* sets. This discrepancy of the relative performance in terms of the two different types of errors τ and ε can be attributed to the "planted nearest-neighbor model". If the nearest-neighbor distance is in fact really small even for some of the queries (as is the case when the query's nearest-neighbor is its (near) duplicate), not locating that neighbor will result in a really high distance error even though the rank error can still be fairly low. This occurrence is exhibited with the *tinyIm384* and the *phy78* sets.

LSH is expected to perform especially well in an extreme version of the planted nearest-neighbor model where the nearest-neighbor(s) of the queries are their respective (near) duplicate(s). Hashing techniques will put the duplicates in the same buckets. In contrast, the random-sampling based searches in Algorithms 8 & 9 can easily miss the (near) duplicates. In such a setting LSH will significantly outperform RS and TSS especially in terms of the distance error. While we do not have a way of extending Algorithm 8 (RS) to work well in this setting, we present a simple way of modifying Algorithm 9 (TSS) to ensure the location of duplicates – only approximate by sampling once the tree-traversal has considered all the points in the leaf node of the tree the query lands in. The duplicates (and possibly near-duplicates) of the query will lie in the leaf node of the tree in which the query lands in. Hence the above modification will ensure that any duplicates (if they exist) will definitely be located.

We would like to note that LSH requires multiple hash tables to obtain any desired level of accuracy, resulting in high preprocessing time and memory/space requirement. In contrast, the tree-based TSS requires a single tree construction and the same tree can be used for all levels of approximation. This implies low preprocessing time and space requirements. The simple RS algorithm has no preprocessing time or space requirement. The search times reported in this section do not include the

preprocessing times of any of the methods.

We also have possible avenues for improving the performance of the tree-based rank-approximate search – (1) While we process multiple queries sequentially in TSS, *dual-tree algorithms* (Gray and Moore, 2000) can be used to improve search-times of TSS for the same amount of error by amortizing the search-cost over multiple queries (a dual-tree rank-approximate search algorithm was presented in Ram et al. (2009b, Figure 2)). On the other hand, LSH does not have any equivalent way of improving query processing for multiple queries. (2) We use the simple mid-point-split *kd*-trees for our experiments. *kd*-trees are traditionally believed to not perform well in high dimensions. More sophisticated trees like ball-trees (Omohundro, 1989), metric-trees (Preparata and Shamos, 1985), random-projection-trees (Dasgupta and Freund, 2008), max-margin-trees (Ram et al., 2012) or cover-trees (Beygelzimer et al., 2006) are reputed to have better performance in higher dimensions and can improve the performance of TSS at the cost of increased preprocessing time.

Caveat of this analysis. To end this section, we should note that this analysis is not a fair analysis because LSH is designed for distance approximate search while RS and TSS are designed for rank approximate search. The analysis here can at best be an apples to oranges comparison.

CHAPTER IV

TIME-CONSTRAINED SEARCH

As seen in the previous chapter, approximate search requires the user to specify the desired approximation level. While rank approximation might make this choice more robust and intuitive, the choice is still not straightforward. Moreover, search is not approximated because we have a desire for answers with some amount of error; it is the lack of time that requires us to approximate search. Hence, it would be useful to have algorithms where you can specify the allowed search time instead of the allowed search error. This would make the choice much easier and usable for the user.

In this chapter, we discuss nearest-neighbor search on a time budget and present new algorithms for this time-constrained search. We begin by explicitly defining time-constrained search in Section 4.1. Then we present a very simplistic time-constrained search algorithm in Section 4.2 that utilizes no data structures and relies solely on a permutation of the data. Then we consider binary space-partitioning tree data structures (BSP-trees) and propose three time-constrained search algorithms in Section 4.3 that utilize these binary trees. This chapter concludes with the relative empirical comparison of the proposed algorithms in Section 4.4.

4.1 Time-constrained Search

In this new setting for the problem of nearest-neighbor search, the true nearest-neighbor would always be preferred but not necessarily required. The actual restriction is the computation time available for search. Given enough time, the algorithm should always return the true nearest-neighbor. But in general, the algorithm is required to return an answer within a fixed amount of time. To formalize, the nearest-neighbor methods would be required to perform the following task:

Time-constrained Search: minimize nearest-neighbor error subject to a limited number of computations.

Suppose that we have a nearest-neighbor search algorithm that can return an answer for any given time constraint $t > 0$. And suppose that the nearest-neighbor error of this answer is $e(t)$. *In this work, we care about minimizing the quantity* $\int_t e(t)dt$. Empirically, if we have the error v. time plot of a method (for example, Figure 21), then we wish to effectively minimize the area under this curve. This would ensure that for any time constraint, the algorithm is performing the best possible. Theoretically, we wish to obtain upper bounds on the nearest-neighbor error for a given time constraint. Intuitively, we would obtain bounds of the form $e(t) \approx O(f(1/t))$ where f is a monotonic non-decreasing function.

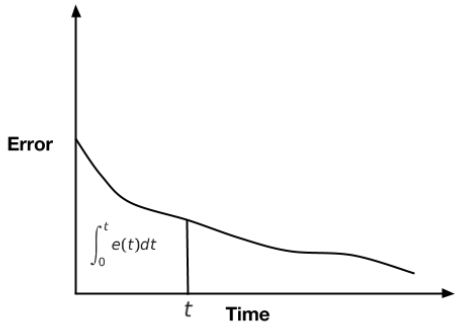


Figure 21: Error v. time plot.

Error combination over queries. Usually the search is performed for various queries on the given dataset. Hence, we wish to minimize the nearest-neighbor error over various queries. The question then is how we combine the errors over all queries. *In this work, we consider the average error and the maximum error over all queries.* One pertains to the average case performance of the method and the other presents the worst case performance of the method.

Notion of error. As discussed in Chapters 2 & 3, there are two possible notions of nearest-neighbor error, rank error and distance error, depending on the application at hand. *In this work, we will present empirical results for both notions of error.* The theoretical guarantees might be with regards to distance or rank error (or both) depending on what we have been able to achieve to date. For clarity, we will redefine the notions of error we will use from here on. For a given query q and a returned neighbor candidate p from the reference set S, the nearest-neighbor **distance error** $\varepsilon(q)$ is defined as

$$\varepsilon(q) = \frac{\mathbf{d}(q, p) - \min_{r \in S} \mathbf{d}(q, r)}{\min_{r \in S} \mathbf{d}(q, r)}, \tag{12}$$

and the nearest-neighbor **rank** $\tau(q)$ is defined as

$$\tau(q) = \text{card}(\{r \in S : \mathbf{d}(q, r) < \mathbf{d}(q, p)\}) + 1. \tag{13}$$

Unit of time. While performing time-constrained search, it is essential to specify the unit of time that we are constraining. There are various quantities that can used as the unit of time, such as, the number of FLOPS or the user time (in seconds) or something more high level like the total number of vector operations (a vector addition and a vector inner-product). *In this work, we will consider the number of vector operations as the unit of time and the time constraint will be in terms of the allowed number of vector operations.* Note that the Euclidean (or ℓ_p) distance computation between two vectors will be considered as a single vector operation. We are assuming that the evaluation of a particular distance function is the same for all pairs of points. Effectively, the time constraint is in terms of the number of $\Omega(\mathcal{D})$ operations where $\mathcal{D}$ is the dimensionality of the dataset in question.

Datasets. For the empirical evaluation of the proposed time-constrained search algorithms, we will use consistently use the following 4 datasets: (i) The Optdigits digit dataset (Blake and Merz, 1998), (ii) a subset of the 80 million tiny images dataset

Table 4: The datasets used for empirical evaluations in this chapter.

Datasets	# points	# queries	# dimensions
Optdigits	3800	1800	64
Tiny Images	100000	100000	384
MNIST	60000	10000	784
Yahoo! Music	100000	1000	50

(Torralba et al., 2008), (iii) the MNIST digits dataset (LeCun, 2000), and (iv) a subset of the Yahoo! music recommendation dataset (Dror et al., 2011). The sizes of the datasets are presented in Table 4. These datasets are again chosen primarily from the applications of nearest-neighbor search such as machine learning, computer vision and collaborative filtering. These 4 datasets are chosen because exact search on these sets with *kd*-trees do not show any significant improvement over brute-force search.

4.2 Permutation-based Time-Constrained Search

A natural way of returning a solution within a fixed time-constraint (of any size) is to structure the search algorithm in an incremental manner. One way of achieving this is to lay the data (the points in the reference set) in some order and then sequentially search through this list of points and return the best candidate encountered once the time runs out (Algorithm 11). In this section, we will present different heuristics for "laying the data" or permuting the data, and present the empirical and, in one case, theoretical performance of these heuristics.

4.2.1 Random Permutation

One heuristic is to randomly permute the data for every query. So every query q sees a permutation of the data and sequentially goes through this permutation. While being extremely simple, this heuristic also provides some theoretical guarantees on the rank-error of the returned best neighbor.

Algorithm 11 Time-constrained search with permutations

Input: Data R, permutation P, query q, allowed time t
Output: Neighbor candidate p

Initialize: Set $i \leftarrow 0$, $d \leftarrow \infty$
while $i < t$ **do**
  if $\mathbf{d}(q, P(R)[i]) < d$ **then**
    $d \leftarrow \mathbf{d}(q, P(R)[i])$; $p \leftarrow P(R)[i]$
  end if
  $i \leftarrow i + 1$
end while

Theorem 4.2.1. *For a given set R of n points and any query q with a time-constraint t, with probability $1-\delta$ for $\delta \in (0, 1)$, the maximum rank for time-constrained search with a random permutation is given by*

$$\tau(q) \leq n\left(1 - \delta^{1/t}\right). \tag{14}$$

Proof. With random sampling, the probability of having at least one candidate from within the top τ neighbors after encountering t candidates is at least $(1 - \tau/n)^t$. Equating this to $1-\delta$ gives the upper bound on the maximum rank-error for a single query. □

4.2.2 Gonzalez Permutation

Gonzalez (1985) presented an approximate clustering algorithm that also serves as method for obtaining an ϵ-net of the dataset of size at most twice the smallest ϵ-net of this dataset. This algorithm can be used to create a permutation (Algorithm 12) of the data that serves as a progressively finer ϵ-net of the data as we proceed along the permutation. The algorithm to generate this permutation is essentially deterministic. The randomness only comes from the selection of the first point in the permutation.

4.2.3 k-means++ Permutation

k-means++ (Arthur and Vassilvitskii, 2007) is a heuristic to pick seeds for the k-means algorithm with interesting guarantees. It can be thought of as a probabilistic

Algorithm 12 Gonzalez permutation

Input: Data $R = \{r_1, \ldots, r_n\}$
Output: Permutation G of $[1 : n]$
Initialize: Pick a random $j \in [1 : n]$; $G[1] \leftarrow j$
for $i \leftarrow 2 : n$ **do**
  $G[i] = \arg \max_{j \in [1:n] \setminus G_{[1:i-1]}} \min_{k \in G_{[1:i-1]}} \mathbf{d}(r_j, r_k)$
end for

version of the Gonzalez permutation (Algorithm 12) where points farther away from points currently in the permutation are given a higher probability for being chosen as the next point in the permutation. The precise method to generate this k-means++ permutation is presented in Algorithm 13.

Algorithm 13 k-means++ permutation

Input: Data $R = \{r_1, \ldots, r_n\}$
Output: Permutation K of $[1 : n]$
Initialize: Pick a random $j \in [1 : n]$; $K[1] \leftarrow j$
for $i \leftarrow 2 : n$ **do**
  Create a probability distribution P_i over $X \in [1 : n]$ such that

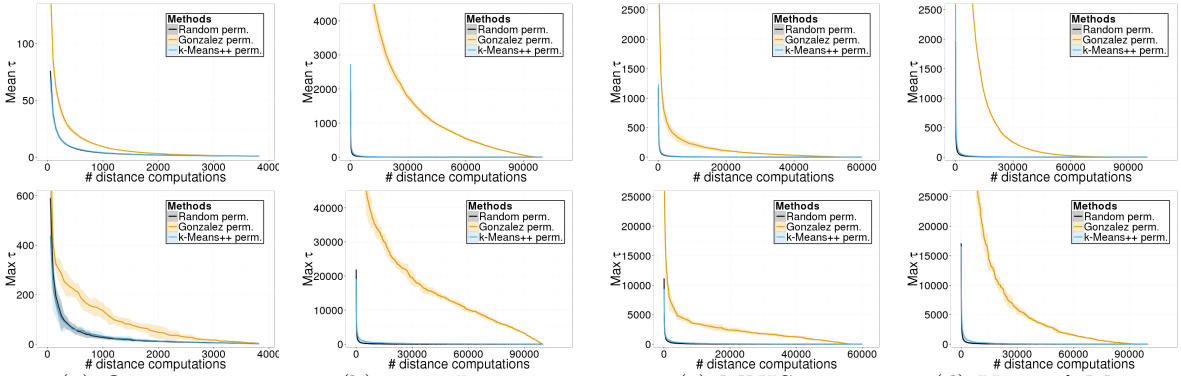
$$\Pr_{P_i}(X = j) = \frac{\min_{k \in K_{[1:i-1]}} \mathbf{d}^2(r_j, r_k)}{\sum_{j' \in [1:n]} \min_{k \in K_{[1:i-1]}} \mathbf{d}^2(r_{j'}, r_k)}$$

  Randomly pick a sample $X \sim P_i$.
  $K[i] = X$
end for

4.2.4 Empirical Evaluation and Discussion

In this section we will evaluate the performance of time-constrained search (TCS) with permutations (Algorithm 11) for the three different heuristics presented to create permutations. We will consider the datasets in Table 4. As mentioned earlier, the nearest-neighbor error (τ and ε) are noted for chosen time-constraints (the time constraints are selected in terms of the number of distance computations or vector operations). There are no parameters that need to be set for any of these algorithms.

Since there is randomness involved in each of the permutations, a single experiment is repeated 10 times and the mean and maximum nearest-neighbor error are averaged over these 10 repetitions and are presented along with the standard error in the form of a translucent ribbon (of the same color) around the average in each of the plots. Figures 22 & 23 present the performance of TCS with different permutations in terms of the rank and distance error respectively. The top row in each figure corresponds to the mean error over all the queries while the bottom row corresponds to the maximum error.

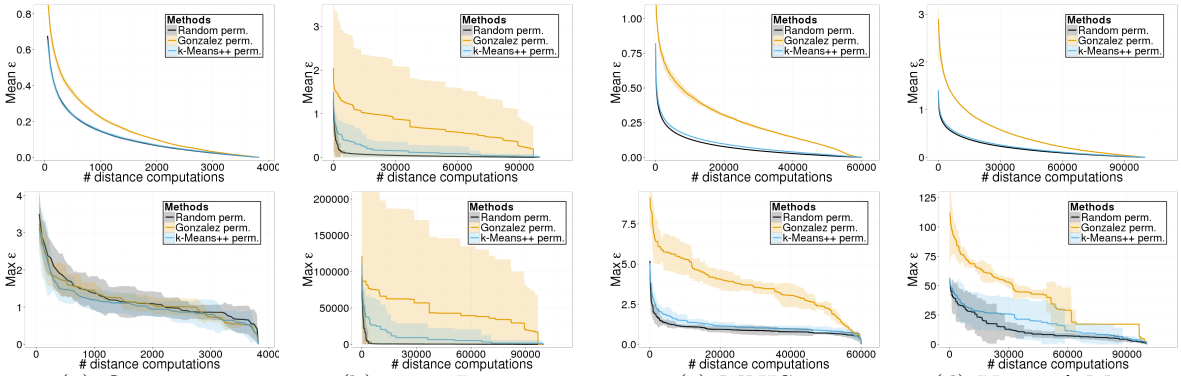


(a) Optdigits (b) Tiny Images (c) MNIST (d) Yahoo! Music

Figure 22: Rank with increasing time constraint. The top row corresponds to the mean rank over all queries and the bottom row corresponds to the maximum rank over all queries (*please view in color*).

The results in Figure 22 indicate that the random permutation and the k-means++ permutation (Algorithm 13) perform significantly better than the Gonzalez permutation (Algorithm 12) both in terms of the mean rank and maximum rank. The performance of the Gonzalez permutation with respect to the random permutation (and the k-means++ permutation) appears to degrade with increasing size of the set. The relative performance in the larger sets (Tiny Images and Yahoo! Music) are much worse than in the smaller set (Optdigits). In all the cases, the performance of TCS with permutations does not seem to be significantly affected with repetitions,

seen by the very low (often not clearly visible) standard error of the statistics (mean and maximum τ).



(a) Optdigits (b) Tiny Images (c) MNIST (d) Yahoo! Music

Figure 23: Distance error with increasing time constraint. The top row corresponds to the mean distance error over all queries and the bottom row corresponds to the maximum distance error over all queries (*please view in color*).

The results in Figure 23 indicate that the random and k-means++ permutations perform better than the Gonzalez permutation also in terms of the nearest-neighbor distance-error. However, in many cases, the difference is not as significant as in the case of nearest-neighbor rank. In all the cases except for Tiny Images, the mean ε appears to be not affected by random restarts (validated by the insignificant standard error). However, in the case of Tiny Images, the standard error is fairly significant, especially in the case of the Gonzalez permutation. This can be attributed to the fact that in the Tiny Images set, there are many instances of near-duplicates (instances where the nearest-neighbor of the query is its almost duplicate), in which case, the distance error for any object other than the (near) duplicate is very high (possibly infinite). Hence, for TCS with the Gonzalez permutation, if some queries find their (near) duplicate in one instantiation of the Gonzalez permutation but do not in another instantiation, the standard error of the mean distance error can become very large (as can be seen in our results).

The relative performance of the three heuristics appear to be the similar in case of the maximum distance error. The only difference is that the standard error (depicted by the translucent ribbon around the maximum distance error averaged over 10 repetitions) is significant in almost all situations. The random permutation appears to perform better than the k-means++ permutation in terms of the maximum distance error; significantly with the TINY IMAGES dataset, and slightly with the MNIST and YAHOO! MUSIC datasets.

4.2.5 Going Forward

Permutation based TCS is a very simple algorithm for TCS. But empirical results indicate that the random permutation appears to perform best compared to more sophisticated permutations such as the Gonzalez and the k-means++ permutations. However, the possible reason why the random permutation performs better than others is that in TCS with random permutation, every query gets a (possibly) new permutation, whereas, with the other two heuristics, all the queries get the same permutation. If the completely random permutation performs better than a fixed permutation, it seems intuitive that a more query adaptive permutation scheme (where every query gets a permutation designed specifically for it) would have a much better TCS performance. For this reason, we will explore TCS algorithms that utilize indexing schemes to get query-dependent permutations.

4.3 Binary Tree based Time-Constrained Search

Binary trees have been extensively studied and used for the task of nearest-neighbor search. But as mentioned in Chapter 2, the usual tree-based depth-first branch-and-bound algorithm (presented here again in Algorithm 14) not only needs to find the

neighbor within the desired level of approximation, but also validate (or get a certificate) *during the algorithm* that the returned neighbor is in fact within the desired approximation (line 6 and line 8 accomplish this validation by visiting all nodes that cannot be ignored with proper guarantee provided by the 'cannot_prune_node(q, N, d, e)' function).

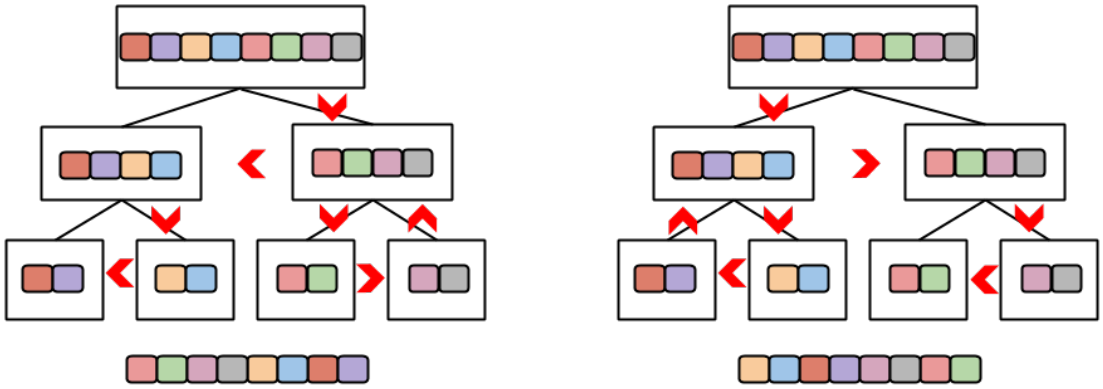
Algorithm 14 Error-constrained search with trees. Every non-leaf node N in the tree T is partitioned into a left (N.left) and a right (N.right) child with the hyperplane $(N.\mathbf{w}, N.b)$. The function 'cannot_prune_node(q, N, d, e)' decides if the node N can be safely ignored for query q with distance d to the current best neighbor candidate p and allowed approximation e.

Input: Data R, Tree T Query q, Allowed approximation e
Output: Neighbor candidate p

```
3: Initialize: Set d ← ∞, set stack S.push(T)
   while S is non-empty do
     Node N ← S.pop
6:   if cannot_prune_node(q, N, d, e) then
       if is_a_leaf_node(N) then
         for r ∈ N ∩ R do
9:         if d(q, r) < d then
             p ← r; d ← d(q, r)
           end if
12:      end for
       else
         if ⟨N.w, q⟩ + N.b < 0 then
15:        S.push(N.right); S.push(N.left)
         else
           S.push(N.left); S.push(N.right)
18:      end if
       end if
     end if
21: end while
```

For any query, the tree based error-constrained search (ECS) traverses the tree in a greedy manner, encountering those points first that the tree predicts to be closer to the query. So the greedy tree traversal effectively generates a query dependent permutation of the data. Figure 24 demonstrates how a tree traversal results in a permutation. The search algorithm then has to go through the whole permutation,

only ignoring chunks that can be safely pruned from the computation without violating the error constraint.



(a) Example 1 (b) Example 2

Figure 24: Generating adaptive permutations. Examples of a tree-traversal effectively generating a permutation (*please view in color*).

In the following subsections, we will present three ways of utilizing the query-adaptive permutation produced by the tree for time-constrained search. The first algorithm, greedy tree search (Section 4.3.1), simply duplicates the error-constrained search algorithm with zero error and just stops computation when time runs out. The second algorithm, lazy defeatist tree search (Section 4.3.2), is a simplification of the first algorithm which adversely affects its performance with respect to the first algorithm, but allows for intuitive theoretical guarantees. The third algorithm, defeatist forest search (Section 4.3.3), utilizes multiple random binary trees instead of a single one to give a simpler time-constrained search algorithm.

4.3.1 Greedy Time-Constrained Search with Binary Trees

A natural way to convert error-constrained search with a tree to a time-constrained search algorithm is to keep track of the elapsed time and return the current best candidate when the time constraint is met. The time-constrained search algorithm resulting from this modification is shown in Algorithm 15. Here lines 12, 19 and 22 keep track of the time spent (in terms of the number of vector operations performed).

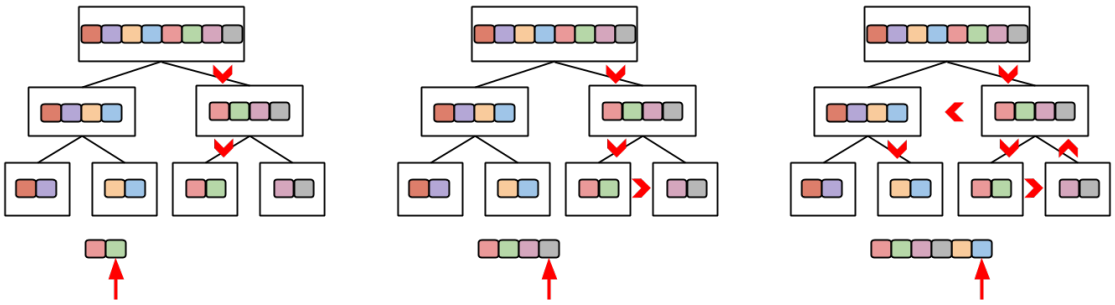
Lines 13-15 returns the current best candidate if the time has run out. We call this algorithm *greedy tree search* since this algorithm greedily traverses through the tree until time runs out.

Algorithm 15 Greedy tree search. Every non-leaf node N in the tree T is partitioned into a left (N.left) and a right (N.right) child with the hyperplane $(N.\mathbf{w}, N.b)$. The function 'cannot_prune_node$(q, N, d, 0)$' decides if the node N can be safely ignored for query q with distance d to the current best neighbor candidate p and no allowed approximation.

Input: Data R, Tree T Query q, Allowed time t
Output: Neighbor candidate p

Initialize: Set $d \leftarrow \infty$, elapsed time $i \leftarrow 0$, set stack S.push(T)
while S is non-empty & $i < t$ **do**
  Node $N \leftarrow S$.pop
  if cannot_prune_node$(q, N, d, 0)$ **then**
    if is_a_leaf_node(N) **then**
      for $r \in N \cap R$ **do**
        if $\mathbf{d}(q, r) < d$ **then**
          $p \leftarrow r$; $d \leftarrow \mathbf{d}(q, r)$
        end if
        $i \leftarrow i + 1$
        if $i = t$ **then**
          BREAK out of search loop
        end if
      end for
    else
      if $\langle N.\mathbf{w}, q\rangle + N.b < 0$ **then**
        $i \leftarrow i + 1$
        S.push(N.right); S.push(N.left)
      else
        $i \leftarrow i + 1$
        S.push(N.left); S.push(N.right)
      end if
    end if
  end if
end while

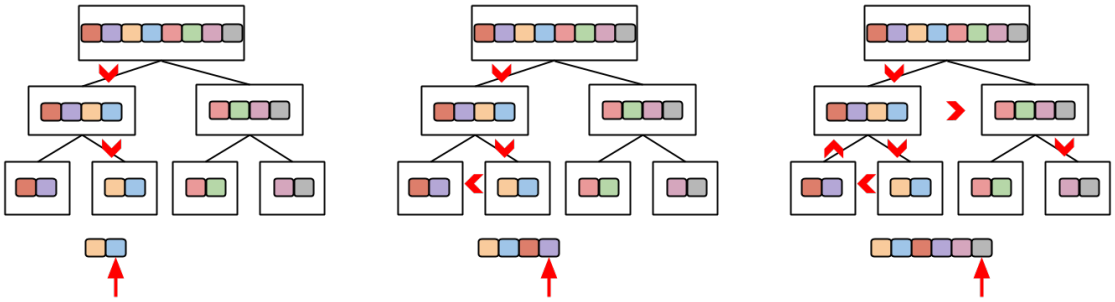
We do assume here that the time constraint is large enough to allow the search algorithm to reach the first leaf node. This is because in a binary tree, the actual points usually reside in the leaf node. Up until the leaf node, the search algorithm does not encounter any actual point in R. To obtain a TCS algorithm that can have



(a) Traversal to first leaf (b) Traversal to second leaf (c) Traversal to third leaf

Figure 25: Progress of the *greedy tree search* (Algorithm 15). Example I (*please view in color*).

any time constraint, we can easily remedy the binary tree to make sure that every tree node has some pivot points (at least one pivot point associated with it).



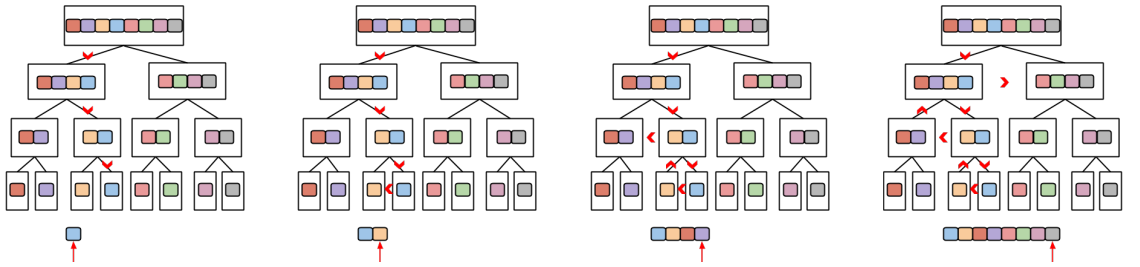
(a) Traversal to first leaf (b) Traversal to second leaf (c) Traversal to third leaf

Figure 26: Progress of the *greedy tree search* (Algorithm 15). Example II (*please view in color*).

The greedy tree search (Algorithm 15) traverses the tree in the following way (this is demonstrated pictorially in Figures 25 & 26): It first visits the leaf in the tree that contains the query. Then the algorithm backtracks up the tree, again greedily traversing down subtrees that cannot be provably pruned for not containing any better candidate than the current one. The algorithm continues until the time runs out, at which point it returns the current best candidate.

4.3.2 Lazy-Defeatist Time-Constrained Search with Binary Trees

In the greedy tree search algorithm (Algorithm 15), after visiting the leaf of the tree containing the query, the algorithm greedily traverses the rest of the tree that cannot be pruned. This tree traversal can be simplified by greedily traversing only to the leaf containing the query. This is usually called the defeatist tree search. Since the leaves of the trees usually only contain few points, there might be time leftover. In this case, we can backtrack up the tree, but instead of greedily traversing subtrees that cannot be pruned, we will just consider all the points in the subtree in the sequence that we encounter them, without pruning any subtree or any further greedy traversal. We call this the *lazy-defeatist tree search* (Algorithm 16), because we start with the greedy algorithm, but resort to a non-greedy traversal of the tree once we have visited the first leaf node that would contain the query. The progress of the search algorithm with time is pictorially demonstrated in Figures 27 & 28.



(a) Visit the 1^{st} node (b) Visit the 2^{nd} node (c) Visit the 3^{rd} node (d) Visit the 4^{th} node

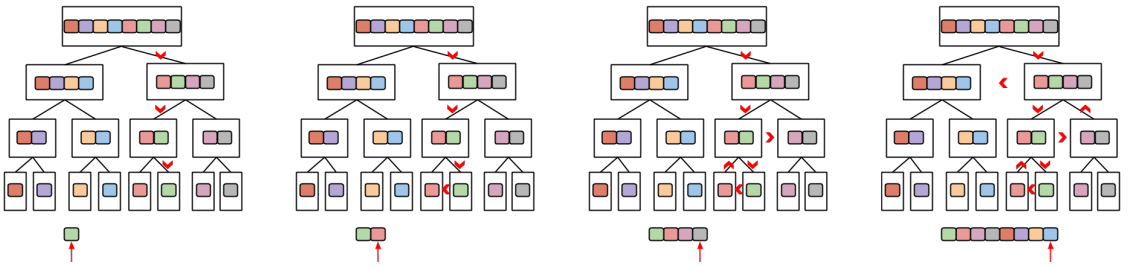
Figure 27: Progress of *lazy-defeatist tree search* (Algorithm 16). Example I (*please view in color*).

This algorithm is simpler than the greedy tree search algorithm for time-constrained search because neither is there any pruning involved nor is there any greedy traversal in the tree beyond the defeatist traversal to the leaf containing the query. Unlike the tree traversal of the greedy tree search (Algorithm 15), the tree traversal in the lazy-defeatist search is quite predictable and hence can allow for some theoretical analysis. In terms of performance, it is possible that the lazy-defeatist search does

Algorithm 16 Lazy-defeatist tree search. Every non-leaf node N in the tree T is partitioned into a left (N.left) and a right (N.right) child with the hyperplane $(N.\mathbf{w}, N.b)$.

Input: Data R, Tree T Query q, Allowed time t
Output: Neighbor candidate p

Initialize: Set $d \leftarrow \infty$, elapsed time $i \leftarrow 0$, stack $S \leftarrow \emptyset$, current node $N \leftarrow T$
while N is not a leaf node **do**
  if $\langle N.\mathbf{w}, q \rangle + N.b < 0$ **then**
    $i \leftarrow i + 1$
    S.push(N.right); $N \leftarrow N$.left
    // greedily traversing down the left node and saving the right node for later
  else
    $i \leftarrow i + 1$
    S.push(N.left); $N \leftarrow N$.right
    // greedily traversing down the right node and saving the left node for later
  end if
end while
S.push(N)
while S is not empty **do**
  Node $N \leftarrow S$.pop
  for $r \in N \cap R$ **do**
    if $\mathbf{d}(q, r) < d$ **then**
      $p \leftarrow r$; $d \leftarrow \mathbf{d}(q, r)$
    end if
    $i \leftarrow i + 1$
    if $i = t$ **then**
      BREAK out of search loop
    end if
  end for
end while



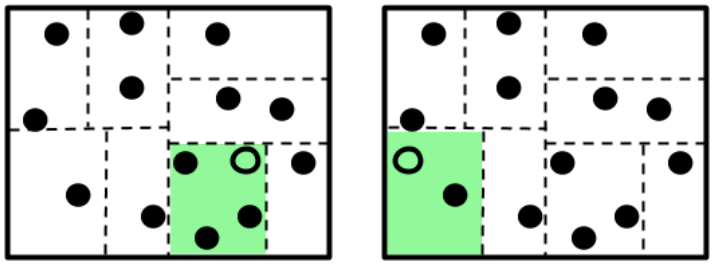
(a) Visit the 1^{st} node (b) Visit the 2^{nd} node (c) Visit the 3^{rd} node (d) Visit the 4^{th} node

Figure 28: Progress of *lazy-defeatist tree search* (Algorithm 16). Example II (*please view in color*).

some extra work by visiting subtrees it could have provably pruned in the greedy tree search algorithm. However, in the situations where no pruning was achieved in a subtree, the lazy-defeatist search would encounter the same set of points without paying the price of the tree traversal. Hence, it is not apparent as to which of the algorithms is superior for time-constrained search. We will compare the relative empirical performances of these two and other time-constrained search algorithms in the Section 4.4 of this chapter.

4.3.3 Defeatist Time-Constrained Search with Space Partitioning Forests

A binary space-partitioning tree (BSP-tree) hierarchically partitions the space containing the data (as suggested by the name) using some heuristic. This heuristic can be deterministic or randomized. Effectively, this tree assigns a region of the space for every point in the data. Each region can be shared by multiple points but a point cannot belong to more than one region. Figure 29 gives two examples of query points assigned to regions in the space by a BSP-tree.

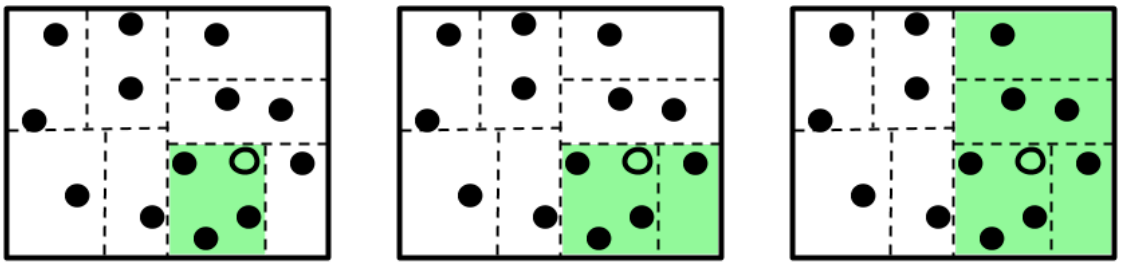


(a) Example I (b) Example II

Figure 29: Region assigned to a query by the BSP-tree.

When a query point is assigned to a region of the space by a BSP-tree, the hope is that the points in the same region as the query are potentially the nearest-neighbors of the query. We will call these the candidate neighbors for the query. Since we might need and can afford more candidate neighbors, we effectively look at larger and larger regions of the space when we backtrack up the tree in greedy tree search

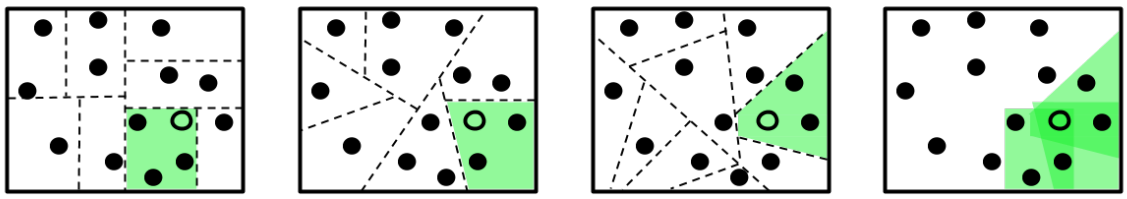
and lazy-defeatist tree search (Algorithms 15 & 16). Figure 30 demonstrates this way of encountering more (and potentially better) neighbor candidates if time permits.



(a) Deepest level (b) Backtrack one level up (c) Backtrack two levels up

Figure 30: Investigating larger and larger regions in search for the nearest-neighbor.

Another way of securing more candidate neighbors is to introduce some randomness in the tree construction so that the same query is assigned different (but overlapping) regions of the space for every new tree constructed. In this situation, we can keep a collection of trees, a forest, and perform a defeatist search on each of the trees to gather candidate neighbors for the query until we run out of time. This process is pictorially demonstrated in Figure 31. We call this time-constrained search algorithm *defeatist forest search* and is formally presented in Algorithm 17.



(a) Region from tree I (b) Region from tree II (c) Region from tree III (d) Combined regions

Figure 31: Defeatist-forest search. Generating neighbor candidates by looking at different regions assigned to the query by three BSP-trees.

This defeatist forest search makes the tree traversal extremely simple and predictable, while still being capable of obtaining more candidate neighbors for queries. This makes defeatist forest search easier to implement (and theoretically analyze)

Algorithm 17 Defeatist forest search. Every non-leaf node N in each tree T of the forest F is partitioned into a left (N.left) and a right (N.right) child with the hyperplane $(N.\mathbf{w}, N.b)$.

Input: Data R, Forest F of trees T Query q, Allowed time t
Output: Neighbor candidate p

Initialize: Set $d \leftarrow \infty$, elapsed time $i \leftarrow 0$
for each $T \in F$ **do**
  Set node $N \leftarrow T$
  while N is not a leaf node **do**
    if $\langle N.\mathbf{w}, q \rangle + N.b < 0$ **then**
      $i \leftarrow i + 1$; $N \leftarrow N$.left
    else
      $i \leftarrow i + 1$; $N \leftarrow N$.right
    end if
  end while
  for $r \in N \cap R$ **do**
    if $r \notin S$ **then**
      if $\mathbf{d}(q, r) < d$ **then**
        $p \leftarrow r$; $d \leftarrow \mathbf{d}(q, r)$
      end if
      $i \leftarrow i + 1$
      if $i = t$ **then**
        BREAK out of search loop
      end if
    end if
  end for
end for

compared to greedy search and lazy-defeatist search. However, there are two potential issues with defeatist forest search – (1) Unlike greedy search and lazy-defeatist search, defeatist forest search is not guaranteed to find the nearest-neighbor of the query even if ample time is provided to the algorithm. (2) While the previous two proposed tree-based time-constrained search algorithms only required a single tree data structure, defeatist forest search needs a forest (collection of trees) instead of a single tree, increasing the space requirements of this algorithm.

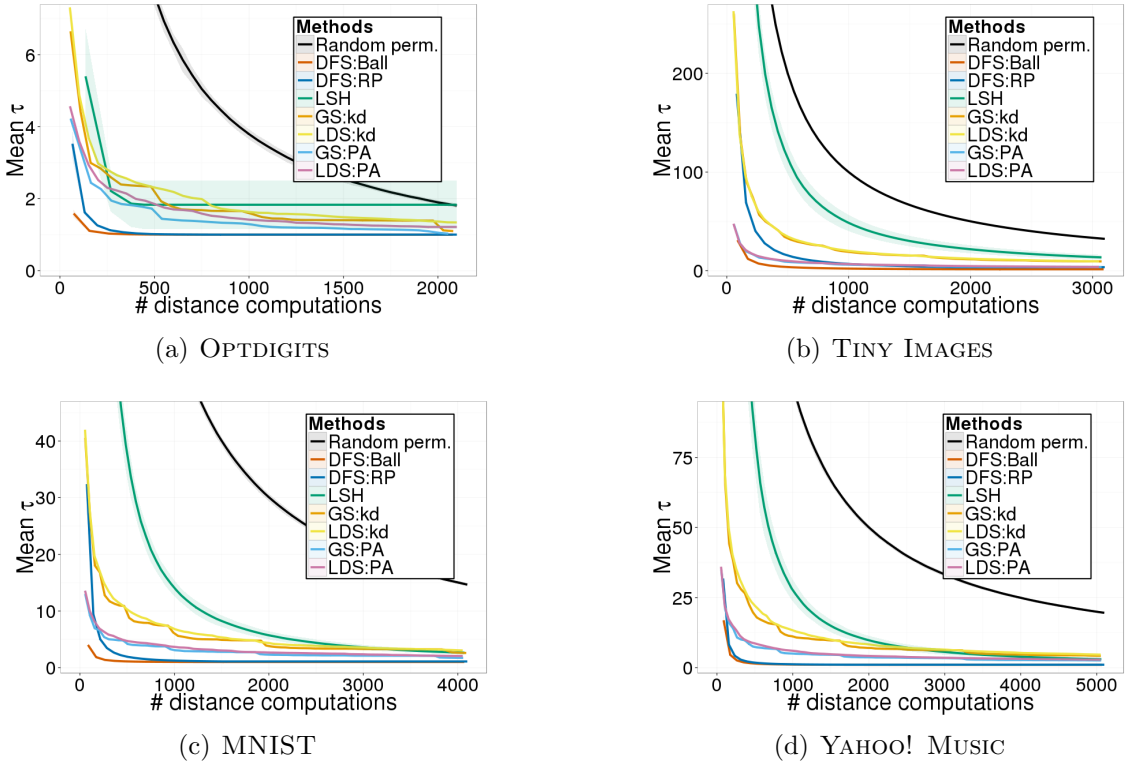
4.4 Empirical Evaluation of Tree-based Algorithms

In this section we compare the three proposed tree-based time-constrained search (TCS) algorithms on the 4 datasets described in Table 4. We consider the performance of TCS with random permutations (Algorithm 11) as the baseline. For greedy tree search (Algorithm 15) and lazy-defeatist tree search (Algorithm 15), we use the *kd*-tree (Friedman et al., 1977) and the *PA*-tree (Sproull, 1991; McNames, 2001). For defeatist forest search (Algorithm 17), we use two random binary space-partitioning trees – the *RP*-tree (Dasgupta and Freund, 2008) and the metric ball-tree (Preparata and Shamos, 1985). We will again consider both the distance error and rank as the notions of nearest-neighbor error. For random methods (such as defeatist forest search and random permutations), we repeat the experiment 10 times.

4.4.1 Comparison to Locality-Sensitive Hashing

In this first setup, we compare all the proposed algorithms to each other as well as to the performance of p-stable locality-sensitive hashing (LSH) (Datar et al., 2004). Time-constrained search with LSH is fairly straightforward – we sequentially lookup hash tables one at a time, considering more hash tables when time permits.

Figure 32 presents the relative performance of all the proposed binary tree based TCS algorithms relative to TCS with random permutations and LSH in terms of the rank. The results indicate that in 3 of the datasets, the tree-based TCS algorithms significantly outperform the baseline (TCS with random permutations) and LSH. In the OPTDIGITS set, LSH performs comparably to the greedy tree search and lazy-defeatist tree search algorithms. For all the datasets, the greedy tree search with a particular tree (*kd*-tree or *PA*-tree) appears to perform almost identically to lazy-defeatist tree search (with the corresponding tree). However, between the two deterministic trees, *kd*-tree and *PA*-tree, *PA*-tree consistently outperforms *kd*-tree significantly for both TCS algorithms (greedy or lazy-defeatist). The defeatist forest



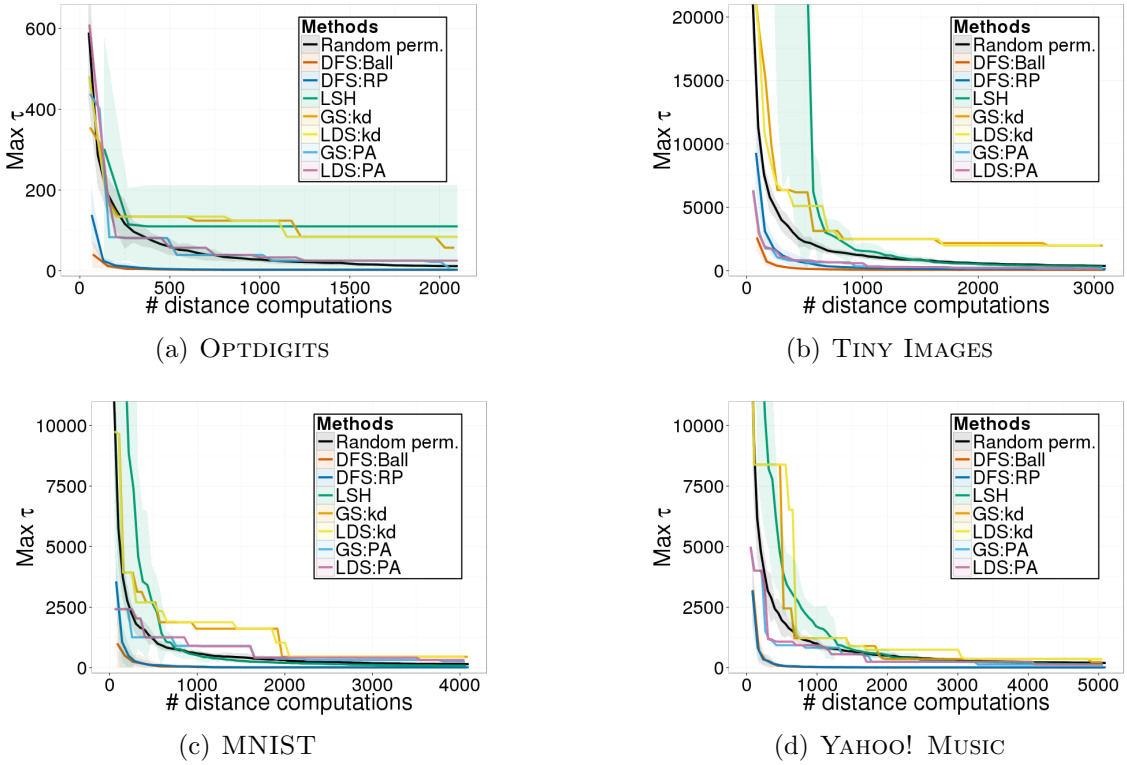
(a) Optdigits (b) Tiny Images

(c) MNIST (d) Yahoo! Music

Figure 32: Mean rank with increasing time constraint. Each sub-figure corresponds to the mean rank over all queries for different datasets. GS:*kd* & GS:*PA* refer to Algorithm 15 with a *kd*-tree and *PA*-tree respectively. LDS:*kd* & LDS:*PA* refer to Algorithm 16 with a *kd*-tree and *PA*-tree respectively. DFS:*RP* & DFS:Ball refer to Algorithm 17 with a forest of *RP*-trees and ball-trees respectively (*please view in color*).

search with ball-trees performs the best in all instances in terms of the mean rank. Defeatist forest search with *RP*-trees performs significantly better than greedy search and lazy-defeatist search for all trees in two of the cases (Optdigits & Yahoo! Music), while performing worse than the *PA*-tree in case of the Tiny Images set. In case of the MNIST set, the defeatist forest search with *RP*-trees begin worse than the *PA*-tree, but quickly start outperforming the *PA*-trees once the time-constraint crosses a certain point.

In terms of the maximum rank (presented in Figure 33), the performance of the tree-based TCS algorithms and LSH is much more comparable to the baseline. The



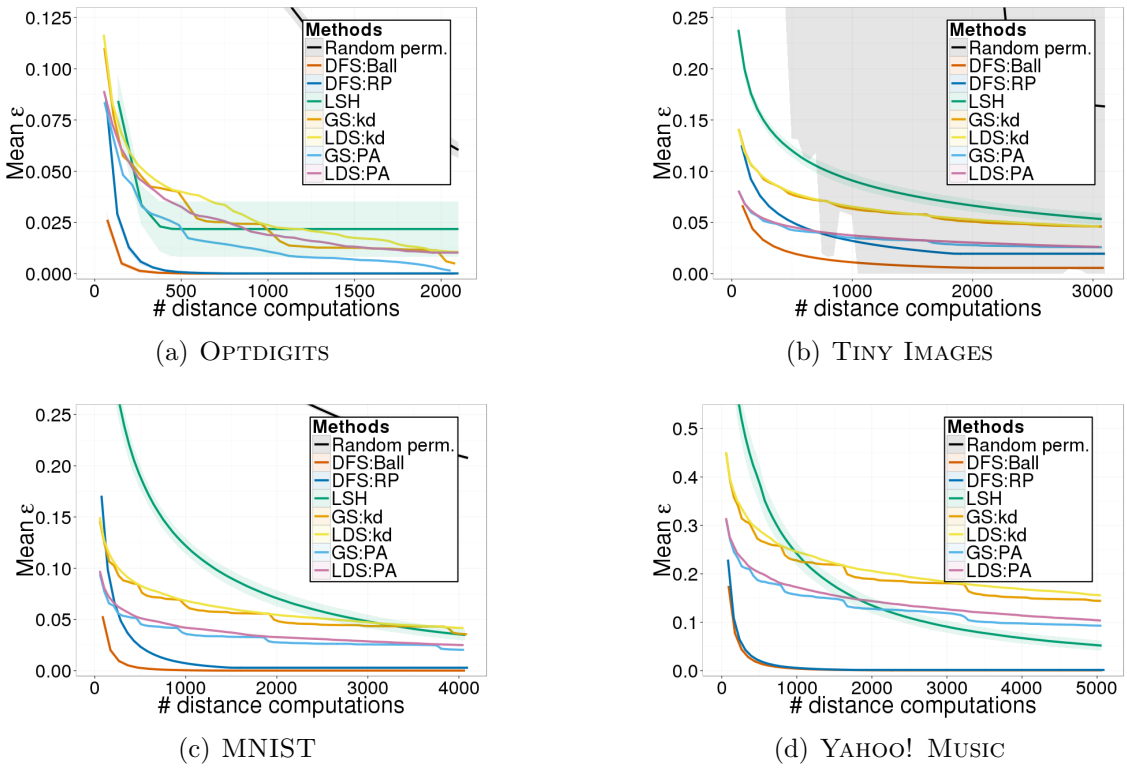
(a) Optdigits (b) Tiny Images

(c) MNIST (d) Yahoo! Music

Figure 33: Maximum rank with increasing time constraint. Each sub-figure corresponds to the maximum rank over all queries for different datasets. GS:*kd* & GS:*PA* refer to Algorithm 15 with a *kd*-tree and *PA*-tree respectively. LDS:*kd* & LDS:*PA* refer to Algorithm 16 with a *kd*-tree and *PA*-tree respectively. DFS:*RP* & DFS:Ball refer to Algorithm 17 with a forest of *RP*-trees and ball-trees respectively (*please view in color*).

greedy and lazy-defeatist tree search with *kd*-tree performs slightly worse than the baseline, and so does LSH. The greedy and lazy-defeatist search with *PA*-tree performs at least comparably to the baseline in terms of maximum rank, outperforming the baseline in 2 of the instances. Defeatist forest search with ball-trees again performs the best in all instances, while defeatist forest search with *RP*-trees performs slightly worse. The relative performance of the proposed tree-based TCS algorithms are consistent both in the average case (mean rank) and the worst case (maximum rank).

Figure 34 presents the performance of all the proposed binary tree based TCS

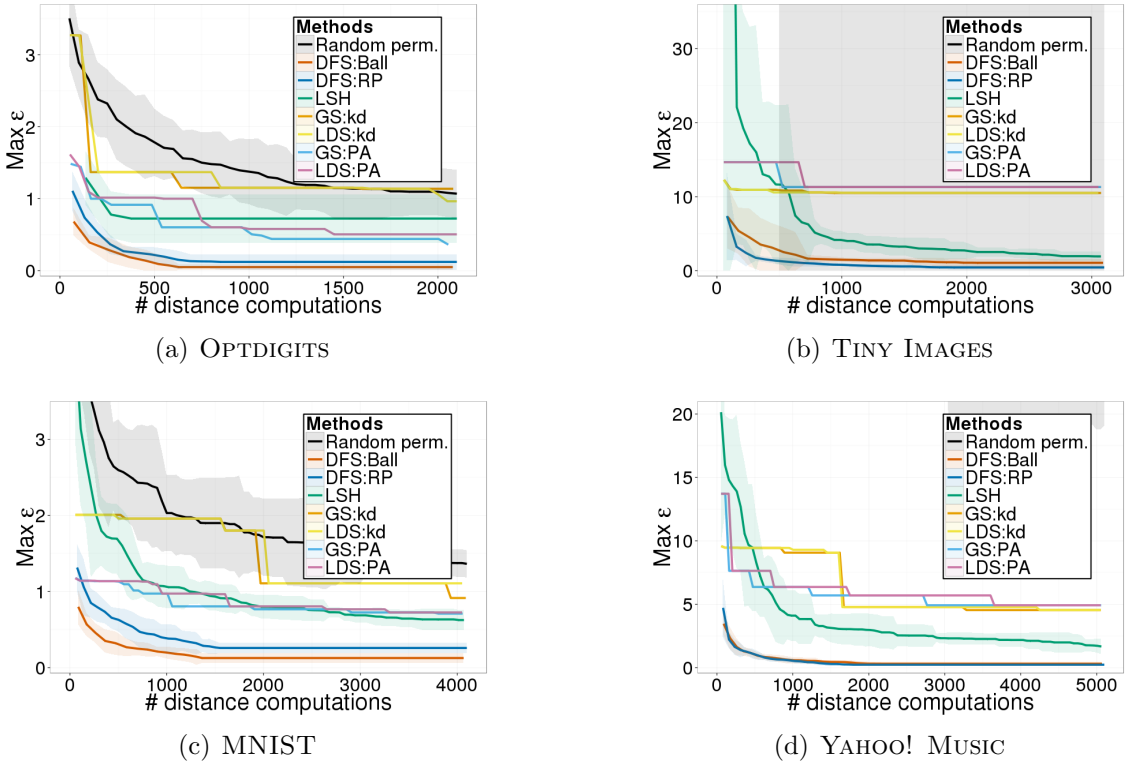


(a) Optdigits (b) Tiny Images

(c) MNIST (d) Yahoo! Music

Figure 34: Mean distance error with increasing time constraint. Each sub-figure corresponds to the mean distance error over all queries for different datasets. GS:*kd* & GS:*PA* refer to Algorithm 15 with a *kd*-tree and *PA*-tree respectively. LDS:*kd* & LDS:*PA* refer to Algorithm 16 with a *kd*-tree and *PA*-tree respectively. DFS:*RP* & DFS:Ball refer to Algorithm 17 with a forest of *RP*-trees and ball-trees respectively (*please view in color*).

algorithms relative to TCS with random permutations and LSH in terms of the mean distance error over all queries. The proposed tree-based TCS algorithms and LSH outperforms the baseline by a significant margin. As in the case of nearest-neighbor rank, the performances of the greedy search and lazy-defeatist search algorithm for the same binary tree (*kd*-tree or *PA*-tree) appear to be very similar in all the cases, with the algorithms performing significantly better with the *PA*-tree. Defeatist forest search with ball-trees again perform the best among all algorithms in terms of the mean distance error. Defeatist forest search with *RP*-trees perform slightly worse than with ball-trees, but still performing better than the other proposed tree-based



(a) Optdigits (b) Tiny Images

(c) MNIST (d) Yahoo! Music

Figure 35: Maximum distance error with increasing time constraint. Each sub-figure corresponds to the maximum distance error over all queries for different datasets. GS:*kd* & GS:*PA* refer to Algorithm 15 with a *kd*-tree and *PA*-tree respectively. LDS:*kd* & LDS:*PA* refer to Algorithm 16 with a *kd*-tree and *PA*-tree respectively. DFS:*RP* & DFS:Ball refer to Algorithm 17 with a forest of *RP*-trees and ball-trees respectively (*please view in color*).

TCS algorithms in most cases. Defeatist forest search with both the random trees significantly outperform LSH. The greedy and lazy-defeatist search with both the deterministic trees perform comparably to LSH, outperforming LSH in two of the cases.

The performance of the TCS algorithms with respect to maximum distance error is presented in Figure 35. The relative performance of LSH is much better in terms of the maximum distance error. LSH outperforms greedy and lazy-defeatist search in two of the cases, performing comparably in the rest of the cases. However, the defeatist forest search with both the trees outperform the rest of the algorithms by

a significant margin, with ball-trees doing better than *RP*-trees in two cases and the *RP*-tree doing better in one of the cases. As in all the previous cases, the greedy tree search and lazy-defeatist tree search have similar performances for the same tree, with *PA*-tree outperforming *kd*-tree in most of the cases.

The results indicate that the greedy tree search (Algorithm 15) and lazy-defeatist tree search (Algorithm 16) have empirically similar performance for the same tree in the high dimensional datasets considered. But between trees, the data-dependent *PA*-tree has significantly better performance than the more data-oblivious *kd*-tree. The greedy search and lazy-defeatist search with a single deterministic *kd*-tree performs at least comparably to LSH in all cases, with the *PA*-tree outperforming LSH in most cases. The defeatist forest search has the best relative performance, justifying the cost of the space requirement of storing a forest instead of a single tree. The forest of ball-trees empirically appears to perform better than the forest of *RP*-trees in almost all the cases. While these are all empirical evidence, we do not have any theoretical results that can allow us to compare the performance of these different proposed tree-based TCS algorithms. The results for LSH are expected because LSH is designed to guarantee maximum distance error (not the mean distance error or any form of rank error). This is why the relative performance of LSH with respect to the maximum distance error is much more competitive than with respect to the rest of the evaluation metrics.

4.4.1.1 Space/Search Quality Tradeoff

In the previous experiment, we saw that defeatist forest search with random forests consistently outperform locality sensitive hashing (LSH) under all notions of error across different datasets. While the tree based methods do not have a straightforward way of increasing the space requirement for better search performance, both defeatist forest search and LSH are capable of tuning the space used by the method. Here we

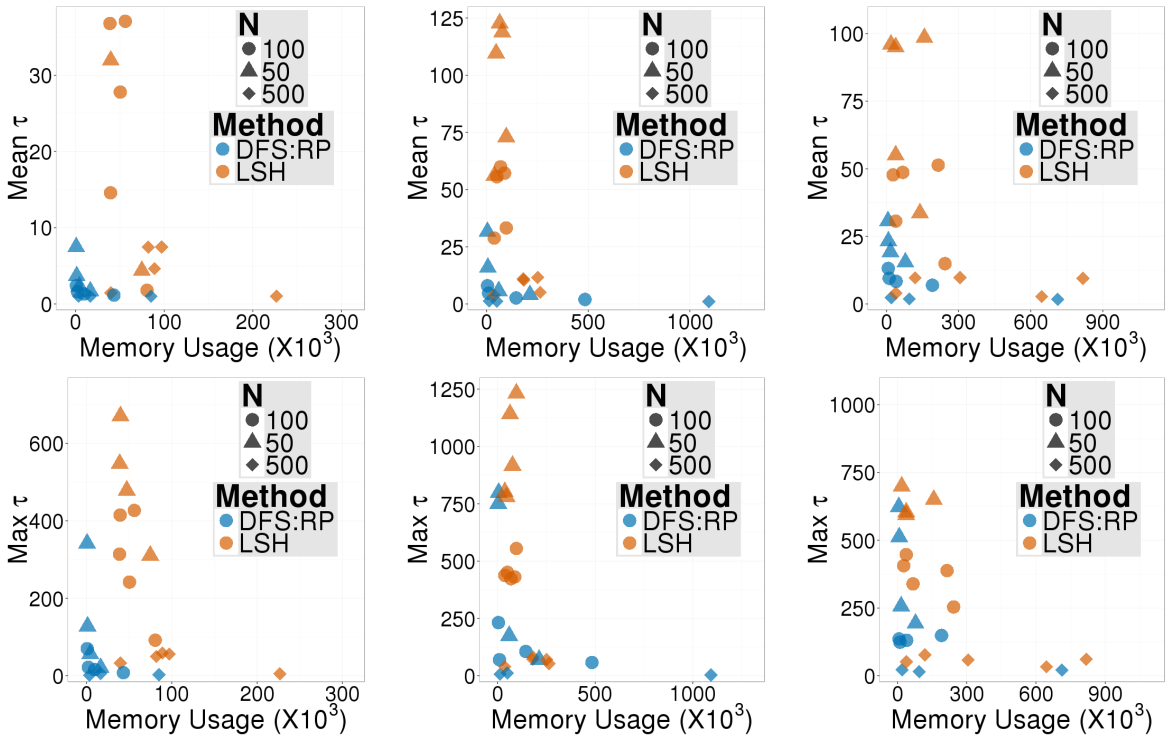
evaluate the relative space/search quality tradeoff of defeatist forest search and LSH.

To study this tradeoff, we fix the number of potential neighbor candidates returned by each method (the best among these neighbor candidates is returned as the approximate nearest-neighbor). This fixes the computation time of the search algorithms once the candidate selection process is complete. We select different values for the sizes of this potential neighbor set (we choose N, the potential neighbor set size, to be $50, 100, 500$).

Given the fixed size of the potential neighbor set, these algorithms can possibly adjust the precision[1] of the preprocessing step in the following way for better quality candidate sets: (i) in LSH, the precision can be increased by increasing the number of projections for each hash table, and (ii) in defeatist forest search, the precision can be increased by decreasing the leaf size of each tree in the forest. Now increasing the precision implies that the number of potential neighbor candidates recovered for the query will decrease (in case of LSH, this implies that smaller number of candidates are obtained from each hash table; in case of defeatist forest search, this implies that smaller number of candidates are obtained from each tree in the forest). In this scenario, to obtain the desired number of potential candidates N, the space usage of the search algorithms will increase in the following way: (i) with LSH, the space usage increases since more hash tables are required to be stored to return a large enough potential candidate set size, and (ii) with defeatist forest search, the space usage also increases since more trees are required to be stored to return a large enough candidate set.

We will evaluate the quality of these recovered potential neighbor sets. Increased precision should imply better quality candidates, but would also imply increased space requirements. We note the memory usage of the method to return the candidate set

[1]Precision here vaguely refers to the probability of finding a query's true nearest-neighbor in this fixed-sized candidate set.

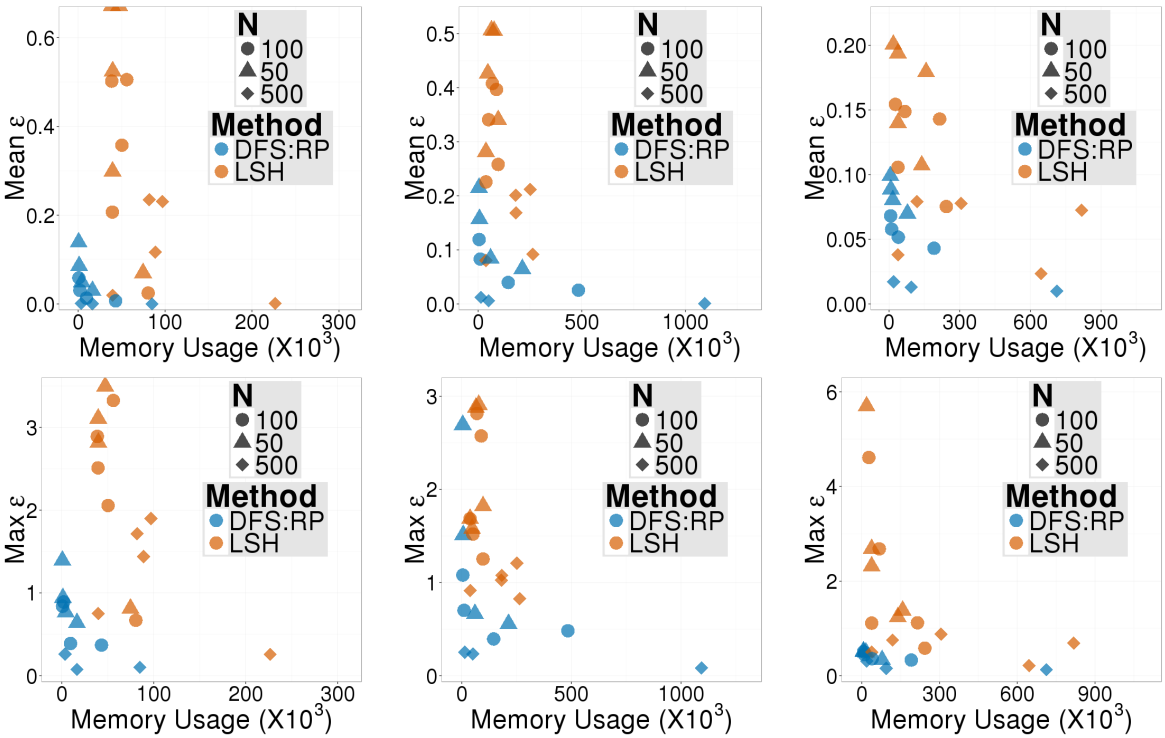


(a) Optdigits (b) MNIST (c) Tiny Images

Figure 36: Rank with increasing space. The top row corresponds to the mean rank over all queries and the bottom row corresponds to the maximum rank over all queries. DFS:*RP* refers to Algorithm 17 with a forest of *RP*-trees (*please view in color*).

of desired size and evaluate the search quality of the candidate set. This is how we study the space/search quality tradeoff of these methods. Here we will only consider forests of *RP*-trees for clarity. The ball-trees have exactly the same memory usage as the *RP*-trees, only the choice of randomness varies between these two trees. The time-constrained search performance of defeatist forest search with ball-trees are always comparable to (or better than) the performance of the forest of *RP*-trees. We also consider subsets of three of the previously used datasets from Table 4.

The relative space/search performance of LSH and defeatist forest search (with *RP*-trees) with respect to the rank of the approximate neighbor is presented in Figure 36. The results indicate that, for large enough candidate set ($N = 500$), defeatist



(a) Optdigits (b) MNIST (c) Tiny Images

Figure 37: Distance error with increasing space. The top row corresponds to the mean rank over all queries and the bottom row corresponds to the maximum rank over all queries. DFS:*RP* refers to Algorithm 17 with a forest of *RP*-trees (*please view in color*).

forest search has low error in terms of rank (both mean and maximum) even for really low memory usage, and the search performance is not affected by the memory usage. The same is also true for LSH with $N = 500$, where the space usage does not significantly affect the search performance. However, for $N = 500$, defeatist forest search always appears to have better performance than LSH with respect to the rank.

For small candidate sets ($N = 50, 100$), the search performance is significantly affected by the memory usage. As expected, both mean and maximum rank (over all queries) decrease with increasing memory usage. This behavior is easily seen in case of defeatist forest search; in case of LSH, this behavior is slightly more noisy. However,

in almost all cases, for the same candidate set size N, defeatist forest search has a better space/search performance tradeoff than LSH. For the same level of accuracy (and N), defeatist forest search always has smaller memory usage (the data points in the plots for defeatist forest search always lie on the left of the data points for LSH).

This same relative performance is also indicated by the space/search time tradeoff with respect to the distance error in Figure 37, with defeatist forest search requiring lesser memory than LSH for similar search performance in all cases (mean and maximum distance error) across all datasets. This implies that defeatist forest search has a better space/search performance tradeoff than LSH for all notions of error (mean/maximum rank or mean/maximum distance error).

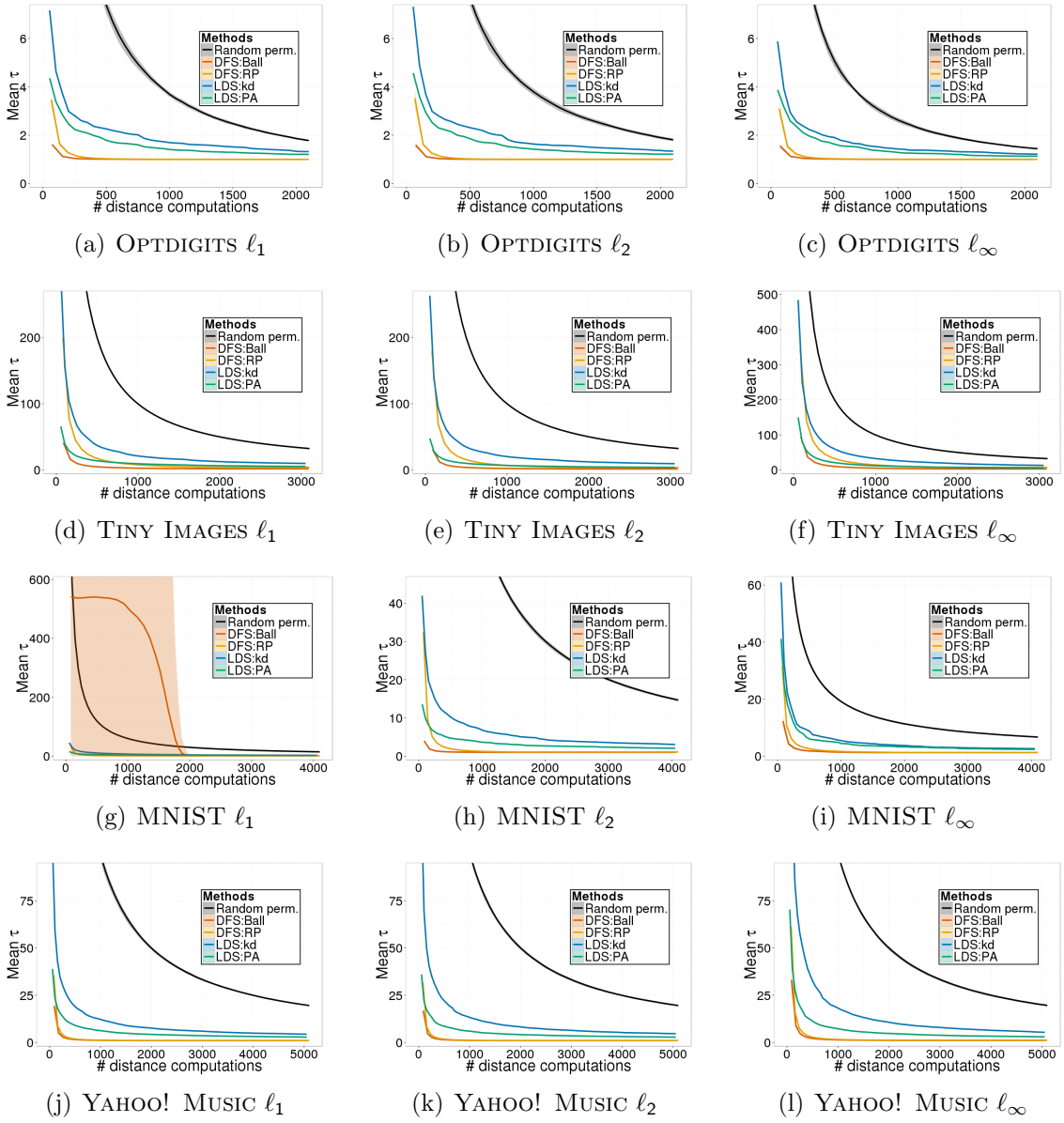
4.4.2 Performance Dependency on the Metric

We consider a different axis of variation in the problem of time-constrained search (TCS) and evaluate the performance of our proposed method along this axis. We consider the performance of the tree based algorithms for three different metrics and study how the relative performance of the proposed methods are affected by the choice of the metric. This means that the candidate nearest-neighbor and the nearest-neighbor error of the candidate nearest-neighbor is chosen with respect to different distance metrics.

Remark. An important point to note here is that the tree construction is not modified for each of the metric. So for the deterministic trees (such as the *kd*-tree and the *PA*-tree), the indexing of the points remains the same. This implies that the tree index is still defined by the space partition in the ℓ_2 (or Euclidean) space containing the points (and the subsequent paths in the tree chosen for the query remains the same). Now for some algorithms, such as the defeatist forest search with *RP*-trees and ball-trees, can metric-specific modifications can be made for the tree construction. However, the *kd*-tree construction is entirely metric independent

and the *PA*-tree is implicitly constructed for the ℓ_2 metric because *PA*-tree aims to capture the covariance structure of the data, which is closely related to the ℓ_2 metric, and there are no straightforward extensions of the *PA*-tree for other metrics. To be consistent, we consider the same construction (and hence the same indexing of the data) for search with different metrics. The hope is that methods allowing for metric-dependent construction should exhibit better performance when appropriate metrics are chosen for tree construction. Another point of note is that we will no longer consider the greedy tree search (Algorithm 15) because its performance is almost identical to the performance of the lazy-defeatist tree search (Algorithm 16) for the same tree.

Figure 38 presents the time-constrained search performance of the proposed algorithms in terms of the mean rank over all queries for the ℓ_1 (Manhattan), ℓ_2 (Euclidean) and the ℓ_∞ (Chebyshev) metric. For three out of the four datasets, the performance of the proposed methods are almost identical across all three metrics, and algorithms with better relative performance in one metric perform relatively better in other metrics. The defeatist forest search performs the best with lazy-defeatist search with *PA*-tree outperforming lazy-defeatist search with *kd*-tree. The only situation where the performance changes is with the MNIST set and the ℓ_1 metric. The only algorithm affected is defeatist forest search with ball-trees, where the performance is drastically worse than the rest of the algorithms. This is possible when, for several queries, the candidate neighbor has a small rank (low error) with respect to the ℓ_2 metric and a really high rank with respect to the ℓ_1 metric (large error). Since the ball-tree used for this experiment partitions the data with respect to the ℓ_2 metric, the query is usually placed in a leaf that is thought to contain close neighbors with respect to the ℓ_2 metric. And since these leaves are of constant size, it is possible that points which are ranked low with respect to the ℓ_1 metric but ranked high with respect to the ℓ_2 metric are always contained in a different leaf node. This should be



(a) Optdigits ℓ_1 (b) Optdigits ℓ_2 (c) Optdigits ℓ_∞

(d) Tiny Images ℓ_1 (e) Tiny Images ℓ_2 (f) Tiny Images ℓ_∞

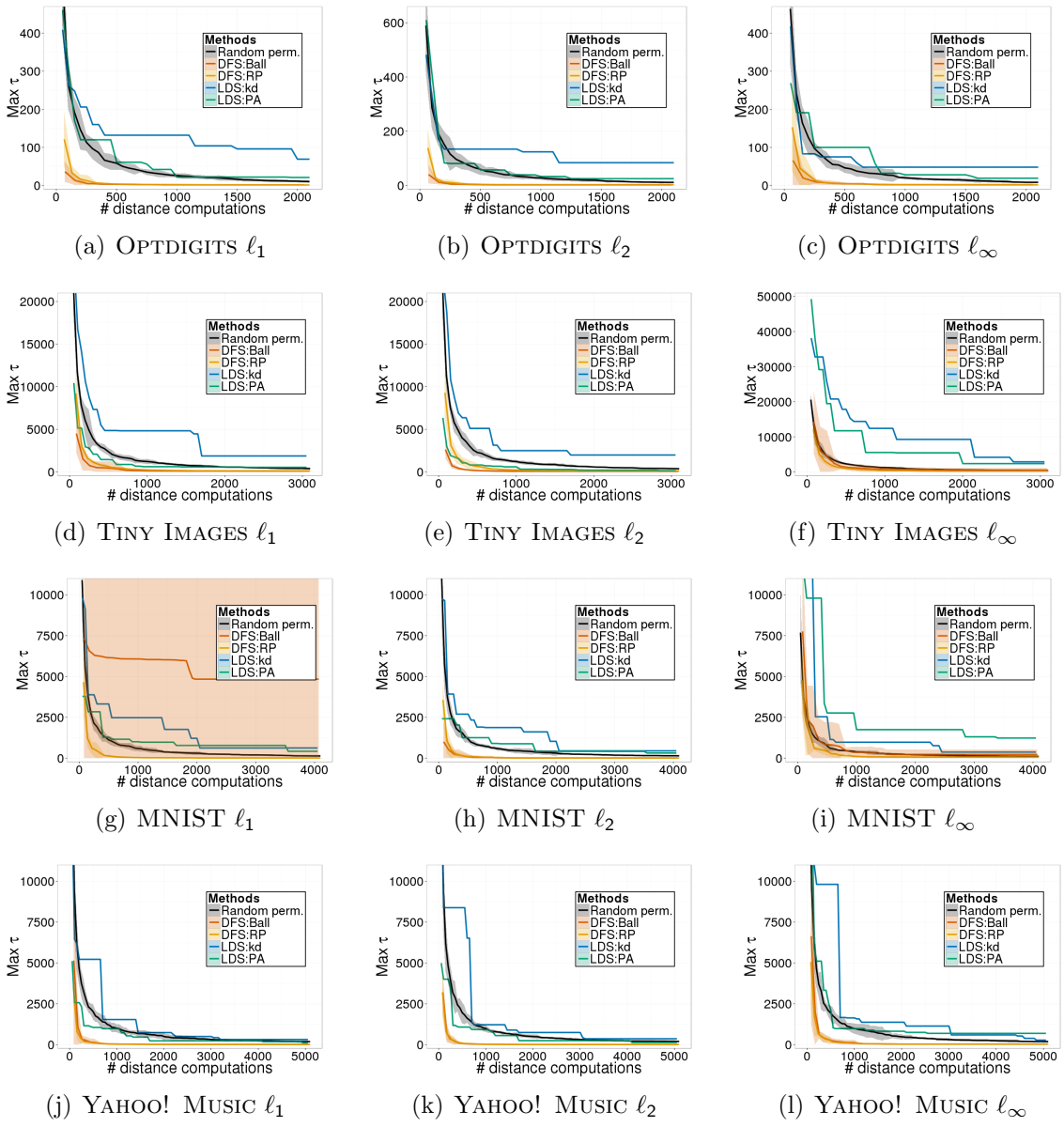
(g) MNIST ℓ_1 (h) MNIST ℓ_2 (i) MNIST ℓ_∞

(j) Yahoo! Music ℓ_1 (k) Yahoo! Music ℓ_2 (l) Yahoo! Music ℓ_∞

Figure 38: Mean rank with respect to different distance metrics with increasing time constraint. Each row corresponds to a different dataset and each column corresponds to a different metric. LDS:*kd* & LDS:*PA* refer to Algorithm 16 with a *kd*-tree and *PA*-tree respectively. DFS:*RP* & DFS:Ball refer to Algorithm 17 with a forest of *RP*-trees and ball-trees respectively (*please view in color*).

easily mitigated by constructing the ball-tree with respect to the ℓ_1 metric instead of the ℓ_2 metric.

The effect of the different metrics is a bit more stark with respect to the maximum



(a) Optdigits ℓ_1 (b) Optdigits ℓ_2 (c) Optdigits ℓ_∞

(d) Tiny Images ℓ_1 (e) Tiny Images ℓ_2 (f) Tiny Images ℓ_∞

(g) MNIST ℓ_1 (h) MNIST ℓ_2 (i) MNIST ℓ_∞

(j) Yahoo! Music ℓ_1 (k) Yahoo! Music ℓ_2 (l) Yahoo! Music ℓ_∞

Figure 39: Maximum rank with respect to different distance metrics with increasing time constraint. Each row corresponds to a different dataset and each column corresponds to a different metric. LDS:*kd* & LDS:*PA* refer to Algorithm 16 with a *kd*-tree and *PA*-tree respectively. DFS:*RP* & DFS:Ball refer to Algorithm 17 with a forest of *RP*-trees and ball-trees respectively (*please view in color*).

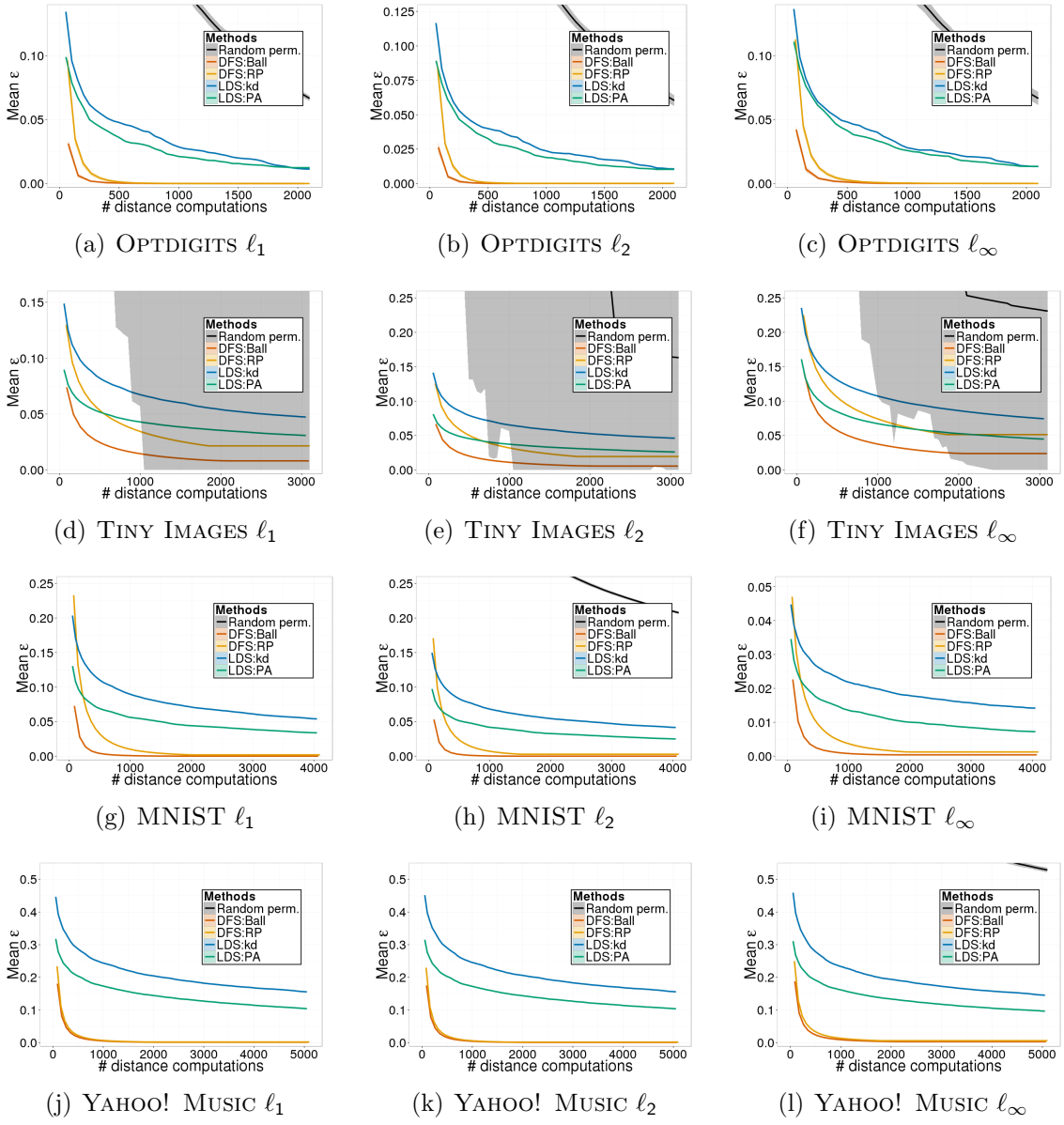
rank shown in Figure 38. For all cases, except the MNIST set with the ℓ_1 metric, defeatist forest search performs the best, and in most cases, lazy-defeatist search with *PA*-tree performs better than lazy-defeatist search with *kd*-tree. Only with respect

to the ℓ_∞ metric does the performance of the *kd*-tree appear comparable or better than the performance of the *PA*-tree with respect to the maximum rank, with the random permutation baseline performing comparably to lazy-defeatist search in two of the cases and outperforming lazy-defeatist search in rest (two) of the cases. The relative performance does not change significantly between the ℓ_1 and ℓ_2 metric.

Figure 40 indicates that the relative performance of all the proposed algorithms with respect to the mean distance error over all queries is identical across all metrics. While the actual values of the mean distance error changes between metrics (seen by the vertical axis in each plot), the relative performances appear consistent. The defeatist forest search performs best with the forest of ball-trees outperforming the forest of *RP*-trees. The lazy-defeatist search outperforms the baseline of random permutations, with lazy-defeatist search with a *PA*-tree exhibiting significantly better performance than the lazy-defeatist search with a *kd*-tree.

The story remains almost the same in Figure 41 with the relative performance of the proposed methods in terms of the maximum distance error over all queries. For two of the datasets (Optdigits and Yahoo! Music), the relative performances appear identical across all the three metrics. For the other two datasets (Tiny Images and MNIST), the relative performances with respect to the maximum distance error are identical between the ℓ_1 and ℓ_2 metrics. Similar to the case with maximum rank (Figure 39), for the ℓ_∞ metric, the lazy-defeatist search with *kd*-tree appears to have comparable (or slightly better) performance than lazy-defeatist search with *PA*-tree with respect to maximum distance error.

The results indicate that for the metrics considered for this experiment (ℓ_1, ℓ_2 and ℓ_∞), the relative performance of the proposed time-constrained search algorithms is usually not affected significantly by the choice of the metric. This might be because the good neighbor candidates with respect to one metric were still good neighbor candidates for the other metrics. However, there are situations where this might not



(a) Optdigits ℓ_1 (b) Optdigits ℓ_2 (c) Optdigits ℓ_∞

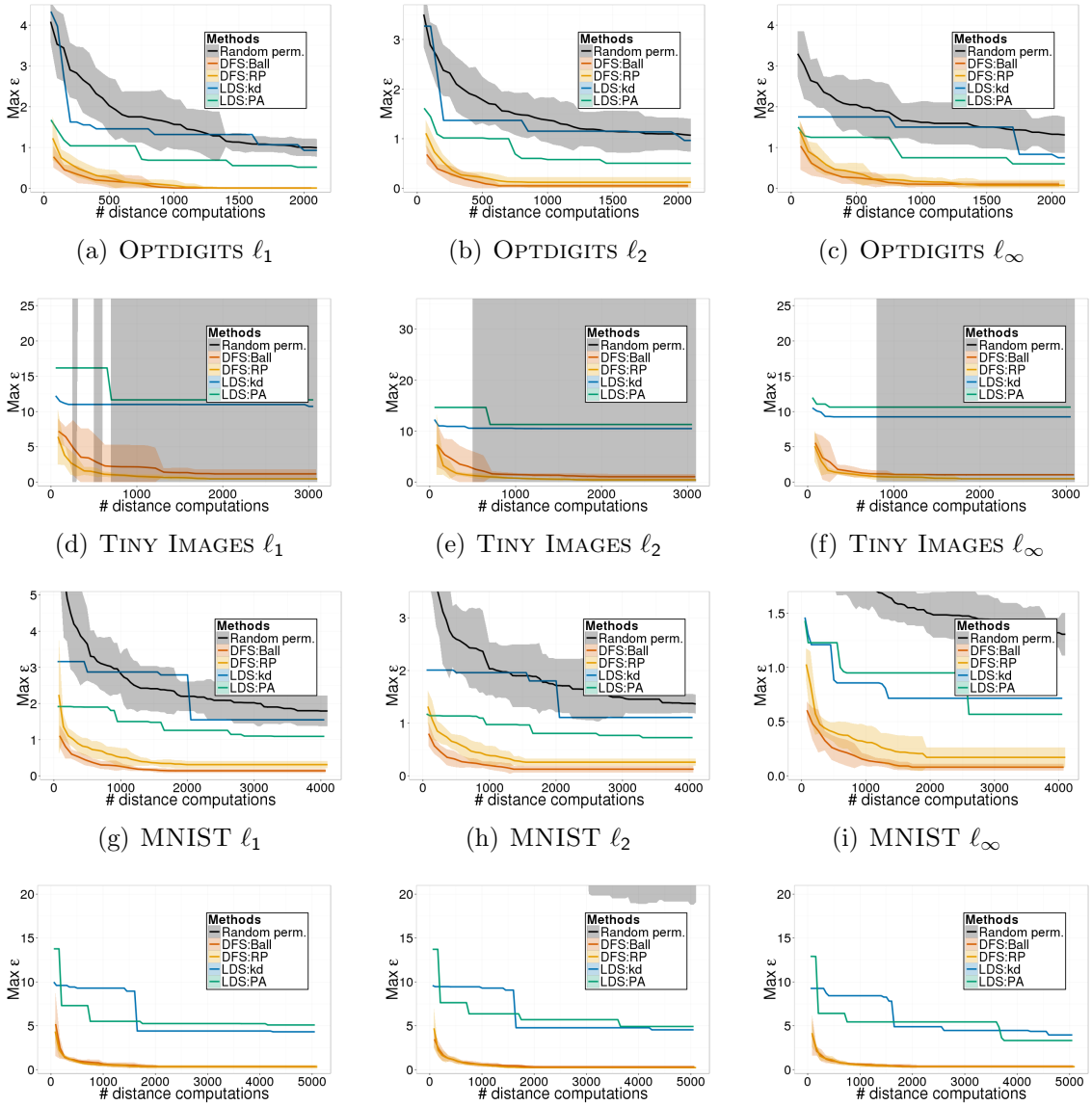
(d) Tiny Images ℓ_1 (e) Tiny Images ℓ_2 (f) Tiny Images ℓ_∞

(g) MNIST ℓ_1 (h) MNIST ℓ_2 (i) MNIST ℓ_∞

(j) Yahoo! Music ℓ_1 (k) Yahoo! Music ℓ_2 (l) Yahoo! Music ℓ_∞

Figure 40: Mean distance error with respect to different distance metrics with increasing time constraint. Each row corresponds to a different dataset and each column corresponds to a different metric. LDS:*kd* & LDS:*PA* refer to Algorithm 16 with a *kd*-tree and *PA*-tree respectively. DFS:*RP* & DFS:Ball refer to Algorithm 17 with a forest of *RP*-trees and ball-trees respectively (*please view in color*).

be true, as can be seen with the MNIST set and ℓ_1 metric for defeatist forest search with ball-trees. However, such a potential problem can be mitigated by considering metric specific indexing schemes (that is, utilize metric-specific tree constructions).



(a) Optdigits ℓ_1 (b) Optdigits ℓ_2 (c) Optdigits ℓ_∞

(d) Tiny Images ℓ_1 (e) Tiny Images ℓ_2 (f) Tiny Images ℓ_∞

(g) MNIST ℓ_1 (h) MNIST ℓ_2 (i) MNIST ℓ_∞

(j) Yahoo! Music ℓ_1 (k) Yahoo! Music ℓ_2 (l) Yahoo! Music ℓ_∞

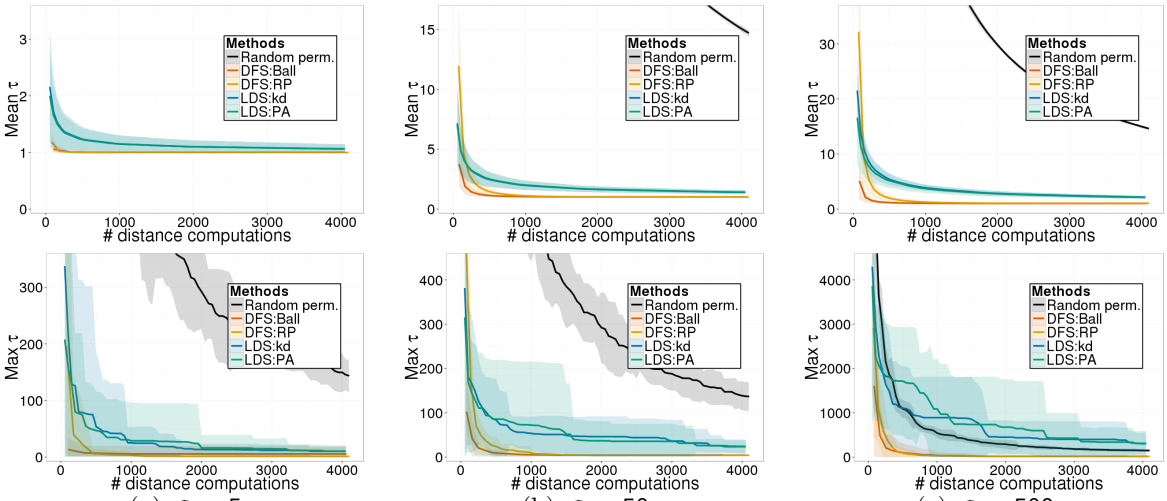
Figure 41: Maximum distance error with respect to different distance metrics with increasing time constraint. Each row corresponds to a different dataset and each column corresponds to a different metric. LDS:*kd* & LDS:*PA* refer to Algorithm 16 with a *kd*-tree and *PA*-tree respectively. DFS:*RP* & DFS:Ball refer to Algorithm 17 with a forest of *RP*-trees and ball-trees respectively (*please view in color*).

4.4.3 Performance Dependency on the Data Dimensionality

In this subsection, we consider the relative performance of the proposed algorithms for time-constrained search for datasets with different dimensionality. One possible source of datasets with varying dimensionality are synthetic datasets. But the behavior of algorithms on synthetic datasets can be very misleading. So I decided to use real datasets for the evaluation. However, just selecting real datasets with different dimensionality would not present a fair image because different real datasets usually have some specific intrinsic structure that drive the search performance on those datasets. For this reason, I considered a single real dataset, the 784-dimensional MNIST set (this is generally considered to be a hard dataset), to ensure that datasets with different dimensionality approximately have the same intrinsic structure. This would allow me to study how the relative performance of the proposed algorithm is affected just by the dimensionality.

The next piece is to create "real looking" datasets of varying dimensionality from this chosen dataset. I considered the eigenvectors of the data covariance matrix. To create a d-dimensional "real looking" dataset, I randomly choose d of these eigenvectors and project the original dataset and the queries along these d eigenvectors. The hope is that this technique allows me to vary dimensionality of a dataset while still retaining most of the intrinsic structure present in the original real dataset. Then I evaluate my proposed algorithms on this synthetic instantiation of a real dataset. For any chosen dimensionality, I create 10 random synthetic instantiations of the MNIST dataset and average the performance of the proposed algorithms over all 10 instantiations.

The relative performance of the proposed algorithms in terms of the candidate neighbor rank for datasets with varying dimensionality ($\mathcal{D} = 5, 50$ & 500) is presented in Figure 42. The defeatist forest search appears to perform best, with the forest

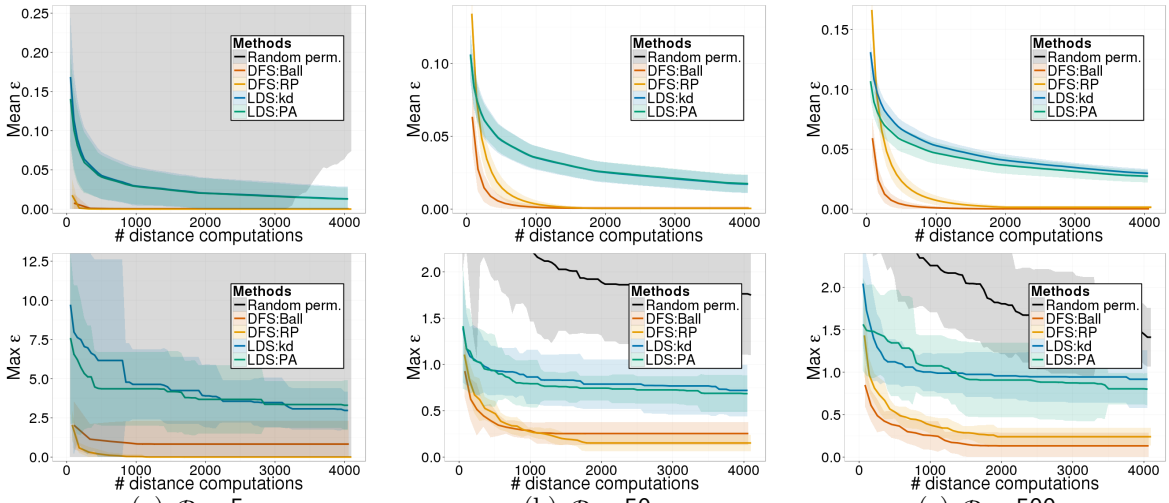


(a) $\mathcal{D} = 5$ (b) $\mathcal{D} = 50$ (c) $\mathcal{D} = 500$

Figure 42: Rank with increasing time constraint for datasets with varying dimensionality. The top row corresponds to the mean rank over all queries and the bottom row corresponds to the maximum rank over all queries. Each column corresponds to a particular dimensionality. LDS:*kd* & LDS:*PA* refer to Algorithm 16 with a *kd*-tree and *PA*-tree respectively. DFS:*RP* & DFS:Ball refers to Algorithm 17 with a forest of *RP*-trees and ball-trees respectively (*please view in color*).

with ball-trees slightly outperforming the forest with *RP*-trees on average. The lazy-defeatist tree search performs second best with both the *PA*-tree and the *kd*-tree having comparable performance to each other. But improvement of the forest search over the tree search increases with increasing dimensionality (as can be seen by the scales on the vertical axis of the plots in Figure 42). An interesting observation is that the relative performance of the random permutation baseline with respect to the maximum rank improves with increasing dimensionality (this improvement is less apparent in case of the mean rank). The performance of the baseline is independent of the dimensionality of the dataset. Hence this observation suggests that increasing dimensionality worsens the performance of the proposed algorithms and brings it closer to the performance of the baseline, as one would usually expect.

The relative performance of the proposed methods with respect to the distance error, presented in Figure 43, tell the same story, with the defeatist forest search



(a) $\mathcal{D} = 5$ (b) $\mathcal{D} = 50$ (c) $\mathcal{D} = 500$

Figure 43: Distance error with increasing time constraint for datasets with varying dimensionality. The top row corresponds to the mean distance error over all queries and the bottom row corresponds to the maximum distance error over all queries. Each column corresponds to a particular dimensionality. LDS:*kd* & LDS:*PA* refer to Algorithm 16 with a *kd*-tree and *PA*-tree respectively. DFS:*RP* & DFS:Ball refers to Algorithm 17 with a forest of *RP*-trees and ball-trees respectively (*please view in color*).

performing the best. The lazy-defeatist tree search outperforms the random permutations baseline by a significant margin, with *PA*-tree and *kd*-tree performing comparably to each other. Defeatist forest search with ball-tree appears to perform significantly better than the defeatist forest search with *RP*-trees in most of the cases (maximum distance error at $\mathcal{D} = 5$ being an exception where the *RP*-trees outperform the ball-trees).

The comparable performance of the *PA*-tree and the *kd*-tree can be attributed to the following fact: The *PA*-tree builds tree by splitting the data along the eigenvectors of the data covariance matrix. A dataset of any desired dimensionality is created by projecting the data along the eigenvectors of the original data covariance. So the eigenvectors of the covariance matrix of this projected dataset are axis-aligned. Hence, the axis-aligned splits of the *kd*-tree have the same effect as the eigenvector

aligned splits of the *PA*-tree, resulting in similar performance for time-constrained search. But when the eigenvectors of the data covariance are not axis aligned (as is the case with usual real datasets), the *PA*-tree has significantly better performance than the *kd*-tree (as shown in the experiments in the previous subsections).

4.4.4 Lessons Learned

These empirical results indicate that in the situation where space (memory) is not an issue, the defeatist forest search is the method of choice for time-constrained search because they have consistently had the best performance with respect to all forms of nearest-neighbor error (mean/maximum rank/distance error) across all datasets and experimental conditions. Between the two forms of forests (one with *RP*-trees and other with ball-trees), ball-trees appear to outperform *RP*-trees empirically in most cases. Another advantage of ball-trees is that they have a metric specific construction, making them more adaptable to the problem at hand.

If only enough space is available for a single tree data structure, lazy-defeatist tree search performs quite well for time-constrained search with respect to the random permutation baseline. Obviously, the performance is significantly affected by the choice of the space partitioning tree. This will studied in detail (both theoretically and empirically) in Chapter 5 Section 5.2, where we try to understand the precise intrinsic properties of the data and the tree which affect the search performance.

If no space or time is available for indexing/preprocessing the data, the random permutation based time-constrained search appears to be the best option at hand. Moreover, if the notion of error that is crucial is the maximum rank, then random permutation based time-constrained search has comparable performance to the tree based time-constrained search methods.

CHAPTER V

LEARNING TO SEARCH

In the time-constrained setting, we wish to preprocess the data in such a way that the better neighbor candidates for a query are encountered earlier during the search process. In trying to achieve this, we have explored certain heuristic permutations such as the Gonzalez and the k-means++ and heuristic binary trees such as the *kd*-tree and *PA*-tree. These heuristics are intuitive and provide significant improvements in some cases. However, it is usually not clear as to what structure in the data (or combination of structures in the data) and what adaptability of the indexing scheme determine the performance of a time-constrained search algorithm.

The task of classification in machine learning involves labelling objects into relevant categories. The usual way to solve this task is with a set of training examples – example objects with known categories. These training examples coupled with some heuristic (such as a choice of the model, regularizer of the model, some structural assumption) are then used to "learn" a classifier. In case of time-constrained search, a similar approach can be utilized where some "training queries" are used to preprocess the data in a way that the good candidate neighbors of future queries are encountered very early during the search algorithm. This technique has been used to learn skip-lists (Clarkson, 1999), *kd*-trees (Maneewongvatana and Mount, 2002) and bucketing schemes in locality sensitive hashing (Cayton and Dasgupta, 2007). We will use a similar technique to learn a permutation for time-constrained search in the Section 5.1 of this chapter.

It is also essential to understand the structural properties of the data which influence time-constrained search performance. Understanding these structural properties

can allow us to do better preprocessing by explicitly optimizing on these properties. We will find two such structural properties of the data that influence the time-constrained search performance of lazy-defeatist search with binary trees (Algorithm 16) in the Section 5.2 of this chapter, and utilize these factors to motivate a new binary space-partitioning tree, the max-margin tree, for improved search performance. In Section 5.3, we present some theoretical properties of the max-margin tree and its efficient variant. Finally we will empirically evaluate the advantage of learning in Section 5.4 of this chapter.

5.1 Learning Permutations for Improved Performance

As mentioned earlier, instead of preprocessing the data with respect to some heuristic, we can use a set of "training queries", queries which serve as an exemplar for the future queries, to preprocess the data in a way which minimizes the search error in the time-constrained setting. There are two notions of search error, distance error and rank error, and in each of the situation, we may care about the average case or the worst case performance. Depending on our application, we can choose the type of error and the aggregate performance over all the queries.

Algorithm 18 presents a simple way to greedily learn a permutation of the data using a set of training queries Q given an error matrix E, where the $(j, i)^{th}$-entry $E[j, i]$ corresponds to the search error incurred if the point r_i is returned as the neighbor candidate for query q_j. This error matrix can correspond to the distance error or the rank error. In lines 3 and 10, the current error of each of the training queries is aggregated using an aggregation function. This function can be the sum (or mean) or the maximum over all the queries.

The algorithm begins by selecting the point in the dataset R that incurs the lowest error aggregated over all training queries. Once a point is put into the permutation, the effective error for any other point in the set is the minimum of the error due

Algorithm 18 Learning a permutation with "training" queries.

Input: Data R, Training set Q, Error matrix $E \in \mathbb{R}^{|Q| \times |R|}$
Output: Permutation T of $[1 : n]$

```
3:  T[1] ← arg min_{i∈[1:n]} aggregate(E[1, i], ..., E[m, i])
    for j ← [1 : m] do
      for i ← [1 : n] \ {T[1]} do
6:      E[j, i] ← min{E(j, T[1]), E(j, i)}
      end for
    end for
9:  for i ← 2 : n do
      T[i] ← arg min_{i∈[1:n]\T[1:i−1]} aggregate(E[1, i], ..., E[m, i])
      for j ← [1 : m] do
12:     for i ← [1 : n] \ {T[1 : i − 1]} do
          E[j, i] ← min{E(j, T[i]), E(j, i)}
        end for
15:   end for
    end for
```

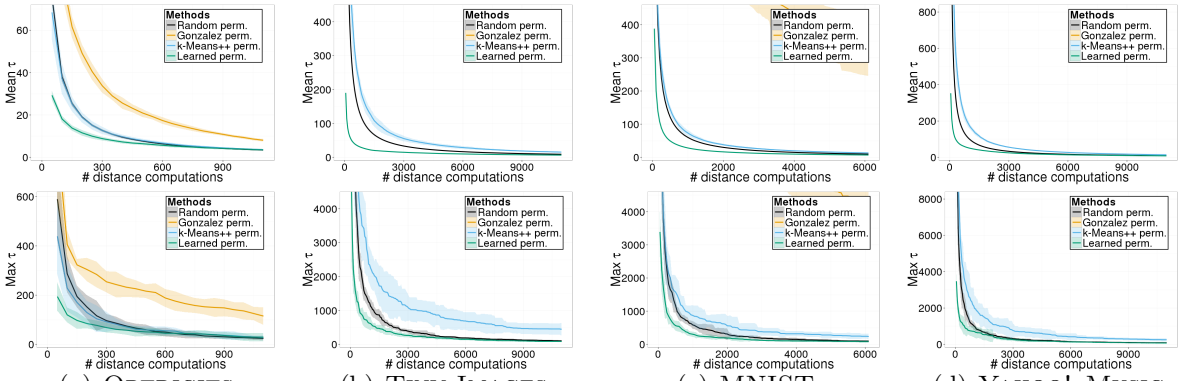
to the selected point and the error incurred by that point. This is because when the search procedure goes through the final permutation, the best candidate from the encountered set of points is returned as the neighbor candidate (Algorithm 11). Hence, after each selection of a point for the learned permutation, the error matrix is accordingly updated. This is done in Lines 4-8 and Lines 11-15 for Algorithm 18.

5.1.1 Empirical Evaluation and Discussion

We continue with the 4 datasets used in the previous chapter. We create the trained permutation using Algorithm 18. The training queries Q are obtained by the randomly selecting 1% of the dataset R. The error-matrix E corresponds to the rank and the "aggregate" function gives the mean rank over all queries. Similar to the previous empirical evaluations of the time-constraint algorithms, the nearest-neighbor error is noted for increasing time-constraints. In our training procedure, there is randomness in the way the training queries are selected. So the experiment is performed 10 times and the results are averaged over the 10 runs. The learned permutation is compared to the previously proposed permutation schemes for time-constrained search (TCS)

in Section 4.2 of Chapter 4.

Figures 44 & 45 present the relative performance of the permutation based TCS in terms of the rank and distance error respectively. The top row in each figure corresponds to the mean error and the bottom row corresponds to the maximum error. Note that both the figures correspond to the permutation trained with respect to the mean rank over all training queries. The same experiment can be performed for the rest of the combinations of error, maximum rank, mean distance-error and maximum distance-error, but is not presented here.

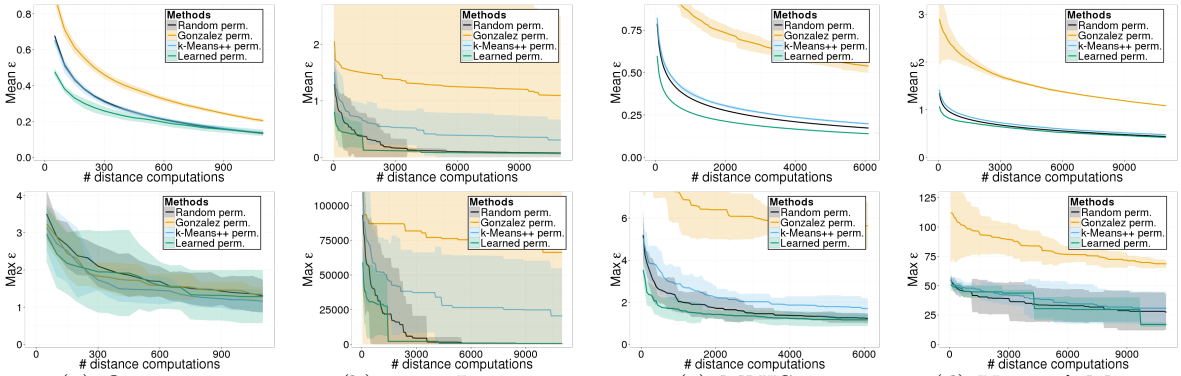


(a) Optdigits (b) Tiny Images (c) MNIST (d) Yahoo! Music

Figure 44: Rank with increasing time constraint. The top row corresponds to the mean rank over all queries and the bottom row corresponds to the maximum rank over all queries (*please view in color*).

The results in the top row of Figure 44 indicate that the performance of the permutation learned with some training queries (as low as 1% of the dataset) has significantly better performance than the other permutation schemes with respect to the mean rank. The improvement is prominent for smaller time-constraints (smaller values on the horizontal axis). This is expected since we explicitly learned a permutation that minimizes the mean rank over all training queries, and the actual queries are not that different from the training queries (this is effectively the basic assumption of machine learning).

While the permutation is learned to (greedily) minimize the mean rank, the results in the bottom row of Figure 44 indicate that this learned permutation also outperforms the heuristic permutation schemes with respect to the maximum rank. The improvements are again more prominent for smaller time-constraints. The hope is that in the scenario where we explicitly train with respect to the maximum rank (i.e., the aggregate function returns the maximum rank over all queries), the performance with respect to the maximum rank should be further improved for the trained permutation.



(a) Optdigits (b) Tiny Images (c) MNIST (d) Yahoo! Music

Figure 45: Distance error with increasing time constraint. The top row corresponds to the mean distance-error over all queries and the bottom row corresponds to the maximum distance-error over all queries (*please view in color*).

The relative performances of the permutation schemes with respect to the mean distance error are shown in the top row of Figure 45. The results indicate that the permutation trained with respect to the mean rank performs better than the heuristic permutation schemes even in terms of the mean distance error, even though the difference is less apparent. The improvement of the trained permutation is again less apparent in the results for the maximum distance error in the bottom row of Figure 45. The trained permutation slightly outperforms the rest of the permutation schemes in the MNIST and Tiny Images datasets. For the rest of the datasets, the

random permutation and the k-means++ permutation perform comparably to the trained permutation.

It should be noted that the permutations was learned with respect to the mean rank. Depending on the application, an appropriate notion of error and an aggregate function can be used to train the permutation. In that situation, the trained performance should outperform the heuristic based permutations, as indicated by the results with respect to the mean rank error (the top row in Figure 44).

5.2 Analysis of Lazy-Defeatist Tree Search

A simplified version of the lazy-defeatist binary tree-based time-constrained search (Algorithm 16) is shown in Algorithm 19. Here the time-constraint t is replaced by the depth l to which the tree will be traversed, where $t = l + (n/2^l)$. For a balanced binary tree on a dataset R of n points with leaf nodes containing a single point, the depth l can be chosen for the given time-constraint t as follows: for $t \leq \log_2 n + 1$, $l = \log_2 n - 1$; for $t \in (\log_2 n + 1, \log_2 n + 2]$, $l = \log_2 n - 2$; for $t \in (\log_2 n + 2, \log_2 n + 2^3 - 3]$, $l = \log_2 n - 3$; for $t \in (\log_2 n + 2^3 - 3, \log_2 n + 2^4 - 4]$, $l = \log_2 n - 4$ as so on.

Algorithm 19 Binary tree based candidate selection: Lazy-defeatist tree search

Input: Dataset R, Binary tree T, Query q and Desired depth l
Output: Candidate neighbor p for q

Initialize: current depth $l_c \leftarrow 0$, current node $T_c \leftarrow T$
while $l_c < l$ **do**
  if $\langle T_c.\mathbf{w}, q \rangle + T_c.b \leq 0$ **then**
    $T_c \leftarrow T_c$.left_child
  else
    $T_c \leftarrow T_c$.right_child
  end if
  $l_c \leftarrow l_c + 1$
end while
$C_q \leftarrow T_c \cap S$
$p \leftarrow \arg\min_{r \in C_q} \|q - r\|$.

We will be building on the existing results (Verma et al., 2009) presented in Section 2.3 in Chapter 2 to present the worst case error guarantee for Algorithm 19.

The following are definitions I will use to state the main result of this section:

Definition 5.2.1 (Quantization error improvement)**.** *Given a set of points (vectors) $S \subset \mathbb{R}^{\mathcal{D}}$, a region A split into two disjoint regions $\{A_l, A_r\}$, and a quantity β depending on the data in the region A, the quantization error improvement is characterized by:*

$$\mathcal{V}_S(\{A_l, A_r\}) < (1 - 1/\beta)\, \mathcal{V}_S(A). \tag{15}$$

where $\mathcal{V}_S(A)$ is the quantization error at node A defined as

$$\mathcal{V}_S(A) = \frac{1}{|A \cap S|} \sum_{x \in A \cap S} \|x - \mu(A)\|_2^2, \tag{16}$$

and $\mathcal{V}_S(\{A_l, A_r\})$ the average quantization error of a collection of regions $\{A_1, A_2\}$ given by:

$$\mathcal{V}_S(\{A_1, A_2\}) = \frac{|A_1 \cap S|\mathcal{V}_S(A_1) + |A_2 \cap S|\mathcal{V}_S(A_2)}{|A \cap S|}. \tag{17}$$

Definition 5.2.2 (ω-balanced split)**.** *A split partitioning any region A into disjoint regions A_1 and A_2 is called an ω-balanced split if*

$$||A_1 \cap S| - |A_2 \cap S|| \leq \omega |A \cap S|. \tag{18}$$

For an entirely balanced tree corresponding to recursive median splits, such as the *PA*-tree and *kd*-tree, $\omega \approx 0$. Non-zero values of $\omega \ll 1$ corresponds to approximately balanced trees. Approximately balanced trees have a depth bound of $O(\log n)$ (Ram et al., 2012, Theorem 3.1). For a tree with ω-balanced splits, the worst case runtime of Algorithm 19 is $O\left(l + \left(\frac{1+\omega}{2}\right)^l n\right)$ instead of $O\left(l + \frac{n}{2^l}\right)$.

Definition 5.2.3 (Expansion constant)**.** *Let $\mathcal{B}_{\ell_2}(p, \Delta) = \{r \in S \colon \|p - r\| < \Delta\}$ denote the points in S contained in a ball of radius Δ around some $p \in S$ with respect to the ℓ_2 metric. Then, the expansion constant of (S, ℓ_2) is defined as the smallest $c \geq 2$ such $\left|\mathcal{B}_{\ell_2}(p, 2\Delta)\right| \leq c \left|\mathcal{B}_{\ell_2}(p, \Delta)\right|$ $\forall p \in S$ and $\forall \Delta > 0$.*

The expansion constant is the maximum ratio of the measure in a ball centered at a point to the measure in the ball centered at the same point with half the radius. A bounded expansion constant corresponds to a growth restricted metric (Karger and Ruhl, 2002). The expansion constant effectively characterizes the distribution of the data. It is closely related to the doubling dimension or the Assouad dimension but is more fragile. The expansion constant $c \sim 2^{O(d)}$ where d is the doubling dimension of the set S with respect to the ℓ_2 metric. The relationship is exact for uniformly distributed points in $\mathcal{D}$-dimensional space (i.e., $c = 2^{\mathcal{D}}$).

Given these definitions, I present the following guarantee for Algorithm 19:

Theorem 5.2.1 (Candidate-neighbor distance bound)**.** *Consider a dataset $R \subset \mathbb{R}^{\mathcal{D}}$ of n points with $\psi = \frac{1}{2n^2}\sum_{x,y\in R}\|x-y\|^2$, a query $q \in \mathbb{R}^{\mathcal{D}}$ and the BSP-tree T built on R with the following conditions:*

(C1) For any $\mathcal{D}$-dimensional convex region $A \subset \mathbb{R}^{\mathcal{D}}$, let $(A \cap (R \cup \{q\}), \ell_2)$ have an expansion constant at most $\tilde{c}$.

(C2) Let T be complete till a depth $L < \left(\log_2 \frac{n}{\tilde{c}}\right)/(1-\log_2(1-\omega))$ with ω-balanced splits throughout the tree.

(C3) Let β^* correspond to the worst quantization error improvement over all splits in T.

Further assume the following for any node A in the tree T:

(A1) $\max_{x,y\in A\cap R}\|x-y\|^2 \leq \eta\mathcal{V}_R(A)$ for a fixed $\eta > 8$.

Then the worst-case upper bound d^u on the distance of q to the neighbor candidate p returned by Algorithm 19 with depth $l \leq L$ is given by

$$\|q-p\| \leq d^u = \frac{2\sqrt{\eta\psi}}{n^{\frac{1}{\log_2 \tilde{c}}} - 2\left(\frac{2}{1-\omega}\right)^{\frac{l}{\log_2 \tilde{c}}}} \left(\frac{2}{1-\omega}\right)^{l\left(\frac{1}{2}+\frac{1}{\log_2 \tilde{c}}\right)} \exp\left(-\frac{l}{2\beta^*}\right). \qquad (19)$$

The theorem is proved in Section 5.2.2. This result implies that for a balanced tree ($\omega = 0$) and a query q landing in a node A at depth l of the tree, the maximum distance between q and its closest point in $A \cap R$ is at most

$$\frac{2\sqrt{\eta\psi}}{n^{\frac{1}{\log_2 \tilde{c}}} - 2^{1+\frac{l}{\log_2 \tilde{c}}}} 2^{l\left(\frac{1}{2}+\frac{1}{\log_2 \tilde{c}}\right)} \exp\left(-\frac{l}{2\beta^*}\right).$$

This implies that between two types of BSP-trees on the same set and the same query, Algorithm 19 has a better (worst-case) guarantee on the candidate neighbor distance for the tree with better quantization performance (corresponding to smaller β^*). The last exponential term in definition of d^u (Equation (19)) implies a smaller upper bound for lower values of β^*.

Moreover, for the same tree, d^u is non-decreasing in l. This is as expected because as we go deeper down the tree, we can never improve the candidate neighbor distance. At the root level ($l = 0$), the candidate neighbor distance is equal to the nearest-neighbor distance. As we descend down the tree, the candidate neighbor distance will worsen if a tree split separates the query from its closer neighbors. This behavior is implied by the above bound.

A less apparent dependence is on the expansion constant. For the set of size n with a node at depth l in a BSP-tree, the candidate neighbor distance is inversely proportional to $\left(\frac{n}{2^l}\right)^{1/\log_2 \tilde{c}}$. This implies that larger $\tilde{c}$ lead to worse upper bounds on the candidate neighbor distance. Since $\log_2 \tilde{c} \sim O(d)$, larger intrinsic dimensionality implies worse guarantees, as one would expect from the curse of dimensionality.

The upper bound in Equation (19) will have similar dependence on all the properties of the data and the tree (β^*, l & $\tilde{c}$) for trees with ω-balanced splits for a non-zero but fixed ω.

Explanation of the conditions and assumptions in Theorem 5.2.1. Condition *C1* implies that the data lying within *any* convex subset A of $\mathbb{R}^D$ has an expansion constant at most $\tilde{c}$ with respect to the ℓ_2 metric. This condition implies that the whole

set (R, ℓ_2) also has an expansion constant at most $\tilde{c}$. However, if the expansion constant of the whole set is c, this does not imply that the data lying within any convex subset of $\mathbb{R}^{\mathcal{D}}$ has an expansion constant at most c. Hence a bounded expansion constant assumption for every convex subset of $\mathbb{R}^{\mathcal{D}}$ is stronger than a bounded expansion constant assumption on the whole set. We do not yet have a way to remove this condition. However, note that condition *C1* does not imply that all subsets of R have an expansion constant at most $\tilde{c}$ with respect to ℓ_2, which is a stronger condition.

The condition *C2* ensures that the tree is complete so that for every query q and a depth $l \leq L$, there exists a node in the tree at depth l which contains q. The condition *C3* gives us the worst quantization improvement rate over all the splits in the tree. The worst case quantization performance allows us to give the worst-case search performance.

Assumption *A1* implies that the squared distance between any pair of points in a region A is within a constant factor η of the quantization error of that node. This refers to the assumption that the node A contains no outliers as described in Section 2.3. This assumption ensures that only 'split-by-hyperplane' (Figure 8(a)) was invoked to construct the tree and 'split-by-distance' (Figure 8(b)) was never used during the construction of the tree T. So by making assumption *A1* we are effectively ruling out any split-by-distances. If we do allow a small fraction of the l splits encountered by the query to be split-by-distance, then a similar result can be obtained by using Proposition 2.3.1. The differences would be the following – (1) the dependence on the ψ term will be replaced by the dependence on $\Delta^2(R) = \max_{x,y \in R} \|x - y\|^2$, and (2) the dependence on l would change.

For two trees with different kinds of hyperplane splits, if split-by-distance is invoked the same number of times in the tree, the difference in the worst-case bounds for both the trees would again be governed by their worst-case quantization performance (β^*). However, for a fixed η, a harder question is whether one type of

split-by-hyperplane forces more split-by-distance invocations than another type of split-by-hyperplane. And we do not yet have an answer for this.

Equipped with this bound on the candidate neighbor distance, we can easily bound the worst-case nearest-neighbor errors as follows:

Corollary 5.2.1 (Distance error guarantee)**.** *Under the conditions, definitions and assumptions of Theorem 5.2.1, the distance error* $\epsilon(q)$ *for any query* q *at a desired depth* $l \leq L$ *in Algorithm 19 is bounded as*

$$\epsilon(q) \leq (d^u/d_q^*) - 1, \tag{20}$$

where $d_q^* = \min_{r \in R} \|q - r\|$.

This follows from the definition of distance error $\epsilon(q)$.

Corollary 5.2.2 (Rank guarantee)**.** *Consider the conditions, definitions and assumptions of Theorem 5.2.1. Then the worst-case rank* $\tau(q)$ *for any query* q *at a desired depth* $l \leq L$ *in Algorithm 19 is bounded as*

$$\tau(q) \leq \tilde{c}^{\lceil \log_2(d^u/d_q^*) \rceil}, \tag{21}$$

where $d_q^* = \min_{r \in R} \|q - r\|$.

Proof. We can see from Definition 5.2.3 and Assumption $A1$ that

$$|\mathcal{B}_{\ell_2}(q, d^u)| \leq \tilde{c}^{\lceil \log_2(d^u/d_q^*) \rceil} |\mathcal{B}_{\ell_2}(q, d_q^*)|.$$

Let $S = \{r \in R\colon \|q - r\| < d^u\}$. By definition, the rank of the candidate neighbor is $|S| + 1$. Also $\mathcal{B}_{\ell_2}(q, d^u)$ contains q and S, and $q \notin S$. Then the number of points contained in $\mathcal{B}_{\ell_2}(q, d^u)$ is $|S| + 1$ which is equal to the rank of the candidate neighbor. Now using the fact that $|\mathcal{B}_{\ell_2}(q, d_q^*)| = |\{q\}| = 1$, we have the rank bound in Equation (21). □

The bounds on both kinds of nearest-neighbor error have direct correspondence to d^u. Hence, from Theorem 5.2.1, the BSP-tree with better quantization performance guarantee allows for better nearest-neighbor performance guarantee. Moreover, for a given tree and a query, the nearest-neighbor error of Algorithm 19 can never decrease with the increasing depth l of the tree since every child node of a BSP-tree is contained within its parent node, and a child node can not have better nearest-neighbor error for a query that the parent. Since d^u is non-decreasing in l, our bounds in Corollaries 5.2.1 & 5.2.2 are nondecreasing with increasing depth. Note that any dependence of this approximation guarantee on the ambient or intrinsic dimensionality of the data is subsumed in the dependence on β^* and the region-wise expansion constant $\tilde{c}$.

While our result bounds the worst-case performance of Algorithm 19, a bound on average case performance of the algorithm can be given by

$$\mathop{\mathbb{E}}_{q} \epsilon(q) \leq d^u \left(\mathop{\mathbb{E}}_{q} 1/d_q^* \right) - 1, \qquad \mathop{\mathbb{E}}_{q} \tau(q) \leq c^{\lceil \log_2 d^u \rceil} \left(\mathop{\mathbb{E}}_{q} c^{-\left(\log_2 d_q^*\right)} \right), \quad (22)$$

since d^u does not depend on q and the expectation is over the queries q. For the purposes of relative comparison among binary trees, the bounds on the expected error for any tree depends solely on d^u since the term within the expectation over q does not depend on the tree.

The error bounds in Equations (20) and (21) also depend on the true nearest-neighbor distance d_q^* of any query q of which we have no prior knowledge. However, if we try to learn splits (decision boundaries) such that it ensures a large margin between the two regions, then we can say, with some confidence, that either the candidate neighbor is the true nearest-neighbor of q or that d_q^* is greater than the size of the margin. We characterize the influence of a large margin split on the approximation guarantee with the following result:

Corollary 5.2.3 (Margin-based nearest-neighbor error guarantee)**.** *Consider the conditions, definitions and assumptions of Theorem 5.2.1. Further assume that* γ *is the*

smallest geometric margin size on both sides of any split in the tree. Then the worst-case nearest-neighbor error for q in terms of distance error $\epsilon(q)$ is

$$\epsilon(q) \leq d^u/\gamma - 1, \tag{23}$$

and in terms of rank $\tau(q)$ is

$$\tau(q) \leq c^{\lceil \log_2(d^u/\gamma) \rceil}. \tag{24}$$

Proof. For a margin of size 2γ, if the query q's true nearest-neighbor is not in the node A, then the true nearest-neighbor is at least at a distance greater than γ from q. Hence $d_q^* > \gamma$. Applying this in Corollaries 5.2.1 & 5.2.2 gives the corresponding desired results. □

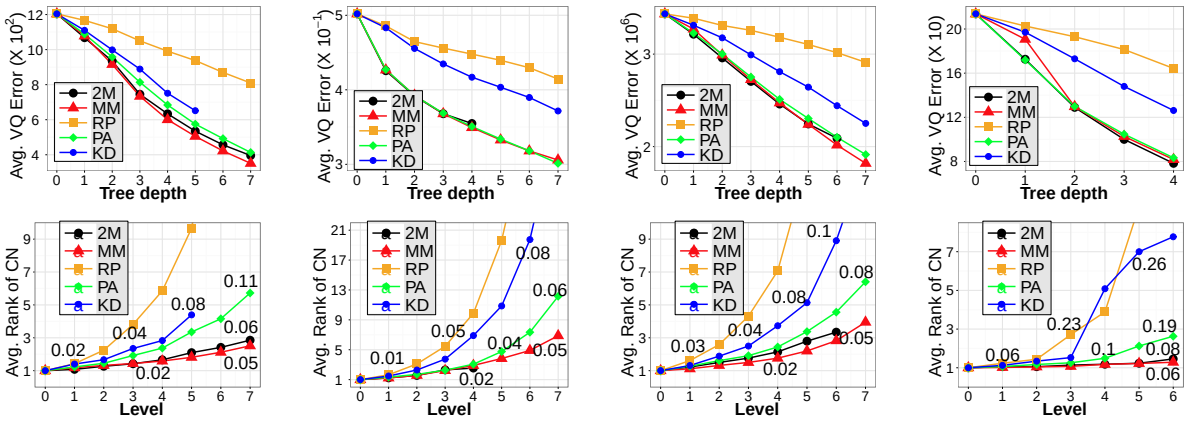
This result indicates that if the margins of the splits in a tree can be increased without adversely affecting its quantization performance, the nearest-neighbor error guarantees will improve for Algorithm 19. While this result also does not show any explicit dependence on the ambient or intrinsic dimensionality of the data, margin sizes can become larger in high dimensions because points are usually more sparse in the high dimensional data space. This motivated us to build a space partitioning tree that explicitly maximizes the margin of the split for every split in the tree. We call this the *max-margin tree* (Ram et al., 2012). The details of its construction are presented in Section 5.3.

5.2.1 Empirical Validation

We examine our theoretical results with subsets of 4 real datasets (described previously in Chapter 3 Section 3.4) – Optdigits ($\mathcal{D} = 64$, $n = 3823$, 1797 queries), Tiny Images ($\mathcal{D} = 384$, $n = 5000$, 1000 queries), MNIST ($\mathcal{D} = 784$, $n = 6000$, 1000 queries) and Images ($\mathcal{D} = 4096$, $n = 500$, 150 queries). We consider the following BSP-trees: *kd*-tree (Algorithm 2), random-projection tree or *RP*-trees (Algorithm 3), principal axis tree or *PA*-tree (Algorithm 4) and the two-means tree or

2M-tree (Algorithm 5). We also consider our proposed BSP-tree, the max-margin tree or *MM*-tree. For the empirical evaluation, we only use split-by-hyperplane for the tree construction. This is because, firstly, the check for the presence of outliers ($\Delta_R^2(A) > \eta \mathcal{V}_R(A)$) is quite expensive to compute, and, secondly, split-by-distance is mostly for the purposes of obtaining theoretical guarantees.

The performance of these BSP-trees on the 4 datasets are presented in Figure 46. Note that the queries are distinct from the points indexed in the tree. Trees with missing data points for higher depth levels (for example, *kd*-tree in Figure 46(a) and *2M*-tree in Figures 46 (b) & (c)) imply that we were unable to grow complete BSP-trees beyond that depth.



(a) Optdigits (b) Tiny Images (c) MNist (d) Images

Figure 46: Performance of binary trees with increasing depth. The top row corresponds to quantization performance of the trees and the bottom row presents the nearest-neighbor error (in terms of mean rank of the candidate neighbor (CN)) of Algorithm 19 with these trees. The nearest-neighbor error plots are also annotated with the mean distance error of the CN (*please view in color*).

The quantization performance of the *2M*-tree, *PA*-tree and *MM*-tree are significantly better than the performance of the *kd*-tree and *RP*-tree and, as suggested by Theorem 5.2.1, this is also reflected in their search performance. The *MM*-tree has comparable quantization performance to the *2M*-tree and *PA*-tree. However,

in the case of search, the *MM*-tree outperforms *PA*-tree in all datasets. This can be attributed to the large margin splits used to build the *MM*-tree. The comparison to *2M*-tree is not as apparent. The *MM*-tree and *PA*-tree have balanced splits enforced algorithmically, resulting in bounded depth and bounded computation of $O(l + n(1+\omega)^l/2^l)$ for any given depth l. No such balance constraint is enforced in the 2-means algorithm, and hence, the *2M*-tree can be heavily unbalanced. The absence of complete *2M*-tree beyond depths 4 and 6 in Figures 46 (b) & (c) respectively is evidence of the lack of balance in the *2M*-tree. This implies possibly more computation and hence lower errors. Under these conditions, the *MM*-tree with an explicit balance constraint performs comparably to the *2M*-tree (slightly outperforming in 3 of the 4 cases) while still maintaining a balanced tree (and hence returning smaller candidate sets on average).

5.2.2 Proof of Theorem 5.2.1

For the proof of Theorem 5.2.1, we utilize the following results:

Fact 5.2.1. *For any set of points S and a region A containing m of the points in S and quantized by the centroid $\mu(A)$ of these points,*

$$\mathcal{V}_S(A) = \frac{1}{2}\frac{1}{m^2}\sum_{x,y\in A\cap S}\|x-y\|^2 . \tag{25}$$

Proof.

$$\begin{aligned}
\frac{1}{m^2}\sum_{x,y\in(A\cap S)}\|x-y\|^2 &= \frac{1}{m^2}\sum_{x,y\in(A\cap S)}\|x-\mu(A)+\mu(A)-y\|^2 \\
&= \frac{1}{m^2}\sum_{x,y\in(A\cap S)}\|x-\mu(A)\|^2+\frac{1}{m^2}\sum_{x,y\in(A\cap S)}\|y-\mu(A)\|^2 \\
&\quad +\frac{2}{m^2}\sum_{x,y\in(A\cap S)}\langle x-\mu(A),\mu(A)-y\rangle \\
&= \frac{2}{m}\sum_{x\in(A\cap S)}\|x-\mu(A)\|^2+\frac{2}{m^2}\sum_{x,y\in(A\cap S)}\langle x-\mu(A),\mu(A)-y\rangle \\
&\quad \text{since}\sum_{x,y\in(A\cap S)}\|x-\mu(A)\|^2 = m\sum_{x\in(A\cap S)}\|x-\mu(A)\|^2 \\
&= \frac{2}{m}\sum_{x\in(A\cap S)}\|x-\mu(A)\|^2+\frac{2}{m^2}\sum_{y\in(A\cap S)}\langle\sum_{x\in(A\cap S)}(x-\mu(A)),\mu(A)-y\rangle \\
&= \frac{2}{m}\sum_{x\in(A\cap S)}\|x-\mu(A)\|^2 = \mathcal{V}_S(A) \quad \text{since}\sum_{x\in(A\cap S)}(x-\mu(A))=0.
\end{aligned}$$

□

Lemma 5.2.1 (Multi-level diameter decrease)**.** *Under the conditions and definitions of Theorem 5.2.1, for any node A at a depth l in the BSP-tree T on R,*

$$\mathcal{V}_R(A) \leq \psi(2/1-\omega)^l exp(-l/\beta^*). \tag{26}$$

Proof. By definition, for any node A at depth l, its sibling node A_s and parent node A_p,

$$\mathcal{V}_R(\{A,A_s\}) \leq \left(1-\frac{1}{\beta^*}\right)\mathcal{V}_R(A_p).$$

For ω-balanced splits, assuming that $|A|<|A_s|$,

$$\mathcal{V}_R(\{A,A_s\}) = \frac{1-\omega}{2}\mathcal{V}_R(A)+\frac{1+\omega}{2}\mathcal{V}_R(A_s) \leq \left(1-\frac{1}{\beta^*}\right)\mathcal{V}_R(A_p).$$

This gives us the following:

$$\frac{\mathcal{V}_R(A)}{\mathcal{V}_R(A_p)} \leq \left(\frac{2}{1-\omega}\right)\frac{\mathcal{V}_R(\{A,A_s\})}{\mathcal{V}_R(A_p)} \leq \left(\frac{2}{1-\omega}\right)\left(1-\frac{1}{\beta^*}\right).$$

The same holds for A_s and for every node at depth l and its corresponding parent at depth $l-1$. Recursing this up to the root node of the tree A_0, we have:

$$\frac{\mathcal{V}_R(A)}{\mathcal{V}_R(A_0)} \leq \left(\frac{2}{1-\omega}\right)^l\left(1-\frac{1}{\beta^*}\right)^l \leq \left(\frac{2}{1-\omega}\right)^l e^{-l/\beta^*}. \tag{27}$$

for any node A at a depth l of the tree. Using Fact 5.2.1, $\mathcal{V}_R(A_0) = \psi$. Substituting this in Equation (27) gives us the statement of Lemma 5.2.1. □

Proof of Theorem 5.2.1. Consider the node A at depth l in the tree where the query q lands, and let m be the number of points of R in the region corresponding to the node A. Let $D = \max_{x,y \in R \cap A} \|x - y\|$, and let $d = \min_{x \in R \cap A} \|q - x\|$. Then, by the Definition 5.2.3 and condition *C1*,

$$\left|\mathcal{B}_{\ell_2}(q, D + d)\right| \leq \tilde{c}^{\log_2 \left\lceil \frac{D+d}{d} \right\rceil} |\mathcal{B}_{\ell_2}(q, d)| = \tilde{c}^{\log_2 \left\lceil \frac{D+d}{d} \right\rceil} \leq \tilde{c}^{\log_2 \left(\frac{D+2d}{d} \right)}.$$

Now $\left|\mathcal{B}_{\ell_2}(q, D + d)\right| \geq m$. Using this above gives us

$$m^{1/\log_2 \tilde{c}} \leq \frac{D}{d} + 2.$$

By condition *C2*, $m^{1/\log_2 \tilde{c}} > 2$. Hence we have

$$d \leq \frac{D}{m^{1/\log_2 \tilde{c}} - 2}.$$

By construction and assumption *A1*, $D \leq \sqrt{\eta \mathcal{V}_R(A)}$. Now $m \geq n(1 - \omega)^l / 2^l$. Plugging this above and utilizing Equation (27) in Lemma 5.2.1 gives us the statement of Theorem 5.2.1. □

5.3 Large Margin Trees

We established that the nearest-neighbor error depends on the quantization performance of the tree as well as the margin of the splits in the tree. This motivated us to consider a binary space partitioning tree (or BSP-tree) which is constructed by explicitly maximizing the margin of every split in the tree (Algorithm 20) while maintaining some form of balance in the tree. We call this the max-margin tree (Figure 47). While the large margin heuristic might be the reason for the favorable empirical performance of the *MM*-tree in Section 5.2, good search performance requires good quantization performance as well. Empirically, the *MM*-tree has comparable

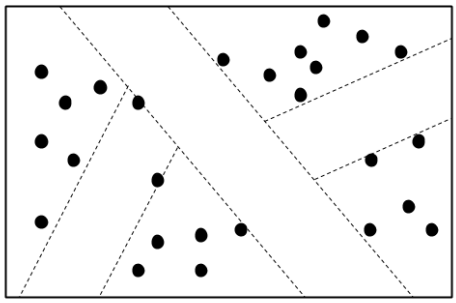


Figure 47: Max-margin tree.

performance to the *2M*-tree and *PA*-tree. In this section, we establish a theoretical guarantee for the quantization performance of the *MM*-tree. Moreover, we discuss efficient ways of utilizing large margin splits in trees and show that, for any chosen direction, utilizing the one-dimensional max-margin split instead of the usual median split does not significantly degrade the quantization performance of the resulting tree, while improving the search performance.

Algorithm 20 A large margin *MM*-tree split.

Input: Set S, Region A
Output: Regions A_l and A_r
$(\mathbf{w}, b) \leftarrow$ a large margin split of $(A \cap S)$
$A_l \leftarrow \{x \in A \cap S : \langle \mathbf{w}, x \rangle + b \leq 0\}$
$A_r \leftarrow \{x \in A \cap S : \langle \mathbf{w}, x \rangle + b > 0\}$

5.3.1 *MM*-tree quantization performance guarantee

The large margin split in the *MM*-tree is obtained by performing max-margin clustering (MMC) with 2 clusters. The task of MMC is to find the optimal hyperplane $(\mathbf{w}^*, b^*)$ from the following optimization problem[1] given a set of points $S = \{x_1, \ldots, x_n\} \subset \mathbb{R}^D$:

$$\min_{\mathbf{w}, b, \xi_i} \frac{1}{2} \|\mathbf{w}\|_2^2 + C \sum_{i=1}^{n} \xi_i \tag{28}$$

[1]This is an equivalent formulation (Zhao et al., 2008) to the original form of max-margin clustering proposed by Xu et al. (2005). The original formulation also contains the labels y_is and optimizes over it. We consider this form of the problem since it makes our analysis easier to follow.

$$s.t. \qquad |\langle \mathbf{w}, x_i \rangle + b| \geq 1 - \xi_i \quad \forall i = 1, \ldots, n \tag{29}$$

$$\xi_i \geq 0 \quad \forall i = 1, \ldots, n \tag{30}$$

$$-\omega n \leq \sum_{i=1}^{n} \operatorname{sgn}\big(\langle \mathbf{w}, x_i \rangle + b\big) \leq \omega n. \tag{31}$$

This formulation finds a soft max-margin split in the data to obtain two clusters. The balance constraint (Equation (31)) avoids trivial solutions and enforces an ω-balanced split. The margin constraints (Equation (29)) enforce a robust separation of the data while allowing for some points to be within the margin. Given a solution to the MMC, we establish the following quantization error improvement rate for the *MM*-tree:

Theorem 5.3.1. *Given a set of points $S \subset \mathbb{R}^D$, consider an ω-balanced max-margin split $(\mathbf{w}, b)$ of the region A into $\{A_l, A_r\}$ with αn support vectors and a split margin of size $\gamma = 1/\|\mathbf{w}\|$. Then the quantization error improvement is given by:*

$$\mathcal{V}_S(\{A_l, A_r\}) \leq \left(1 - \frac{\gamma^2 (1-\alpha)^2 \left(\frac{1-\omega}{1+\omega}\right)}{\sum_{i=1}^{D} \lambda_i}\right) \mathcal{V}_S(A), \tag{32}$$

where $\lambda_1, \ldots, \lambda_D$ are the eigenvalues of the covariance matrix of $A \cap S$.

The result indicates that larger margin sizes (large γ values) and a smaller number of support vectors (small α) implies better quantization performance. The dependence on ω is not significant because ω is generally restricted algorithmically. This result also indicates that if $\gamma = O(\sqrt{\lambda_1})$ then this rate matches the best possible quantization performance of the *PA*-tree (see Table 1). We present the proof of Theorem 5.3.1 in Section 5.3.4. A caveat of this result is that we do assume that we have a feasible solution of the MMC problem, which might not always be possible in practice.

5.3.2 Efficient large margin discriminative trees

Finding the unsupervised maximum-margin split is a non-convex optimization problem and all current algorithms for MMC perform some convex relaxation of the problem

and use convex solvers. Moreover, these convex relaxations are not only approximate (without any formal approximation bounds) but also are computationally expensive to solve in practice. The most efficient algorithm for approximate MMC requires $O(n)$ time to find an unsupervised max-margin split for a set of n points (Zhao et al., 2008)[2].

Hence max-margin splits are very useful, but too computationally expensive to utilize with large datasets. One strategy is to use *kd*-tree splits (median splits along some coordinate axis) for larger nodes in the tree and utilize the max-margin splits only on the small nodes (where this threshold size can be chosen according to the available resources). We call this the *Hybrid-tree* (HY-tree).

Another strategy is the following – (I) first split the data at the median along a direction of our choice (for example, it can be the principal axis like in a *PA*-tree or simply some coordinate axis like in a *kd*-tree), (II) then project the data onto the direction along the line joining the two centroids of the initial partition, (III) finally perform an unsupervised max-margin split of the one-dimensional projections of the points. The one-dimensional MMC is an easy task and can be computed exactly in $O(n \log n)$ by sorting the projections. We call the corresponding trees the *PA*+-tree and the *kd*+-tree. We make use of the following result to quantify the degradation in the quantization error improvement:

Lemma 5.3.1. *Give a set S, for any partition $\{A_1, A_2\}$ of a set A,*

$$\mathcal{V}_S(A) - \mathcal{V}_S(\{A_1, A_2\}) = \frac{|A_1 \cap S||A_2 \cap S|}{|A \cap S|^2} \left\|\mu(A_1) - \mu(A_2)\right\|^2, \tag{33}$$

where $\mu(A)$ is the centroid of the points in the region A.

This result is due to Dasgupta and Freund (2008) and implies that the improvement in the quantization error depends on the distance between the centroids of the

[2]Note that a lot of constants are hidden in the $O(\cdot)$ and we refer the readers to Theorem 4.4 of Zhao et al. (2008).

two regions in the partition. The following result shows that our proposed second strategy for efficient large-margin tree construction will only degrade the quantization performance of the original median-split tree by a very limited amount:

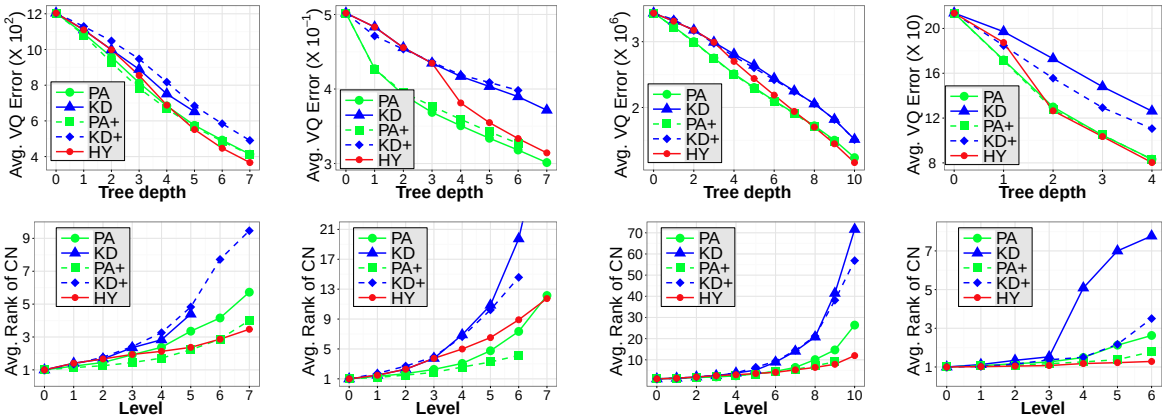
Lemma 5.3.2. *Consider a region A contained in $\mathcal{B}_{\ell_2}(0, \psi) \subset \mathbb{R}^{\mathcal{D}}$ which is split along a chosen direction at the median into $\{A_l, A_r\}$. Let $\mathbf{w}$ denote the unit direction along the line joining the centroids $\mu(A_l)$ and $\mu(A_r)$ of the data in the two regions A_l and A_r. Now consider an ω-balanced split resulting from a max-margin split of the one-dimensional projections of the points in A along the direction $\mathbf{w}$ with a margin of γ and αn support vectors. Let the new partition result in the regions $\{A'_l, A'_r\}$. Then*

$$\|\mu(A'_l) - \mu(A'_r)\| \geq (1 - \omega) \|\mu(A_l) - \mu(A_r)\| - \omega^2 \psi. \tag{34}$$

The above result shows that the centroids of the new partitions resulting from the one-dimensional max-margin split are at least almost as far apart as the centroids of the original split. Since the improvement in the quantization error is directly proportional to the distance between the centroids of the two sides of the partition, Lemma 5.3.2, in conjunction with Lemma 5.3.1, implies that the quantization performance of the new partition is at most only slightly worse than the quantization performance of the original partition. The split balance ω controls the level of degradation in the quantization performance. For the max-margin splits, ω can be directly controlled by the user. The proof of Lemma 5.3.2 is presented in Section 5.3.4.2.

5.3.3 Empirical evaluation

For our empirical validation, we consider a simpler version of our proposed second strategy. We simply perform the one-dimensional max-margin split on the projections of the points on the initially chosen direction (for example, some coordinate axis for *kd*+-trees). This is simpler than the original strategy of projecting the points in the direction of the line joining the two centroids of the initial partition. We again try to validate our theoretical results with the datasets considered in Section 5.2.1.



(a) Opt-digits (b) Tiny Images (c) MNist (d) Images

Figure 48: Performance of large margin trees with increasing depth. The top row corresponds to quantization performance of the trees and the bottom row presents the nearest-neighbor error (in terms of mean rank of the candidate neighbor (CN)) of Algorithm 19 with these trees. Note that the line corresponding to *PA*+-tree almost coincides with the HY-tree for the MNIST set in Figure (c) bottom row (*Please view in color*).

We compare the performance of the discriminative trees with efficient max-margin splits to their non-discriminative original versions in Figure 48. Trees with missing data points for higher depth levels (for example, *kd*-tree in Figure 48(a) and *kd*+-tree in Figures 48 (b)) imply that we were unable to grow complete BSP-trees beyond that depth. The top row shows that, in all cases, the quantization performance of trees with max-margin splits along a chosen direction is not much worse than the quantization performance of the original trees even though we used a simpler version of our strategy. Hence the quantization performance of *PA*+-tree and *kd*+-tree is very close to that of *PA*-tree and *kd*-tree respectively. The HY-tree mirrors the performance of the *kd*-tree for lower depths and does significantly better than the *kd*-tree for higher depths because of the more involved general dimensional max-margin splits.

The bottom row in Figure 48 exhibits the advantage of large margin splits in *PA*-tree and *kd*-tree. The *PA*+-tree outperforms the *PA*-tree in all the considered datasets, with the difference becoming more apparent for higher depths (implying

lower nearest-neighbor error for small amounts of computation). The *kd*+-tree outperforms the *kd*-tree for the high dimensional Images set. The improvement is less significant in the MNIST and Tiny Images set while being slightly worse than the *kd*-tree for the OptDigits set. However, for the OptDigits set, the original *kd*-tree is very unbalanced (even though we used median splits) because of the presence of many points with coinciding dimensions. This unbalanced structure allowed for a complete *kd*-tree of maximum depth 5. The *kd*+-tree performs comparably to the *kd*-tree while being complete and balanced. As expected, the HY-tree mirrors the performance of the *kd*-tree for lower depths, but quickly improves to even outperform the *PA*-tree for higher depths in all datasets. All these results show that in most cases, a large margin split does improve the nearest-neighbor performance without degrading the quantization performance significantly (as predicted by our theoretical results).

5.3.4 Proofs of the theoretical results presented in this section

5.3.4.1 Proof of Theorem 5.3.1

We use these following results to prove Theorem 5.3.1:

Fact 5.3.1. *Given a set S and a region A in $\mathbb{R}^{\mathcal{D}}$, and let $\lambda_1, \ldots, \lambda_{\mathcal{D}}$ be the eigenvalues of the covariance matrix of the points $(A \cap S)$. Then,*

$$\mathcal{V}_S(A) = \sum_{i=1}^{\mathcal{D}} \lambda_i, \tag{35}$$

Proof. Let $\mathrm{Cov}(A \cap S)$ be the covariance matrix of the data contained in a cell A and $\lambda_1, \ldots, \lambda_{\mathcal{D}}$ be the eigenvalues of $\mathrm{Cov}(A \cap S)$. Then, we have:

$$\mathcal{V}_S(A) = \frac{1}{|A \cap S|} \sum_{x \in A \cap S} \|x - \mu(A)\|^2 = \mathrm{tr}\left(\mathrm{Cov}(A \cap S)\right) = \sum_{i=1}^{\mathcal{D}} \lambda_i.$$

□

Proof of Theorem 5.3.1. The solution for the MMC $(\mathbf{w}^*, b^*, \xi_i^*|_{i=1,\ldots,n})$ imply that

$$\sum_{i=1}^{n} |\langle \mathbf{w}^*, x_i \rangle + b^*| \geq n - \sum_{i=1}^{n} \xi_i^*.$$

Let $\tilde{x_i} = \langle \mathbf{w}^*, x_i \rangle + b^*$ and $n_+ = |\{i \colon \tilde{x_i} > 0\}|$ and $n_- = |\{i \colon \tilde{x_i} \leq 0\}|$ and $\tilde{\mu}_+ = (\sum_{i \colon \tilde{x_i} > 0} \tilde{x_i})/n_+$ and $\tilde{\mu}_- = (\sum_{i \colon \tilde{x_i} \leq 0} \tilde{x_i})/n_-$. Then the above equation can be written as:

$$n_+ \tilde{\mu}_+ - n_- \tilde{\mu}_- \geq n - \sum_{i=1}^{n} \xi_i^*.$$

Without loss of generality, we assume that $n_+ \geq n_-$. Then the balance constraint (Equation (31)) tells us that $n_+ \leq n(1+\omega)/2$ and $n_- \geq n(1-\omega)/2$. Using this, the above equation can written as:

$$\tilde{\mu}_+ - \tilde{\mu}_- + \omega(\tilde{\mu}_+ + \tilde{\mu}_-) \geq 2 - \frac{2}{n} \sum_{i=1}^{n} \xi_i^*.$$

Since $\tilde{\mu}_+ > 0$ and $\mu_- \leq 0$, $|\tilde{\mu}_+ + \tilde{\mu}_-| \leq (\tilde{\mu}_+ - \tilde{\mu}_-)$. Hence

$$(1+\omega)(\tilde{\mu}_+ - \tilde{\mu}_-) \geq 2 - \frac{2}{n} \sum_{i=1}^{n} \xi_i^*. \tag{36}$$

For an unsupervised split, the data is always separable since there is no misclassification. This implies that $\xi_i^* \leq 1 \forall i$. Combining this with the above inequality in Equation (36), we get

$$\tilde{\mu}_+ - \tilde{\mu}_- \geq \frac{2 - \frac{2}{n} |i \colon \xi_i^* > 0|}{1+\omega}. \tag{37}$$

The term $|\{i \colon \xi_i^* > 0\}|$ corresponds to the number of support vectors in the final max-margin solution. For the means $\mu(A_l)$ and $\mu(A_r)$, Cauchy-Schwartz gives us:

$$\|\mu(A_l) - \mu(A_r)\| \geq \frac{|\langle \mathbf{w}, \mu(A_l) - \mu(A_r) \rangle|}{\|\mathbf{w}\|} = (\tilde{\mu}_+ - \tilde{\mu}_-)\gamma,$$

where $\tilde{\mu}_- = \langle \mathbf{w}, \mu(A_l) \rangle + b$ and $\tilde{\mu}_+ = \langle \mathbf{w}, \mu(A_r) \rangle + b$. For Equation (37), we can say that

$$\|\mu(A_l) - \mu(A_r)\|^2 \geq 4\gamma^2 \left(\frac{1-\alpha}{1+\omega} \right)^2. \tag{38}$$

Also, for ω-balanced splits, $|A_l||A_r| \geq (1-\omega^2)n^2/4$. Substituting this and Equation (38) in Equation (33) from Lemma 5.3.1, we have

$$\mathcal{V}_S(A) - \mathcal{V}_S(\{A_l, A_r\}) \geq (1-\omega^2)\gamma^2 \left(\frac{1-\alpha}{1+\omega} \right)^2 = \gamma^2 (1-\alpha)^2 \left(\frac{1-\omega}{1+\omega} \right).$$

Reformulating and utilizing Equation (35) from Fact 5.3.1 gives us the statement of the theorem. □

5.3.4.2 Proof of Lemma 5.3.2

Proof of Lemma 5.3.2. Let $\{x_1, \ldots, x_m\}$ be the points in the region A and let $\tilde{x}_i = \langle \mathbf{w}, x_i \rangle$ be the points projected along $\mathbf{w}$. Also, let $\tilde{\mu}(B)$ denote the mean of the points in any region B projected along $\mathbf{w}$. Without loss of generality, assume that the median of $\{\tilde{x}_1, \ldots, \tilde{x}_m\}$ is zero and that $\tilde{\mu}(A_l) < 0$ and $\tilde{\mu}(A_r) > 0$.

Also assume that the max-margin split is at a value $b^* < 0$. Then $|A'_l| = (1 - \omega)m/2$ and $|A'_r| = (1 + \omega)m/2$. Expanding $\tilde{\mu}(A_l)$, we have

$$\begin{aligned} \tilde{\mu}(A_l) &= \frac{\sum_{\tilde{x} \le b^*} \tilde{x} + \sum_{b^* < x \le 0} \tilde{x}}{n/2} \\ &= \tilde{\mu}(A'_l) + \frac{-\tilde{\mu}(A'_l)\frac{n\omega}{2} + \sum_{b^* < \tilde{x} \le 0} \tilde{x}}{n/2}. \end{aligned} \tag{39}$$

Similarly expanding $\tilde{\mu}(A'_r)$, we have

$$\begin{aligned} \tilde{\mu}(A'_r) &= \frac{\sum_{\tilde{x} > 0} \tilde{x} + \sum_{b^* < x \le 0} \tilde{x}}{n(1+\omega)/2} \\ &= \tilde{\mu}(A_r) + \frac{-\tilde{\mu}(A_r)\frac{n\omega}{2} + \sum_{b^* < \tilde{x} \le 0} \tilde{x}}{n(1+\omega)/2}. \end{aligned} \tag{40}$$

Combining Equation (39) and (40) and simplifying, we have

$$\tilde{\mu}(A'_r) - \tilde{\mu}(A'_l) = (\tilde{\mu}(A_r) - \tilde{\mu}(A_l)) + \frac{-\tilde{\mu}(A_r)\frac{n\omega}{2} - \tilde{\mu}(A'_l)\frac{n\omega}{2}(1+\omega) + (2+\omega)\sum_{b^* < \tilde{x} \le 0} \tilde{x}}{n(1+\omega)/2}.$$

Now $b^* \ge \tilde{\mu}(A'_l)$ and $|\{\tilde{x} \colon b^* < \tilde{x} \le 0\}| = n\omega/2$. Hence

$$\begin{aligned} \tilde{\mu}(A'_r) - \tilde{\mu}(A'_l) &\ge (\tilde{\mu}(A_r) - \tilde{\mu}(A_l)) - \frac{\omega(\tilde{\mu}(A_r) - \tilde{\mu}(A'_l))}{1+\omega} \\ &= \left(1 - \frac{\omega}{1+\omega}\right)(\tilde{\mu}(A_r) - \tilde{\mu}(A_l)) - \frac{\omega(\tilde{\mu}(A_l) - \tilde{\mu}(A'_l))}{1+\omega}. \end{aligned} \tag{41}$$

Again utilizing Equation (39), we have

$$\tilde{\mu}(A_l) - \tilde{\mu}(A'_l) \le -\omega\tilde{\mu}(A'_l) \le \omega\Psi \tag{42}$$

since $\tilde{\mu}(A'_l) < 0$ and that $|\tilde{x}| \le \Psi \ \forall x \in A$. By definition of $\mathbf{w}$, $\|\mu(A_l) - \mu(A_r)\| = \tilde{\mu}(A_r) - \tilde{\mu}(A_l)$ and $\|\mu(A'_l) - \mu(A'_r)\| \ge \tilde{\mu}(A'_r) - \tilde{\mu}(A'_l)$. Using this fact and Equation (42) in Equation (42) with the fact that $-\omega/(1+\omega) > -\omega$ for any $\omega \in [0, 1]$ gives us Equation (34) in the statement of the theorem. □

5.4 Empirical Evaluation of "Learned" Binary Trees

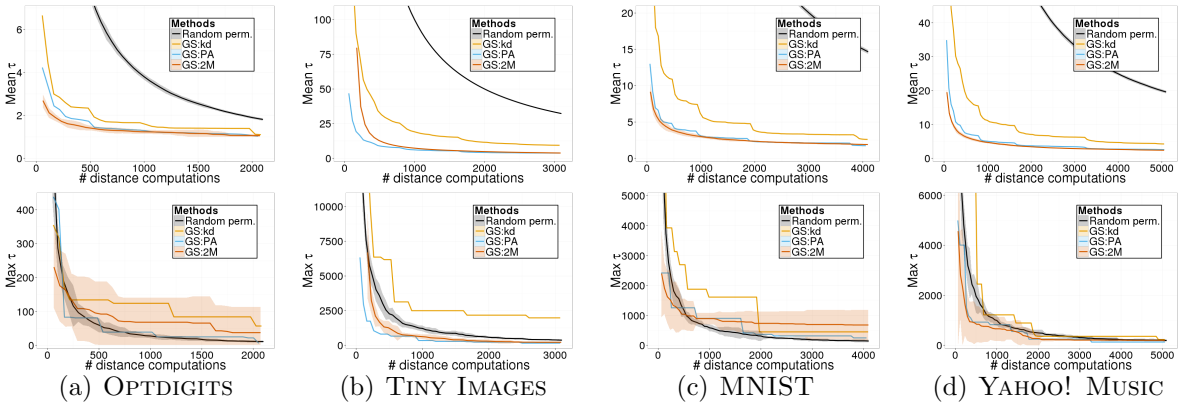
In this section we consider the two trees learned from optimizing the factors affecting search performance. As mentioned earlier, greedily minimizing the quantization error at each level gives us the *2M*-tree, and using MMC to recursively obtain an unsupervised large margin split gives us the *MM*-tree. In this section, we evaluate the *2M*-tree, and the efficient but less accurate version of the *MM*-tree, the HY-tree (the hybrid tree).

5.4.1 Greedy Tree Search

We continue with the 4 datasets previously described in Table 4 in Chapter 4 to evaluate the greedy tree-based time-constrained search (Algorithm 15) with the learned trees. First we compare the performance of the greedy tree search with the *2M*-tree to the performance of time-constrained search (TCS) with random permutations (Algorithm 11) and the greedy tree search with the *kd*-tree and the *PA*-tree. Since we use Lloyd's algorithm (Lloyd, 1982) to solve the 2-means problem, there is randomness in the construction of the *2M*-tree. So we average the performance of the tree over 10 repetitions.

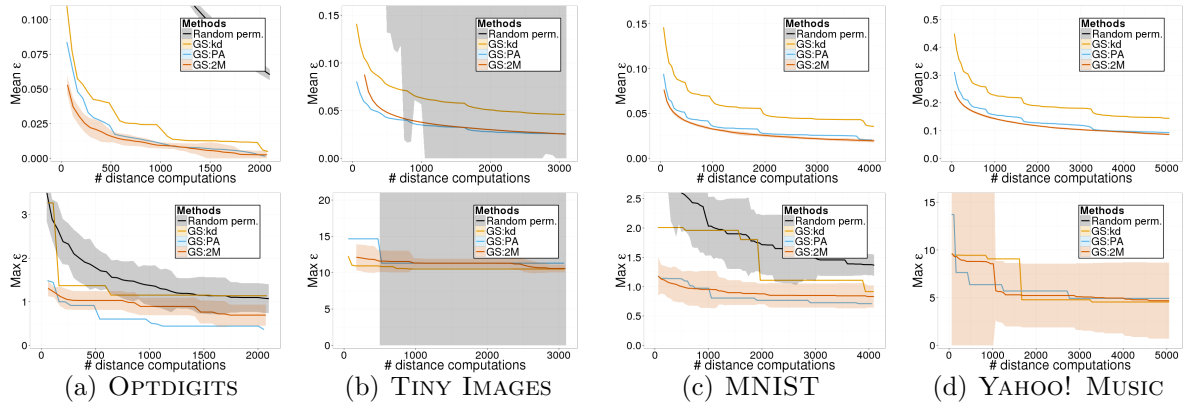
Figure 49 presents the performance of the *2M*-tree with respect to the baseline methods in terms of the rank. The top row indicates that, in terms of the mean rank, the *2M*-tree outperforms the *PA*-tree significantly on one occasion and slightly on two occasions while performing worse than the *PA*-tree on one occasion. The *2M*-tree performs significantly better than the simple *kd*-tree. In terms of the maximum rank, the *2M*-tree performs comparably to the *PA*-tree on three of occasions while performing worse on one occasion.

Figure 50 presents the performance of the *2M*-tree with respect to the baseline methods in terms of the distance error. Similar to the situation with rank, the *2M*-tree outperforms the *PA*-tree in terms of the mean distance error on three occasions



(a) Optdigits (b) Tiny Images (c) MNIST (d) Yahoo! Music

Figure 49: Rank with increasing time constraint. The top row corresponds to the mean rank over all queries and the bottom row corresponds to the maximum rank over all queries. GS:*kd* & GS:*PA* refer to Algorithm 15 with a *kd*-tree and *PA*-tree respectively. GS:*2M* refers to Algorithm 15 with a *2M*-tree (*please view in color*).



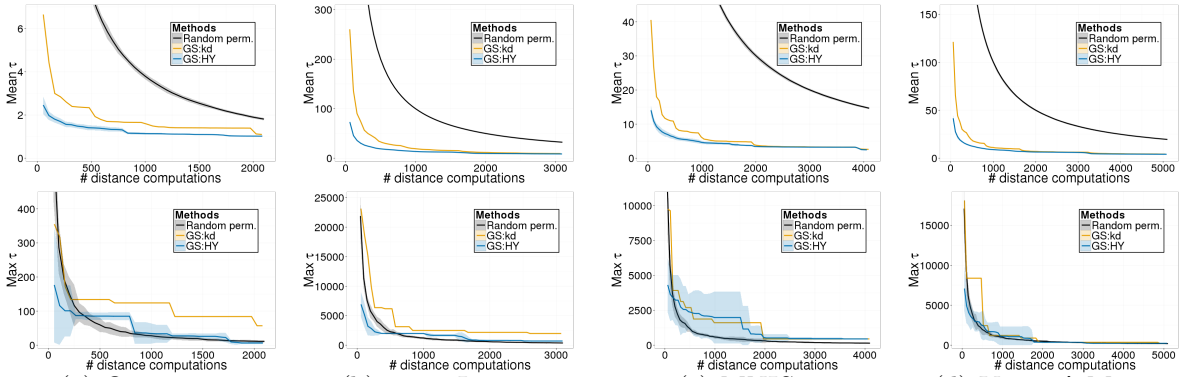
(a) Optdigits (b) Tiny Images (c) MNIST (d) Yahoo! Music

Figure 50: Distance error with increasing time constraint. The top row corresponds to the mean distance error over all queries and the bottom row corresponds to the maximum distance error over all queries. GS:*kd* & GS:*PA* refer to Algorithm 15 with a *kd*-tree and *PA*-tree respectively. GS:*2M* refers to Algorithm 15 with a *2M*-tree (*please view in color*).

(significantly once and slightly twice) and performs worse than the *PA*-tree on one occasion. In terms of the maximum distance error, the *2M*-tree appears to perform comparably to the *PA*-tree in all occasions, especially for small time-constraints (small values on the x-axis).

The results indicate that, in most cases, the *2M*-tree performs comparably to the *PA*-tree. This can be attributed to the following observations – (1) the *PA*-tree implicitly tries to reduce the quantization error by splitting along the principal axis, (2) the *2M*-tree intends to greedily minimize the quantization error, but Lloyd's algorithm solves the 2-means problem approximately and we do not perform random restarts for each split to pick the split reducing the quantization error the most; we just perform a single run of the Lloyd's algorithm, and (3) unlike the *PA*-tree, the *2M*-tree does not enforce any kind of balance in the binary tree, possibly resulting in trees with depth much worse than $O(\log n)$ (this is the possible reason for the adverse behavior of the *2M*-tree in the TINY IMAGES dataset).

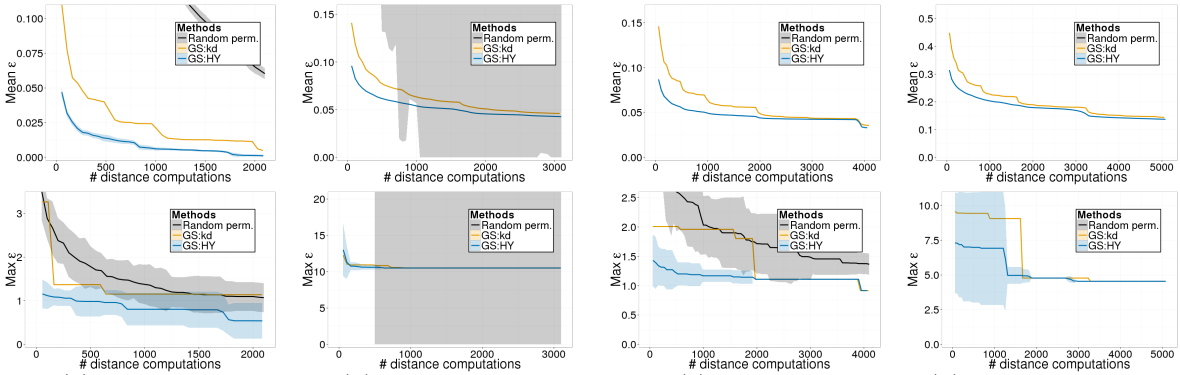
We also compare the performance of the greedy tree search with our proposed HY-tree to the performance of TCS with random permutations (Algorithm 11) and the greedy tree search with the *kd*-tree. Since the HY-tree is same as the *kd*-tree in the higher levels of the tree, we compare the improvement of the HY-tree only over the *kd*-tree. For the implementation of the HY-tree, we perform a split using MMC only if the number of points in the node is less than 5000.



(a) OPTDIGITS (b) TINY IMAGES (c) MNIST (d) YAHOO! MUSIC

Figure 51: Rank with increasing time constraint. The top row corresponds to the mean rank over all queries and the bottom row corresponds to the maximum rank over all queries. GS:*kd* & GS:*PA* refer to Algorithm 15 with a *kd*-tree and *PA*-tree respectively. GS:HY refers to Algorithm 15 with a HY-tree (*please view in color*).

Figure 51 indicates that the HY-tree shows remarkable improvement over the *kd*-tree both in terms of the mean rank and the maximum rank. In terms of the mean rank, HY-tree outperforms the *kd*-tree, while in terms of the maximum rank, HY-tree outperforms the *kd*-tree significantly for small time-constraints (small values on the x-axis). This improvement is obviously not surprising since the HY-tree is more flexible than the *kd*-tree. On one occasion (for the MNIST set with maximum rank), the HY-tree and the *kd*-tree have comparable performance. However, the curve for the HY-tree has a much smaller starting value compared to the *kd*-tree.



(a) Optdigits (b) Tiny Images (c) MNIST (d) Yahoo! Music

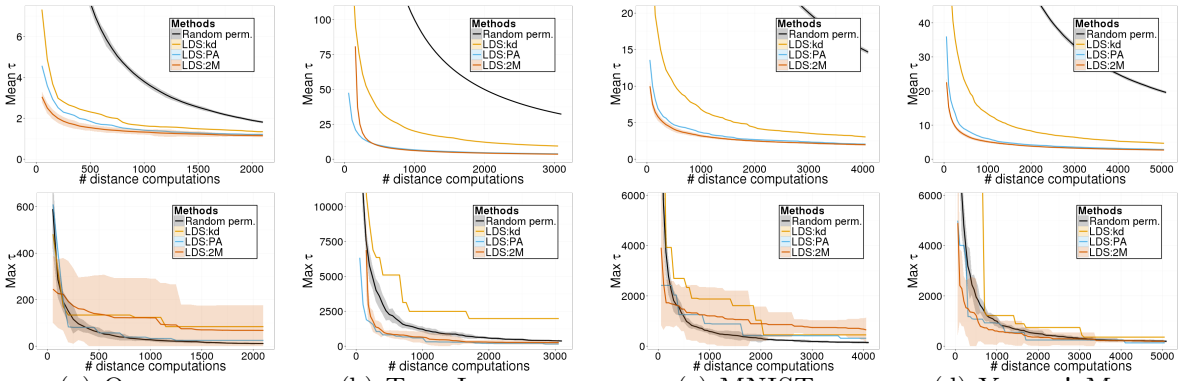
Figure 52: Distance error with increasing time constraint. The top row corresponds to the mean distance error over all queries and the bottom row corresponds to the maximum distance error over all queries. GS:*kd* & GS:*PA* refer to Algorithm 15 with a *kd*-tree and *PA*-tree respectively. GS:HY refers to Algorithm 15 with a HY-tree (*please view in color*).

The relative performance of the HY-tree with respect to the *kd*-tree in terms of the distance error is shown in Figure 52. As in the case with rank, the HY-tree outperforms the *kd*-tree by a significant margin in all cases in terms of the mean distance error. In terms of the maximum distance error, the HY-tree does not outperform the *kd*-tree on one occasion with the Tiny Images set. This can possibly attributed to the fact that when the data in the node is uni-modal or uniformly spread, an axis-aligned split might be as good as a large margin split. This is because the margin

size from a large margin split would not be significantly greater than the margin size of a heuristic axis-aligned split. A large margin split is only useful if there is a large margin to be found.

5.4.2 Lazy-Defeatist Tree Search

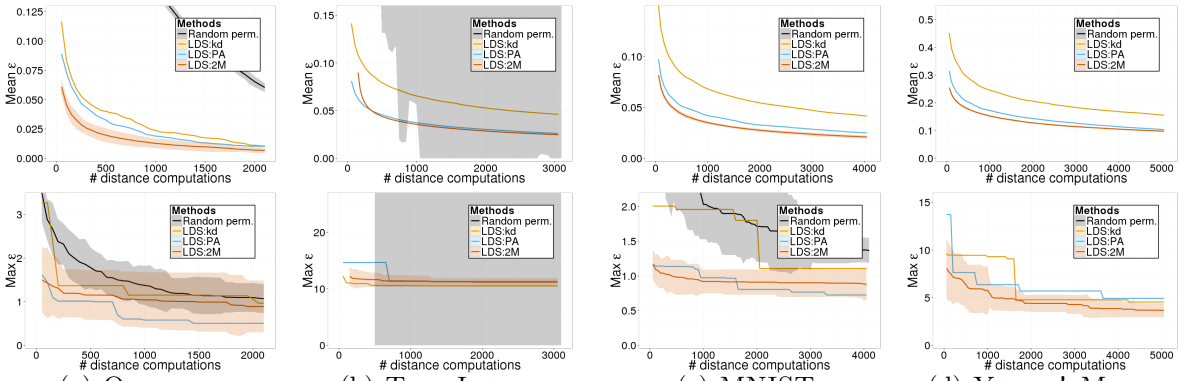
Even though the empirical performance of the lazy-defeatist search (Algorithm 16) was seen to be similar to the performance of the greedy search (Algorithm 15) in Chapter 4, we will present the performance of the learned trees, namely the *2M*-tree and the HY-tree, with lazy-defeatist search for completeness. Again, we first compare the *2M*-tree to TCS with random permutations (Algorithm 11) and the lazy-defeatist tree search with the *kd*-tree and the *PA*-tree. The results are average over 10 repetitions to cover for the randomness in the construction of the *2M*-tree.



(a) Optdigits (b) Tiny Images (c) MNIST (d) Yahoo! Music

Figure 53: Rank with increasing time constraint. The top row corresponds to the mean rank over all queries and the bottom row corresponds to the maximum rank over all queries. LDS:*kd* & LDS:*PA* refer to Algorithm 16 with a *kd*-tree and *PA*-tree respectively. LDS:*2M* refers to Algorithm 16 with a *2M*-tree (*please view in color*).

As with greedy tree search, Figure 53 indicates that the *2M*-tree outperforms the baseline methods in three of the four cases. In terms of the mean rank, the *2M*-tree outperforms the *PA*-tree significantly on one occasion and slightly on two occasions while performing worse than the *PA*-tree on one occasion. In terms of the maximum



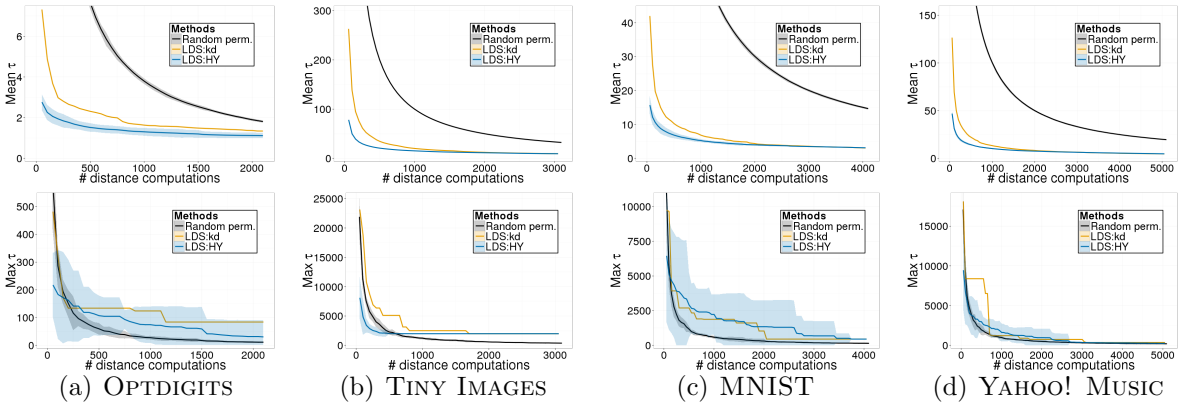
(a) Optdigits (b) Tiny Images (c) MNIST (d) Yahoo! Music

Figure 54: Distance error with increasing time constraint. The top row corresponds to the mean distance error over all queries and the bottom row corresponds to the maximum distance error over all queries. LDS:*kd* & LDS:*PA* refer to Algorithm 16 with a *kd*-tree and *PA*-tree respectively. LDS:*2M* refers to Algorithm 16 with a *2M*-tree (*please view in color*).

rank, the *2M*-tree performs comparably to the *PA*-tree on three of occasions while performing worse on one occasion.

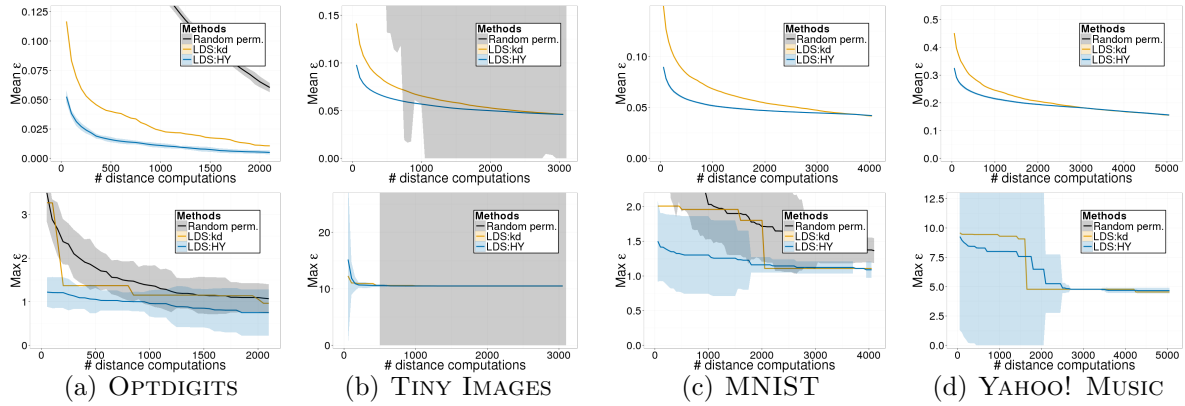
Similar to the performance with greedy tree search, Figure 54 indicates that the *2M*-tree outperforms the *PA*-tree in terms of the mean distance error in three occasions (significantly once and slightly twice) and performs worse than the *PA*-tree on one occasion. In terms of the maximum distance error, the *2M*-tree appears to perform comparably to the *PA*-tree in all occasions, especially for small time-constraints (small values on the x-axis). As expected, the relative performance of the lazy-defeatist search with the *2M*-tree is very similar to the relative performance of the greedy search with the *2M*-trees. Hence the reasons for such performance is also the same.

We also compare the performance of the lazy-defeatist tree search with our proposed HY-tree to the performance of TCS with random permutations (Algorithm 11) and the lazy-defeatist tree search with the *kd*-tree. Since the HY-tree is same as the *kd*-tree in the higher levels of the tree, we compare the improvement of the HY-tree only over the *kd*-tree. For the implementation of the HY-tree, we perform a split



(a) Optdigits (b) Tiny Images (c) MNIST (d) Yahoo! Music

Figure 55: Rank with increasing time constraint. The top row corresponds to the mean rank over all queries and the bottom row corresponds to the maximum rank over all queries. LDS:*kd* & LDS:*PA* refer to Algorithm 16 with a *kd*-tree and *PA*-tree respectively. LDS:HY refers to Algorithm 16 with a HY-tree (*please view in color*).



(a) Optdigits (b) Tiny Images (c) MNIST (d) Yahoo! Music

Figure 56: Distance error with increasing time constraint. The top row corresponds to the mean distance error over all queries and the bottom row corresponds to the maximum distance error over all queries. LDS:*kd* & LDS:*PA* refer to Algorithm 16 with a *kd*-tree and *PA*-tree respectively. LDS:HY refers to Algorithm 16 with a HY-tree (*please view in color*).

using MMC only if the number of points in the node is less than 5000.

Figure 55 indicates that, as in the case with greedy search, the HY-tree shows remarkable improvement over the *kd*-tree both in terms of the mean rank and the maximum rank. On one occasion (for the MNIST set), the HY-tree and the *kd*-tree

have comparable performance in terms of the maximum rank. However, the curve for the HY-tree has a much smaller starting value compared to the *kd*-tree. The relative performance of the HY-tree with respect to the *kd*-tree in terms of the distance-error is shown in Figure 56. As in the case with rank and in the case of greedy tree search, the HY-tree outperforms the *kd*-tree in lazy-defeatist search by a significant margin in all cases in terms of the mean distance error. In terms of the maximum distance error, the HY-tree does not outperform the *kd*-tree on one occasion with the TINY IMAGES set.

CHAPTER VI

MAX-KERNEL SEARCH

Max-kernel search. Formally, max-kernel search is defined as follows: for a given set S of n objects (the *reference set*), a Mercer kernel function $\mathcal{K}(\cdot,\cdot)$, and a query q, find the object $p \in S$ such that:

$$p = \arg\max_{r \in S} \mathcal{K}(q, r). \tag{43}$$

This general form of similarity search problem is ubiquitous in computer science. As mentioned Section 2.5, efficient approximate solutions to this problem exist for the situation where the kernels are normalized. These techniques accomplish this by performing locality-sensitive hashing of the objects directly in the kernel space. In this chapter, we focus on the general problem for any Mercer kernel and we begin by focusing on exact search. In Section 6.1, we try to understand the problem of max-kernel search and its inherent hardness. This allows us to find an appropriate indexing scheme that can index the objects directly in the (Hilbert) space induced by the Mercer kernel. Using this index, we propose an exact branch-and-bound algorithm for max-kernel search in Section 6.2 which we theoretically and empirically analyze in Sections 6.3 and 6.4. We conclude with approximate extensions of our proposed branch-and-bound algorithm in Section 6.5.

6.1 Understanding Max-Kernel Search

A Mercer kernel implies that the kernel value for a pair of objects (x, y) corresponds to an inner-product between the vector representation of the objects $\varphi(x), \varphi(y)$ in some inner-product space $\mathcal{H}$ ($\mathcal{K}(x, y) = \langle \varphi(x), \varphi(y) \rangle_{\mathcal{H}}$). Hence every Mercer kernel

induces the following metric in $\mathcal{H}$:

$$\mathbf{d}_{\mathcal{K}}(x, y) = \|\varphi(x) - \varphi(y)\| = \sqrt{\mathcal{K}(x, x) + \mathcal{K}(y, y) - 2\mathcal{K}(x, y)}. \tag{44}$$

Whenever max-kernel search can be reduced to nearest-neighbor search in $\mathcal{H}$, nearest-neighbor search methods for general metrics can be used for efficient max-kernel search. In this section, we show that this reduction is possible only under certain conditions. Subsequently, we discuss the hardness of max-kernel search and contrast it to the hardness of nearest-neighbor search. Finally, we discuss desirable properties of certain nearest-neighbor search techniques and their applicability to max-kernel search.

Possible reductions and conditions. The nearest neighbor for a query q in $\mathcal{H}$ $(\arg\min_{r \in S} \mathbf{d}_{\mathcal{K}}(q, r))$ is the max-kernel candidate (Equation (43)) if $\mathcal{K}(\cdot, \cdot)$ is a normalized kernel. The two problems can have very different answers with un-normalized kernels. Every kernel also induces a cosine similarity $\left(\mathcal{K}(x, y)/\sqrt{\mathcal{K}(x, x)\mathcal{K}(y, y)}\right)$ in $\mathcal{H}$. Similar to the previous case, the object with the maximum cosine-similarity in $\mathcal{H}$ is the max-kernel candidate only for normalized kernels.

Hardness of nearest-neighbor search. However, even if max-kernel search can be reduced to nearest-neighbor search, nearest-neighbor search is still hard for general metrics ($\Omega(n)$ for a single query) without any assumption on the structure of the metric and the set $(S, \mathbf{d})$. Here we present one notion of characterizing the hardness in terms of the structure of the metric (Karger and Ruhl, 2002):

Definition 6.1.1. *Let* $B_S(p, \Delta) = \{r \in S : \mathbf{d}(p, r) \leq \Delta\}$. *Then, the* ***expansion constant*** *of* $(S, \mathbf{d})$ *is defined as the smallest* $c \geq 2$ *such that* $|B_S(p, 2\Delta)| \leq c\,|B_S(p, \Delta)|$ $\forall p \in S$ *and* $\forall \Delta > 0$.

The expansion constant effectively bounds the number of points that could be sitting close to the surface of a hyper-sphere of any radius around any point. If c is

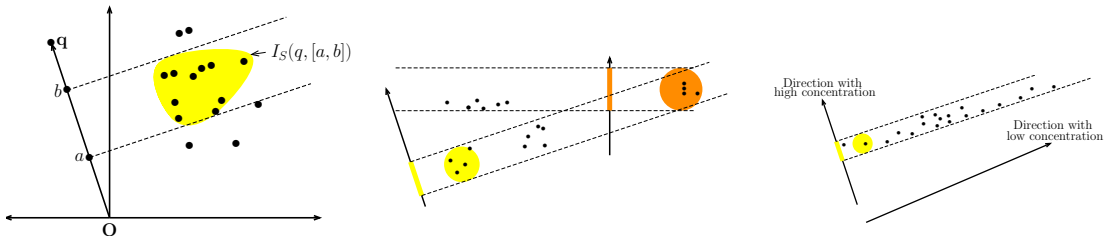
high, nearest-neighbor search could be expensive. A value of $c \sim \Omega(n)$ implies that, in the worst case, the search cannot be better than linear scan asymptotically. Under the assumption of bounded expansion constant, nearest-neighbor search methods with sub-linear/logarithmic theoretical runtime guarantees have been presented (Karger and Ruhl, 2002; Beygelzimer et al., 2006).

6.1.1 Characterizing the Hardness of max-kernel search

The kernel values for a query are proportional to the length of the projections in the direction of the query in $\mathcal{H}$. While the hardness of nearest-neighbor search depends on how concentrated the surface of spheres are (as characterized by the expansion constant), the hardness of max-kernel search should depend on the distribution of the projections in the direction of the query. This distribution can be characterized using the distribution of points in terms of distances. *If two points are close in distance, then their projections in any direction are close as well. However, if two points have close projections in a direction, it is not necessary that the points themselves are close to each other.* It is to characterize this reverse relationship between points (closeness in projections to closeness in distances) that we present a new notion of concentration in any direction:

Definition 6.1.2. *Let $\mathcal{K}(x, y) = \langle \varphi(x), \varphi(y) \rangle_{\mathcal{H}}$ be a Mercer kernel, $\mathbf{d}_{\mathcal{K}}(x, y)$ be the induced metric from $\mathcal{K}$ and let $B_S(p, \Delta) = \{r \in S : \mathbf{d}_{\mathcal{K}}(p, r) \leq \Delta\}$ for some $p \in \mathcal{H}$. Let $I_S(v, [a, b]) = \{r \in S : \langle v, \varphi(r) \rangle_{\mathcal{H}} \in [a, b]\}$ be the set of objects in S projected onto a direction v in $\mathcal{H}$ lying in the interval $[a, b]$ along v. Then, the **directional concentration constant** of $(S, \mathcal{K})$ is defined as the smallest $\gamma \geq 1$ such that $\forall u \in \mathcal{H}$ such that $\|u\|_{\mathcal{H}} = 1$, $\forall p \in S$ and $\forall \Delta > 0$, the set $I_S(u, [\langle u, \varphi(p) \rangle_{\mathcal{H}} - \Delta, \langle u, \varphi(p) \rangle_{\mathcal{H}} + \Delta])$ can be covered by γ balls of radius Δ.*

The directional concentration constant is not a measure of the number of points projected into a small interval, but rather a measure of the number of "patches" of



(a) Projection interval set. (b) Low value of γ. (c) High value of γ.

Figure 57: Concentration of projections.

the data in a particular direction. The definition implies that for a set of points with a directional concentration constant of γ, if a subset of points are close in projection in some direction, then there are at most γ subsets of these points which are close in distances as well. Consider the set of points $B = I_S(q, [a, b])$ projected into an interval in some direction (Figure 57(a)). A high value of γ implies that the number of points in B is high but the points are spread out and the number of balls (with diameter equal to $|b - a|$) required to cover all these points is high as well (each point possibly requiring an individual ball). Figure 57(c) provides one such example. A low value of γ implies that either B has a small size or the size of B is high and B can be covered with a small number of balls (of diameter $|b - a|$). Figure 57(b) is an example of a set with low γ. The directional concentration constant bounds the number of balls of a particular radius required to index points that project onto an interval of size twice the radius.

6.1.2 Desirable Existing Techniques

The notion of bounded directional concentration implies that indexing schemes capable of efficiently indexing points in a particular direction might be useful for the task of max-kernel search. Indexing schemes like space-partitioning trees have been widely used for nearest-neighbor search with success in many cases. Trees offer good hierarchical indexing schemes in low to medium dimensions; for high-dimensional data, trees which exploit some low-dimensional structure have been developed (Dasgupta

and Freund, 2008; Beygelzimer et al., 2006). These hierarchical indexing schemes lend easily to intuitive branch-and-bound algorithms for efficient solutions (especially approximate solutions (Ciaccia and Patella, 2000; Ram et al., 2009b)). In addition, tree-based branch-and-bound algorithms are essentially incremental algorithms, and can thus be easily extend to anytime algorithms (Ram et al., 2012). Most importantly, trees only require a single construction – different algorithms can work on the same tree; in addition, once a tree is built, point insertions and deletions are generally cheap.

Which tree to use for max-kernel search? Given the numerous advantages of trees, we wish to select an appropriate tree for efficient max-kernel search. The following two properties are desired of any indexing scheme used for max-kernel search:

- Explicit representation of the objects is not required.
- It should have sub-quadratic construction time.

The *kd*-tree (Friedman et al., 1977) and the metric tree (Preparata and Shamos, 1985) exhibit good characteristics and are widely used in nearest-neighbor search. However, the *kd*-tree requires the explicit representation of the points $\varphi(x)$ in $\mathcal{H}$ for its axis-aligned splits. For similar reasons, random-projection trees (Dasgupta and Freund, 2008) and principal-axis trees (McNames, 2001; Sproull, 1991) cannot be used for max-kernel search. Metric trees can be constructed without the explicit representations since they can work with only the ability to evaluate the induced metric $\mathbf{d}_{\mathcal{K}}$ (Moore, 2000)[1].

We will consider the recently formulated *cover tree* (Beygelzimer et al., 2006) for max-kernel search. It provides a systematic way to build a tree without explicit object

[1]Calculation of the explicit mean μ of a node is avoidable by using the kernel trick for operations on the mean: $\left(\langle \mu, \varphi(q) \rangle = \sum_{x \in T} \mathcal{K}(x, q)/|T|\right)$ (where T is the set of points in the node). However, this makes the tree construction and tree search computational prohibitive for efficient max-kernel search.

representations and with a sub-quadratic construction time. A key difference between cover trees and the aforementioned trees is that the *kd*-tree and the metric tree are binary trees while the cover trees can have multiple number of children. Moreover, the time complexities of building and querying a cover trees have been analyzed extensively (Beygelzimer et al., 2006; Ram et al., 2009a), whereas *kd*-trees and metric trees have been analyzed only under very limited settings (Friedman et al., 1977).

Cover tree. A cover tree stores a dataset S of size n in the form of a levelled tree. The structure has $O(n)$ space requirement (Theorem 1 in (Beygelzimer et al., 2006)). Each level is a "cover" for the level beneath it and is indexed by an integer scale i which decreases as the tree is descended. Let C_i denote the set of nodes at scale i. For all scales i, the following invariants hold: (i) (nesting invariant) $C_i \subset C_{i-1}$, (ii) (covering tree invariant) $\forall p \in C_{i-1}, \exists q \in C_i : \mathbf{d}(p, q) \leq 2^i$, and exactly one such q is a parent of p. (iii) (separation invariant) For all $p, q \in C_i$, $\mathbf{d}(p, q) > 2^i$.

Although the cover tree is defined as infinite levelled sets, the tree has a precise finite representation. As explained in (Ram et al., 2009a), "The *implicit representation* consists of infinitely many levels C_i with the level C_∞ containing a single node which is the root and the level $C_{-\infty}$ containing every point in the dataset as a node. The *explicit representation* is required to store the tree in $O(n)$ space. It coalesces all nodes in the tree for which the only child is the self-child. This implies that every explicit node either has a parent other than the self-parent or has a child other than a self-child."

6.1.3 Tree construction in the kernel space $\mathcal{H}$.

For a cover tree built on dataset S, each node is only associated with a single point $p \in S$. Therefore, because the only distance computations involve points in S, the explicit representation of the objects in $\mathcal{H}$ is not required (by the kernel trick). Given a Mercer kernel $\mathcal{K}$ for a class of objects, the distances between points in $\mathcal{H}$ required for

the tree construction in $\mathcal{H}$ can be evaluated using the distance metric $\mathbf{d}_{\mathcal{K}}$ (Equation 44) induced from the kernel. Three kernel evaluations are required to compute the distance; however, if the self-kernel values $\mathcal{K}(x, x) \forall\ x \in S$ are precomputed and saved, $\mathbf{d}_{\mathcal{K}}(x, y)$ can be evaluated with a single evaluation of $\mathcal{K}(x, y)$. The cover tree construction is given in its entirety in Algorithm 21.

Algorithm 21 The cover tree construction (Beygelzimer et al., 2006)

Input: Dataset S, metric $\mathbf{d}_K(\cdot, \cdot)$
Output: Cover tree root p

Initialize: $p \leftarrow$ random point in S
$p \leftarrow$ **Construct**$(p, \langle \mathrm{R}, \emptyset \rangle, \log_2 \max_{r \in S} \mathbf{d}(p, r))$

Construct$(p, \langle \textsc{Near}, \textsc{Far} \rangle, i)$
  if $\textsc{Near} = \emptyset$ **then**
    return $\{p, \emptyset\}$.
  else
    $\{\textsc{Self}, \textsc{Near}\} =$ **Construct**$(p, \textbf{Split}(\mathbf{d}(p, \cdot), 2^{i-1}, \{\textsc{Near}\}), i - 1)$
    add Self to Children(p)
    while $\textsc{Near} \neq \emptyset$ **do**
      pick q in Near
      $\{\textsc{Child}, \textsc{Unused}\} =$ **Construct**$(q, \textbf{Split}(\mathbf{d}(q, \cdot), 2^{i-1}, \{\textsc{Near}, \textsc{Far}\}), i - 1)$
      add Child to Children(p)
      $\{\textsc{New-near}, \textsc{New-far}\} =$ **Split**$(\mathbf{d}(p, \cdot), 2^i, \{\textsc{Unused}\})$
      $\textsc{Far} \leftarrow \textsc{Far} \cup \textsc{New-far}$; $\textsc{Near} \leftarrow \textsc{Near} \cup \textsc{New-near}$
    end while
    return $\{p, \textsc{Far}\}$.
  end if

Split$(d(p, \cdot), r, \{S_1, S_2, \ldots\})$
  $\textsc{Near} = \bigcup_i \{q \in S_i : \mathbf{d}(p, q) \leq r\}$; $\textsc{Far} = \bigcup_i \{q \in S_i : 2r > \mathbf{d}(p, q) > r\}$
  $\forall\ i, S_i \leftarrow S_i \setminus (\textsc{Near} \cup \textsc{Far})$.
  return $\{\textsc{Near}, \textsc{Far}\}$.

The algorithm calls the recursive function **Construct**$(p, \langle \textsc{Near}, \textsc{Far} \rangle, i)$ where p is a point, $\langle \textsc{Near}, \textsc{Far} \rangle$ are the point sets and i is the current level of the tree. This recursive function calls a subroutine **Split**$(\mathbf{d}(p, \cdot),\ r, \{S_1, S_2, \ldots\})$ with $\mathbf{d}(p, \cdot)$ representing the set of distances of the points in the point sets $\{S_1, S_2, \ldots\}$ to the point p and r is the splitting distance to form the Near and Far sets. Hence, the

cover tree construction only requires distances between the points in the set; and therefore construction in $\mathcal{H}$ *does not require any explicit representation in* $\mathcal{H}$. The tree construction time is bounded by the following theorem:

Theorem 6.1.1. *(Theorem 3.6 in (Beygelzimer et al., 2006)) For a data set* S *of* n *objects and a metric* $\mathbf{d}_{\mathcal{K}}$ *with expansion constant* c*, the tree construction requires at most* $O(c^6 n \log n)$ *time.*

Remark. We would like to point out that while we consider the cover tree to implicitly index points in $\mathcal{H}$ without any modification, the cover tree has only been used for nearest-neighbor search to this date. The novelty of this work is a new branch-and-bound *algorithm* using this tree to solve the general task of max-kernel search with provable theoretical guarantees and supporting empirical evidence.

6.2 FastMKS: The Branch-and-Bound Algorithm

In this section, we present a simple branch-and-bound algorithm on the cover tree for max-kernel search. Branch-and-bound is widely used in nearest-neighbor search with the help of the triangle inequality of the distance metric. In the absence of the triangle inequality for kernel functions, we obtain a novel bound on the max-kernel value possible between a query and any subtree of a cover tree. Then we present the algorithm for exact search.

Bounding the max-kernel value. A cover tree node is defined by an object p and a level i. Let S_p^i denote the set of objects in the subtree rooted at a node defined by object p at level i. In the following theorem, we bound the maximum possible kernel value between a query and an object in the subtree of the cover tree. For notational convenience, for any set R, we will denote $\max_{r \in R} \mathcal{K}(q, r)$ as $\mathcal{K}(q, R)$.

Theorem 6.2.1. *Given a cover tree node rooted at an object* p *at level* i *in the kernel space* $\mathcal{H}$ *and a (query) object* q*, the maximum kernel function value between* q *and*

any object in the set S_p^i is bounded as:

$$\mathcal{K}(q, S_p^i) \leq \mathcal{K}(q, p) + 2^{i+1}\sqrt{\mathcal{K}(q, q)}. \tag{45}$$

Proof. Suppose that p^* is the best possible match in the set S_p^i for q and let $\tilde{u}$ be a unit vector in the direction of the line joining $\varphi(p)$ to $\varphi(p^*)$ in $\mathcal{H}$. Then $\varphi(p^*) = \varphi(p) + \Delta \cdot \tilde{u}$ where $\Delta = \mathbf{d}_{\mathcal{K}}(p, p^*)$ is the distance between $\varphi(p)$ and the best possible match $\varphi(p^*)$ (see Figure 58). Then we have the following:

$$\begin{aligned} \mathcal{K}(q, S_p^i) &= \mathcal{K}(q, p^*) = \langle \varphi(q), \varphi(p^*) \rangle_{\mathcal{H}} \\ &= \langle \varphi(q), \varphi(p) + \Delta \cdot \tilde{u} \rangle_{\mathcal{H}} \\ &\leq \langle \varphi(q), \varphi(p) \rangle_{\mathcal{H}} + \Delta \left\| \varphi(q) \right\|_{\mathcal{H}}, \end{aligned} \tag{46}$$

where the last inequality follows from the Cauchy-Schwartz inequality ($\langle x, y \rangle \leq \|x\| \, \|y\|$) and the fact that $\|\tilde{u}\| = 1$. From the definition of the kernel function, Equation (46) gives us $\mathcal{K}(q, S_p^i) \leq \mathcal{K}(q, p) + \Delta\sqrt{\mathcal{K}(q, q)}$. We bound Δ from above using the covering invariant – for any cover tree node p at level i, the distance to the farthest child node is bounded by 2^i. Applying this bound recursively with the triangle inequality of $\mathbf{d}_{\mathcal{K}}$ gives us $\Delta = \mathbf{d}_{\mathcal{K}}(p, p^*) \leq \sum_{j=-\infty}^{i} 2^j = 2^{i+1}$. The statement of the theorem follows. □

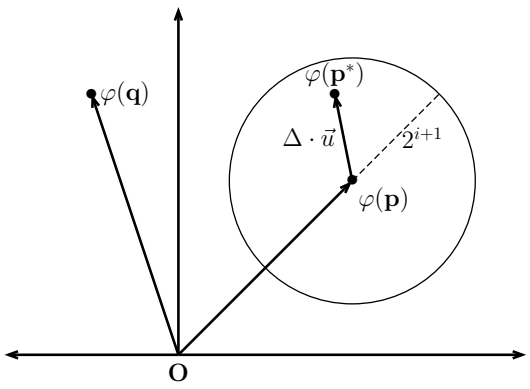


Figure 58: Max-kernel upper bound.

For normalized kernels ($\mathcal{K}(x,x) = 1\ \forall x \in S$), all the points are on the surface of a hyper-sphere in $\mathcal{H}$. In this case, the above bound in Theorem 6.2.1 is correct but possibly loose. In the following theorem, we present a tighter bound specifically for this condition:

Theorem 6.2.2. *Consider a kernel $\mathcal{K}$ such that $\mathcal{K}(x,x) = 1 \forall\ x \in S$. Given a cover tree node rooted at an object p at level i in $\mathcal{H}$ and a (query) object q, the maximum kernel function value between q and any object in the set S_p^i is bounded as:*

$$\mathcal{K}(q, S_p^i) \leq \begin{cases} \mathcal{K}(q,p)(1-2^{2i+1}) + 2^{i+1}\sqrt{(1-\mathcal{K}(q,p)^2)\,(1-2^{2i})}, \\ \quad \textit{if } \mathcal{K}(q,p) \leq 1 - 2^{2i+1} \\ 1.0, \quad \textit{otherwise} \end{cases} \tag{47}$$

The proof is similar to the proof of Theorem 6.2.1.

Proof. Since all the points are sitting on the surface of a hypersphere in $\mathcal{H}$, $\mathcal{K}(q,p)$ denotes the cosine of the angle made by $\varphi(q)$ and $\varphi(p)$ at the origin. If we first consider the case where q lies within the ball bounding cover tree node p at level i (that is, if $d_K(q,p) < 2^{i+1}$), it is clear that the maximum possible kernel evaluation should be 1, because there could exist a point in S_p^i whose angle to q is 0. We can easily modify this condition to an easier condition on $\mathcal{K}(q,p)$, which is $\mathcal{K}(q,p) < 1 - 2^{2i+1}$.

Now, for the other case, let $\cos\theta_{qp} = \mathcal{K}(q,p)$ and $p^* = \arg\max_{r \in S_p^i} \mathcal{K}(q,r)$. Let θ_{pp^*} be the angle between $\varphi(p)$ and $\varphi(p^*)$ and θ_{qp^*} be the angle between $\varphi(q)$ and $\varphi(p^*)$ at the origin. Then

$$\mathcal{K}(q,p^*) = \cos\theta_{qp^*} \leq \cos(\{\theta_{qp} - \theta_{pp^*}\}_+). \tag{48}$$

Now we know that $\mathbf{d}_{\mathcal{K}}(p,p^*) \leq 2^{i+1}$ and that $\mathbf{d}_{\mathcal{K}}(p,p^*) = \sqrt{2 - 2\cos\theta_{pp^*}}$. Hence $\cos\theta_{pp^*} \geq 1 - 2^{2i+1}$, and $\theta_{pp^*} \leq |\cos^{-1}(1-2^{2i+1})|$ and take $\theta = |\cos^{-1}(1-2^{2i+1})|$. Combining this with equation (48), we get:

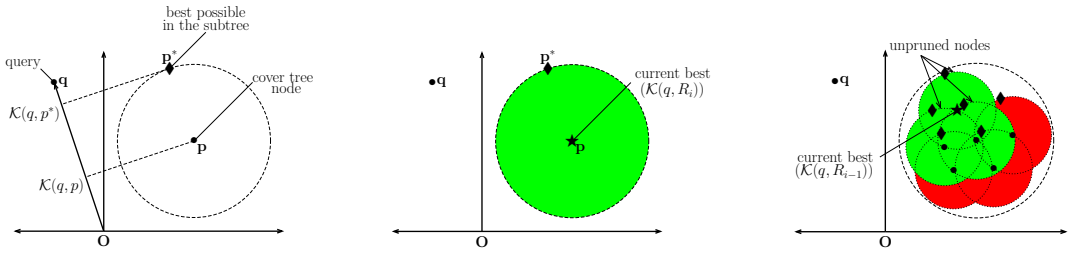
$$\mathcal{K}(q, S_p^i) \leq \cos(\{\theta_{qp} - \theta_{pp^*}\}_+) \leq \cos(\{\theta_{qp} - \theta\}_+).$$

Substituting the value of θ above and simplifying gives us the statement of the theorem. □

Algorithm 22 FastMKS

Input: Cover tree T, Query q
Output: p: $\mathcal{K}(q, p) = \max_r \mathcal{K}(q, r)$

Initialize: $R_\infty = C_\infty$ // *the root of the tree* T
for $i = \infty$ **to** $-\infty$ **do**
  $R = \{Children(r) : r \in R_i\}$ // *tree descend*
  $R_{i-1} = \{r \in R : \mathcal{K}(q, r) \geq \mathcal{K}(q, R) - 2^i \sqrt{\mathcal{K}(q, q)}\}$
end for
return $\arg\max_{r \in R_{-\infty}} \mathcal{K}(q, r)$ // *the leaf nodes.*



(a) Best possible kernel value (b) At level i (c) At level $i - 1$

Figure 59: Branch-and-bound tree-traversal. The green nodes are retained and the red nodes are pruned (*please view in color*).

The branch-and-bound algorithm. The bound on $\mathcal{K}(q, S_p^i)$ is used to decide whether a node is retained for further exploration or removed ("pruned") from consideration. The branch-and-bound algorithm **FastMKS**(T, q) is presented in Algorithm 22. If the maximum possible kernel value from a subtree is less than the current best kernel value, that subtree is not explored any further. The best possible kernel value from a subtree and a step of the algorithm is depicted in Figure 59. For a retained node, all its children are explored. The correctness of **FastMKS**(T, q) follows from:

Theorem 6.2.3. *If T is a cover tree on S, then* ***FastMKS***(T, q) *(Algorithm 22) returns* $\arg\max_{r \in S} \mathcal{K}(q, r)$.

Proof. Let $p^* = \arg\max_{r \in S} \mathcal{K}(q, r)$ and $\kappa^* = \mathcal{K}(q, p^*)$. At the beginning of the iteration at level ∞, p^* is in consideration since $p^* \in S$ and for $p \in C_\infty$, $S_p^\infty = S$. At the end of the iteration at level i, all points $p \in S$ such $\mathcal{K}(q, p) \geq \mathcal{K}(q, R_{i-1}) - 2^i\sqrt{\mathcal{K}(q, q)}$ are still in consideration. Since $\kappa^* \geq \mathcal{K}(q, R_{i-1})$, p^* is still in consideration. Either $p^* \in R_{i-1}$ or p^* is the grandchild of some point $p \in R_{i-1}$. Hence, by theorem 4.1, $\mathcal{K}(q, R_{i-1}) \geq \kappa^* - 2^i\sqrt{\mathcal{K}(q, q)}\ \forall\ i$. Now $\lim_{i \to -\infty} \mathcal{K}(q, R_{i-1}) \geq \lim_{i \to -\infty} \left(\kappa^* - 2^i\sqrt{\mathcal{K}(q, q)}\right) = \kappa^*$ (assuming $\mathcal{K}(q, q) < \infty$). Hence $\mathcal{K}(q, R_{-\infty}) = \kappa^*$ and **FastMKS**(T, q) returns $\arg\max_{r \in R_{-\infty}} \mathcal{K}(q, r) = p^*$. □

6.3 Runtime Guarantee for FastMKS

The runtime analysis of **FastMKS** will make use of the following results from Beygelzimer et al. (2006):

Lemma 6.3.1. *The number of children of any node p is bounded by c^4.*

Lemma 6.3.2. *The maximum depth of any point p in the explicit representation is $O(c^2 \log n)$.*

The main result of this section is the search time complexity of **FastMKS**(T, q) in terms of the number of objects in S and the properties of $(S, \mathcal{K})$:

Theorem 6.3.1. *Given a Mercer kernel $\mathcal{K}$, if the dataset S of size n has an expansion constant c (with the metric $\mathbf{d}_\mathcal{K}$) and a directional concentration constant γ, **FastMKS**(T, q) requires $\mathbf{O}(c^{12}\gamma^2 \log n)$ time.*

Proof. The first part of the proof is similar to the runtime analysis of NNS with cover trees (Beygelzimer et al., 2006). Let R^* denote the last explicit R_i considered by the algorithm. By Lemma 6.3.2, the explicit depth of any point in R^* is at most $k = O(c^2 \log n)$. The maximum number of iterations required would be at most $k|R^*| \leq k \max_i |R_i|$. The amount of work done in each iteration is at most $O(\max_i |R_i|)$, hence resulting in a total of $O(k \max_i |R_i|^2)$ work. Moreover, from Lemma 6.3.1,

the total number of children encountered throughout the whole algorithm is at most $k \max_i |R_i| \cdot c^4$. Hence, the pruning/retaining rule does at most $O(k \max_i |R_i| \cdot c^4)$ work. Also, $R_{-\infty} \leq \max_i |R_i|$. Therefore, the algorithm requires at most $O(k \max_i |R_i|^2 + k \max_i |R_i| \cdot c^4)$ time.

Now we bound $\max_i |R_i|$. Let $u = \varphi(q)/\|\varphi(q)\|$. Then

$$I_S(\varphi(q), [a, b]) = I_S\left(u, [a/\|\varphi(q)\|, b/\|\varphi(q)\|]\right).$$

For any level i, let $R = \{\text{Children}(r) \colon r \in R_i\}$ and let $\kappa = \mathcal{K}(q, R)$ and $\kappa^* = \mathcal{K}(q, S)$. Then

$$\begin{aligned} R_{i-1} &= \{r \in R \colon \mathcal{K}(q, r) \geq \kappa - 2^i \|\varphi(q)\|\} = I_S(\varphi(q), [\kappa - 2^i \|\varphi(q)\|, \kappa^*]) \cap R \\ &\subseteq I_S(\varphi(q), [\kappa - 2^i \|\varphi(q)\|, \kappa^*]) \cap C_{i-1} \subseteq I_S(\varphi(q), [\kappa^* - 2^{i+1} \|\varphi(q)\|, \kappa^*]) \cap C_{i-1} \end{aligned}$$

since $\kappa^* \leq \kappa + 2^i \|\varphi(q)\|$. Then for any $r \in I_S(\varphi(q), [\kappa^* - 2^{i+1} \|\varphi(q)\|, \kappa^*])$,

$$\begin{aligned} &I_S(\varphi(q), [\kappa^* - 2^{i+1} \|\varphi(q)\|, \kappa^*]) \subseteq I_S(q, [\mathcal{K}(q, r) - 2^{i+1} \|\varphi(q)\|, \mathcal{K}(q, r) + 2^{i+1} \|\varphi(q)\|]) \\ &= I_S(u, [\langle u, \varphi(r)\rangle - 2^{i+1}, \langle u, \varphi(r)\rangle + 2^{i+1}]). \end{aligned}$$

By the definition of the directional concentration constant, there exists r_js such that:

$$I_S(u, [\langle u, \varphi(r)\rangle - 2^{i+1}, \langle u, \varphi(r)\rangle + 2^{i+1}]) \subseteq \cup_{j=1}^{\gamma} B_S(r_j, 2^{i+1}).$$

Bounding the size of $I_S(q, [K - 2^i \|\varphi(q)\|, \mathcal{K}(q, S)]) \cap C_{i-1}$ amounts to bounding the number of disjoint balls of radius 2^{i-2} that can be packed into each of the γ balls $B_S(r_j, 2^{i+1} + 2^{i-2})$. For each of the r_j, we have:

$$|B_S(r_j, 2^{i+1} + 2^{i-2})| \leq |B_S(r', 2^{i+2} + 2^{i-1})| \leq |B_S(r', 2^{i+3})| \leq c^5 |B_S(r, 2^{i-2})|.$$

Hence, $\forall\ i$, $|R_i| \leq \gamma c^5$. Hence $\max_i |R_i| \leq \gamma c^5$, thus giving us the statement of the theorem. □

Comparing to the query time $O(c^{12} \log n)$ for nearest-neighbor search (Beygelzimer et al., 2006), it is clear that **FastMKS** has similar $\log n$ scaling, but also has an extra price of γ^2 for solving the more general problem of max-kernel search.

6.4 Empirical Evaluation of FastMKS

We evaluate **FastMKS** with different kernels and datasets. For every experiment, we query the top $\{1, 2, 5, 10\}$ max-kernel candidates and report the speedup over linear search (in terms of the number of kernel evaluations performed). The cover tree and the algorithm is developed in MLPACK[2](Curtin et al., 2011).

Table 5: Details of the vector datasets.

Datasets	**# queries**	**# points**	**# dimensions**
Y! Music	100000	600000	50
MovieLens	6040	3706	50
Optdigits	450	1347	64
Physics	37500	112500	78
Bio	75000	210409	74
Covertype	150000	431012	55
LiveJournal	10000	10000	25327
MNIST	10000	60000	784
Netflix	480189	17770	50
Corel	10000	27749	32
LCDM	6000000	10777216	3
TinyImages	5000	1000000	384

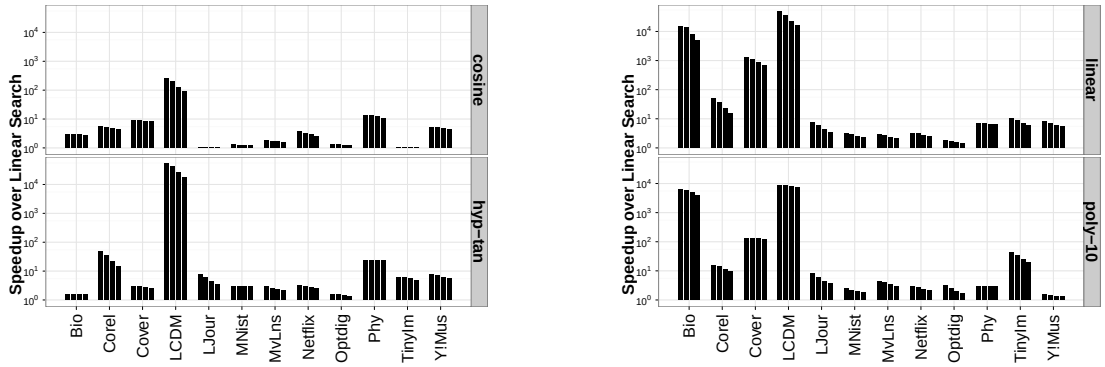
Datasets. We use two different classes of datasets – (1) Datasets with fixed-length objects: These include the MNIST dataset (LeCun, 2000), the Isomap "Images" dataset, several datasets from the UCI machine learning repository (Blake and Merz, 1998), three collaborative filtering datasets (MovieLens, Netflix (Bennett and Lanning, 2007), Yahoo! Music (Dror et al., 2011)), the LCDM astronomy dataset (Lupton et al., 2001), the LiveJournal blog moods text dataset (Kim et al., 2011) and a subset of the 80 million tiny images dataset (Torralba et al., 2008) (the sizes of the datasets are presented in Table 5). (2) Datasets without fixed length representation: We consider protein sequences from the GenBank[3]. The datasets are again selected based

[2]More information on MLPACK is at `http://mlpack.org`.

[3]`ftp://ftp.ncbi.nih.gov/refseq/release/complete/`

on the usual applications of max-kernel search – computer vision, machine learning and bioinformatics.

Kernels. We consider all of the following kernels for the vector datasets: *cosine*, *hyperbolic tangent*[4], *linear*, and *polynomial* (with degree 10). While **FastMKS** is applicable to any kernel, max-kernel search with the Gaussian kernel reduces to nearest-neighbor search in the input space; hence, we omit this kernel from our experiments. The p-spectrum kernel (Leslie et al., 2002) is used for the protein sequence data.



(a) Cosine and hyperbolic tangent kernels

(b) Linear and polynomial (degree 10) kernels

Figure 60: Speedups over linear scan for **FastMKS** with $k = 1, 2, 5, 10$ with 4 kernels and 12 datasets.

Results. The results for the vector datasets are summarized in Figure 60. While the speedups range from anywhere between 1 to over 10^4, a speedup close to an order of magnitude is seen in most large datasets (except MNIST). It is also quite clear that different kernels give very different speedups for the same dataset. This can be attributed to the fact that the expansion constant and the directional concentration constant are properties of the dataset-kernel pair. The results for the protein sequence data (Figure 61) indicate that the speedups increase with increasing reference set

[4]While the hyperbolic tangent kernel is not Mercer in general, our proposed algorithm works correctly with non-Mercer kernels if the kernel is positive definite when restricted to the dataset.

size, recording a speedup of over two orders of magnitude for a set of around 100000 sequences, exhibiting the logarithmic scaling of **FastMKS**.

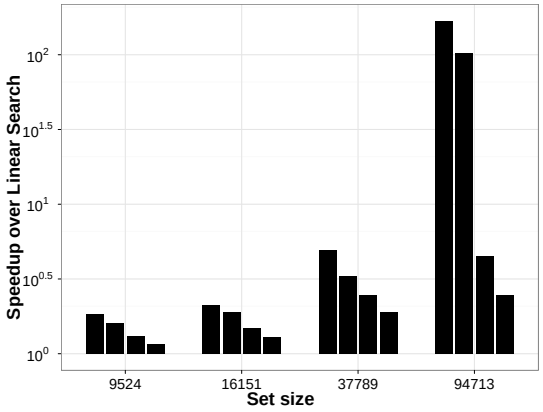


Figure 61: Speedups over linear scan for **FastMKS** with $k = 1, 2, 5, 10$ with sets of protein sequences of increasing size to demonstrate the scaling of **FastMKS**.

6.5 Fast Approximate Max-Kernel Search

Similarity search problems can be approximated for further scalability. Even though we are focusing on exact max-kernel search, we wish to demonstrate that the tree based method can be very easily extended to perform the approximate max-kernel search. Approximation can be achieved in the following ways:

1. Absolute value approximation (AVA) – return $p \in S : \mathcal{K}(q, p) \geq \mathcal{K}(q, S) - \epsilon$.
2. Relative value approximation (RVA) – return $p \in S : \mathcal{K}(q, p) \geq (1 - \epsilon)\mathcal{K}(q, S)$[5].
3. Rank approximation (RA) – return $p \in S : |\{r \in S : \mathcal{K}(q, r) > \mathcal{K}(q, p)\}| \leq \tau$.

Care has to be taken for relative value approximation since there is no guarantee that $\mathcal{K}(q, S) > 0$. The following stopping rules can be used at the end of an iteration in the for loop in **FastMKS**(T, q) (Algorithm 22) for the **value approximations**. The best candidate up until then is the approximate solution.

[5]Here we are assuming that $\mathcal{K}(q, S) > 0$. In case $\mathcal{K}(q, S) < 0$, we seek a $p \in S$ such that $\mathcal{K}(q, p) > \mathcal{K}(q, S) - \epsilon|\mathcal{K}(q, S)|$

1. AVA stopping: if $\epsilon \geq 2^i \sqrt{\mathcal{K}(q,q)}$, stop.

2. RVA stopping: assuming $\mathcal{K}(q,S) > 0$, if $\mathcal{K}(q,R_{i-1}) \geq \left(2^i/\epsilon\right) \sqrt{\mathcal{K}(q,q)}$, stop.[6]

Theorem 6.5.1. *The AVA stopping rule returns a point $p \in S$ such that $\mathcal{K}(q,p) \geq \mathcal{K}(q,S) - \epsilon$.*

Proof. At the end of iteration at level i, $\mathcal{K}(q,R_{i-1}) \geq \mathcal{K}(q,S) - 2^i\sqrt{\mathcal{K}(q,q)}$. If $2^i\sqrt{\mathcal{K}(q,q)} \leq \epsilon$, the AVA condition is satisfied. □

Theorem 6.5.2. *Assuming $\mathcal{K}(q,S) > 0$, the RVA stopping rule returns a point $p \in S$ such that $\mathcal{K}(q,p) \geq (1-\epsilon)\mathcal{K}(q,S)$.*

Proof. When $\mathcal{K}(q,R_{i-1}) \geq \left(2^i/\epsilon\right)\sqrt{\mathcal{K}(q,q)}$, then $2^i\sqrt{\mathcal{K}(q,q)} \leq \epsilon\mathcal{K}(q,R_{i-1}) \leq \epsilon\mathcal{K}(q,S)$. Using this, we have $\mathcal{K}(q,R_{i-1}) \geq \mathcal{K}(q,S) - 2^i\sqrt{\mathcal{K}(q,q)} \geq \mathcal{K}(q,S) - \epsilon\mathcal{K}(q,S)$. □

Algorithm 23 RAFastMKS

Input: Cover tree T, Query q, Rank error τ, Failure probability δ
Output: p: $|\{r: \mathcal{K}(q,r) > \mathcal{K}(q,p)\}| \leq \tau$
Initialize $R_\infty \leftarrow C_\infty$ *// the root of the tree T*
Initialize $k \leftarrow \left\lceil \frac{\log 1/\delta}{\log 1/\left(1-\frac{\tau}{n}\right)} \right\rceil$ *// the number of samples*
for $i = \infty$ **to** $-\infty$ **do**
  $R = \{Children(r): r \in R_i\}$ *// tree descend*
  $R' = \{r \in R: \mathcal{K}(q,r) \geq K(q,R) - 2^i\sqrt{\mathcal{K}(q,q)}\}$ *// the cover set*
  $R_{i-1} = \{r \in R': |S_r^i| > \frac{n}{k}\}$ *// approximate by sampling*
  $R_{-\infty} = R_{-\infty} \cup \{R' \setminus R_{i-1}\}$.
end for
return $\arg\max_{r \in R_{-\infty}} \mathcal{K}(q,r)$ *// the leaf nodes.*

Rank-approximation can be achieved by performing stratified sampling on the cover tree. The **rank-approximation** of the max-kernel search is defined as follows: for a given set S of n objects (the reference set), a (Mercer) kernel function $\mathcal{K}(\cdot,\cdot)$, and a query q, find the object $p \in S$ such that

$$|\{r \in S: \mathcal{K}(q,r) > \mathcal{K}(q,p)\}| \leq \tau.$$

[6]The stopping rule can be easily modified for $\mathcal{K}(q,S) < 0$.

As mentioned in Ram et al. (2009b) (as well as Chapter 3), the idea is to draw enough samples S' from the tree such that

$$\Pr\left(|\{r \in S : \mathcal{K}(q, r) > \mathcal{K}(q, S')\}| < \tau\right) \geq 1 - \delta.$$

The stratified sampling on a tree is presented in Algorithm 23. We assume that at each node of the tree, we have access to the number of points in the subtree S_r^i rooted at node r at level i for every node in the tree. This algorithm returns a τ-rank approximate solution to the max-kernel operation with probability $(1 - \delta)$.

CHAPTER VII

UNANSWERED QUESTIONS

My thesis has left me with more unanswered questions than questions I have actually answered. The obvious next step is to improve upon the current answers. But there are certain questions I find interesting and will briefly discuss these question in this chapter. These are some of the questions I wish to pursue at some point in my future research.

- **Can we do rank-sensitive hashing?** In the usual setting of nearest-neighbor search in Euclidean or Hamming space, I wish to have a hashing based search technique that guarantees any desired rank error (say τ). This would be possible if we could devise hash functions with the following properties:

  - if two points are both among the τ-nearest-neighbors of each other, they should be hashed into the same bucket, and
  - if two points are neither among the τ-nearest-neighbor of each other, they should be hashed into separate buckets.

  While the idea seems simple, this needs to be done efficiently in the preprocessing step. And the hashing functions should be such that they can efficiently hash new queries. It would also be interesting to explore a randomized technique to create hash functions with these desired properties. An issue with this idea is that the τ-neighborhood relationship is not symmetric – it is possible for a (query) point to not be in the τ-nearest-neighborhood of any of its τ-nearest-neighbors. In this case, the hash functions with the proposed properties will not return any neighbor for such a query point.

- **What is the true time complexity of nearest-neighbor search?** Quoting Ken Clarkson (Clarkson, 2006), "The task of nearest-neighbor searching can be viewed as having two parts, finding the nearest-neighbor, and proving that all other sites are **not** the nearest-neighbor". An interesting question is to find out the actual time taken by the algorithm to find (or locate) the nearest-neighbor. I do not have a precise form of the bound I am seeking. But under distributional assumptions on the dataset and the queries, it would be great to have bounds of the form "with probability $1 - \delta$, the nearest-neighbor is located in time at most t", where decreasing the failure probability δ implies increasing the location time t. This kind of a result would allow us to provide lower bounds for the time-constrained search performance of any algorithm (in terms of the minimum nearest-neighbor error incurred for a given time constraint).

- **What is the best strategy for time-constrained search with a batch of queries?** In time-constrained search, for an allowed amount of time, different queries can get different amounts of search error. I aggregated the error over all the queries with each query getting the same amount of time. It might be possible to get improved aggregated error over all the queries for the same amount of cumulative time if the available amount of time is allocated in an adaptive way – allocating more time to certain queries and less to others. This would require us to develop a notion of "hardness" for queries and a way to compute this hardness on the fly without eating too much into the time constraint. The allocation of the time (dynamic or static) should also have to be extremely efficient.

- **Can we solve the problem of weighted similarity search in a general way?** There are two forms of weighted similarity search where every point r in the dataset R is associated with a weight $w(r)$, and the goal is to efficiently

solve the following weighted versions of similarity search:

- Additive weights: $\arg\max_{r \in R} (\mathcal{S}(q, r) + w(r))$
- Multiplicative weights: $\arg\max_{r \in R} w(r) \cdot \mathcal{S}(q, r)$

These weights can be arbitrary or be continuous functions with bounded Lipschitz constants. In case of data in Euclidean space or the Hamming cube, arbitrary additive weights can easily be dealt with – artificial features can be added to each of the points in the dataset such that the pairwise distances between any query and these points with these additional features is equal to the (additive) weighted distance between the query and the point in the original space. At this point, usual methods for nearest-neighbor search can be applied to the modified dataset and the search performance guarantees would probably depend on the properties of the dataset as well as the properties of the weights (or weighing functions). Such a straightforward transformation is not possible for multiplicatively weighted similarity search even with Euclidean space or Hamming cube.

For max-kernel search with Mercer kernels, additive weights can be dealt with seamlessly by adding artificial features again. However, multiplicative weights can also be dealt with by artificially scaling the points with their corresponding weights in the Hilbert space induced by the Mercer-kernel. In this case, the exact (and approximate) search algorithm presented in Chapter 6 can be directly used for multiplicatively weighted similarity search. I believe that an empirical and theoretical study of these methods for weighted similarity search would be interesting.

REFERENCES

Ailon, N. and Chazelle, B. (2006). Approximate nearest neighbors and the fast johnson-lindenstrauss transform. In *Proceedings of the thirty-eighth annual ACM Symp. on Theory of computing*. ACM.

Altschul, S., Gish, W., Miller, W., Myers, E., Lipman, D., et al. (1990). Basic Local Alignment Search Tool. *Journal of Molecular Biology*.

Andoni, A. and Indyk, P. (2006). Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *Foundations of Computer Science, 2006. FOCS'06. 47th Annual IEEE Symp. on*. IEEE.

Arthur, D. and Vassilvitskii, S. (2007). k-means++: The advantages of careful seeding. In *Proceedings of the eighteenth annual ACM-SIAM Symp. on Discrete algorithms*, pages 1027–1035. Society for Industrial and Applied Mathematics.

Arya, S. and Mount, D. (1993). Approximate nearest neighbor queries in fixed dimensions. In *Proceedings of the fourth annual ACM-SIAM Symp. on Discrete algorithms*. Society for Industrial and Applied Mathematics.

Arya, S., Mount, D., Netanyahu, N., Silverman, R., and Wu, A. (1998). An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *Journal of the ACM (JACM)*.

Balcan, M.-F., Blum, A., and Vempala, S. (2006). Kernels as features: On kernels, margins, and low-dimensional mappings. *Machine Learning*, 65(1):79–94.

Bennett, J. and Lanning, S. (2007). The Netflix Prize. In *Proc. KDD Cup and Workshop*.

Beygelzimer, A., Kakade, S., and Langford, J. (2006). Cover Trees for Nearest Neighbor. *Proceedings of the International Conference on Machine Learning*.

Blake, C. L. and Merz, C. J. (1998). UCI Machine Learning Repository. http://archive.ics.uci.edu/ml/.

Borgwardt, K., Ong, C., Schönauer, S., Vishwanathan, S. V. N., Smola, A., and Kriegel, H. (2005). Protein Function Prediction via Graph Kernels. *Bioinformatics*.

Cayton, L. and Dasgupta, S. (2007). A Learning Framework for Nearest Neighbor Search. *Advances in Neural Info. Proc. Systems 20*.

Charikar, M. S. (2002). Similarity Estimation Techniques from Rounding Algorithms. In *Proceedings of the 34th annual ACM Symp. on Theory of Comp.*

Ciaccia, P. and Patella, M. (2000). PAC Nearest Neighbor Queries: Approximate and Controlled Search in High-dimensional and Metric spaces. *Proceedings of 16th International Conference on Data Engineering*.

Clarkson, K. (1994). An algorithm for approximate closest-point queries. In *Proceedings of the tenth annual Symp. on Computational geometry*. ACM.

Clarkson, K. (1999). Nearest neighbor queries in metric spaces. *Discrete and Computational Geometry*.

Clarkson, K. (2006). Nearest-neighbor Searching and Metric Space Dimensions. *Nearest-Neighbor Methods for Learning and Vision: Theory and Practice*.

Curtin, R. R., Cline, J. R., Slagle, N. P., Amidon, M. L., and Gray, A. G. (2011). MLPACK: A Scalable C++ Machine Learning Library. In *BigLearning: Algorithms, Systems, and Tools for Learning at Scale*.

Curtin, R. R., Cline, J. R., Slagle, N. P., March, W. B., Ram, P., Mehta, N. A., and Gray, A. G. (2012). MLPACK: A Scalable C++ Machine Learning Library. *CoRR*, abs/1210.6293.

Curtin, R. R., Ram, P., and Gray, A. G. (2013). Fast Exact Max-Kernel Search. *SIAM International Conference on Data Mining*.

Dasgupta, S. and Freund, Y. (2008). Random projection trees and low dimensional manifolds. In *Proc. of ACM Symp. on Theory of Comp.*

Datar, M., Immorlica, N., Indyk, P., and Mirrokni, V. (2004). Locality-sensitive Hashing Scheme based on p-stable Distributions. In *Proceedings of the 12th Annual Symposium on Computational Geometry*. ACM.

Dror, G., Koenigstein, N., Koren, Y., and Weimer, M. (2011). The Yahoo! Music Dataset and KDD-Cup'11. *Journal Of Machine Learning Research*.

Freund, Y., Dasgupta, S., Kabra, M., and Verma, N. (2007). Learning the structure of manifolds using random projections. *Advances in Neural Information Processing Systems*.

Friedman, J. H., Bentley, J. L., and Finkel, R. A. (1977). An Algorithm for Finding Best Matches in Logarithmic Expected Time. *ACM Trans. Math. Softw.*

Fukunaga, K. and Nagendra, P. M. (1975). A Branch-and-Bound Algorithm for Computing k-Nearest-Neighbors. *Transactions on Computing, IEEE*.

Gionis, A., Indyk, P., and Motwani, R. (1999). Similarity Search in High Dimensions via Hashing. *Proceedings of the 25th International Conference on Very Large Data Bases*.

Gong, Y. and Lazebnik, S. (2011). Iterative Quantization: A Procrustean Approach to Learning Binary Codes. In *Computer Vision and Pattern Recognition, IEEE Conference on*. IEEE.

Gonzalez, T. (1985). Clustering to minimize the maximum intercluster distance. *Theoretical Computer Science*, 38:293–306.

Goyal, N., Lifshits, Y., and Schütze, H. (2008). Disorder inequality: a combinatorial approach to nearest neighbor search. In *Proceedings of the International Conference on Web search and web data mining*. ACM.

Gray, A. G. and Moore, A. W. (2000). '*N*-Body' Problems in Statistical Learning. In *Advances in Neural Info. Proc. Systems 13*.

Hammersley, J. M. (1950). The Distribution of Distance in a Hypersphere. *Annals of Mathematical Statistics*, 21:447–452.

Har-Peled, S. (2001). A replacement for voronoi diagrams of near linear size. In *Foundations of Computer Science, 2001. Proceedings. 42nd IEEE Symp. on*. IEEE.

Har-Peled, S. and Mendel, M. (2005). Fast construction of nets in low dimensional metrics, and their applications. In *Proceedings of the twenty-first annual Symp. on Computational geometry*. ACM.

He, J., Kumar, S., and Chang, S. (2012). On the Difficulty of Nearest Neighbor Search. In *Proceedings of the International Conference on Machine Learning*.

Hellerstein, J. M., Haas, P. J., and Wang, H. J. (1997). Online aggregation. In *ACM SIGMOD Record*, volume 26, pages 171–182. ACM.

Indyk, P. and Motwani, R. (1998). Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *Proceedings of the 30th annual ACM Symp. on Theory of Comp.*

Johnson, W. and Lindenstrauss, J. (1984). Extensions of lipschitz mappings into a hilbert space. *Contemporary mathematics*, 26(189-206):1–1.

Joly, A. and Buisson, O. (2011). Random Maximum Margin Hashing. In *Computer Vision and Pattern Recognition (CVPR), IEEE Conference on*. IEEE.

Karger, D. R. and Ruhl, M. (2002). Finding Nearest Neighbors in Growth-Restricted Metrics. *Proceedings of ACM Symposium on Theory of Computing*.

Kim, S., Li, F., Lebanon, G., and Essa, I. (2011). Beyond Sentiment: The Manifold of Human Emotions. *Arxiv preprint* `arXiv:1202.1568`.

Klaas, M., Lang, D., and de Freitas, N. (2005). Fast Maximum-a-posteriori Inference in Monte Carlo State Spaces. In *Artificial Intelligence and Statistics*.

Kleinberg, J. (1997). Two algorithms for nearest-neighbor search in high dimensions. In *Proceedings of the twenty-ninth annual ACM Symp. on Theory of computing*. ACM.

Krauthgamer, R. and Lee, J. R. (2004). Navigating Nets: Simple Algorithms for Proximity Search. *Proceedings of the 15th Annual ACM-SIAM Symposium on Discrete Algorithms*.

Kulis, B. and Grauman, K. (2009). Kernelized Locality-sensitive Hashing for Scalable Image Search. In *IEEE 12th International Conference on Computer Vision*.

Kushilevitz, E., Ostrovsky, R., and Rabani, Y. (1998). Efficient search for approximate nearest neighbor in high dimensional spaces. In *Proceedings of the thirtieth annual ACM Symp. on Theory of computing*. ACM.

LeCun, Y. (2000). MNIST dataset. http://yann.lecun.com/exdb/mnist/.

Leslie, C., Eskin, E., and Noble, W. (2002). The Spectrum Kernel: A String Kernel for SVM Protein Classification. In *Proc. of the Pacific Symp. on Biocomputing*.

Li, Z., Ning, H., Cao, L., Zhang, T., Gong, Y., and Huang, T. S. (2011). Learning to Search Efficiently in High Dimensions. In *Advances in Neural Info. Proc. Systems 24*.

Lifshits, Y. (2009). Combinatorial framework for similarity search. In *Second International Workshop on Similarity Search and Applications.*, pages 11–17. IEEE.

Lifshits, Y. and Zhang, S. (2009). Combinatorial algorithms for nearest neighbors, near-duplicates and small-world design. In *Proceedings of the 20th Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 318–326. Society for Industrial and Applied Mathematics.

Liu, T., Moore, A. W., Gray, A. G., and Yang, K. (2005). An Investigation of Practical Approximate Nearest Neighbor Algorithms. In *Advances in Neural Info. Proc. Systems 17*.

Lloyd, S. (1982). Least squares quantization in pcm. *Information Theory, IEEE Transactions on*, 28(2):129–137.

Lupton, R., Gunn, J., Ivezic, Z., Knapp, G., Kent, S., and Yasuda, N. (2001). The SDSS Imaging Pipelines. *Arxiv preprint astro-ph/0101420*.

Maneewongvatana, S. and Mount, D. (2002). Analysis of approximate nearest neighbor searching with clustered point sets. *Data structures, near neighbor searches, and methodology: fifth and sixth DIMACS implementation challenges: papers related to the DIMACS challenge on dictionaries and priority queues (1995-1996) and the DIMACS challenge on near neighbor searches (1998-1999)*.

McNames, J. (2001). A fast nearest-neighbor algorithm based on a principal axis search tree. *IEEE Transactions on Pattern Analysis and Machine Intelligence*.

Moore, A. (2000). The Anchors Hierarchy: Using the triangle inequality to survive high dimensional data. In *Proc. of Conf. on Uncertainty in Artificial Intelligence*.

Müller, K., Smola, A., Rätsch, G., Schölkopf, B., Kohlmorgen, J., and Vapnik, V. (1997). Predicting Time Series with Support Vector Machines. *Intl. Conf. on Artificial Neural Networks*.

Nister, D. and Stewenius, H. (2006). Scalable Recognition with a Vocabulary Tree. In *Computer Vision and Pattern Recognition, IEEE Computer Society Conference on*. IEEE.

Norouzi, M. and Fleet, D. J. (2011). Minimal Loss Hashing for Compact Binary Codes. *Proceedings of the 28th International Conference on Machine Learning*.

Omohundro, S. M. (1989). Five Balltree Construction Algorithms. Technical report, International Computer Science Institute.

Preparata, F. P. and Shamos, M. I. (1985). *Computational Geometry: An Introduction*. Springer.

Pugh, W. (1990). Skip Lists: A Probabilistic Alternative to Balanced Trees. *Communications of the ACM*.

Raginsky, M. and Lazebnik, S. (2009). Locality-Sensitive Binary Codes from Shift-Invariant Kernels. *Advances of Neural Information Processing Systems*, 22.

Rahimi, A. and Recht, B. (2007). Random Features for Large-scale Kernel Machines. *Advances in Neural Info. Proc. Systems 20*.

Rakthanmanon, T., Campana, B., Mueen, A., Batista, G., Westover, B., Zhu, Q., Zakaria, J., and Keogh, E. (2012). Searching and mining trillions of time series subsequences under dynamic time warping. In *SIGKDD International Conference on Knowledge Discovery and Data Mining*.

Ram, P. and Gray, A. (2012). Maximum Inner-Product Search Using Cone Trees. In *SIGKDD International Conference on Knowledge Discovery and Data Mining*.

Ram, P. and Gray, A. G. (2013). Efficient Nearest-Neighbor Search and Classification in High Dimensions via Rank-Approximation.

Ram, P., Lee, D., and Gray, A. G. (2012). Nearest-neighbor Search on a Time Budget via Max-Margin Trees. In *SIAM International Conference on Data Mining*.

Ram, P., Lee, D., March, W., and Gray, A. (2009a). Linear-time Algorithms for Pairwise Statistical Problems. In *Advances in Neural Info. Proc. Systems 22*.

Ram, P., Lee, D., Ouyang, H., and Gray, A. G. (2009b). Rank-Approximate Nearest Neighbor Search: Retaining Meaning and Speed in High Dimensions. In *Advances in Neural Info. Proc. Systems 22*.

Salakhutdinov, R. and Hinton, G. (2007). Learning a Nonlinear Embedding by Preserving Class Neighbourhood Structure. In *AI and Statistics*.

Sproull, R. (1991). Refinements to Nearest-Neighbor Searching in k-dimensional Trees. *Algorithmica*, 6(1):579–589.

Tenenbaum, J. B., Silva, V., and Langford, J. (2000). A Global Geometric Framework for Nonlinear Dimensionality Reduction. *Science*, 290(5500):2319–2323.

Torralba, A., Fergus, R., and Freeman, W. (2008). 80 Million Tiny Images: A large data set for nonparametric object and scene recognition. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*.

Verma, N., Kpotufe, S., and Dasgupta, S. (2009). Which spatial partition trees are adaptive to intrinsic dimension? In *Proceedings of the Twenty-Fifth Conf. on Uncertainty in Artificial Intelligence*.

Wang, J., Kumar, S., and Chang, S. (2010). Semi-Supervised Hashing for Scalable Image Retrieval. In *Computer Vision and Pattern Recognition (CVPR), IEEE Conference on*. IEEE.

Weiss, Y., Torralba, A., and Fergus, R. (2008). Spectral Hashing. Advances of Neural Information Processing Systems.

Xu, L., Neufeld, J., Larson, B., and Schuurmans, D. (2005). Maximum margin clustering. *Advances in neural information processing systems*, 17:1537–1544.

Zhao, B., Wang, F., and Zhang, C. (2008). Efficient maximum margin clustering via cutting plane algorithm. In *The 8th SIAM Intl. Conf. on Data Mining*, pages 751–762.

VITA

Parikshit Ram was born in Durgapur, India, and, through a string of good fortune, found himself in the Georgia Institute of Technology. He attended the School of Computational Science and Engineering in the College of Computing, Georgia Tech, and is a member of the FASTlab which focuses on developing fundamental algorithms and statistical tools for machine learning and data mining. Pari joined Georgia Tech in 2007 after completing his BS and MS in Mathematics and Computing in the Department of Mathematics at the Indian Institute of Technology, Kharagpur.