# HIGH-DIMENSIONAL DATA INDEXING WITH APPLICATIONS

by

Michael Arthur Schuh

A dissertation submitted in partial fulfillment
of the requirements for the degree

of

Doctor of Philosophy

in

Computer Science

MONTANA STATE UNIVERSITY
Bozeman, Montana

July 2015

# DEDICATION

*To my parents...*

*Who taught me that above all else, sometimes you just need to have a little faith.*

# ACKNOWLEDGEMENTS

It was six years ago that I first met Dr. Rafal Angryk in his office in Bozeman, Montana, and I have been thankful ever since for the opportunity and privilege to work as one of his students. As an advisor, he motivated me to pursue my goals and maintained the honesty to critique me along the way. As a mentor, his wisdom to focus my energy towards (albeit still many) worthwhile pursuits, while driving me to finish each task to completion, has impacted my life in immeasurable ways beyond academics. I also owe a deep thank you to Dr. Petrus Martens for his research guidance and motivational enthusiasm towards science.

I am grateful to the many people who have helped me during my time at Montana State University. A special thanks to: Dr. Rocky Ross, Ms. Jeanette Radcliffe, Mr. Scott Dowdle, Ms. Shelly Shroyer, as well as former graduate student labmates: Dr. Tim Wylie, Dr. Karthik Ganesan Pillai, and Dr. Juan Banda, all of whom made my life a little bit easier and much more enjoyable. I would be remiss if I did not thank my undergraduate advisors Dr. Tom Naps and Dr. David Furcy for recognizing interests in computer science and pushing me towards graduate school.

I will be forever grateful to my parents, Arthur L. Schuh Jr. and Cheryl A. Schuh, for the countless opportunities they gave me to experience the world and better myself. Last but not least, I wish to sincerely thank all of my friends and extended family for their endless support and encouragement over the years. Nobody should go it alone.

## Funding Acknowledgement

This work was supported in part by two NASA Grant Awards: 1) No. NNX09AB03G, and 2) No. NNX11AM13A.

TABLE OF CONTENTS

TABLE OF CONTENTS – CONTINUED

TABLE OF CONTENTS – CONTINUED

# LIST OF TABLES

Table Page

# LIST OF FIGURES

Figure Page

LIST OF FIGURES – CONTINUED

Figure Page

LIST OF FIGURES – CONTINUED

Figure Page

# LIST OF ALGORITHMS

Algorithm Page

# ABSTRACT

The indexing of high-dimensional data remains a challenging task amidst an active and storied area of computer science research that impacts many far-reaching applications. At the crossroads of databases and machine learning, modern data indexing enables information retrieval capabilities that would otherwise be impractical or near impossible to attain and apply. One such useful retrieval task in our increasingly data-driven world is the $k$-nearest neighbor ($k$-NN) search, which returns the $k$ most similar items in a dataset to the search query provided. While the $k$-NN concept was popularized in every-day use through the sorted (ranked) results of online text-based search engines like Google, multimedia applications are rapidly becoming the new frontier of research.

This dissertation advances the current state of high-dimensional data indexing with the creation of a novel index named ID* ("ID Star"). Based on extensive theoretical and empirical analyses, we discuss important challenges associated with high-dimensional data and identify several shortcomings of existing indexing approaches and methodologies. By further mitigating against the negative effects of the curse of dimensionality, we are able to push the boundary of effective $k$-NN retrieval to a higher number of dimensions over much larger volumes of data. As the foundations of the ID* index, we developed an open-source and extensible distance-based indexing framework predicated on the basic concepts of the popular iDistance index, which utilizes an internal $B^+$-tree for efficient one-dimensional data indexing. Through the addition of several new heuristic-guided algorithmic improvements and hybrid indexing extensions, we show that our new ID* index can perform significantly better than several other popular alternative indexing techniques over a wide variety of synthetic and real-world data.

In addition, we present applications of our ID* index through the use of $k$-NN queries in Content-Based Image Retrieval (CBIR) systems and machine learning classification. An emphasis is placed on the NASA sponsored interdisciplinary research goal of developing a CBIR system for large-scale solar image repositories. Since such applications rely on fast and effective $k$-NN queries over increasingly large-scale and high-dimensional datasets, it is imperative to utilize an efficient data indexing strategy such as the ID* index.

# CHAPTER 1

# INTRODUCTION

Big data is everywhere, and massive image collections are quickly becoming a ubiquitous aspect of life. Countless examples exist in every facet of society, from smartphone cameras on every individual, to weather and traffic cameras around a city, security cameras in schools and businesses, and medical databases of mammogram x-rays and other radiography images, to even national weather and military surveillance satellites, and international scientific missions to capture images of the Sun and beyond. Clearly, there is already no shortage of image data available.

The modern world is composed of an ever-increasing amount of large-scale and high-dimensional data. While collecting and storing this data is becoming more routine, efficiently indexing and retrieving it is still an important concern. A frequently desired but costly retrieval task on these datasets is $k$-nearest neighbor ($k$-NN) search, which returns the $k$ most similar records to any given query record. This is especially useful in growing multimedia domains, such as photo, audio, and video, where the data objects are often represented by high-dimensional feature vectors for content-based retrieval. Popular examples already common to the public include music recommendation services, visual shopping search websites, and even Google's relatively new "search by image" feature, which lets a user query with an example image instead of using traditional text keywords, very much like a full-fledged Content-Based Image Retrieval (CBIR) system.

The $k$-NN query has far-reaching applications in information retrieval, from highly specialized user-driven CBIR-like systems in multimedia databases to more general data-driven decision support systems using machine learning classification. The ex-

amples are as numerous as the data sources. Doctors refer to similar x-ray images for tumor diagnoses, surveillance satellites automatically detect and identify possible objects of interest, and even consider yourself attempting to search for old pictures of a friend in your personal image library spanning thousands of photos over several decades. In all such large-scale applications, efficient data indexing is crucial to support the necessary $k$-NN queries in a timely manner.

## 1.1 Motivation

The ability to efficiently index and retrieve data has become a silent backbone of modern society, and it defines the capabilities and limitations of practical data usage. While a database management system (DBMS) is highly optimized for a few dimensions and equality-based retrieval, the traditional indexing algorithms (*e.g.*, the B-tree and R-tree families) degrade quickly as the number of dimensions increases. Compounding the issue, $k$-NN queries can be quite useful and effective in these high-dimensional spaces, increasing their demand and pushing the limits on an already difficult task. Often in these circumstances, without highly specialized indexes, the most efficient retrieval method available is actually a brute force sequential scan of every single record in the database.

Many algorithms have been proposed in the past with limited success for high-dimensional indexing beyond 20-30 dimensions, and this general problem is commonly referred to as *the curse of dimensionality* [1]. In practice, this issue is often mitigated by applying dimensionality reduction techniques before using popular multi-dimensional indexing methods, and sometimes even by adding application logic to combine multiple independent indices or requiring user involvement during search. However, modern applications are increasingly employing highly-dimensional tech-

niques to effectively and accurately represent massive data coming from complex real-life systems. For example, 128-dimensional features produced by the Scale-Invariant Feature Transform (SIFT) algorithm [2] are quite popular and effective in computer vision applications, bringing a real challenge to large-scale CBIR systems where it is of critical importance to be able to comprehensively index all dimensions for a unified similarity-based retrieval model.

With the steady march forward of technological progress, these already large-scale image repositories (and other multimedia archives) will only continue to grow in size and complexity. The modern perspective of this new-found scientific paradigm welcomes greater opportunities for knowledge discovery from data (KDD) never before considered possible [3]. However, only through effective indexing can we attain these possibilities and avoid drowning in the deluge of data.

## 1.2 Overview

The work in this dissertation focuses on high-dimensional data indexing to better facilitate $k$-NN queries for use in application. Through interdisciplinary research motivations to create a large-scale CBIR system for solar images, we advance the frontier of data indexing research with the creation of the ID* index, specifically suited for exact $k$-NN queries in high-dimensional and large-scale datasets.

We begin with a background in Chapter 2. Here we briefly review important concepts necessary to our work, as well as other related indexing techniques to better understand the general research landscape. Special attention is focused on the practical difficulties of high-dimensional data through several experimental results highlighting key aspects of the curse of dimensionality.

Chapter 3 presents the ID* indexing algorithm and our extensive research and development efforts. We start with an overview of the fundamental algorithm components and then establish a performance benchmark and best practices for use over a variety of algorithmic options. These initial findings led us to identify several directions of continued research and development, such as indexing extensions and algorithmic improvements, which are detailed in the remainder of this chapter.

In chapter 4, we present several applications of the ID* index through the use of $k$-NN queries in Content-Based Image Retrieval (CBIR) systems and machine learning classification. Here we focus on our NASA sponsored interdisciplinary research goal of developing a CBIR system for large-scale solar image repositories.

Lastly, we close with conclusions in Chapter 5. Here we also highlight several aspects of future work, such as ongoing improvements to the ID* algorithms and related heuristics, more optimal clustering for the sake of indexing, and the integration of ID* into an existing Database Management System (DBMS) for use in applications.

# CHAPTER 2

# BACKGROUND

In this chapter we present the basic concepts necessary to understand the work presented in this dissertation. We begin with an overview of essential mathematical foundations and provide the basic context of the research problem and a detailed explanation of what our objective is and the difficulties associated with it. Then we briefly review several important and related indexing techniques that define the research landscape of our specific problem scope.

## 2.1 Notation

We refer the reader to Table 2.1 for an overview of the mathematical notations and symbol usage used throughout this dissertation. In general, parameter-like variables are capital letters, such as $N$ data points and $M$ reference points with $D$ dimensions. Subscripted variables are usually an item within a set, such as each dimension $d_i$ in the set of dimensions $\mathcal{D}$, where $i = 1, \ldots, D$. Typically $p$ and $q$ are used as example data and query points, respectively.

Note the overloaded use of the variable "k". While used in the name of $k$-means as the number of clusters ($k$), we use $M$ as this term for our work. Unless otherwise explicitly stated, $k$ refers to the number of nearest neighbors to return from a $k$-NN query. With respect to the size of datasets $N$, we often use a non-italicized k to represent $N$ in the thousands, *e.g.*, 1,000 = 1k. We also note the possible confusion over the use of $D$ and "D", where the former represents the variable with some value and the latter is only an abbreviation of "dimension". For example, consider the

Table 2.1: Mathematical notation

| Symbol | Definition |
|---|---|
| $D$ | Dimensionality of dataspace |
| $N$ | Number of data points in the dataset |
| $M$ | Number of reference points |
| $\mathcal{D}$ | Set of dimensions |
| $d_i$ | The $i$th dimension |
| $\mathcal{P}$ | Set of partitions |
| $P_i$ | The $i$th partition |
| $\mathcal{O}$ | Set of reference points |
| $O_i$ | The $i$th reference point |
| $distmax_i$ | The farthest distance value assigned to $P_i$ |
| $p$ | A data point |
| $q$ | A query point |
| $r$ | The radius of a query sphere |
| $r_\Delta$ | The query radius increment value during $k$-NN search |
| $k$ | Number of nearest neighbors to retrieve |
| $S$ | The set of nearest neighbors |
| $distmax_S$ | The farthest distance of any object in $S$ |
| $dist()$ | A metric distance function for two points |

difference between $2D$ number of reference points and a 2D dataspace. Care was taken to make sure the various uses are as clear as possible in context.

## 2.2 Preliminaries

In this section we describe the mathematical foundations of the space in which we index data. We also present a brief look at several common types of searches performed by indexes in these spaces, including the $k$-NN query.

### 2.2.1 Metric Spaces

We define our general space as the $D$-dimensional unit hypercube such that the set of dimensions is $\mathcal{D} = \{d_1, \ldots, d_D\}$, where each dimension $d_i$ is a real value in the

closed range $[0.0, 1.0]$. A metric space can be defined as a set of objects for which the pair-wise distances between all objects are defined. Typically these objects live in some $D$-dimensional real-valued space with a set of $D$ features (or attributes) which comprise their place and respective distance from other objects in the dataspace. Given a finite metric space $(S, dist)$, and objects $x, y, z \in S$, the distance function $dist$ must satisfy the following properties.

$$dist(x, y) \geq 0 \qquad \text{(non-negativity)} \tag{2.1}$$

$$dist(x, y) = 0 \iff x = y \qquad \text{(identity)} \tag{2.2}$$

$$dist(x, y) = dist(y, x) \qquad \text{(symmetry)} \tag{2.3}$$

$$dist(x, z) \leq dist(x, y) + dist(y, z) \qquad \text{(triangle inequality)} \tag{2.4}$$

While some of these properties can be relaxed for certain situations, we assume all hold for datasets used in this work. Note that the underlying feature space may not be explicitly given, and only the distance function is essential here, termed the *metric* on the set. An abundance of various metrics can be used, such as any $L^p$*-norm*, but for our purposes we will use the well known Euclidean distance function ($L^2$*-norm*) unless otherwise stated. The Euclidean norm satisfies the above properties and is also translation and rotation invariant. It is also a generalized case of the magnitude (or length) of a vector in space, which in this case is a feature vector in the dataspace. Given two $D$-dimensional points $x$ and $y$, the Euclidean distance between them is defined in Equation 2.5, where $i$ represents the dimensional index in the feature vectors.

$$dist(x, y) = \sqrt{\sum_{i=1}^{D} (y_i - x_i)^2} \tag{2.5}$$

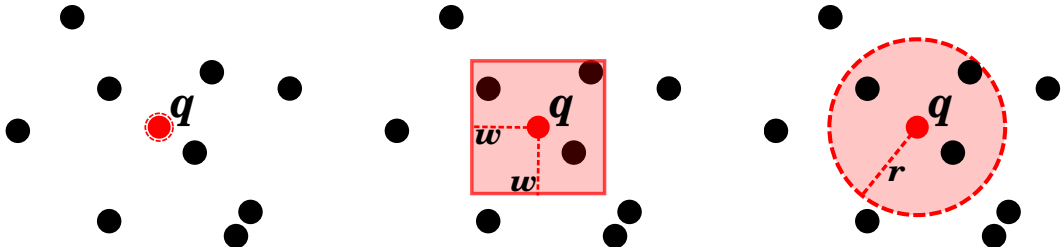
The most useful property of metric spaces is the triangle inequality relationship, which enables trigonometric pruning. Essentially, this enables a bounding of the exact distance between data points by using other already known distances. Often, the lower bound of this distance is sufficient enough to prune a data point from search without ever requiring an explicit distance calculation.

For example, suppose we are looking for the closest neighbor to $x$ (1-NN) and considering $y$ as a candidate with an existing current nearest neighbor distance as $dist_1$. If we already know the distances to $z$ for both $x$ and $y$, then we can rearrange Equation 2.4 such that $dist(x, y) \geq dist(x, z) - dist(y, z)$. If $dist_1 \leq dist(x, z) - dist(y, z)$, then we can guarantee $y$ is not the nearest neighbor of $x$ and prune the candidate from further consideration. As an aside, if $dist(x, y) = dist_1$, then $x$ is equally as close as the current nearest neighbor. In practice these ties can be solved arbitrarily, such as keeping the first found for efficiency sake, or keeping the object with the lowest index ID for query consistency and result validation.

### 2.2.2 Types of Search Queries

Here we briefly outline several common types of searches, or queries, performed on data in a metric space. We focus most of our attention on k-nearest neighbor ($k$-NN) search, but it is important to understand the relation to other query types as well. An example of each type of query is presented in Figure 2.1.

2.2.2.1 Point. A point query $q = \{q_1, ..., q_D\}$ is a point in $D$-dimensional space. We want to retrieve all objects, if any exist, from the dataspace that exactly match all




(a) (b) (c)

Figure 2.1: Example search types: (a) point query, (b) range query with per-dimensional range width $w$, and (c) $k$-NN query with radius $r$.

dimensional values specified. This type of query can be efficiently processed through a number of different methods because it requires every dimensional value to exactly match the query's value. In other words, data points can be guaranteed to not belong to the result set as soon as one conflicting dimensional value is found. An example is shown in Figure 2.1a.

2.2.2.2 Range. An orthogonal (or axis-aligned) range query in a $D$ dimensional space is the set $\mathcal{R} = \{R_1, \ldots R_D\}$. Each range $R_i$ is a tuple of the minimum and maximum values that define the range over that dimension, *i.e.*, $R_i = [v_{min}, v_{max}]$, so the range of dimension $i$ is $[v_{min}, v_{max}]$. An example is shown in Figure 2.1b. Here we see each dimension has the same range width $w$, resulting in the range query $\mathcal{R} = \{R_1, R_2\}$, where $R_i = [q_i - w, q_i + w]$.

We also include the spherical range query, which is defined as a query point $q$ with a radius $r$ that returns all objects that are within the ball $B(q, r)$, which is defined as $B(q, r) = \{x | dist(q, x) \leq r\}$. In other words, all objects that are no farther than $r$ from $q$ are returned. This $D$ dimensional hypersphere shape is a fundamental concept for distance-based orientation, such as nearest neighbor queries. The main

distinction between a spherical range query and a nearest neighbor query is the static and pre-defined radius of the range query.

2.2.2.3 Nearest Neighbor. A nearest neighbor query attempts to find all data objects whose features are deemed similar to the given query. This type of query requires a distance function to determine the similarity between two data points in space. This is much more costly than a point or range query, because it must explicitly analyze all dimensions of the space (or already have a metric defined between all points).

A $k$-NN query is defined as a query point $q$ with a radius $r$, which returns the $k$-nearest objects to $q$ within the ball $B(q, r)$, based on a specified distance metric $dist()$. The difficulty of a $k$-NN query is not knowing before-hand how large of a radius around the query point needs to be searched to find the specified $k$ nearest results. Typically this involves an iterative search approach where the radius is incrementally expanded until the query is satisfied.

An example $k$-NN query is shown in Figure 2.1c. Here we emphasize the uncertain length of radius $r$, which currently encompasses the three nearest neighbors from $q$. If $k > 3$ for this $k$-NN query, then $r$ must be increased to find the next nearest neighbor(s) to satisfy the query.

## 2.3 High-Dimensional Data

Traditional databases focus on low-dimensional data indexing. To put into perspective the prohibitive costs of traditional methods on high-dimensional data, we present Figure 2.2, which shows the total time (in log-based minutes) to create and test (calculate pair-wise distances) synthetic data on a modern CPU (3.0GHz quad-core CPU with 16GB RAM, where only one core is utilized and RAM is not a limiting

factor). Both datasets are uniform, where the first is only 1,000 (1k) points, and the second is 10,000 (10k) points. We can see that the time to generate the datasets is essentially nil, but the time to perform all pair-wise distance calculations dramatically increases with dimensionality. Of course this is expected by theory, but note that practically it took almost 90 minutes to check 10k data points in $64D$, whereas much of our work is focused on significantly larger datasets in much higher dimensions.

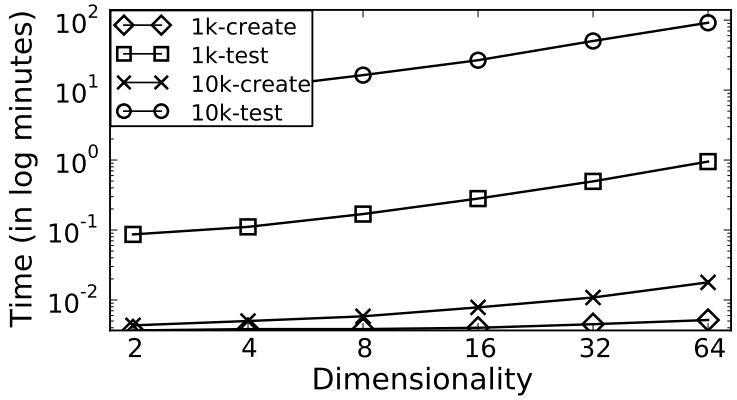



Figure 2.2: The time taken on uniform datasets to generate (create) the data and calculate (test) all pair-wise distances.

### 2.3.1 Curse of Dimensionality

Many typical database and information retrieval tasks are made more difficult in higher dimensional spaces. These challenges are collectively referred to as the *curse of dimensionality*, which was originally coined by Bellman [1] to indicate that the number of samples needed to estimate an arbitrary function with a given level of accuracy grows exponentially with the number of variables (dimensions) that it comprises [4]. An interesting related fact of this result is that real world high-dimensional data is invariably not uniform, because the sample (dataset) is much smaller than $O(v^D)$, where $v$ is a constant representing the possible values a data point can have in each dimension, and $D$ is the total dimensions.

For example, if we let $v = 2$, then at $D = 20$ dimensions we would already require over 1 million data points ($2^{20}$) to uniformly cover the dataspace. Clearly, the feasibility of uniform datasets beyond 20-30 dimensions is impractical and highly unlikely. In practice, our synthetic data has a precision of four digits, which in only $2D$ would still require 1 million data points ($1000^2$) to fully blanket the dataspace—although in this case, a reasonable sample could likely be uniform.

### 2.3.2 Similarity Search

All three types of queries discussed in the previous section are examples of similarity search, whereby point and range queries require the distance difference (similarity) to be zero. The curse of dimensionality directly affects the ability to properly perform similarity searches, as the distance to the nearest and farthest neighbors of a given query point tend to converge as the dimensionality of the underlying dataspace increases [5]. An example of this can be seen in Figure 2.3 where we plot the min, mean, and max pair-wise distances between all data points in a uniform 10k dataset—essentially calculating the metric. Note that we normalized all distances by the maximum distance of the unit dataspace ($\sqrt{D}$), which is represented in the figure by the "space" line.

### 2.3.3 Query Selectivity

Another related problem with high-dimensional spaces is the selectivity of a query over the dataspace. This can best be described in the context of orthogonal range queries, where the selectivity of a rectangular query $q$ is the ratio of volume of $q$ to the volume of the dataspace. Since we assume the dataspace is a unit hypercube, the entire dataspace volume is always equal to 1.0. The volume of $q$ can be calculated

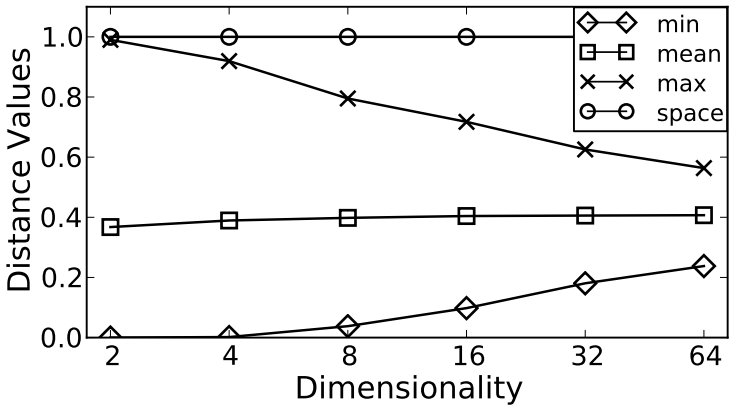



Figure 2.3: The curse of dimensionality in a uniform dataset.

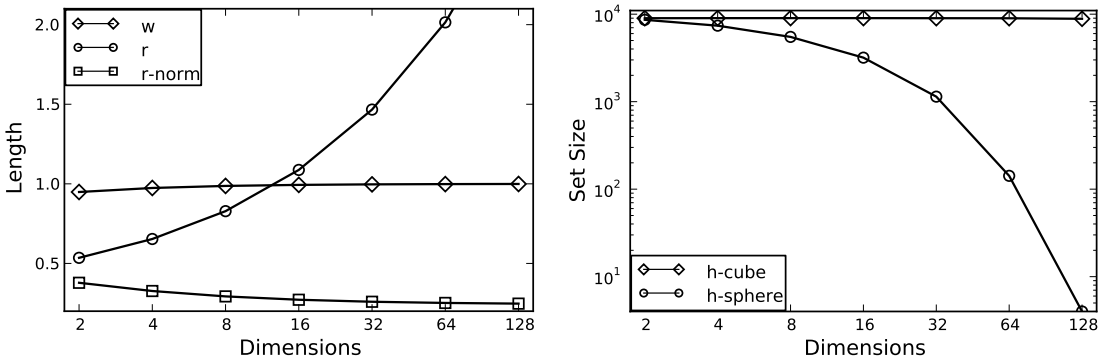
by multiplying the individual dimensional side-lengths together. If we assume each side-length is an equal length $w$, then given dimensionality $D$, the volume of $q$ is $w^D$.

For example, if we let $w = 0.9$ and $D = 10$, then our resultant query $q$ will cover just under 35% of the dataspace. Further, with $D = 100$, our query selectivity is nearly nil, at 0.002% of the dataspace. Therefore, to maintain any reasonable query selectivity in high dimensions, the side-length of said query becomes nearly the entire range of the dataspace. This is a fundamental reason typical per-dimensional space-based partitioning methods break down as dimensionality increases.

We highlight two other factors related to query selectivity. The first is the relation between hypersphere and hypercube volumes, which represent the two types of range queries. If we set $w = r$, we note that as dimensionality increases, the sphere represents proportionally less volume in the dataspace. As $D$ approaches infinity, this ratio approaches 0. In other words, more volume is being accumulated in the corners of the dataspace where the hypersphere does not reach.

In Figure 2.4a, we present the hypercube side-length (w) and the hypersphere radius (r) to create their respective shape containing 90% of the dataspace volume, with $q$ centered at 0.5 over a varying number of dimensions. Although difficult to

see, we note the value of $w$ is increasingly approaching 1.0. We also present the radius value normalized over the maximum distance in the $D$-dimensional dataspace (r-norm), which shows relatively less total radial coverage of the dataspace as dimensionality increases, likely due to the "stretching" effect of the corners of the underlying dataspace hypercube.




(a) (b)

Figure 2.4: Comparing hypercubes and hyperspheres over dimensions.

A related consequence of these high-dimensional dataspace effects is the general movement of all data points to the edges and corners of the dataspace. We show this empirically in Figure 2.4b, which presents the result set size for the hypercube (h-cube) and hypersphere (h-sphere) queries just explained. We can see that although the hypersphere is 90% of the volume of a the dataspace, the uniform dataset returns increasingly fewer results – from 8,620 in 2D to only 4 in 128D. On the other hand, the hypercube consistently returns approximately 90% of the dataset.

## 2.4 Related Index Techniques

Here we discuss several indexing methods related to the work we present in the later chapters of this dissertation. We begin with two of the most fundamental,

and crucial to understand, indexing techniques that laid the foundations for much of the indexing-related research of the past decades. We also highlight several of the important research directions that led to the work that continues to this day.

### 2.4.1 Sequential Search

We should first include a brief note about sequential search, or *sequential scan*, which is itself not an indexing technique, but a simple brute force examination of every object in the dataset in a linear manner. While trivial to implement, and naïve in its methodology, we will come to find later that in high-dimensional data, sequential scan is quite often the most efficient retrieval method available. This is because most indexing techniques break down as the dimensionality increases and eventually they require all nodes to be individually accessed during traversal of the search tree. This results in an increased time due to additional disk seeks for individual records compared to sequential search which utilizes a more efficient I/O of reading contiguous pages on disk.

While there exists research on more efficient methods to perform sequential search, such as the VA-file [6], these are beyond the scope of our discussion. It is also beneficial in our case to use the standard sequential search algorithm as a definitive worst-case benchmark. Note we often abbreviate sequential search (SS) for experimental presentation and discussion. We also use sequential "scan" interchangeably with "search" throughout the dissertation, which could be considered a more descriptive name for the method, but otherwise has identical meaning.

### 2.4.2 Low Dimensions

Several foundational indexing techniques were developed to deal with low-dimensional spaces quite effectively. Here we mention two of the most famous and popular types of methods, both of which are still integral in many applications.

2.4.2.1 B-tree Family. The B-tree [7] is a balanced search tree designed for minimizing I/O operations on direct-access secondary storage devices, such as hard disks. B-trees are similar to red-black trees, except they allow more than two (binary) children. This usually large amount of node children, termed the branching factor of the tree, allows for a much shorter height of the overall tree, and therefore fewer total disk accesses during traversal of the tree. The height of an $n$ node B-tree is $O(\log n)$, where the log base is the branching factor.

This work uses an underlying $B^+$-tree [8] to facilitate our one-dimensional distance-based indices. The main improvement of the $B^+$-tree is that all data objects are stored solely in the leaves of the tree, rather than intermixed in internal nodes like a traditional B-tree. By only keeping keys and child pointers in the internal nodes, it maximizes the branching factor and therefore minimizes the overall tree height and traversal costs. Additionally, it also allows us to visit neighboring leaf nodes through a doubley-linked list without missing any data objects stored in the tree.

2.4.2.2 R-tree Family. While the one-dimensional $B^+$-tree is foundational to the modern relational DBMS, most real-life data has many dimensions (attributes) that would be better indexed together than individually. Mathematics has long-studied the partitioning of multi-dimensional metric spaces, most notably Voronoi Diagrams

and the related Delaunay triangulations [9], but these theoretical solutions can be too complex for applications in big data settings.

The R-tree [10] was originally developed as a multi-dimensional spatial access method to suit a variety of real world applications. Similar to B-trees, it is a balanced search tree designed for disk-based data retrieval, where each internal node contains a number of key-pointer pairs and each leaf node contains a number of individual object representations. The essence of the R-tree, where the R stands for "rectangle," is to use Minimum Bounding Rectangles (MBRs) whereby each level of the tree working up to the root contains the MBR coverage of its children. Often if the objects are complex polygons, the leaves will contain only the MBR of said object, with an additional pointer to the actual object on disk for final retrieval and comparison. Extended versions of R-trees, called R*-trees [11], enhanced search efficiency by minimizing MBR overlap. There exist many other extensions to improve upon the R-tree foundations and to apply the algorithms to new data types.

The biggest drawback of R-trees is the directly inverse relationship of dimensionality and branching factor. Because an MBR requires two data points in any $D$ dimension space, the cost of storing an MBR increases as the dimensionality of the space increases. This results in fewer MBRs fitting into a node (related to a page on a disk), and a decrease in tree fan-out. These trees were found to quickly degrade in performance as dimensionality increased beyond 8-10 dimensions [12, 13] and are therefore not suitable for high-dimensional data.

### 2.4.3 Higher Dimensional Approaches

A tremendous amount of algorithmic research exists in creating better indexing techniques for higher dimensional dataspaces. Much of this work was accomplished

in the late 1980's to early 2000's, when the growth of data had already surpassed the limits of prior methods. However, like all technological growth, the amount of data in that era pales in comparison to modern standards and future prospects.

Initial research pursued multiple paths to combat the curse of dimensionality. Here we note a few important works. The TV-tree [14] used a *telescoping vector* to analyze only a subset of dimensions at each level of the tree. The X-tree [15] used a *super node* concept to maintain over-full nodes that had no ideal splitting point. The SS-tree [16] (and others) used hyperspheres instead of hyperrectangles.

More recently, research has focused on creating indexing methods that define a one-way lossy mapping function from a multi-dimensional space to a one-dimensional space that can then be indexed efficiently in a standard $B^+$-tree. These lossy mappings require filter-and-refine strategies to produce exact query results, where the one-dimensional index is used to quickly and efficiently retrieve a subset of the data points as candidates (the filter step). Then each of these candidates is checked to be within the specified query region (within a distance of $r$ from point $q$) in the original multi-dimensional space (the refine step). Because checking the candidates in the actual dataspace is a costly task, the goal of the filter step is to return as few candidates as possible while retaining the exact results.

The Pyramid Technique [12] was one of the first prominent methods to effectively use this strategy by dividing up the $d$-dimensional space into $2d$ pyramids with the apexes meeting in the center of the dataspace. This was later extended by moving the apexes to better balance the data distribution equally across all pyramids [17].

For greater simplicity and flexibility, iMinMax($\theta$) [13, 18] was developed with a global partitioning line $\theta$ that can be moved based on the data distribution to create more balanced partitions leading to more efficient retrieval. The simpler transformation function also aids in faster filter-step calculations for finding candidate sets.

Lastly, note that the Pyramid Technique and iMinMax($\theta$) were designed for range queries in a multi-dimensional space, and extending them to high-dimensional $k$-NN queries is not a trivial task.

### 2.4.4 Distance-based Indexing

A number of methods have been introduced in recent years to facilitate searching by indexing objects with respect to their distances to other objects or one another. This is advantageous for many applications which may not have easily quantifiable features in a dataspace, whereby a provided distance function can gauge the similarity (or dissimilarity) between all objects in the dataset. Typically these methods assume a metric space, allowing the search to prune away portions of the dataset through basic trigonometric properties and thereby reduce overall computational costs. Two basic types of distance-based indexing were initially proposed by Uhlmann [19] based on the manner of in which the dataspace is partitioned: ball partitioning and generalized hyperplane partitioning.

A ball partitioning scheme specifies an object, often called a *vantage point*, by which all other points in the dataset are ranked by their distance to the respective object. Alternatively, generalized hyperplane partitioning specifies a set of *pivot objects* by which the distance from each pivot to all data points is determined. The pivots need not be actual data points in the dataset, and are often defined as cluster centers with hyperplane partitioning following traditional Voronoi diagram, closest-point, assignment. This can be seen as a more general version of the ball scheme, where each pivot object implicitly defines a ball of dataspace coverage based on data point assignment.

2.4.4.1 M-tree. A brief discussion of the M-tree indexing algorithm [20] is given here. Originally developed in 1997, the primary motivation of this technique was to combine the recent advancements in incremental distance-based indexing algorithms with the utility and performance of the balanced tree-based approaches. The main distinction to the other distance-based algorithms we discuss is the internal insertion and deletion algorithms that maintain this balanced tree, much like the R-tree.

2.4.4.2 iDistance. Published in 2001 and 2005, iDistance [21, 22] specifically addressed $k$-NN queries in high-dimensional space and claimed to be one of the most efficient and state-of-the-art high-dimensional indexing techniques available for exact $k$-NN search. In recent years, iDistance has been used in a number of demanding applications, including large-scale image retrieval [23], video indexing [24], mobile computing [25], peer-to-peer systems [26], and surveillance system video retrieval [27]. As the precursor to our ID* index, we discuss a few additional details here.

The basic concept of iDistance is to segment the dataspace into disjoint partitions, where all points in a specific partition are *indexed by their distance* ("iDistance") to the reference point of that partition. This results in a set of one-dimensional distance values, each related to one or more data points, for each partition that are all together indexed in a single standard $B^+$-tree. The algorithm was motivated by the ability to use arbitrary reference points (*pivot objects*) to determine the (dis)similarity between any two data points in a metric space, allowing single dimensional ranking and indexing of data points no matter what the dimensionality of the original space [21]. The algorithm also contains many adjustable parameters and run-time options, making the overall complexity and performance highly dependent on the choices selected.

Several different comparative evaluations were performed in both original iDistance works. In [22], iDistance was shown to be similar in performance to iMinMax [13] on a 30-dimensional (100K) uniform dataset and greatly outperformed it on a 30-dimensional (100K) clustered dataset. The authors also compared iDistance against the A-tree algorithm [28], which itself has been shown to out perform both the SR-tree [29] and the VA-file [6]. The results, again on a 30-dimensional clustered dataset, show iDistance is clearly more efficient than the A-tree [22]. In [21], the authors showed that in most cases iDistance greatly outperformed two popular metric based indexes, the M-tree [20] and the Omni-sequential [30], as well as the in-memory bd-tree [31] structure. Therefore, there is significant evidence that the original iDistance algorithm was already a highly-competitive indexing algorithm.

2.4.4.3 SIMP. Quite recently, the SIMP algorithm (created in 2012) was developed to combine the concepts of distance-based indexing with the benefits of additional filtering through hash-based techniques[32]. Similar to iDistance, the algorithm uses partition coverage for trigonometric pruning of candidates during search. However, the candidate list is first filtered by the intersection of multiple subspace projections of the query through polar-coordinate-mapped hash buckets. This reduction of candidates requiring distance calculations is at the expense of additional pre-processing and non-trivial storage overhead.

### 2.4.5 Approximation Methods

In our previous discussions we assumed exact results for a given type of search or query. It is worth noting that this condition can be relaxed, allowing for approximate results if desired. This is most typically seen for nearest neighbor queries, where an

error or approximation term is specified such that if the $k$-th (farthest) neighbor is at a distance of $dist_k$, then all approximate results are within $dist_k + \epsilon$ distance away. The same concept can be applied to point and range queries.

In many applications, approximate results are good enough for the given tasks and algorithms can leverage these relaxed criteria to retrieve results faster than could be done otherwise. Many works are focused on returning approximate nearest neighbors [33, 34], but these are outside the scope of exact $k$-NN retrieval, which we focus on in this dissertation.

# CHAPTER 3

# THE ID* INDEX

In this chapter we describe the ID* index. We present the overall process through a series of steps shared by many related indexing algorithms, with emphasis on the specific aspects of each step pertinent to our research and development. We begin with a high-level look at the most essential concepts of distance-based indexing: building and querying the index structure. While these fundamentals of the ID* index are similar to those of its predecessor iDistance, the same is true of most any general distance-based indexing technique.

We greatly differentiate our research and the ID* index through comprehensive experimental analyses and performance benchmarks over wide ranges of dataset characteristics and algorithmic tuning options. Based on these results, we introduce several novel indexing extensions to further segment the dataspace partitions and improve overall query performance for high-dimensional and tightly-clustered spaces. Armed with insights from the prior studies, we revisit the fundamental algorithms with detailed theoretical and experimental analysis. Several algorithmic improvements are introduced to better bound expected query performance and greatly reduce the total amount of costly distance calculations required for each query.

Lastly, we introduce our preliminary work on a new extension to reduce dataspace partition overlap during index creation. By utilizing specialized heuristic-guided outlier detection and a small sequential scan bucket to hold a set of data points outside of the index, we can greatly reduce overlap for little additional cost.

## 3.1 Foundations

The foundations of the ID* index are traced to the iDistance index, which itself originated from the GH-tree [19] and other general concepts of distance-based indexing in metric spaces [19, 35]. The goal is to partition the data in a way that allows the dimensionally-reduced (lossy) single-valued distances between objects to be both minimized in number of total calculations and maximized in total candidate pruning potential. Here we introduce the basic concepts of the ID* index.

### 3.1.1 Building the Index

We build the index up front on the entire dataset of $N$ points in $D$ dimensions. While data insertion and deletion capabilities are available, the index structure does not reorganize itself based on these actions—like a $B^+$-tree does for example.

One of the most important algorithmic options is the partitioning of the dataspace. Two general types of partitioning strategies exist: *space-based* which assumes no knowledge of the actual data, and *data-based*, which adjusts the size and location of partitions based on the data distribution. For any strategy, each partition requires a representative reference point, which may or may not also be present in the dataset. Data points are then assigned to a single partition based on the closest reference point in Euclidean distance.

Formally, we have a set of partitions $\mathcal{P} = \langle P_1, \ldots, P_M \rangle$ with respective reference points $\mathcal{O} = \langle O_1, \ldots, O_M \rangle$. We let the number of data points assigned to partition $P_i$ be denoted as $|P_i|$ and the total number of points in the dataset as $N$. After the partitions are defined, a mapping scheme is applied to create separation in the under-

lying $B^+$-tree between each partition, ensuring that any given index value represents a unique distance for exactly one partition.

A mapping scheme is required to create separation between the partitions in the underlying $B^+$-tree, ensuring any given index value represents a unique distance in exactly one partition. Given a partition $P_i$ with reference point $O_i$, the index value $y_p$ for a point $p$ assigned to this partition is defined by Equation 3.1, where $i$ is the partition index, and $c$ is a constant multiplier for creating the partition separation. While constructing the index, each partition $P_i$ records the distance of its farthest point as $distmax_i$.

$$y_p = i \times c + dist(O_i, p) \tag{3.1}$$

We can safely set $c = 2\sqrt{D}$, assuming the reference points are inside the dataspace, which is twice the maximum possible distance of two data points in the $D$-dimensional unit space, and therefore no index value in partition $P_i$ will clash with values in any other partition $P_{j \neq i}$. Importantly, we note that setting this value larger than necessary has no significant effect due to efficient leaf packing by the underlying $B^+$-tree. In other words, the index sparsely stores only values present in the data, so if large ranges of index values do not exist (whether by design or data presence), the tree will simply and implicitly ignore this empty value range.

3.1.1.1 Pre-processing. Here we discuss several pre-processing methods that are commonly used in high-dimensional data indexing. The general goal is to be better informed of the underlying data distribution and tune indexing structures accordingly for better performance. The idea is to incur a higher pre-processing cost of additional analysis on the data while building the index, with the benefit of saving time during

on-demand query tasks. The most common analysis is some form of clustering algorithm to find good locations for the partition reference points. While the underlying indexing algorithms are unchanged, the selection of reference points can greatly affect query performance, emphasizing the importance of pre-processing.

In our work we chose to use an iterative $k$-means clustering algorithm [36] for several reasons. First, it is an extremely well-studied and popular choice across many domains in practice, as well as the preferred choice of comparable indexing algorithms iDistance [21] and SIMP [32]. Second, while this particular type of clustering has its downsides, they mostly relate to the spherical distance-based nature of the resultant clusters. From our perspective, this is quite possibly an advantage, as it is the same data partitioning methodology inherent in distance-based indexing. Lastly, while true $k$-means clustering is an NP-Hard problem, we use an implementation of Lloyd's algorithm [37], which is relatively fast with a complexity of $O(nkid)$, where $n$ is the number of data points, $k$ is the number of clusters, $i$ is the number of iterations, and $d$ is the dimensionality. Therefore, the dominating factor is a simple linear growth by $n$. In general, little work has been done with other clustering algorithms.

We note that hierarchical clustering is commonly discussed in literature [4]. From a theoretical perspective, it adds an additional type of object above the "flat" set of clusters that itself only contains other clusters. The advantage is the ability to prune away entire sets of clusters with one higher-level check in the hierarchy. Unfortunately, the polynomial time complexity, generally $O(n^3)$, is infeasible for large datasets.

Many of the heuristics presented in this dissertation that guide better algorithmic choices are also reliant on scalability. Therefore, we typically start with fast yet simple distributed statistical measures and greedy decisions. As heuristics are inherently contextual, we present a more detailed looks at our methods when they are applicable

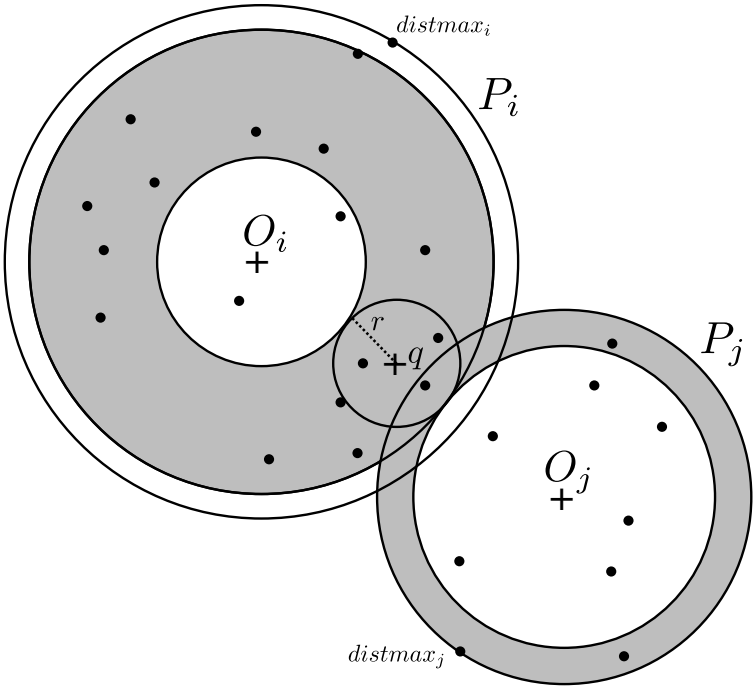



Figure 3.1: An example query search in 2D. A query sphere $q$ with radius $r$ and the searched regions (shaded) in the two overlapping partitions $P_i$ and $P_j$ defined by their reference points $O_i$ and $O_j$, and radii $distmax_i$ and $distmax_j$.

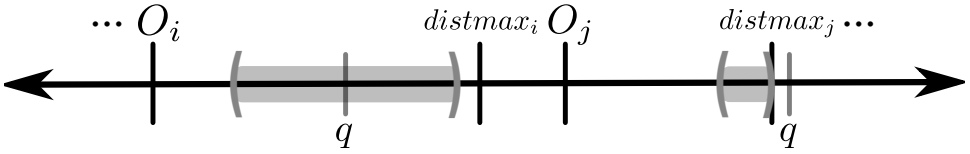



Figure 3.2: An example query search in 1D, corresponding to the 2D shaded region.

in later sections. We note these heuristics here because they are also pre-processing activities of the same nature and intent.

### 3.1.2 Querying the Index

The index should be built in such a way that the filter step returns the fewest possible candidate points without missing the true $k$-nearest neighbor points. Fewer candidates reduces the costly refinement step which must verify the true multi-dimensional

distance of each candidate from the query point. Performing a query $q$ with radius $r$ consists of three basic steps:

1. Determine the set of overlapping partitions to be searched.
2. Calculate the 1D $B^+$-tree search range for each partition in the set.
3. Retrieve the candidate points and refine by their true distances to $q$.

Figure 3.1 shows an example query point $q$ with radius $r$ contained completely within partition $P_i$ and intersecting partition $P_j$, as well as the shaded ranges of each partition that need to be searched. Figure 3.2 shows the same shaded query regions in the corresponding line plot of one-dimensional distance values. For each partition $P_i$ and its $distmax_i$, the query sphere $B(q, r)$ overlaps the partition if the distance from the edge of the query sphere to the reference point $O_i$ is less than $distmax_i$, as defined in Equation 3.6. If the query sphere does not overlap the partition, then the partition can be pruned from the search. There are two possible cases of overlap: 1) $q$ resides within $P_i$, or 2) $q$ is outside of $P_i$, but the query sphere intersects it. In the first case, the partition needs to be searched both inward and outward from the query point over the range $(q \pm r)$, whereas an intersected partition is only searched inward from the edge of the partition to the farthest point of intersection. Equation 3.3 combines both overlap cases into a single search range for each partition.

$$dist(O_i, q) - r \leq distmax_i \tag{3.2}$$

$$[dist(O_i, q) - r, MIN(dist(O_i, q) + r, distmax_i)] \tag{3.3}$$

Note that partition $P_i$ is searched inward from $q$ to $dist(O_i, q) - r$ as well as outward from $q$ to $q + r$, and partition $P_j$ is only searched inward from $distmax_j$ to

$dist(O_j, q) - r$. It should be fairly clear to see how the lossy transformation affects efficient retrieval.

3.1.2.1 Alternative Types of Search. We briefly mention satisfying the other types of search queries presented in the background. A point query is straightforward and uses the same mechanism as finding the center point of the $k$-NN query. Spherical range query is also integral to the $k$-NN query and therefore implicitly available. Axis-aligned range queries are achieved by first performing a spherical range query on the minimal hypersphere that encloses the true range query hypercube, and then applying a second filter of the per-dimensional range boundaries. While beyond the scope of this dissertation, we also developed an optimal data-based approach to facilitate range queries using an existing $k$-NN-based indexing algorithm [38].

## 3.2 Establishing a Benchmark

Our research started with the introduction of a new and open-source implementation of the original iDistance algorithm, including detailed documentation, examples, visualizations, and extensive test scripts. In [39], we methodically analyzed iDistance partitioning strategies with the goal of increasing the overall performance of indexing and retrieval, which was determined by the total tree nodes accessed, candidate records returned, and the time taken to perform a query. These metrics are used to quantitatively establish best practices and provide benchmarks for the comparison of new methods and future research. We also contributed research-supporting code for pre-processing datasets and post-processing results, as well as all published algorithmic improvements. This research became the pre-cursor to our ID* codebase, and the results and insights are directly applicable to the basic ID* algorithms as well.

### 3.2.1 Motivation

The motivations addressed in the original iDistance publications have only increased in importance because of the ubiquity of rich, high-dimensional, and large-scale data for information retrieval, such as multimedia databases and the mobile computing market which have exploded in popularity since the last publication in 2005. While there is little doubt that the algorithm remains effective and competitive, a more thorough investigation into performance-affecting criteria is needed to provide a basis for general capabilities and best practices. Without this study, it can be difficult to effectively use iDistance in application and reliably compare it to new methods in future research.

### 3.2.2 Partitioning Strategies

The only space-based methods presented in detail in previous works [21, 22] were *Center of Hyperplane* and *External Point*, which we refer to in this work as *Half-Points* (HP) and *Half-Points-Outside* (HPO), respectively. The HP method mimics the Pyramid-Technique [12] by placing reference points at the center of each dimensional edge of the data space with $2D$ partitions in $D$ dimensions. The HPO method creates the same reference points, but then moves them outside of the dataspace by a preset distance to reduce the overlap volume between partitions. For example, in a 2D space such as Figure 3.3, HP would result in four partitions, based on reference points: $(0.0, 0.5)$, $(0.5, 0.0)$, $(1.0, 0.5)$, and $(0.5, 1.0)$, and HPO-10 (movement of 10.0) would result in reference points: $(-10.0, 0.5)$, $(0.5, -10.0)$, $(11.0, 0.5)$, and $(0.5, 11.0)$.

Here we also introduce random reference point selection (RAND) to create any number of reference points located randomly in the dataspace. While this is a trivial strategy, it has not been shown before and greatly helps compare other strategies

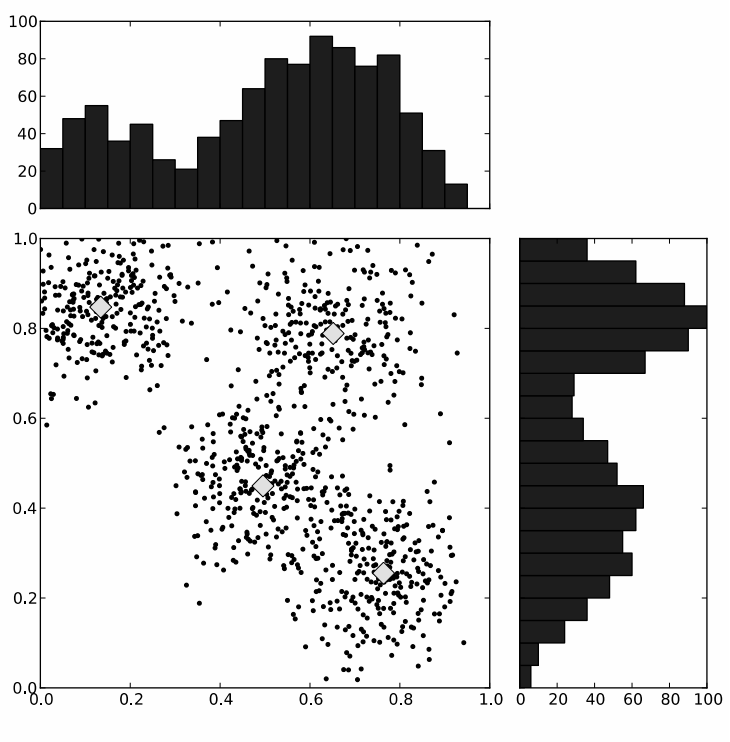



Figure 3.3: A scatter plot of a two dimensional dataset with individual histograms.

by providing a naïve benchmark. We often use some sort of random strategy as a baseline performance marker when testing algorithmic options or heuristics, which should in theory perform better than random if effective.

Two interesting things are worth noting about the HP strategy. First, the number of partitions grows linearly to the dimensions of the space. Imagine instead, for example, placing a reference point in each corner of the data space, which results in an exponential growth of $2^D$ partitions – something infeasible in even a few dimensions. Second, creating the HP index is extremely fast because it only requires $O(D)$ time, by simply setting static values of $0.0, 0.5$, and $1.0$ in one loop over all dimensions, two points at a time.

The primary benefit of data-based methods is their adaptability to data distributions, which greatly increases retrieval performance in real-world settings. Two

methods were originally introduced: *center of cluster* and *edge of cluster*, but only the *center of cluster* method was presented in published results [21, 22]. Essentially, both of these methods use algorithmically derived cluster centers as reference points to create cluster-based partitions in the dataspace.

Approximate cluster centers can be found through a variety of popular clustering algorithms. The original authors of iDistance recommended (without explicit rationale) using $2D$ as the number of partitions (clusters) with the $k$-means [36] or BIRCH [40] algorithms. They hypothesized that using the edges of clusters is intuitively more promising as it should reduce partition overlap (decreasing node accesses) and reduce the number of equi-distant points from any given reference point (decreasing candidates). Unfortunately, they leave us with only implementation suggestions, such as "points on hyperplanes, data space corners, data points at one side of a cluster and away from other clusters, and so on" [21], but many of these methods are simply infeasible in high dimensions and were never presented.

### 3.2.3 Experiments

Before we begin our experiments, we first overview some basic preliminaries. These explanations about experimental design and setup are common to the majority of experiments within the dissertation. Therefore, we dedicate more time upon first presentation to save time reiterating similar concepts and characteristics later. We also refer the reader to Appendix A that contains more details about data pre-processing for the various datasets used throughout this dissertation.

3.2.3.1 Preliminaries. Every run of our implementation of iDistance reports a set of statistics describing the index and query performance of that run. As an attempt

to remove machine-dependent statistics, we use the number of $B^+$-tree nodes instead of page accesses when reporting query results and tree size. Tracking nodes accessed is much easier within the algorithm and across heterogeneous systems, and is still directly related to page accesses through the given machine's page size and $B^+$-tree leaf size. We primarily highlight three statistics from tested queries: 1) the number of candidate points returned during the filter step, 2) the number of nodes accessed in the $B^+$-tree, and 3) the time taken (in milliseconds) to perform the query and return the final results. Often we express the ratio of candidates and nodes over the total number of points in the dataset and the total number of nodes in the $B^+$-tree, respectively, as this eliminates skewed results due to varying the dataset. We report the total number of nodes in the $B^+$-tree as the worst-case benchmark for sequential scan, even though it does not actually access any $B^+$-tree nodes itself.

Many experiments are on synthetic datasets (uniform and clustered) so we can properly simulate specific dataset conditions. Later, we apply these results towards evaluating strategies on a real world dataset. All artificial datasets are given a specified number of points and dimensions in the unit space $[0.0, 1.0]^D$. For clustered data, we provide the number of clusters and the standard deviation of the independent Gaussian distributions centered on each cluster (in each dimension). The cluster centers are randomly generated and an equal amount of data points are generated around each. For uniform data, each dimensional value from each data point is sampled independently over the $[0.0, 1.0]$ range. Unless otherwise stated, we randomly select 500 data points as $k$-NN queries (with $k = 10$) from each dataset, which ensures that our query point distribution follows the dataset distribution.

It is extremely important to understand that in these experiments all data (and the index) reside in main memory, so all comparisons are made here without depending on the behaviors of specific hardware-based disk-caching routines. In real-life however,

disk-based I/O bottlenecks are a common concern for inefficient retrieval methods. Therefore, unless sequential scan runs significantly faster, there is a greater implied benefit when the index method does not have to access every data record, each of which could potentially be a costly look-up on disk.

3.2.3.2 Space-based Strategies in Uniform Data. Our first experiments compare *Sequential Scan* (SS) to space-based methods in uniform datasets ranging from 4 to 64 dimensions and 1,000 (1k) to 1 million (1000k) points. We present *Half-Points* (HP) and *Half-Points-Outside* (HPO), specifically HPO-10 and HPO-100, and also show the RAND method with equivalent $2D$ reference points (R2D).

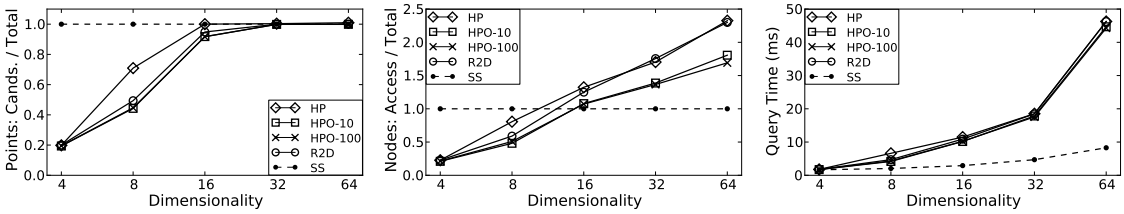



Figure 3.4: Space-based methods on uniform data (10K) over dimensions.

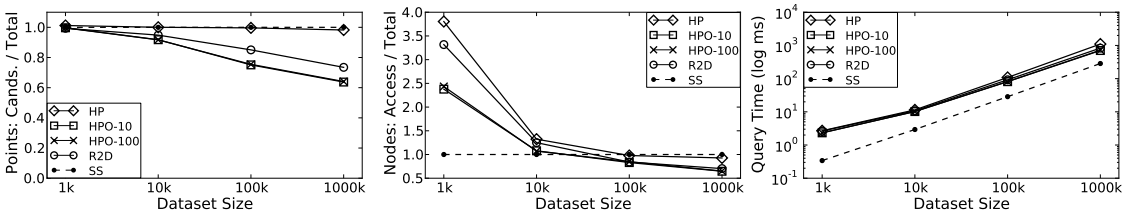



Figure 3.5: Space-based methods on uniform data (16D) over dataset size.

Figures 3.4 and 3.5 validate the original claim that HPO performs better than HP [21, 22], but surprisingly it also shows that R2D works better than HP. We can also see that a movement of 10.0 (HPO-10) outside of the dataspace is sufficient for performance improvements with HPO, and there is minimal gain thereafter. Although space-based methods take longer than SS in 16 dimensions (16D) or less, they access

significantly fewer nodes and return fewer candidates. Note that it is possible to access the same nodes multiple times because data points from disjoint partitions can be stored in the same tree leaves. Another important performance factor is dataset size, shown in Figure 3.5 over a constant 16D. This can be linked to Figure 3.4 at 10k data points. We now log-transform the query time to show that as expected, larger datasets slow down all methods. However, sequential scan grows the fastest (with a linear increase), because at a certain point space-based strategies begin to properly filter the congested space and access fewer nodes while returning fewer candidates.

While still using uniform data, we investigate the effects of varying the number of reference points. Figures 3.6 and 3.7 look at the RAND method with 16 (R16), 64 (R64), 256 (R256), and 1024 (R1024) reference points. We also include dynamic methods of $2D$ (R2D) and $\sqrt{N}$ (RP*) total reference points, which are meant to better account for the specific dataset characteristics. The results highlight the trade-off between dimensions of a space and total points, showing that as the number of dimensions increase, more partitions reduce the number of candidates, but also increase the number of nodes accessed and overall query time. Conversely, as the number of data points increases and dimensionality holds constant, $k$-NN queries become more compact, and the number of candidates and nodes decreases, leading to a shorter query time.

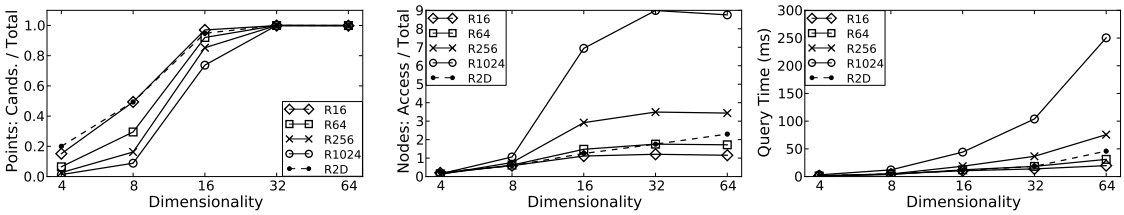



Figure 3.6: Varying the number of random reference points on uniform data (10K) over dimensions.

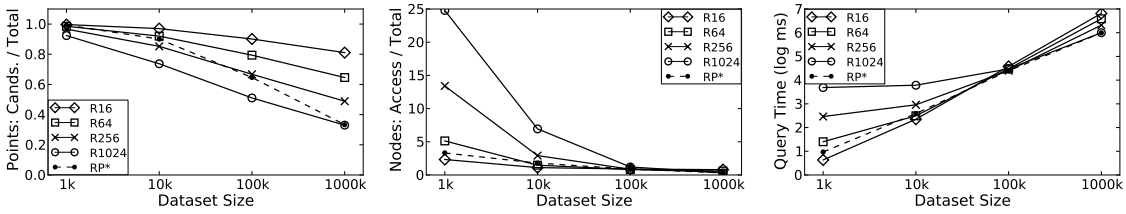



Figure 3.7: Varying the number of random reference points on uniform data (16D) over dataset size.

3.2.3.3 The Transition to Clustered Data. Since most real-world data is not uniform, we turn our attention to clustered data and data-based partitioning strategies. As mentioned by the authors in the original iDistance publications [21, 22], data-adaptive indexing is the primary strength of iDistance, and we too show that it greatly improves overall performance. We start by trying to better understand when data-based strategies overtake space-based strategies through varying cluster densities in the space, which had previously not been investigated. For each dataset, cluster centers (12 total) are randomly generated and then points are sampled with a standard deviation (SD) ranging from 0.40 to 0.005 in each dimension of the 16D space with a total of 100k points equally distributed among clusters. We use the originally generated reference points, *True Centers* (TC), as the reference points. For comparison, we include *Half-Points* (HP) and *Sequential Scan* (SS) as baseline benchmarks. The RAND method was not included because it produces completely unpredictable, although sometimes similar, results depending on the reference point locations and underlying dataset distributions.

In Figure 3.8, we can see the effect that cluster density has as the space transitions from very loose to extremely tight clusters. We do not report candidates because the results closely mirror the nodes accessed ratio. While using the true centers of the clusters as reference points quickly becomes the better technique, it eventually stalls

out and fails to improve once the data is sufficiently dense – but notice that the performance of HP steadily increases to near similar results. Since the space-based reference points are not bound to clusters, they continue to increase in effectiveness by searching smaller and smaller "slices" of each partition.

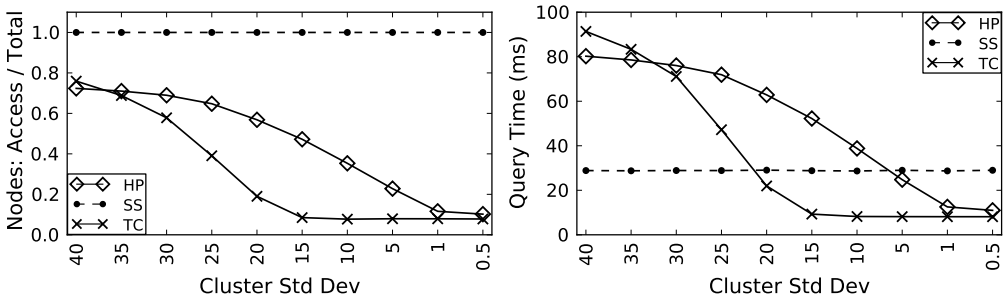



Figure 3.8: Results over varying cluster density by standard deviation (SD).

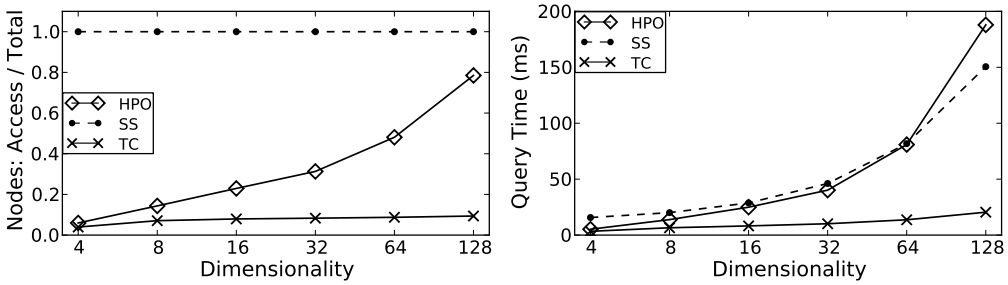



Figure 3.9: Results of 12 clusters (0.1 SD) over dimensions.

We can further see these trade-offs in Figure 3.9. Here we set the SD of all 12 clusters to 0.1 and vary the dimensionality of 100k data points. The 12 equal-sized clusters seem to explain why TC stabilizes with around 8% (or 1/12) of the nodes accessed in both of these figures. In other words, the clusters become so dense that although the $k$-NN queries rarely have to search outside of a single partition, they ultimately have to search through the entire partition containing the query. We confirm this in Figure 3.10, which shows the total partitions checked and candidates returned for three clustered datasets with 6 (TC6), 12 (TC12), and 24 (TC24) clusters over

varying cluster density. Notice that all three start with accessing all partitions and most data points, but all converge to only one checked partition with the respective ratio of candidates.

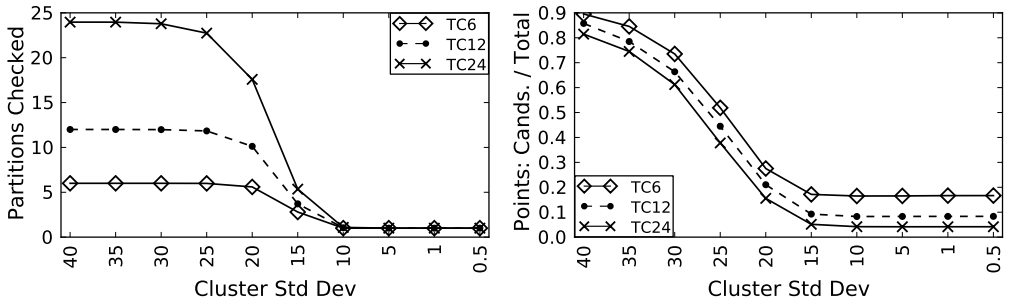



Figure 3.10: Results of various numbers of clusters over cluster density (by SD).

3.2.3.4 Reference Points: Moving from Clusters. We now investigate more advanced data-based partitioning strategies using the *True Centers* (TC) of clusters as our benchmark reference points. Original works make mention of reducing partition overlap, and thereby increasing performance, by moving reference points away from each other [21, 22], but did not investigate it. This approach should perform better than TC in theory, because there will be less equi-distant points for each reference point, meaning the lossy transformation is less destructive for true data point similarity.

We present two strategies for moving reference points away from cluster centers. Since cluster centers are typically found by minimizing inter-cluster similarity while maximizing intra-cluster similarity, by moving reference points away from the cluster centers, one could hypothesize that there should be less equi-distant points in each partition and therefore a more discriminative one-dimensional $B^+$-tree index. The two methods are: 1) *Min-Edge* (M), moving towards the closest edge of the dataspace in any single dimension, and 2) *Random* (R), moving randomly in any/all dimensions. We specify movement by a total distance in the multi-dimensional dataspace and

both methods are capable of pushing reference points outside of the dataspace, which makes the *Min-Edge* method similar to *Half-Points Outside* (HPO). Using *Min-Edge* on the data in Figure 3.3 as an example, the upper-left cluster center will decrease along the $x$-axis, and the upper-right cluster will increase along the $y$-axis.

Figure 3.11 shows the ratio of candidates returned from the two cluster center movement methods (M and R), with movement distances of {0.025, 0.05, 0.1, 0.2, 0.4}, each compared to TC. Each method performs best with a movement distance of 0.2, as shown with TC in the third column chart for better readability. We can see that above 16D (with 12 clusters and 100k points) no methods seem to make a significant difference. However, lower dimensions do support our hypothesis that moving away from the centers can help. Figure 3.12 shows the same methods in 16D over a varying number of data points, and here we see the methods also become ineffective as the number of points increase. Thus, the hypothesis does not seem to hold in large-scale and high-dimensional data.

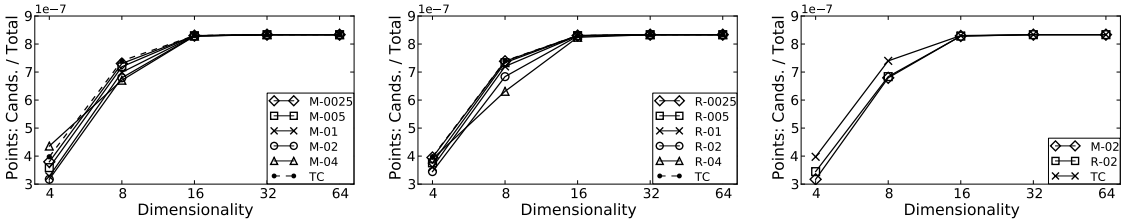



Figure 3.11: Results of center movement methods over dimensionality.

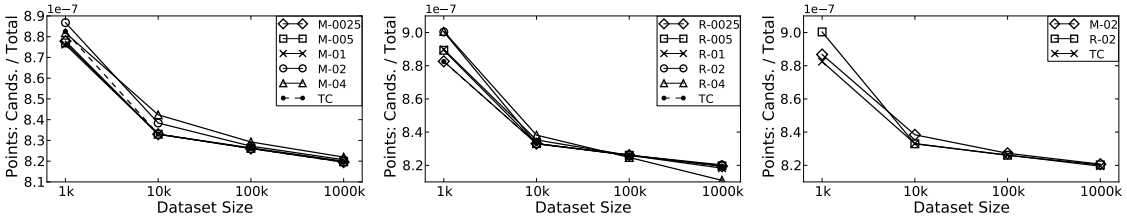



Figure 3.12: Results of center movement methods over dataset size.

3.2.3.5 Reference Points: Quantity vs. Quality. While we know that iDistance performs well on datasets with known clusters, a more common scenario is less knowledge of the data where the size and number of clusters are unknown. This is the focus of the original iDistance works, which suggest the use of any popular clustering algorithm as a pre-processor to identify more optimal reference point placements. The original publications used BIRCH [40] in 2001 [22] and $k$-means [36] in 2005 [21].

In these experiments we investigate the effect of the number of provided clusters during our pre-processing with the $k$-means algorithm. It should be stated that $k$-means is known to be sensitive to the initial starting position of cluster centers, and does not ensure any balance between cluster populations. Here we used a standard MATLAB implementation and mitigate these inherent weaknesses by initializing our cluster centers on a randomly sampled data subset, and forcing all clusters to contain at least one point so the resultant reference points are not accidentally removed and ignored from the space. Although never discussed in previous works, we believe it is very important to address the case of non-empty clusters, especially when analyzing how well a certain number of reference points perform. Otherwise, there is no guarantee that the specified number of reference points actually reflects the same number of partitions as intended.

The authors of iDistance originally suggested a general setting of $2D$ reference points—so $k$-means with $k = 2D$ clusters—which also matches the space-based strategies [21, 22]. In Figure 3.13, we look at the performance of $k$-means (KM) with $D$-relative clusters from $D/2$ to $4D$, in various dimensions over 100k points in 12 clusters. We also include *True Centers* (TC) as our current baseline benchmark, and $k$-means with 12 clusters but without knowledge of the true cluster centers upon initialization (KM-12*). Notice the relatively equal nodes accessed ratio for all meth-

ods in higher dimensions, but the increase in overhead time taken for the methods with more clusters (partitions).

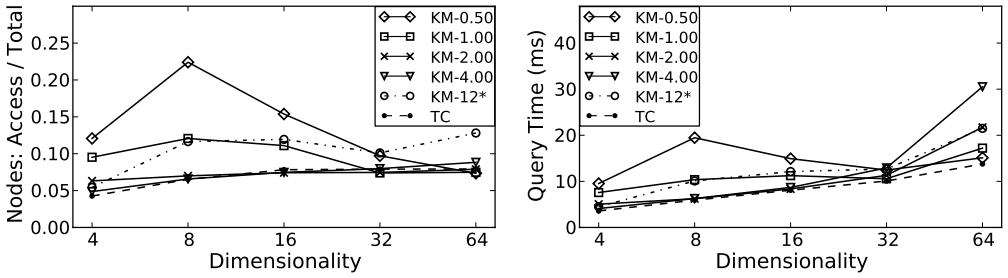



Figure 3.13: Varying number of $k$-means centers with 12 clusters (100k points).

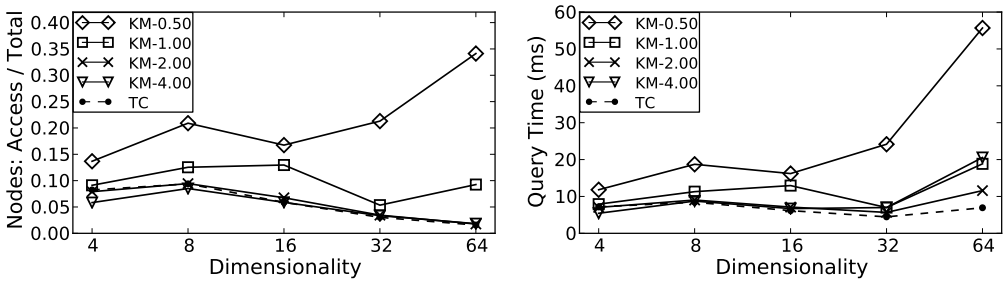



Figure 3.14: Varying number of $k$-means centers with $D$ clusters (100k points).

An important realization here is how beneficial the knowledge of true cluster centers can be, as we see TC performs more consistently (and nearly always better) than other methods over all dimensions. The same can be seen in Figure 3.14, where we now generate $D$ clusters in the dataset instead of only 12. However, here we see that in higher dimensions more clusters make a major difference for the number of nodes accessed, and $2D$ clusters seem in many cases to be an appropriate balance between the number of partitions and the time taken to search all the partitions, as both $1D$ and $4D$ clusters are equally slower. Also note that setting $k$ to the number of known clusters for $k$-means (KM-12* in Figure 3.13) does not guarantee performance because of the variability of discovered clusters from the $k$-means algorithm.

3.2.3.6 Results on Real Data. Our final experiments use a real dataset to determine if any of our findings carry over from synthetic dataset studies. We use a popular real world dataset containing one million 128-dimensional SIFT feature vectors. More information on this dataset can be found in Appendix A.3.3. This dataset was previously used by the authors of the SIMP algorithm [32] to show comparatively better performance over their private implementation of iDistance using 5,000 reference points. However, without knowledge of algorithmic options, or several baseline experiments to show optimal performance results, we have very little insight into the effectiveness (and reproducibility) of their specific comparison.

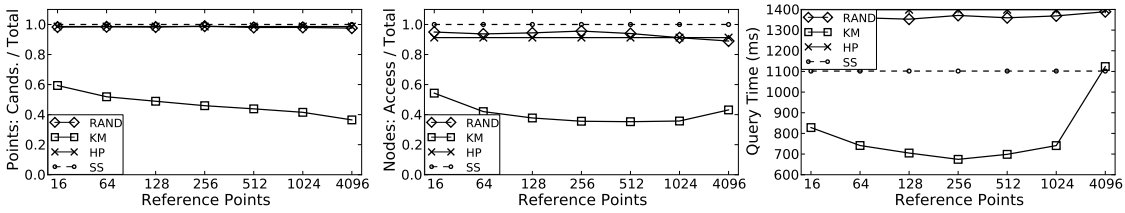



Figure 3.15: Results of $k$-means on real data over varying number of reference points.

In Figure 3.15, we look at RAND and $k$-means (KM) over a varying number of reference points, and include HP and SS methods as comparisons. We can see that the only method that performs significantly better than sequential scan is $k$-means. Although the number of candidates returned continues to decrease as we add reference points, we can see that after a certain point the overhead costs of additional partitions outweighs the filtering benefits, and the number of nodes accessed begins to increase while query time dramatically rises. We note that there exists a clear range of relatively equivalent results from approximately $D/2$ (64) to $4D$ (512) partitions, which might be a combination of many factors including indexing performance and dataset characteristics. This performance plateau also provides an excellent measure for tuning to the proper number of partitions.

We also analyzed the number of partitions that were empty or checked for candidates, and the results of RAND exemplified our concerns over empty partitions and poor reference point placement. Essentially, as the number of random reference points increased, the more empty partitions are created. Worse yet, every non-empty partition is almost always checked due to high overlap and poor placement relative to each other and the data (and query) distributions.

### 3.2.4 Remarks

We have presented many complementary results to that of the original iDistance works, and through extensive experiments on various datasets and data characteristics we uncovered many additional findings that were not presented or discussed in prior works. This work, [39], established a self-standing baseline for the wide variance in performance of partitioning strategies that opens the door for more directed and concise future works grounded on our findings. These results have also helped to establish an up-to-date benchmark and best practices for using the iDistance algorithm in a fair and efficient manner in application or comparative evaluations.

We see a promising result in Figure 3.12 at 1000k data points, hinting that it is still possible to produce better results by moving reference points. This suggests that there may exist a more sophisticated solution than the relatively simple methods we have presented. We note that because of the default closest distance assignment strategy, when reference points are moved the points assigned to them may change. Thus our efforts to reduce the number of equi-distant points may have been confounded, and if reference points are moved outside the dataspace, their partitions may become empty. Unfortunately, we found no significant difference in results by employing a static partition assignment before and after reference point movement, and therefore

did not include the results for discussion. Clearly, more knowledge is required to move reference points in an optimal way that impacts partitioning efficiency. This led to initial investigations in the idea of *clustering for the sake of indexing*, by learning cluster arrangements explicitly for use as reference points within iDistance [41].

In general, we find a trade-off between dimensionality and dataset size, where more dimensions lead to less precise query regions (the classic curse of dimensionality problem), but more points allow smaller regions to fully satisfy a given query. Space-based methods suffer much worse from dimensionality and are really not ideal for use. We agree with the original authors that $2D$ reference points seems appropriate as a general recommendation. In relatively moderate to small datasets and multi-dimensional spaces, $2D$ is probably overkill but far less burdensome than in exceptionally large datasets and high-dimensional spaces where the cost of additional reference points dramatically increases without providing much benefit. Results strongly support an intelligent data-centric approach to the amount and placement of reference points that results in minimally overlapping and non-empty partitions.

## 3.3 Segmentation Extensions

This section establishes the clear separation of our ID* index from its predecessor iDistance. Here we present the development of two algorithmic extensions that further segment the existing distance-based partitions into spatially disjoint sections. These sections add another layer of container objects, much like hierarchical clustering, that can then be quickly and efficiently pruned during search. We also propose several data-driven heuristics to intelligently guide this segmentation process. Extensive experiments confirm that our local extension can significantly improve query performance over both tightly-clustered and high-dimensional datasets.

### 3.3.1 Motivation

Our prior work had shown that iDistance tends to stabilize in performance by accessing an entire partition ($k$-means cluster) to satisfy a given query, despite dataset size and dimensionality [39]. While only accessing a single partition is already significantly more efficient than sequential scan, this shortcoming was the main motivation for exploring further dataspace segmentation to enhance retrieval performance. We achieve this additional segmentation with the creation of intuitive heuristics applied to a novel hybrid index. These extensions are similar to the works of the iMinMax($\theta$) [13] and recently published SIMP [32] algorithms, whereby we can incorporate additional dataspace knowledge at the price of added algorithm complexity and overhead.

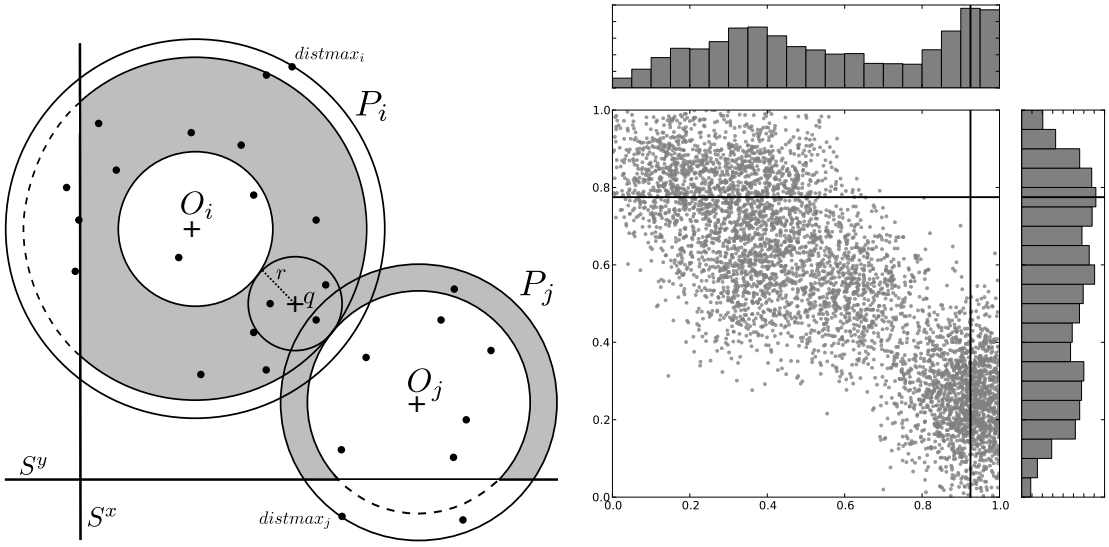
### 3.3.2 Overview

Essentially, we aim to further separate dense areas of the dataspace by *splitting* partitions into disjoint *sections* corresponding to separate *segments* of the B$^+$-tree that can be selectively pruned during retrieval. The previous sentence identifies the technical vocabulary we will use to describe the segmentation process. In other words, we apply a given number of axis-aligned dimensional *splits* in the dataspace which *sections* the partitions into B$^+$-tree *segments*.

We develop two types of segmentation based on the scope of splits: 1) **Global**, which splits the entire dataspace (and consequently any intersecting partitions), and 2) **Local**, which explicitly splits the dataspace of each partition separately and independently. While different concepts, the general indexing and retrieval algorithm modifications and underlying affects on the B$^+$-tree are similar.

$$y_p = i \times c + j \times \frac{c}{2^s} + dist(O_i, p) \tag{3.4}$$

First, the mapping function is updated to create a constant segment separation within the already well-separated partitions. Equation 3.4 describes the new index with the inclusion of the sectional index $j$, and $s$ as the total number of splits applied to the partition. Note that a partition is divided into $2^s$ sections, so we must appropriately bound the number of splits applied. Second, after identifying the partitions to search, we must identify the sections within each partition to actually search, as well as their updated search ranges within the B$^+$-tree. This also requires the overhead of section-supporting data structures in addition to the existing partition-supporting data structures.




(a) (b)

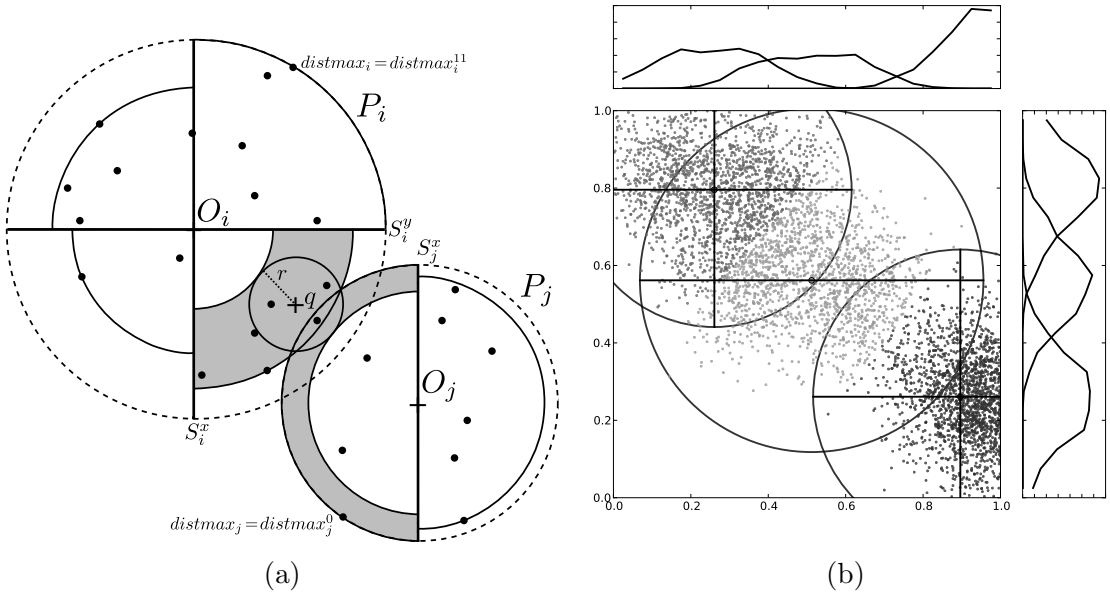
Figure 3.16: (a) Conceptual global splits $S^x$ and $S^y$ and their effect on query search ranges. (b) The histogram-based splits applied by the G2 heuristic on the example dataset.

3.3.2.1 Global. The global segmentation technique is inspired by the Pyramid [12] and iMinMax($\theta$) [13] methods, whereby we adjust the underlying index based on the data distributions in the space. A global split is defined by a tuple of dimension and

value, where the given dimension is split on the specified value. There can be only one split in each dimension, so given the total number of splits $s$, each partition may have up to $2^s$ sections. Because the splits occur across the entire dataspace, they may intersect (and thereby segment) any number of partitions (including zero). An added benefit of global splits is the up front global determination of which sections require searching, since all partitions have the same z-curve ordering. We introduce three heuristics to best choose global splits.

- (**G1**) Calculate the median of each dimension as the proposed split value and rank the top $s$ dimensions to split in order of split values nearest to 0.5, favoring an even 50/50 dataspace split.
- (**G2**) Calculate an equal-width histogram in each dimension, selecting the center of the highest frequency bin as the proposed split value, and again ranking dimensions to split by values nearest to the dataspace center.
- (**G3**) Calculate an equal-width histogram in each dimension, selecting the center of the bin that intersects the most partitions, ranking the dimensions first by total intersections and then by their proximity to the center.

Figure 3.16a shows our partition coverage example with a split in each dimension $(S^y, S^x)$. We can see that a given split may create empty partition sections, or may not split a partition at all. We explicitly initialize these data structures with a known value representing an "empty" flag. Therefore, we only check for query overlap on non-empty partitions and non-empty sections. For the example dataset and partitions in Figure 3.28, the G2 heuristic applies the splits shown in Figure 3.16b.




(a) (b)

Figure 3.17: (a) Conceptual local splits $S_i^x$, $S_i^y$, and $S_j^x$ and their effect on query search ranges in overlapping partitions and sections. (b) The local population-based splits applied to each partition by the L3 heuristic on the example dataset.

3.3.2.2 Local. We developed the local technique as a purposeful partition-specific segmentation method based only on local information of the given partition rather than the global data characteristics of the entire space. Local splits are defined by a tuple of partition and dimension, where the given partition is split in the specified dimension. Unlike the global technique, we do not need the values of each dimensional split because all splits are made directly on the partition reference point location. Figure 3.17a shows a conceptual example of local splits.

During index creation we maintain a $distmax_i$ of each partition $P_i$, and we now we do this for sections too ($distmax_i^j$ for partition $P_i$ and section $j$). This allows the addition of a section-based query overlap filter, which can often prune away entire partitions that would have otherwise been searched. This can be seen in Figure 3.17a, where the original partition *distmax* is now a dotted line and a new *distmax* is shown for each individual partition section. Notice that the original *distmax* data point is

now present in only one section, and all the other sections have an updated *distmax* that is now much smaller.

We introduce three heuristics to choose local splits. Within each partition, we want to split the data as evenly as possible using the added spatial information. Since the split locations are already determined by reference point location, we must look at the assigned population to each partition. Thus, for all three methods, we rank the dimensions to split by those closest to an equal (50/50) population split. We denote $s$ as the maximum number of splits any partition can have.

- (**L1**) Uniformly apply $s$ splits to each partition using the top ranked.
- (**L2**) Use a population threshold as a cutoff criteria, so each partition could have up to $s$ splits. The intuition here is that if a split only segments a reasonably small portion of the data, it probably is not worth the additional overhead to keep track of it.
- (**L3**) Use $s$ to calculate the maximum number of underlying $B^+$-tree segments created by method L1, but then redistribute them based on partition population. This automatically applies additional splits to dense areas of the dataspace while removing splits where they are not needed and maintaining an upper bound on $B^+$-tree segments.

Formally, we have a set of splits $\mathcal{S} = \{S_1, \ldots, S_M\}$ where $S_i$ is the set of dimensions, $|S_i|$ total, to split on for partition $P_i$. Given a maximum granularity for the leaves in the $B^+$-tree, we can calculate how many splits each partition should have as a percentage of its population to the total dataset size. Equation 3.5 gives this assignment used by L3. For example, we can see in Figure 3.17b that the center cluster

did not meet the population percentage necessary to have a second split. Thus, the only split is on the dimension that has the closest 50/50 split of the data.

$$|S_i| = \left\lfloor \lg \frac{|P_i|}{N} \cdot M \cdot 2^s \right\rfloor \tag{3.5}$$

3.3.2.3 Hybrid Indexing. The global and local dataspace segmentation generates a hybrid index with one-dimensional distance-based partitions subsequently segmented in a tuneable subset of dimensions. Due to the exponential growth of the tree segments, we must limit the number of dimensions we split on. For the local methods, L3 maintains this automatically. For the spatial subtree segmentation, we use z-curve ordering of each partition's sections, efficiently encoded as a ternary bitstring representation of all overlapping sections that is quickly decomposable into a list of section indices. Figure 3.17b shows an example of this ordering with $distmax_i^{11}$, which also happens to be the overall partition $distmax_i$. Since $P_i$ is split twice, we have a two-digit binary encoding of quadrants, with the upper-right of '11' equating to section 3 of $P_i$.

### 3.3.3 Experiments

The goal of our experiments is to determine the feasibility and effectiveness of our global and local segmentation techniques and their associated heuristics over datasets of various size and dimensionality. Using these results, we follow up with more varied experiments on additional synthetic and real-world datasets.

3.3.3.1 Preliminaries. The first experiments are on uniform and clustered synthetic datasets in a $D$-dimensional unit space $[0.0, 1.0]^D$. As always, we provide the number

of clusters and the standard deviation (SD) of the independent Gaussian distributions (in each dimension) for each randomly generated cluster center. We then use 500 randomly selected data points as $k$-NN queries (with $k = 10$), which ensures that our query point distribution follows the dataset distribution. Later experiments validate our findings on real-world datasets.

3.3.3.2 First Look: Extensions & Heuristics. We create synthetic datasets with 100,000 (100k) data points equally distributed among 12 clusters with a 0.05 standard deviation (SD) in each dimension, ranging from 8 to 512 dimensions. The true cluster centers are used as partition reference points and each heuristic is independently tested with 2, 4, 6, and 8 splits. Here we compare against regular iDistance (non-optimized ID*) using the same true center (TC) reference points. For spacing considerations, we only present one heuristic from each type of segmentation scope, namely G2 and L1. These methods are ideal because they enforce the total number of splits specified, and do so in a rather intuitive and straightforward manner, exemplifying the over-arching global and local segmentation concepts. We also note that the other heuristics generally perform quite similar to these two, so presentation of all results would probably be superfluous.

Figure 3.18 shows the performance of G2 compared to TC over candidates, nodes accessed, and query time. Above 64 dimensions we do not see any significant performance increase by any global heuristic. Note that we do not show Sequential Scan (SS) results here because they are drastically worse. For example, we see TC returning approximately 8k candidates versus the 100k candidates SS must check.

The same three statistics are shown in Figure 3.19 for L1 compared to TC. Unlike G2, here we can see that L1 greatly increases performance by returning significantly fewer candidates in upwards of 256 dimensions. Above 64 dimensions, we can see the

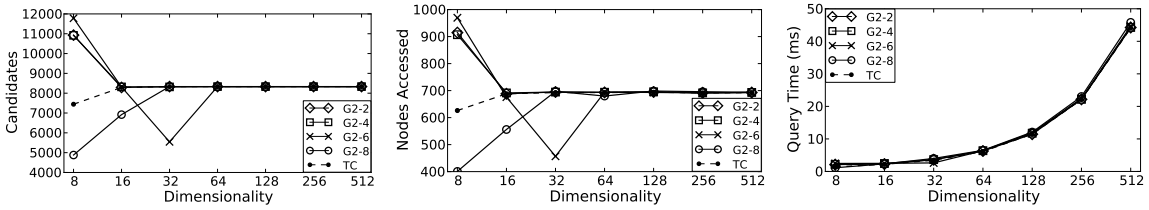



Figure 3.18: Global heuristic G2 with 2, 4, 6, and 8 splits over dataset dimensionality.

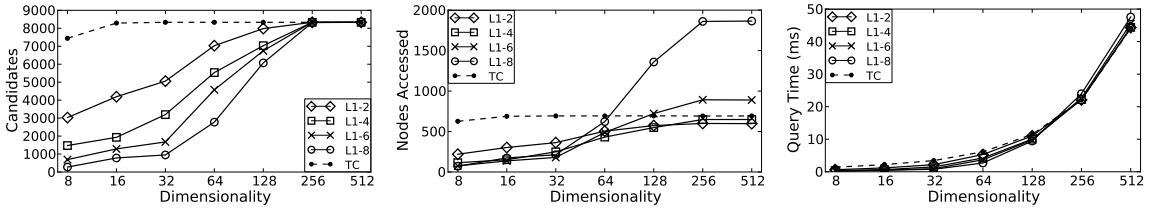
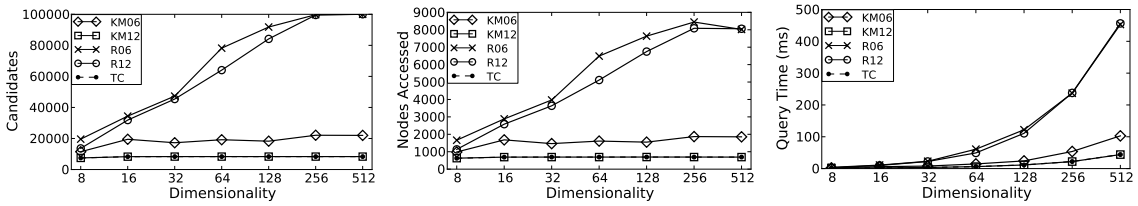



Figure 3.19: Local heuristic L1 with 2, 4, 6, and 8 splits over dataset dimensionality.

increased overhead of nodes being accessed by the larger number of splits (6 and 8), but despite these extra nodes, all methods run marginally faster than TC because of the better candidate filtering. It should also be clear that the number of splits is directly correlated with performance, as we can see that 8 splits performs better than 6, which performs better than 4, and so on. Of course, the increased nodes accessed is a clear indication of the trade-off between the number of splits and the effort to search each partition section to satisfy a given query. Eventually, the overhead of too many splits will outweigh the benefit of enhanced filtering power.

Lastly, Figure 3.20 revisits a finding from our previous work [39] to show the general performance comparison between TC and $k$-means (KM), with the random (RAND) space-based partitioning method included as a naïve data-blind approach. Since we established 12 well-defined clusters in the dataspace, we can use this data knowledge to test KM and RAND, each with 6 and 12 partitions, and compare them to TC (which uses the true 12 cluster centers). Unsurprisingly, we see that RAND (R6



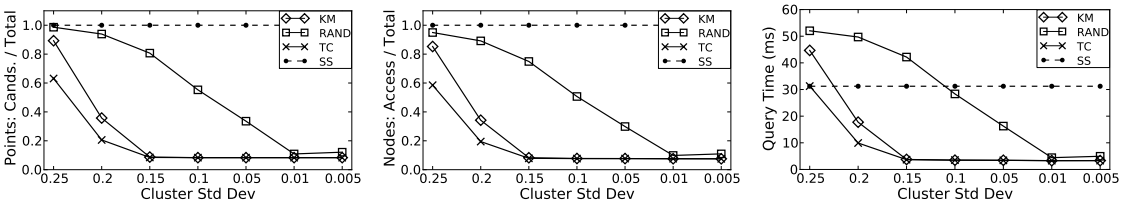

(a) (b) (c)

Figure 3.20: A comparison of TC to KM and RAND with 6 (KM6, R6) and 12 (KM12, R12) partitions.

and R12) quickly degrades in performance as dimensionality increases. We also find that KM12 performs almost equivalent to TC throughout the tests. This provides ample justification for the benefits of data clustering for effective dataspace partitioning, and allows us to focus entirely on $k$-means for the remainder of our experiments, which is also a much more realistic condition for non-synthetic datasets.

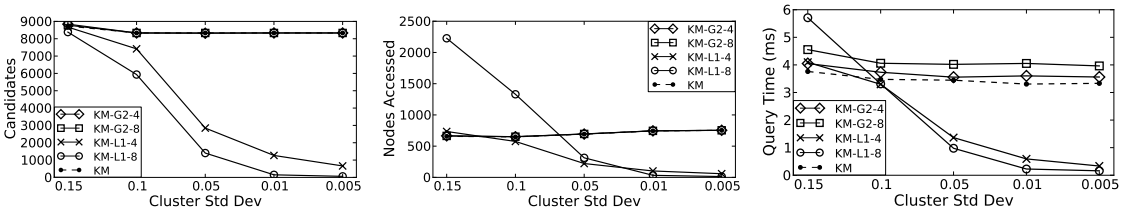
3.3.3.3 Investigating Cluster Density Effects. The second experiment extends the analysis of our heuristics' effectiveness over varying tightness/compactness of the underlying data clusters. We again generate synthetic datasets with 12 clusters and 100k points, but this time with cluster SD ranging from 0.25 to 0.005 in each dimension over a standard 32-dimensional unit space. Figure 3.21 reiterates the scope of our general performance, showing SS and RAND as worst-case benchmarks and the rapid convergence of TC and KM as clusters become sufficiently compact and well-defined. Notice however, that both methods stabilize in performance as the SD decreases below 0.15. Essentially, the clusters become so compact that while any given query typically only has to search in a single partition, it ends up having to search the entire partition to find the exact results.

Figure 3.22 compares our heuristics G2 and L1 applied to $k$-means derived partitions (labeled as KM for a stand-alone comparison). To simplify the charts, we only




(a) (b) (c)

Figure 3.21: A comparison of partitioning strategies over cluster standard deviation.




(a) (b) (c)

Figure 3.22: Heuristics G2 and L1 versus KM over cluster standard deviation.

look at 4 and 8 splits, which still provide an adequate characterization of performance. We also only look at a cluster SD of 0.15 and less, to highlight the specific niche of poor performance we are addressing. Unfortunately, we again see that G2 performs no better than KM, and it even takes slightly more time due to the increased retrieval overhead and lack of better candidate pruning. However, L1 performance drastically improves over the stalled out KM partitions, which more than proves the effectiveness of localized segmentation heuristics.

3.3.3.4 Results on Real Data. Our next experiment used a real-world dataset consisting of 90 dimensions and over 500k data points representing recorded songs from a large music archive, referred to as the Music dataset (see Append A.3.2 for more details). First, we methodically determine the optimal number of clusters to use for $k$-means, based on our previous discovery of a general iDistance performance plateau surrounding this optimal number [39], which represents a best practice for choosing $k$

for iDistance. Although omitted for brevity, we tested values of $k$ from 10 to 1000, and found that iDistance performed best around $k = 120$ clusters (partitions). We also discard global techniques from discussion given their poor performance on previous synthetic tests.

Figure 3.23 shows our local L1 and L3 methods, each with 4 and 8 splits, over a varying number of $k$-means derived partitions. Since this dataset is unfamiliar, we include baseline comparisons of KM and SS, representing standard iDistance performance and worst-case sequential scan performance, respectively. We see that both local methods significantly outperform KM and SS over all tests. Also note that L3 generally outperforms L1 due to the more appropriate population-based distribution of splits, which translates into more balanced segments of the underlying $B^+$-tree.

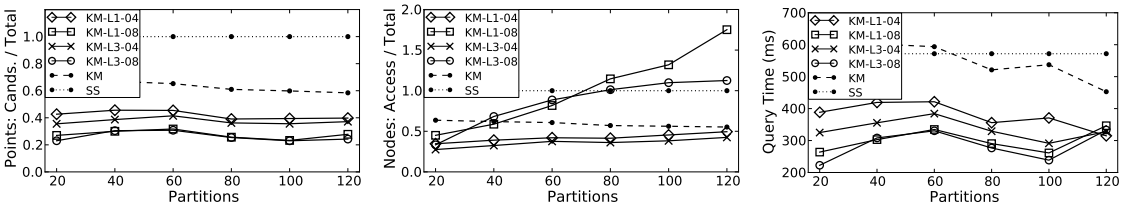



Figure 3.23: Results of local methods L1 and L3 versus iDistance on real data.

### 3.3.4 Extended Experiments

Next we follow up with additional experiments that highlight the difficulties of indexing in high-dimensional and tightly-clustered dataspaces by exploring several important tunable parameters for optimizing $k$-NN query performance using the iDistance and ID* algorithms [42]. We experiment on real and synthetic datasets of varying size, cluster density, and dimensionality, and compare performance primarily

through filter-and-refine efficiency and execution time. Results show great variability over parameter values and provide new insights and justifications in support of prior best-use practices. Local segmentation consistently outperforms iDistance in any clustered space below 256 dimensions, setting a new benchmark for efficient and exact $k$-NN retrieval in high-dimensional spaces.

In previous work, it was shown that in high-dimensional and tightly-clustered spaces, iDistance stabilizes in performance by accessing an entire partition ($k$-means derived cluster) to satisfy a given query [39]. More recent work [43] showed that our novel ID* algorithm using our local segmentation extension improved on these shortcomings. Here we expand this investigation to further analyze how each of these types of dataspaces affect the index algorithms.

3.3.4.1 Curse of Dimensionality. Figure 3.24 illustrates the curse of dimensionality quite well through various pair-wise distance statistics on a synthetic clustered dataset. These are collected for every data point within the same partition, over all partitions of the dataset. In other words, we are explicitly looking at the distance between intra-cluster data points. Notice how the standard deviation (stddev) and relative average distance normalized by the maximum distance in the given dimensional space (relavg) remain essentially constant over dimensionality, but minimum and maximum distances are converging on the increasing average distance.

As dimensionality grows, the average distance between any two intra-cluster data points grows, but the distribution of these distances converges on everything becoming equi-distant from each other. This is similar to tightly-clustered spaces, whereby all points become equidistant, however importantly, here the average distances decrease as cluster density (or compactness) increases.

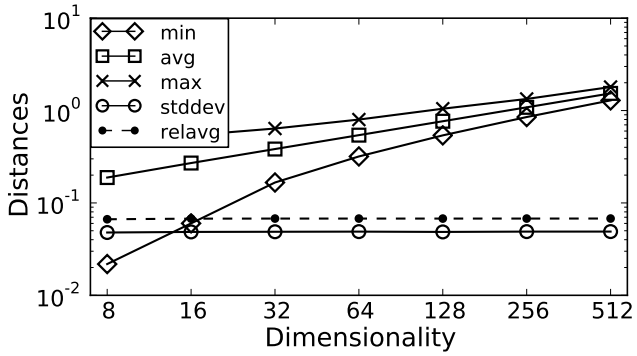
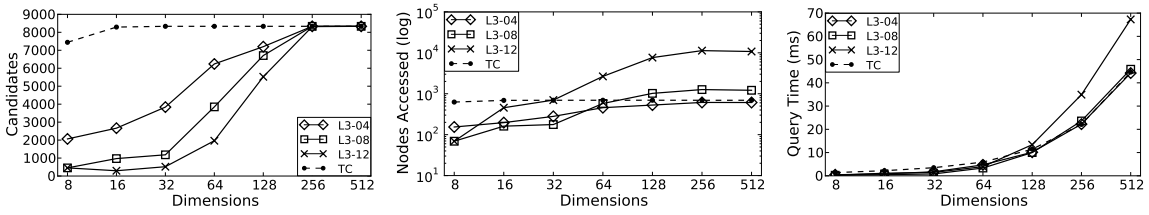



Figure 3.24: The curse of dimensionality through pair-wise intra-cluster distances.

3.3.4.2 High-dimensional Space. Figure 3.25 shows the performance of ID* local segmentation (L3) over varying dimensionality using 4, 8, and 12 splits determined by the L3 heuristic, compared to standard iDistance (TC) using the same underlying true cluster centers as reference points. Here we see a clear trade-off between the filtering power of local segmentation, and the performance overhead of adding additional splits. Under 256 dimensions, the additional filtering power not only works significantly better, but it also takes very little overhead, however above 256 dimensions there is little benefit in using it. One would hypothesize that even more splits would further mitigate the curse of dimensionality for filtering candidates, but the overhead cost in accessed nodes will dramatically increase and likely be impractical.




(a) (b) (c)

Figure 3.25: Comparing iDistance (TC) to ID* (L3) with 4, 8, and 12 splits over dataset dimensionality.

3.3.4.3 Tightly-clustered Space. Figure 3.26 shows the performance of ID* local segmentation (L3) in tightly-clustered data similarly using 4, 8, and 12 splits compared to iDistance (TC). While the distance-phenomenon is somewhat similar in tightly-clustered spaces, we can see a noticeable difference as clusters become tighter. First notice that when clusters are quite spread out (0.25 SD in each dimension) the additional dataspace segmentation cannot filter the candidates any better than without, but it suffers in time because of the overhead of additional nodes accessed. The real benefit is seen from 0.15 SD and smaller, when proper segmentation eliminates many potential candidates, even though the space is extremely dense.

The difference between these two sets of charts exemplifies how difficult indexing high-dimensional spaces is compared to well-clustered, but congested areas. Consider again the conceptual examples of our spherical partitions in Figures 3.1 and 3.17a. As we decrease standard deviation of these clusters, the partitions collapse inward to the reference point, where intra-cluster distances become equally small, but inter-cluster distances increase. On the other hand, as we increase dimensionality, these spherical partitions turn into ring-like bands of equally large distances that continue to expand away from the reference points. This leads to more overlapping of partitions and, given the equi-distant points, more candidates being returned from each search.

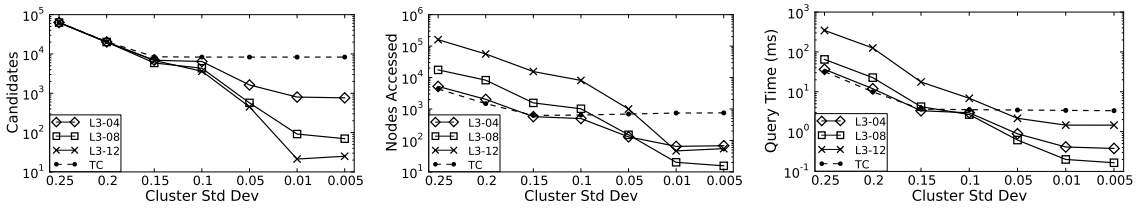


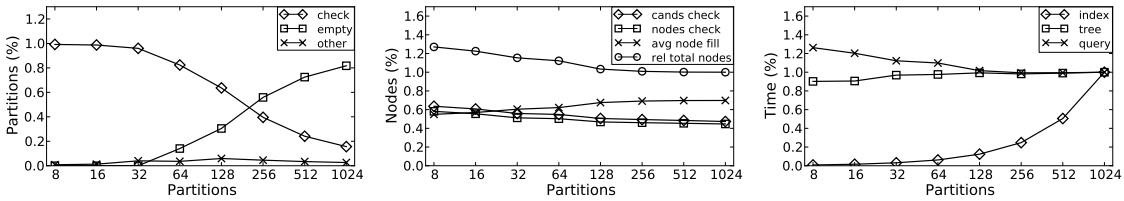

Figure 3.26: Comparing iDistance (TC) to ID* (L3) with 4, 8, and 12 splits over dataset cluster standard deviation.

3.3.4.4 Partition Tuning. Next we look at how the number of partitions affects performance of iDistance (and also ID*). Here we again use a real-world dataset of SIFT descriptors consisting of 1 million points in 128 dimensions, referred to as the Sift dataset (see Appendix A.3.3 for more details). Since the dataset characteristics ($N$ and $D$) do not change, we tune performance by varying the number of partitions used, shown in Figure 3.27. The first chart shows that as we increase above 64 total partitions, queries access fewer and fewer partitions, primarily because more and more are empty. The "other" line represents partitions which are used (non-empty) but not searched.

The second chart in Figure 3.27 uses proportions to present four related statistics: the percentage of candidates checked, the percentage of nodes accessed, the average fill of a node in the tree, and the relative amount of total nodes in the tree, based on the total node count for 1,024 partitions. First, we see the expected result that more total partitions lead to fewer candidates and nodes accessed. However, we can see this reduction stagnates above 256 partitions, which is also where we see the structure of the tree change. Second, notice around 128 partitions where we see a reduction in total nodes, but an increase in average node fill, thereby indicating a more compact tree. This is a somewhat unexpected result, which might be caused in part by the constant partition separator and the increase in unique index values.

Lastly, the third chart in Figure 3.27 shows the time taken to complete three actions: 1) calculate the index values from the dataset, 2) build the tree of index values, and 3) perform a query. Here we use the relative values again (based on values at 1,024 partitions) so we can compare all three side-by-side. Interestingly, the tree time is rather constant, but it does take somewhat longer when more partitions are used. This is likely caused by the more compact tree and possibly more tree

re-organization steps during loading. While query time decreases as expected, given the fewer nodes and candidates, the most important time consideration is clearly the initial index construction. While this is essentially just a pre-process step in our setup, it is worthwhile to note for practical application and index maintenance.




(a) (b) (c)

Figure 3.27: Varying the number of partitions on real data.

### 3.3.5 Remarks

One topic worth discussion is the placement of reference points for creating partitions in the dataspace. Building on previous works [21, 39, 22], we continue to use the $k$-means algorithm to cluster the data and obtain a list of cluster centers to be used directly as reference points. However, Figure 3.28 hints at a possible shortcoming of using $k$-means, specifically when building closest-assignment partitions. Notice how large the center cluster is because of the farthest assigned data point, and how much additional partition overlap that creates. We hypothesize that just a few outliers in the data can have a tremendously negative effect on index retrieval due to this property of $k$-means.

We developed global and local segmentation heuristics in the dataspace and underlying B$^+$-tree, but only localized partition segmentation proved effective. Results have shown that ID* can significantly outperform the iDistance index in tightly-clustered and high-dimensional dataspaces. Furthermore, in unfavorable conditions ID* performs generally no worse than iDistance. This establishes a new performance

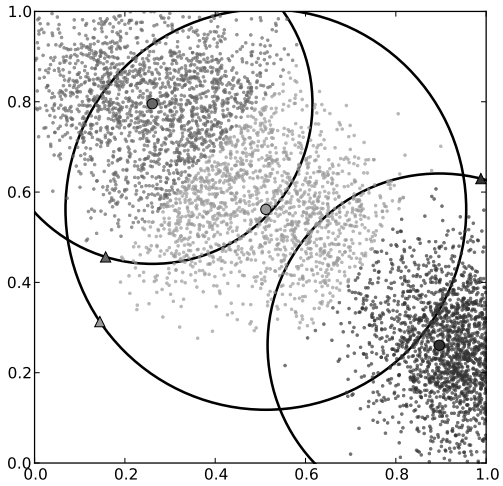



Figure 3.28: An example of $k$-means partitioning with $k = 3$.

benchmark for efficient and exact $k$-NN retrieval that can be independently compared to and evaluated against by the community.

Unfortunately, the curse of dimensionality still remains an issue beyond 256 dimensions, where Euclidean distance offers little differentiation between data points (and index values). We find the number of partitions used can affect the quality of the index beyond the general performance metrics of candidates, nodes, and query time. The pattern of performance changes on real data supports prior findings with new insights into why iDistance performance plateaus around $2D$ partitions.

## 3.4 Algorithmic Improvements

We now thoroughly analyze three crucial algorithmic components of a distance-based index, with specific focus on our own ID* index. All of these algorithms directly affect our newly developed segmentation extensions from the prior section, as well as iDistance, because they rely on the same index foundations. Generally speaking,

however, we refer to ID* as the non-hybrid distance-based index (without local or global segmentations), using our most up-to-date insights into best practices and the current version of the software.

In this section we first present pseudocode to specifically highlight the time complexity of each algorithm. We show how variable tuning best practices can simplify these complexity bounds, and then discuss how several practical improvements can increase theoretical performance. Lastly, we use ID* to test the performance of these improvements and back up our formal analyses with empirical validations. Results on synthetic and real datasets show significant improvement in $k$-NN query performance.

### 3.4.1 Motivation

The task of $k$-NN retrieval is inherently costly due to the necessary calculation of distance between objects. Since distance calculation requires an operation on every dimension of the $D$-dimensional dataspace (see Equation 2.5) it must always be $O(D)$ in time. Consider this compared to point or range queries that can discard candidates by potentially only looking at a single dimension. Therefore, we hypothesize that any reduction in overall distance calculations should improve query performance.

### 3.4.2 Theoretical Analysis

We now take a more detailed look at several critical ID* algorithms first introduced in the beginning of this chapter. Before we begin, we point out that the space costs of ID* and $k$-NN retrieval are minimal and not as important as the time costs. The index stores a key-value pair for every data point, $O(N)$, which can typically occupy main memory (8 bytes per value and 2 values per record is approximately 15.25 MB for 1 million records), and at worst it can be efficiently held on disk since it is already in a

$B^+$-tree. A $k$-NN query itself occupies a negligible constant of $O(k)$ space, because it only needs to retain at most $k$ data point IDs at any given time. Likewise, a distance calculation only needs one summation value, $O(1)$, regardless of dimensionality.

3.4.2.1 Index Creation. An unavoidable and costly task is the actual creation of the index. As seen in Algorithm 3.1, this requires a distance calculation between every data point and every reference point to find the closest partition for assignment. Given $N$ $D$-dimensional points and $M$ reference points, the index creation process is bounded in time by $O(NMD)$. We note that $N$ is typically orders of magnitude larger than $M$ or $D$, and therefore dominates the time costs with $O(N)$.

As an aside, it is important to point out the practical difference between a distance-based indexing algorithm and a pair-wise dataset metric. Essentially, since a metric requires a distance between every point of the dataset, it always takes $O(N^2)$ time. This is the special case where $M = N$, however, we know $M$ will always be much smaller than $N$. While not the main performance factor, also consider the difference in final space costs, which is $O(N^2)$ for a computed pair-wise metric and only $O(N)$ for a distance-based index.

From a practical perspective, we can follow general best practices [21, 39] of using $2D$ reference points to best mitigate the curse of dimensionality in a dynamic (self-tuning manner). This allows the bound to be simplified to $O(ND^2)$. While this now implicates the dimensionality of the space as a higher potential cost, $N$ will always be greater than $D$ and $D^2$, e.g., if $D = 256$ dimensions, then $D^2 = 65{,}536$, which is a relatively quite small dataset of records in an extremely high dimensional space. Therefore, $O(N)$ will always dominate creation time.

Lastly, we note that the process of choosing the set of reference points can come at a cost as well. This pre-processing cost only occurs once before index creation, but if

**Algorithm 3.1** Index Creation

**Input:** $\mathcal{N}$ is the set of points and $\mathcal{O}$ is the set of reference points.
**Output:** An index value for each point in $\mathcal{N}$.

```
1: procedure BuildIndex (N, O)
2:     for each p_i ∈ N do                                  ▷ O(N)
3:         min_dist ← −inf
4:         min_m ← −1
5:         for each O_j ∈ O do                              ▷ O(M)
6:             pdist ← dist(p_i, O_j)                       ▷ O(D)
7:             if pdist < min_dist then
8:                 min_dist ← pdist
9:                 min_m ← j
10:        index_i ← (c × min_m) + min_dist
```

frequent updates occur or dataset characteristics change, general maintenance might require additional periodic runs. As a baseline, consider that randomly generated reference points costs $O(MD)$.

It is generally accepted in the community and the literature to use Lloyd's algorithm [37] for the $k$-means clustering algorithm. Here the average complexity is given by $O(NMDI)$, where $I$ is the number of iterations the algorithm must perform to converge on a solution. Therefore, it is considered a cheap (efficient) clustering algorithm, but also one that is well known to have shortcomings leading to poor results. Several enhancements exist, which we utilize, such as starting $k$-means with more promising centroids (pre-cluster sampling and $k$-means++) than random placement, which leads to quicker convergence and performing several runs of $k$-means on sampled data subsets (batch-$k$-means), which in turn leads to much quicker overall execution time with highly similar results.

3.4.2.2 Index Retrieval. The second component we discuss is the query retrieval algorithm, shown in Algorithm 3.2. Given a $k$-NN query $q$, this algorithm incremen-

**Algorithm 3.2** Query Retrieval

**Input:** A query point $q$ and the set size $k$ of neighbors.
**Output:** The set $S$ of exact $k$-nearest neighbors.

```
1: procedure QueryKNN (q, k)
2:     r ← 0
3:     stop ← False
4:     while stop == False do                                   ▷ O(I)
5:         r ← r + r_Δ
6:         for each P_i ∈ P do                                  ▷ O(M)
7:             distp ← dist(O_i, q)                             ▷ O(D)
8:             if P_i previously searched then
9:                 Resume search in and/or out
10:            else
11:                if distp − r <= distmax_i then
12:                    // q overlaps P_i                        ▷ Partition Filter
13:                    Mark P_i as searched
14:                    if distp < distmax_i then
15:                        // q inside P_i
16:                        Search inward and outward
17:                    else
18:                        // q intersects P_i
19:                        Search inward only
20:        if |S| == k and distmax_S <= r then
21:            stop ← True
```

tally searches for points within a given distance $r$ from $q$, based on trigonometric calculations from the partition reference points and the precomputed point index values, which were generated in the previous algorithm. We can see that over each iteration, we calculate the distance of $q$ to each partition. If the query sphere previously overlapped a partition, the search can be directly continued, and if not, it is checked again to be searched. Both options require the distance to $q$ and the newly enlarged radius $r$. We omit further details of the retrieval algorithms that search and traverse the $B^+$-tree, as they are not central to these complexity bounds.

Two important and mutually-related observations can be made about the complexity of this algorithm. First note that we cannot terminate retrieval until the

exact $k$-set $S$ is found, which we can guarantee is the case when the distance of the farthest item in the set to $q$ ($distmax_S$) is less than or equal to the current query sphere radius $r$. To achieve this efficiently, we iteratively expand the query sphere by increasing $r$ by some value $r_\Delta$ until this termination condition is met. If $r_\Delta$ is too small, we perform many iterations, and if $r_\Delta$ is too large, we likely end up checking many additional candidates while finding the last item(s) in $S$.

By knowing the dimensionality $D$ of the dataspace, and the value of $r_\Delta$, we can upper bound the total possible iterations ($I$) to be $\frac{\sqrt{D}}{r_\Delta}$. Since this assumes $distmax_S = \sqrt{D}$, which is the maximum possible distance between two points in a $D$-dimensional unit space, this value cannot be surpassed and will rarely be reached.

Given a constant value of $r_\Delta$, such as 0.01, we note that the number of iterations increases as dimensionality increases. For example, if $D = 4$, $I = 200$, but if $D = 64$, $I = 800$. This is because the maximum distance in the dataspace increases over dimensionality. Importantly, the curse of dimensionality results in a more sparse dataspace with larger average distances between data points, as well as a convergence of these distances to all equi-distant data points. Therefore, if we instead set $r_\Delta$ to a value dependent on $D$, such as $0.01 \times \sqrt{D}$, we now have a constant bound ($I = 100$) and more consistent retrieval performance.

While better handling the total number of iterations is beneficial, the costs within each iteration can also be mitigated. Notice that for each iteration, we check if the (now larger) query sphere overlaps the partition. However, this overlap calculation is actually broken into two pieces, the distance from $q$ to $O_i$ and the size of the spheres $r$ and $distmax_i$ respectively. Therefore, we can cache the distance values from $q$ to every partition reference point, and only have to account for the incrementing radius $r$ in the overlap calculation. We note that this requires a negligible additional storage cost of $O(M)$.

The general complexity of Algorithm 3.2 is $O(IMD)$. Through these practical adjustments, we can update the complexity to $O(MD + IM)$. Now if we consider $M = 2D$ and $I$ as a constant, we have $O(2D^2 + 2ID) = O(D^2)$.

3.4.2.3 Nearest Neighbor Collection. The third component we discuss is the process by which we check candidates during query retrieval to see if they belong in the current result set $S$. Presented in Algorithm 3.3, this occurs within the search inward and outward functions referenced in the inner loop of Algorithm 3.2. Note that in the worst case, all data points could be at some point retrieved during the process of searching inward and outward over all partitions, which would result in this algorithm being executed $O(N)$ times. However, the essential point of a distance-based filter-and-refine strategy is to prune away candidates and achieve a sub-linear cost with respect to $N$. Empirical results highlight total candidates checked as one of the key performance factors for this reason.

**Algorithm 3.3** Neighbor Collection

**Input:** A candidate point $p$.

```
1: procedure AddNeighbor (p)
2:     distp ← dist(p, q)                          ▷ O(D)
3:     if distp <= distmax_S then
4:         // p passes refinement
5:         Add p to S
6:         Update distmax_S
```

Since the original iDistance pseudocode in [21] presents neighbor addition in a purely set-theoretical manner, here we discuss the similar ID* algorithmic details at the level of actual implementation. Every candidate $p$ is refined by its true distance to the query point $q$. Even if $|S| < k$ and the point will be added to $S$, the $\mathbf{dist}(p, q)$ is necessary for ensuring proper termination of the iterative retrieval algorithm which requires $distmax_S$.

Therefore, within our simplified $O(D^2)$ loop of Algorithm 3.2 we have at worst a total cost of $O(ND)$, which results in a total worst case retrieval cost of $O(D^2+ND)$. Compared to the brute force computation of $\Theta(ND)$, this is essentially just the overhead of the Algorithm 3.2 loop that must calculate distance of the query point to every reference point.

Of course, if we do not yet have $k$ results then $p$ will be added to $S$ regardless of its distance away, which may be farther than the current query radius $r$. This is an important subtly of the lossy mapping scheme and overlapping partition searches, where-as correct results might be added to $S$ very early on, but still require continued search iterations until the query radius is large enough to guarantee the results. On the other hand, the sooner the $k$-nearest neighbors are found, the fewer candidates that will have to be inserted into $S$ in the meanwhile. We enforce this simple condition in line 3 as a final filter step, whereby if the distance is greater than $distmax_S$ (which we already know), then $p$ is discarded and the set is untouched.

Several important points surround the actual addition of $p$ to $S$ and the maintenance of this set in lines 5 and 6. We chose to maintain an ordered result set, which uses a simple insertion sort over the set that requires on average $O(k^2)$ time. As a practical constant speed up, we start from the farthest end to maximize the likelihood of quicker insertion. Unlike sorting over a random set, one could argue this is a much more beneficial improvement due to the nature of retrieving increasingly farther distances from the query point. Also, as $r$ increases, closer items in $S$ become guaranteed results, and the size of $k$ decreases.

As an aside, this could also be improved with a heapsort to $O(klog(k))$, or heavily relaxed to only maintain the $distmax_S$ in $O(k)$ time. Overall though, given the relative small size of $k$ typically used (i.e., $k << N$), these differences are likely negligible in practice.

### 3.4.3 Experiments

Here we empirically verify some of the algorithmic complexities and improvements discussed. We begin with highlighting the costs of reference point selection and index creation. Then we look at the practical improvements of query retrieval and true costs of refining neighbors to the final result set. Lastly, we test our findings on several real world datasets and comparatively evaluate our algorithmic optimizations.

3.4.3.1 Preliminaries. Every experiment reports a set of statistics describing the index and query performance of that test. As an attempt to remove machine-dependent statistics, we use the number of $B^+$-tree nodes instead of page accesses when reporting query results and tree size. Tracking nodes accessed is much easier within the algorithm and across heterogeneous systems, and is still directly related to page accesses through the given machine's page size and $B^+$-tree leaf size.

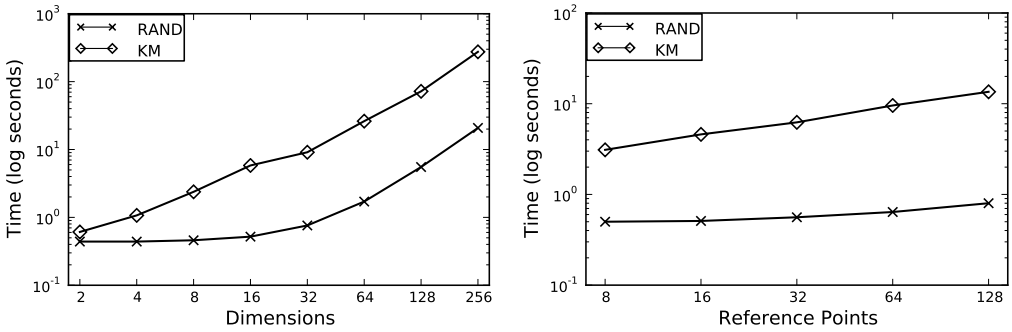
We primarily highlight three statistics from tested queries: 1) the number of candidate points returned during the filter step, 2) the number of nodes accessed in the $B^+$-tree, and 3) the time taken (in milliseconds) to perform the query and return the final exact results. Other descriptive statistics included the $B^+$-tree size (total nodes) and the number of dataspace partitions and sections (of partitions) that were checked during the query. We often express the ratio of candidates and nodes over the total number of points in the dataset and the total number of nodes in the $B^+$-tree, respectively, as this eliminates skewed results due to varying the dataset.

The first experiments are on synthetic datasets so we can properly simulate specific dataset characteristics, followed by further tests on real datasets. All artificial datasets are given a specified number of points and dimensions in the $D$-dimensional unit space $[0.0, 1.0]^D$. For clustered data, we provide the number of clusters and the

standard deviation of the independent Gaussian distributions centered on each cluster (in each dimension). The cluster centers are randomly generated during dataset creation and saved for later use. For each dataset, we randomly select 500 points as $k$-NN queries (with $k = 10$) for all experiments, which ensures that our query point distribution follows the dataset distribution.

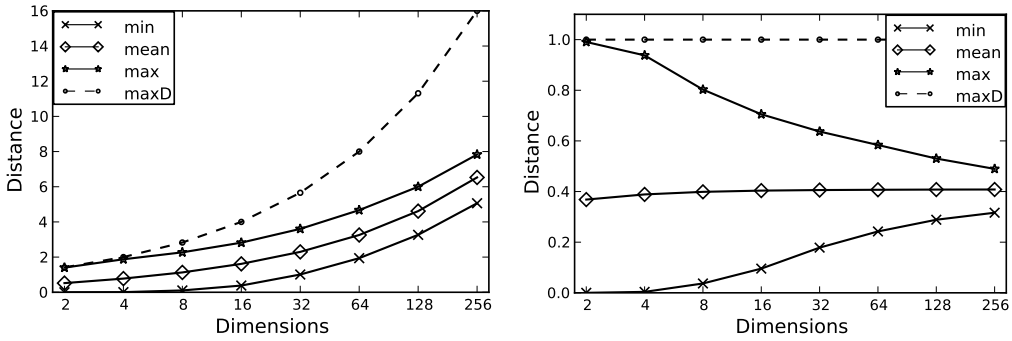
Sequential scan (SS) is often used as a benchmark comparison for worst-case performance. It must check every data point, and even though it does not use the $B^+$-tree for retrieval, total tree nodes provides the appropriate worst-case comparison. Note that all data fits in main memory, so all experiments are compared without depending on the behaviors of specific hardware-based disk-caching routines. In real-life however, disk-based I/O bottlenecks are a common concern for inefficient retrieval methods. Therefore, unless sequential scan runs significantly faster, there is a greater implied benefit when the indexing method does not have to access every data record, which could potentially be on disk.

3.4.3.2 Creation Costs. We first look at uniform data with 10,000 points from 2 to 256 dimensions to explore the inherent costs of dimensionality when creating an ID* index. We use the suggested $2D$ number of reference points, and compare purely random reference point generation to $k$-means clustering generated reference points. In Figure 3.29a, we can see the creation time is quite similar for both methods, but as expected, $k$-means has a much higher overhead cost. Notice, though, that it is asymptotically equal to random generation, and therefore quite promising if the generated reference points prove worthwhile. We also show in Figure 3.29b the time taken to create a varying number of reference points for both methods, which displays similar results.




(a) (b)

Figure 3.29: Total time (log seconds) taken to generate (a) 2D reference points over dimensions and (b) total reference points over 16 dimensions.




(a) (b)

Figure 3.30: Curse of dimensionality for pair-wise distance on uniform data.

While discussing uniform data, we take a moment to revisit and observe the well known curse of dimensionality. Figure 3.30 shows the minimum, average, and maximum pair-wise distance between all data points in the uniform datasets over dimensionality. We also include the maximum possible distance in the given $D$-dimensional unit space (maxD in Fig 3.30). We first show this with real distance values in Fig 3.30a to highlight the growth of all such distance values, indicating an increasing size of the underlying dataspace (although we note the volume remains the same because it is a unit hypercube). Furthermore, we show the same distance values normalized by the maximum dataspace distance in Fig 3.30b, which exemplifies

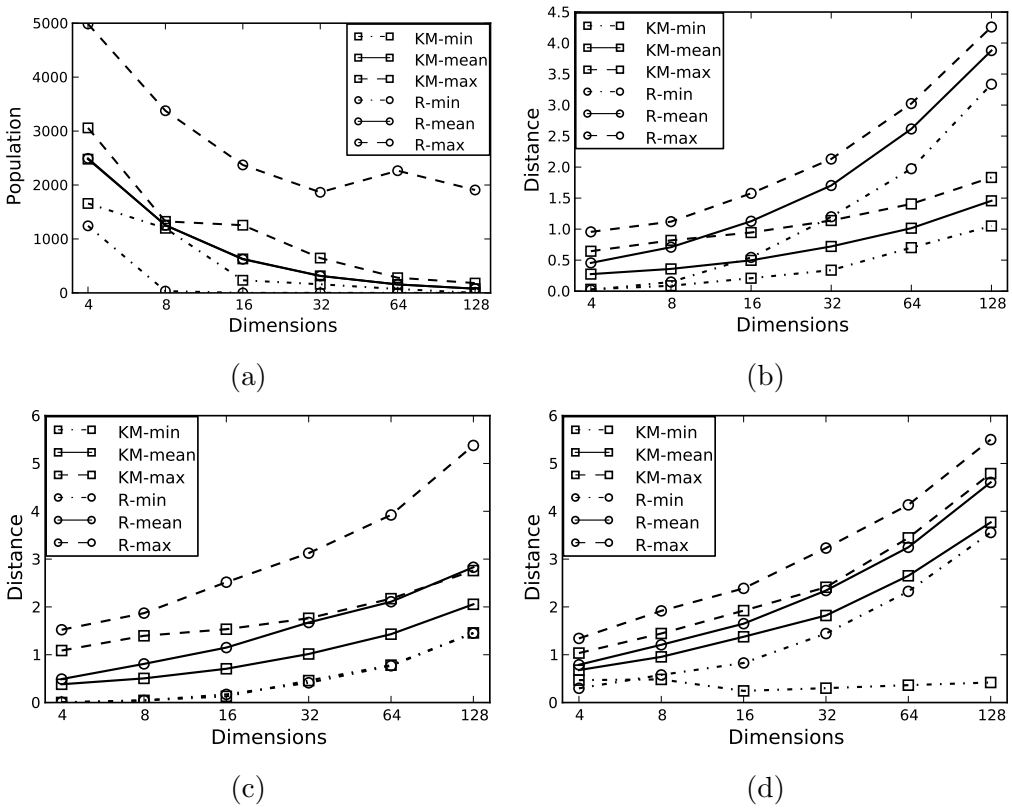



Figure 3.31: Partition statistics for clustered data.

the real crux of the problem. Here we see clearly that the minimum and maximum distances are converging on the average, leading to all equidistant data points in the high-dimensional dataspace.

Since we know that real datasets will likely not be uniform, and that ID* performance hinges on the cluster-ability of the data [39], we chose to omit performance results on these uniform datasets. The comparative results indicate similar findings from prior works [21, 39], and highlight the difficulty of trying to cluster and index high-dimensional uniform data. Results on clustered and real data provide more novelty and worthwhile discussion towards real application.

3.4.3.3 Reference Point Quality. We now move on to synthetically clustered data to explore the impact of reference point choice on the index. We generate 100,000 data points equally distributed over 16 clusters, with dimensionality varying from 4 to 128. Here we compare partition-based statistics to gain a sense of the quality of reference points provided, and show comparative performance results to back up these insights. For each dataset and set of reference points, we compute four statistics: 1) partition population count, 2) closest assignment distance for each data point, 3) pair-wise distance between points within the same partition (intra-partition), and 4) pair-wise distance between all reference points.

In Figure 3.31, we present the minimum, average, and maximum values for these four statistics using reference points created by random and $k$-means. Notice that the min and max provide helpful insight into the tightness of average, which as expected, is much looser for random reference points over all four stats. In Fig 3.31a, we see the population bounds for $k$-means partitions is very consistent, and we can see the data points assigned to each partition are much closer to the reference point (Fig 3.31b) and each other (Fig 3.31c), indicating much better clustering. Lastly in Fig 3.31d, we see the distance between each reference point dominated by the curse of dimensionality, however, the minimum value for $k$-means indicates a much more purposeful placement of partitions due to discovered cluster centers.

It should be mentioned we also compare against reference points *randomly sampled* from the set of data points, rather than *randomly generated* from the dataspace. This provides a more practical generation of "random" reference points that better follows the distribution of the data. It is most advantageous when large areas of sparse space exist, which is typical of high-dimensional data, whereby purely randomly generated reference points will be wasted in unoccupied space and never be assigned data points.

However, simply sampling data points as reference points still does not guarantee proper placement of all reference points in relation to each other in the dataspace. Therefore, both random methods show highly similar results, so we omit this second method for clarity and readability.

3.4.3.4 Optimizations. While still using the same synthetic clustered data, we present the comparative results of our algorithmic optimizations. All comparisons use the exact same $2D$ $k$-means reference points and queries. Four cases are presented in Figure 3.32: 1) original iDistance equivalent ID* (KM2D), 2) bounding search iterations to the dimensionality (KM2D-kinc), 3) caching partition distance checks to the query (KM2D-pc), and 4) optimized ID* using both the improved iterative bounds and distance check caching (KM2D-opt). We note for KM2D, $r_\Delta = 0.01$, and for KM2D-kinc, $r_\Delta = 0.01\sqrt{D}$.

Two important and mutually beneficial results can be seen here. First, note that optimizing $r_\Delta$ reduces the number of iterations from exponential with respect to $D$ to constant regardless of $D$, which matches our theoretical assumptions. Also notice that this leads to a reduction in candidates (Fig 3.32a) checked. Second, we see that, obviously, caching partition distance checks drastically reduces the number of total checks required (Fig 3.32d) which is bounded by both the number of iterations and the number of partitions. However, by applying both optimizations, this cost is near constant, as reflected by total query time in Fig 3.32b, where we also include the time taken to perform sequential scan (SS). Notice that this reduced overhead allows us to run in a similar linear fashion as SS, but the fewer than 100% candidate checks lowers our total costs below SS. These important improvements greatly mitigate the curse of dimensionality for the ID* index.

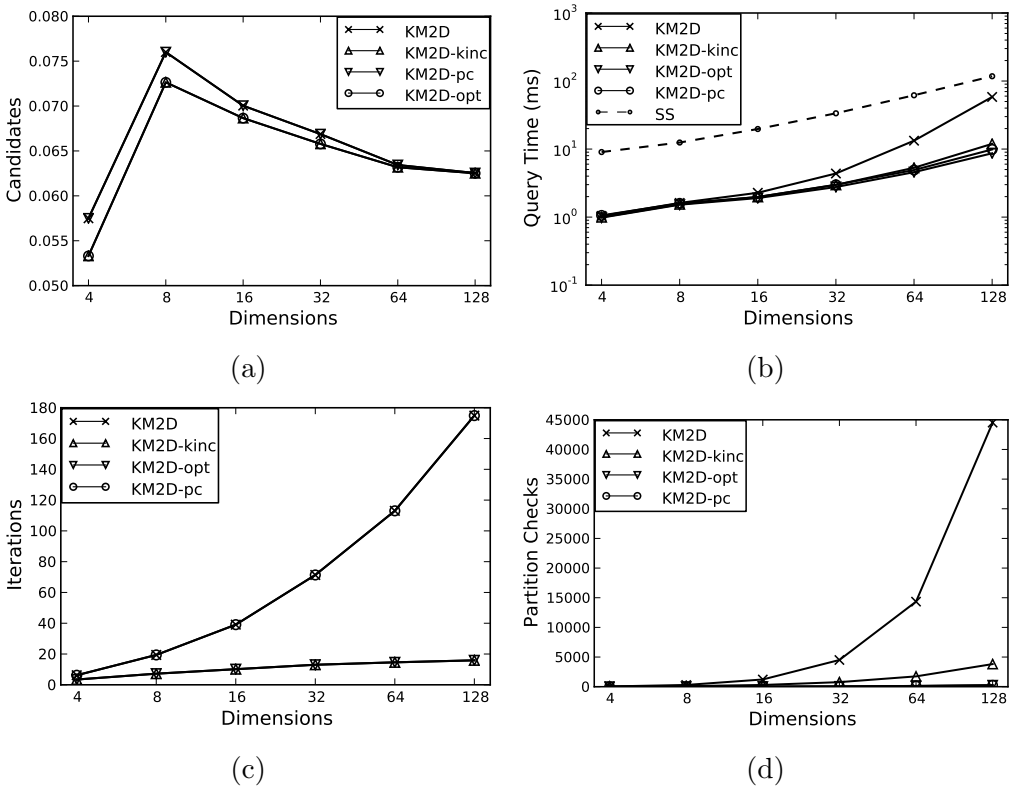



Figure 3.32: Improvement results on clustered data over dimensions.

The final results presented on this same synthetic clustered data is the overall efficiency of the add neighbor function described in Algorithm 3.3. While we must calculate the distance to every candidate, we also provide a final filter step to short circuit the additional and unnecessary overhead of traversing the set to add the candidate if it is farther away than the current farthest item in the full set, assuming $|S| = k$. In other words, by checking $distmax_S$ before attempting insertion into $S$, we reduce the costs of accessing and maintaining the sorted list of results. While only a minor practical consideration, we note that over 98.5% of all candidates (totals presented in Fig 3.32a) over all dimensionalities were eliminated before list insertion. Therefore, while we are checking the distance of approximately 5% to 7% of the data points, less than 0.1% of the data points are inserted into the result list.

3.4.3.5 Real World Data. We continue our experimental evaluations with showcasing ID* performance on two real world datasets, comparing the results of before and after optimizations. Here we vary the number of reference points relative to $D$ because the dataset characteristics are pre-defined and unchanging. As we have done in prior works, this investigates the tuning of $M$, which has been found to perform well around $M = 2D$ independent of the specific dataset being indexed [39].

The first dataset we use is the popular UCI Corel Image Features dataset, referred to as the Color dataset, which contains 68,040 points in 32 dimensions derived from color histograms (see A.3.1 for more details). The second dataset used is the Sift dataset previously described (and detailed in A.3.3), which contains one million points and was specifically created for testing nearest neighbor retrieval on 128-dimensional SIFT feature vectors [2] extracted from images.

To better highlight dataset independent performance improvements, results for both datasets are shown together. For each dataset, we refer to our optimized runs with the appended "-opt" label. In Figure 3.33, we can see across-the-board improvements on both datasets over all number of reference points. Notice that on average, fewer candidates (Fig 3.33a) are checked within fewer searched partitions (Fig 3.33b). To compare the distance checks (Fig 3.33c) and query time (Fig 3.33d), we present the optimized runs as relative values to the original non-optimized runs normalized to 1. Here we see a dramatic reduction in total distance calculations performed as the number of reference points increases, and therefore a similar reduction in total time taken to perform queries.

To gain a better idea of the reduction in distance calculations, we present each dataset separately in Figures 3.34 and 3.35. Now the cause is clear and even more dramatic. A large factor in the prior best practices suggestion of using $2D$ reference

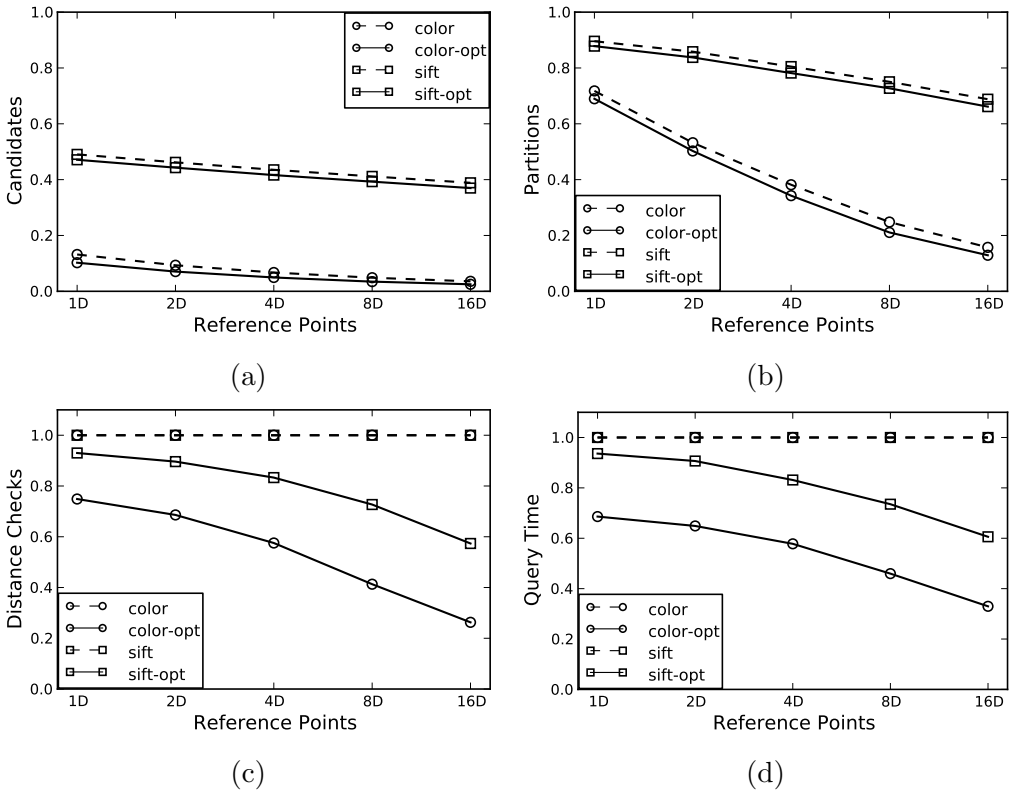
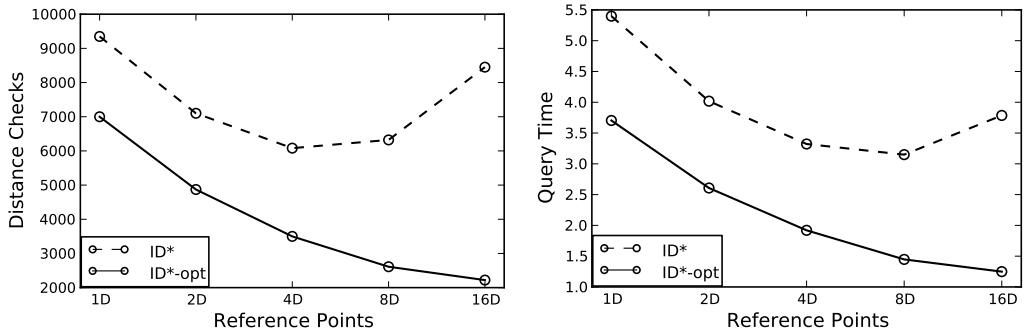



Figure 3.33: Improvement results on real world datasets.

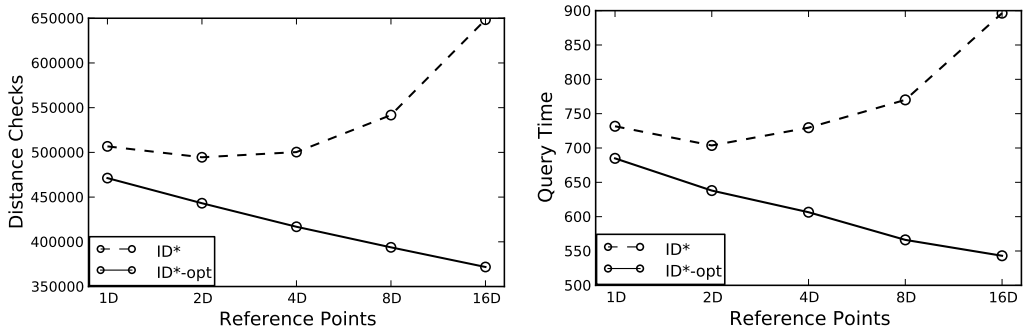
points is seen in the original runs (dotted lines), whereby the distance checks (and overall performance) has a clear concave shape. Essentially, before too few reference points leads to an increase in distance calculations from refining candidates, while too many reference points leads to an increase in distance calculations for partition overlap checks. Now however, we have mitigated most of the partition check costs, leading to a much larger and deeper concave trend in performance. While we don't see the query time turn back upward by $16D$ reference points, we do see the curve flattening out, and by definition the algorithmic overhead of too many reference points will eventually impact performance and outweigh the benefits.

Lastly, in Figure 3.36, we return to our segmentation extensions and present the results of the ID* local extension (L3) using 4 splits on the Color dataset, where again




(a) (b)

Figure 3.34: Comparative performance results for ID* improvements on Color dataset.

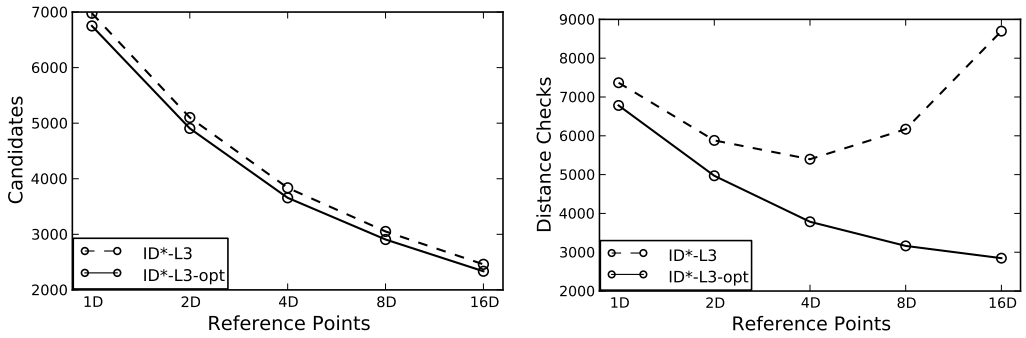



(a) (b)

Figure 3.35: Comparative performance results for ID* improvements on Sift dataset.

the "-opt" label represents the optimized algorithms. As a reminder, this means that we intend to chose 4 dimensions to segment within each partition. By using the L3 metric, we take these total segments and re-arrange them to better fit the congested areas of the dataspace, meaning that some partitions may have more (or less) than 4 splits. As expected, since these segmentation extensions use the same underlying indexing algorithms presented in this section, we see great performance improvements for these as well.

In Figure 3.36a, we see our improved algorithms return consistently fewer candidates over all amounts of reference points. More notably, in Figure 3.36b, we see a significant reduction in total distance calculations performed. This follows the




(a) (b)

Figure 3.36: Comparative performance results for ID* improvements with local extension (L3) using 4 splits on Color dataset.

findings of prior experiments, and given the only minor reduction in candidates, is most likely due to the now bounded number of partition distance checks.

### 3.4.4 Remarks

Several interesting and newly seen results were presented in this work. This might re-open the question of utilizing additional reference points. We saw a general trade off in performance and time overhead in prior experiments, but as we now see much of this overhead can be mitigated. While in general we experience a linear cost increase of additional reference point calculations, we may also be able to gain the benefits of smaller partitions with less overlap, thereby increasing pruning ability. If there exists enough separation in the data, we could also explore hierarchical clustering to have "clusters of clusters". This would be advantageous to enable a drill-down type of partitioning that would likely reduce the number of partition checks to less than the total (linear) number of reference points.

We note that interior pruning strategies for shell-like clusters could be implemented and utilized within ID* (or iDistance) without major algorithmic modifications. Currently we consider the minimum distance of a point to the reference point

as 0, in other words, the reference point itself could be a data point. Therefore, we could modify this assumption and keep track of the minimum distance much the same way we already track the maximum distance. Note however, interior pruning is far less important [4], mostly due to the general nature of data clusters and clustering algorithms. For example, we use $k$-means to find our reference points, which are by definition center weighted by all closest points, and therefore, the odds are quite high that data exist near the cluster center (reference point). Clearly exceptions to this argument can be made, but these are highly unlikely in real-world data.

In summary, we analyzed three crucial algorithmic components to distance-based indexing in regards to high-dimensional $k$-NN query retrieval. Through suggested efficiency improvements and practical input assumptions, we were able to simplify general complexity bounds and reduce theoretical costs. This was verified in experimental results using the ID* algorithm, showing greatly improved performance in synthetic and real world large-scale and high-dimensional datasets.

Future work in this area surrounds the continued investigation of algorithmic complexities and efficiency improvements to exact distance-based indexing strategies. Additional experimental analyses exploring the distances of items in varying-sized $k$ result sets could shed further light on the proper partition coverage strategy in varying data and query conditions. A related topic is the overall quality of reference points and their respective partitions, with promising research towards qualitative cluster (partition) quality expectations currently underway.

## 3.5 Bucketing Extension

In this section we present the creation of a new algorithmic extension for ID* that uses a secondary storage bucket for holding a small portion of the dataset outside

of the index. This subset of data can be intelligently chosen through heuristics to improve the dataspace characteristics for the subsequent ID* index creation. This represents a simple way of improving index performance by reducing partition overlap without sacrificing exactness in query retrieval. Over time, we aim to explore several other well-known factors of poor index performance and additional algorithmic extensions and heuristics to minimize the effects and occurrence of these factors. Here we present the basic motivation and preliminary results for this latest extension.

### 3.5.1 Motivation

We have already seen several factors that affect the performance of our distance-based ID* index. A major factor we have previously mentioned is partition overlap. The problem with overlap is that any query within this space must now be searched for within each overlapping partition, even if that partition has no data points present in the actual query location. Because of closest-point assignment strategy, there may be very distant points assigned to partitions causing their radius ($distmax$) to be quite large. Since partitions are spherical, these assigned outlier points greatly affect the entire coverage of the partition, perhaps causing overlap in an entirely different area of the dataspace than the data distributions would imply.

### 3.5.2 Overview

Our goal here is to explore outlier detection and removal to reduce this unintended (and unwanted) partition overlap. We note that high-dimensional outlier detection is also an active and challenging area of research [44]. Although we cannot simply eliminate outlier data points from the dataset while maintaining exact and complete

results, we can hold them separately in an "overflow" bucket to be searched in conjunction with the index.

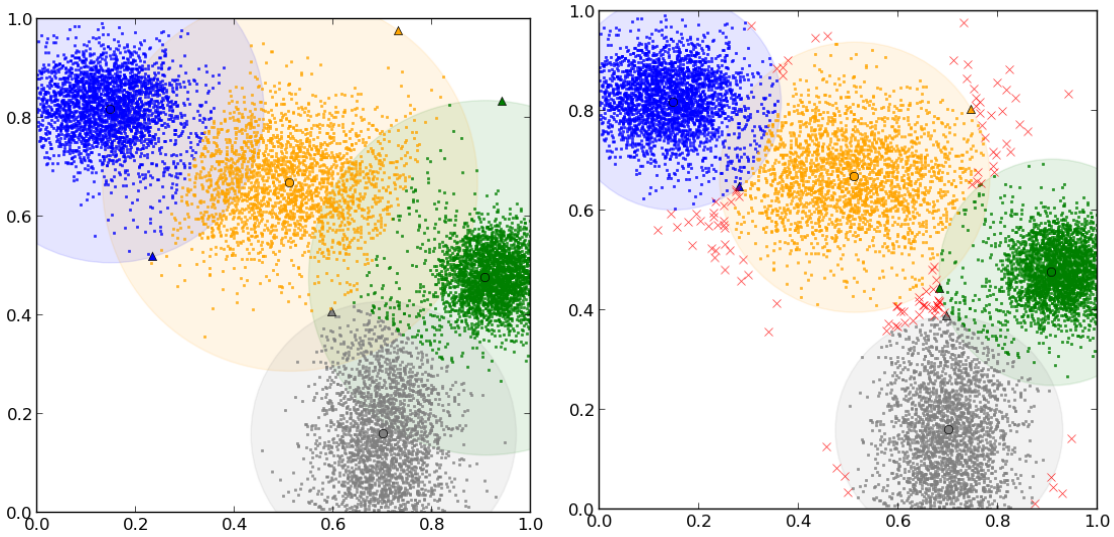
3.5.2.1 Partition Overlap. One of the most fundamental factors that affects performance is partition overlap. As we first saw in Figure 3.1, all partitions that a query sphere covers must be searched. Conceptually, overlap between two partitions occurs when their farthest-assigned point (the *distmax* value) is farther away than the actual reference points are from one another. Since points are assigned in a closest-reference point strategy, like a Voronoi diagram, this can never be caused by a point that lies directly on the vector between the two reference points.

We define an overlap metric to quantify a reflection of the partition coverage, and so we have a real-valued number to reduce. The overlap metric is defined in Equation 3.6 as the total sum of partition radii that overlap. We note this is a preliminary metric with several shortcomings. Most importantly, this metric does not take into account the total size of partitions and the volume of their overlapping areas.

$$OV = \sum_{i=1}^{M} \sum_{j=i+1}^{M} MAX\left[0, ((distmax_i + distmax_j) - dist(O_i, O_j))/2\right] \tag{3.6}$$

3.5.2.2 Outlier Detection and Bucketing. To reduce overlap, we iteratively remove the farthest point from each partition. We note this is a relatively quick operation since we already know the *distmax* point for each partition, and by removing it from assignment, we guarantee equal or smaller *distmax* value from the next farthest point. Currently, over each iteration we naïvely remove a data point from each partition that still overlaps with some other partition to any extent.

Since the $OV$ metric inherently increases in higher dimensionality, we devise an epsilon $\epsilon$ percentage of overlap to reduce during outlier removal. For example, a value of $\epsilon = 0.99$ states that we retain 99% of the total overlap. In other words, we reduce overlap by the amount of $(1 - \epsilon)$.




(a) (b)

Figure 3.37: An example of removing local outliers to decrease partition overlap.

Given the nature of real datasets, we find even a small number of local (partition-specific) outliers can drastically increase the size (and therefore possible overlap) of partitions in the dataspace. An example of this in 2D can be seen in Figure 3.37, where we show 10,000 datapoints randomly distributed into four randomly shaped elliptical clusters. In Figure 3.37a, we show the assignment of all data points to the four partitions, with their farthest points indicated by triangle markers. Notice the large amounts of overlap caused by only a few points.

An example of removing only a few outlier data points from each partition to greatly reduce overlap is shown in Figure 3.37b. Specifically, we reduce overlap to only 25% of the original by removing 108 data points (indicated by the "x" marks),

which represents only approximately 1% of the dataset. Again, while we obviously cannot consider removing data a practical solution here, the process does provide us a way to better quantify and understand the effects of overlap.

### 3.5.3 Experiments

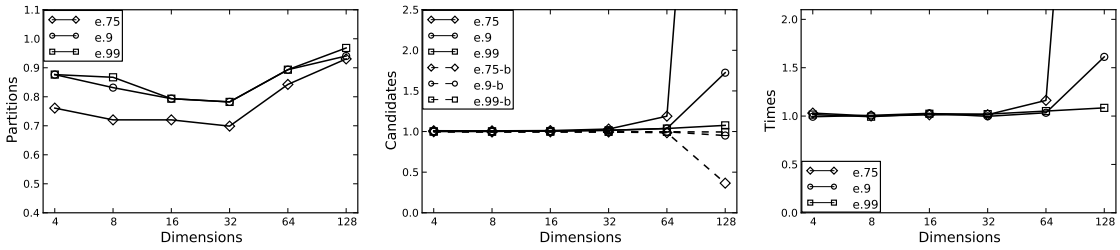
We now scale up a synthetic clustered datasets simlar to other sections, ranging from 2 to 128 dimensions with $N = 100k$ data points distributed equally over 16 randomly generated clusters. The total overlap value and total bucketed points to achieve the remaining ($\epsilon$) percentage overlap of partitions is presented in Table 3.1. We note that because each iteration of our outlier removal algorithm removes a point from every single overlapping partition before the overlap value is recalculated, there are likely circumstances of removing more points than necessary. This can be seen quite clearly by noting the equivalent total bucket points in both the 90% and 99% rows, except for the drastic difference in 128 dimensions.

Table 3.1: Total bucket points and overlap value for the clustered dataset.

| Dimensions | 4 | 8 | 16 | 32 | 64 | 128 |
|---|---|---|---|---|---|---|
| Overlap | 9.4655 | 38.45 | 112.18 | 312.79 | 634.83 | 2568.08 |
| 75% | 96 | 144 | 128 | 256 | 1280 | 63843 |
| 90% | 16 | 48 | 64 | 128 | 256 | 4845 |
| 99% | 16 | 32 | 64 | 128 | 256 | 510 |

In Figure 3.38, we show the first results of our bucketing extension. Note that all values are normalized by the regular ID* run with no bucketing and using the same reference points. The first chart shows that we do in fact search fewer partitions per query with bucketing. However, the second and third charts show that even though our indexed $k$-NN search is more effective (fewer candidates), the total candidates (including the overlap bucket size) is actually more than the original $k$-NN search

without bucketing. Therefore, due to these extra candidate checks, the resultant query time is slightly slower than normal.




(a) (b) (c)

Figure 3.38: Initial results of the outlier bucketing extension over dimensions.

### 3.5.4 Remarks

While we have demonstrated the extension works, we don't have clearly definitive results of performance improvements. This is likely due to our naïve strategy towards outlier identification and overlap metric calculation. If we can better identify the most significant outliers affecting performance, then we should be able to achieve similar relatively small bucket sizes, while even more greatly reducing overlap. One such future approach is to consider volumetric overlap instead of radial length overlap, which would implicitly account for the quite different scenarios of large overlap in small partitions and small overlap in large partitions. Similarly, by using a more greedy heuristic—one that does not need to remove an equal number of points from each partition over each iteration—may also help focus these improvements.

Additional analysis on outliers could indicate the existence of "outlier clusters" that might be better suited for another indexed partition. In other words, increasing the number of $k$-means derived clusters. With outliers removed, one could even consider re-running $k$-means clustering to better tune the partition locations to the remaining data.

# CHAPTER 4

# APPLICATIONS

This chapter presents several application areas for our ID* index. We begin with a direct comparison to several other well-known indexing algorithms that are highly-related and publicly available. Through this comparative evaluation we establish an up-to-date performance benchmark for practical considerations of real-world use.

We then move on to present our long-term research goal of creating a Content-Based Image Retrieval (CBIR) system to facilitate Query By Example (QBE) over large-scale repositories of solar images. Since most CBIR systems are highly specialized and domain specific, a large amount of interdisciplinary research was performed to properly curate solar datasets for applied use, and we take time to discuss these details throughout the chapter. Like other multimedia applications, the intrinsic content of this complex image data is represented by high-dimensional feature vectors that are then indexed in a database for later $k$-NN query retrieval. Therefore, an efficient high-dimensional data index for $k$-NN queries is one of the most critical elements of our responsive user-driven CBIR system. Without such an index, sequential search is the only viable option, and often it is simply too slow for practical use.

Lastly, we present a short case study using labeled solar data to highlight the need of an efficient index such as ID* in this domain and applications. We also briefly discuss the use of $k$-NN queries in machine learning applications as a supervised classification or regression algorithm. Since the $k$-NN classifier is a lazy learner, it delays the computational burden to the testing phase of classification. This is in contrast to many other popular algorithms, such as Naïve Bayes, Decision Trees, or Support Vector Machines, that build and train the entire model before any testing

occurs. So in other words, the speed of a $k$-NN classification algorithm is directly related to the speed of the underlying index (or lack thereof) that is performing the necessary $k$-NN query. While this typically is not a concern in small-scale machine learning research and applications, it is critical in large-scale and high-dimensional datasets common in practice and shown in this dissertation.

## 4.1 Comparative Evaluation

There are many indexing techniques available to choose from for an application. In this section, we take a look at several popular algorithms for high-dimensional data indexing and evaluate their performance against each other. We first compare two alternative indexing algorithms to ID* and establish an interesting performance benchmark across three separate software environments. Then we compare several variations of our own ID* index for a final look at the general performance results.

### 4.1.1 Experiments

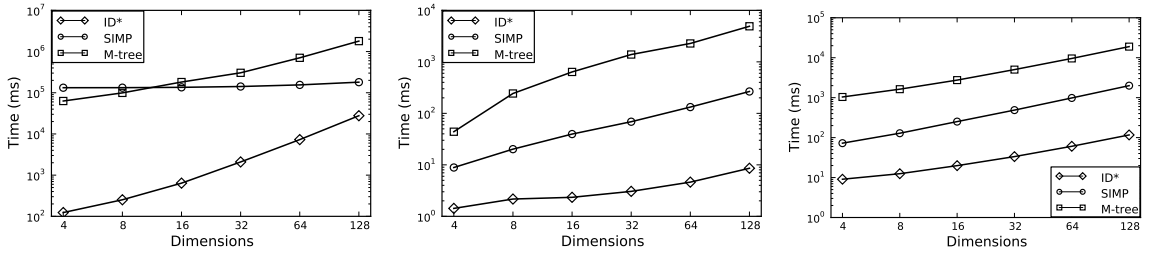
We chose two popular competitor choices as alternatives, the M-tree [20] and SIMP [32]. Since the original M-tree implementation is not maintained or up-to-date with modern language standards, we chose to use an open-source python implementation[1] and then wrote our own python sequential scan method for ensuring correct results and base-lining performance. We utilize the Java-based implementation of SIMP provided by the authors, which is described online[2].

[1] https://github.com/tburette/mtree
[2] http://alumni.cs.ucsb.edu/ vsingh/SIMP/simpsearch.html

4.1.1.1 Preliminaries. We use the same synthetic clustered datasets presented in the previous chapter. These datasets have 16 randomly generated hyper-elliptical clusters over 100k data points in various dimensionality, and we choose 500 random data points as queries. All algorithms are given the same normalized data and queries.

Note that all author-suggested defaults were used for each algorithm. One of the most important algorithmic options for SIMP is the number of viewpoints, which is similar to the number of reference points for ID* and set to a hard-coded constant of 5,000. For the M-tree, the size of leaf nodes is set to at least 8 items per node, which is the same as our $B^+$-tree setting used in ID*.

4.1.1.2 Comparing Alternatives. Figure 4.1 show the log-based times (in milliseconds) to build, query, and sequentially scan the datasets. Notice that SIMP retains a quite stable build time, while M-tree grows with dimensionality. We also see ID* grow even faster with respect to dimensionality, and this is again directly related to our choice of $2D$ reference points versus their dimensionality-agnostic setting.




(a) (b) (c)

Figure 4.1: A comparison of ID* to SIMP and M-tree.

While all three algorithms show similar trends in query time, there is a large disparity in actual clock time taken. We believe that this is mostly due to programming language implementation environment, which can be further understood in the

third chart that shows the time taken to perform sequential scan queries. Notice how similar this trend is to that of the query times.

In an attempt to eliminate language and environmental factors from the true algorithmic comparison of the three methods, we now show build and query times normalized by their respective sequential scan times. Figure 4.2 shows these same build and query times, which, now normalized, greatly highlight the differences in build times. Note that SIMP does not get faster at building the index in high-dimensional space, but only relatively faster compared to its sequential scan alternative. This clearly shows that 5,000 viewpoints is a sensible value in high dimensions, but likely overkill in lower dimensions. Here we can also clearly see that ID* is relatively faster than the other methods—as well as orders of magnitude faster in real wall-clock time.

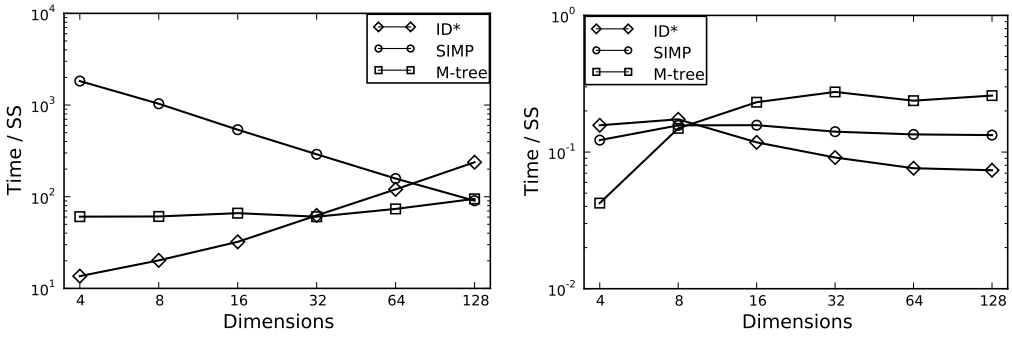



Figure 4.2: A comparison of ID* to SIMP and M-tree, normalized by each independent algorithm's sequential scan time.

4.1.1.3 Benchmarking the ID* Index. Lastly in this section, we look at several variations of our own ID* algorithm. We present results on the Color dataset, and extend upward beyond previous analyses to a total of $32D$ reference points. In Figure 4.3, we see the results of ID* (without any extensions), compared to our own original iDistance implementation and ID* with the local segmentation extension using the L3 heuristic with 4 splits. Note that while we refer to this as "original" iDistance, it

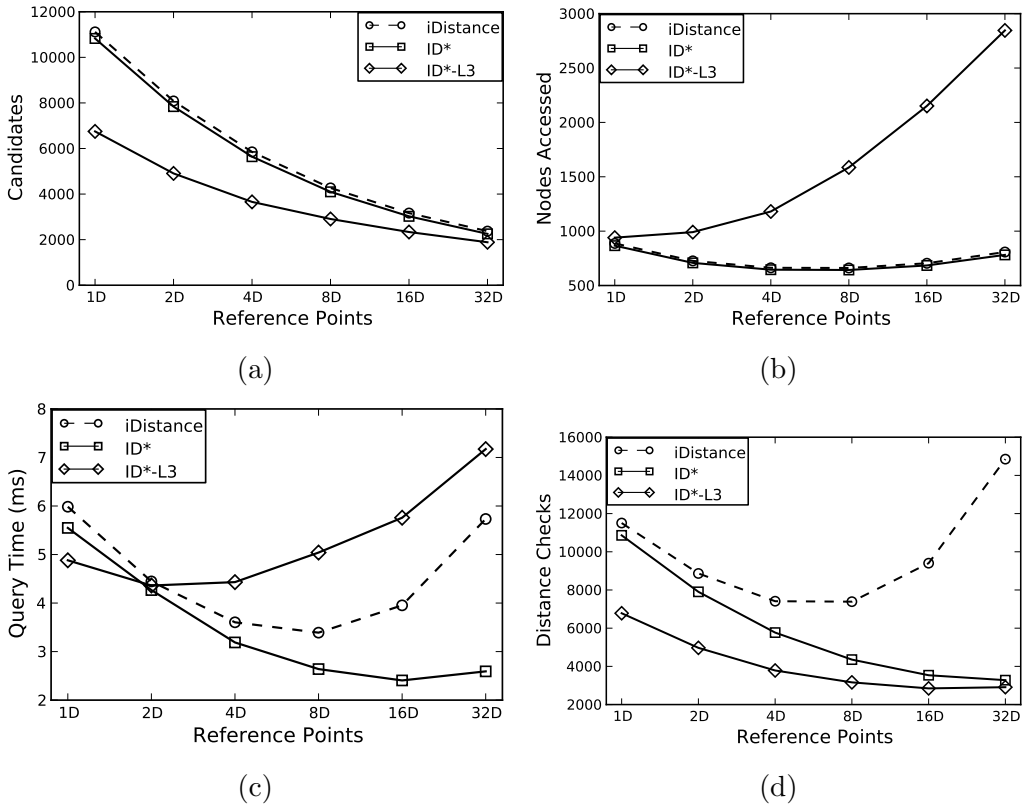



Figure 4.3: Comparative performance results for ID* improvements.

still contains many of the subtle enhancements implemented for ID*, with the specific exception of all the algorithmic improvements presented in Chapter 3.

As expected, we see an impressive reduction in total distance calculations in Figure 4.3d. However, Figure 4.3c indicates this does not always gaurauntee faster execution time, as only ID* remains consistently faster than iDistance. The issue here, as was discussed thoroughly in Chapter 3, is that the partition splits of the local extension are causing a significant increase in algorithmic overhead through additional nodes accessed. We note this trade-off would likely be much more worthwhile if the data resided on disk instead of in memory, where the greatest importance is an overall reduction in candidates retrieved. As an aside, we note that all results here are drastically better than the sequential scan alternative. Although we excluded it from

the presented charts for better readability, we note that it took on average over 23 milliseconds to perform a query, compared to the less than 5 milliseconds generally seen here. Also, the average total nodes in the $B^+$-tree was nearly 6,000, compared to the under 1,000 nodes accessed for everything but select ID*-L3 runs.

### 4.1.2 Remarks

We clearly see that ID* is superior to its predecessor iDistance. We also see that ID* generally performs better than M-tree and SIMP. Although absolute performance superiority is most likely due to implementation, we see that all three algorithms have quite similar performance in relative complexity compared to sequential scan. More work surrounding additional comparative evaluations could help in better understanding the differences in performance. Examples of this could include using the same ID* reference points as SIMP viewpoints, as well as varying the total viewpoints used in SIMP and the node sizes used in the M-tree.

## 4.2 Similarity Retrieval: CBIR Systems

The practical goal of this research is to enhance the performance of $k$-NN searches, with specific application towards facilitating user-defined region-based queries on solar image repositories. While much of this application research is only semi-related to the underlying indexing technique chosen to use or develop for our specific CBIR system, we present here an overview of our entire application developments efforts. Through interdisciplinary research such as this, we find real-world challenges that motivate novel research towards solutions that benefit everyone involved, such as the development of our state-of-the-art ID* indexing technique presented in this dissertation.

This work [45] has also produced standard benchmarks for automated feature recognition using solar image data from the Solar Dynamics Observatory (SDO) mission. We combined general purpose image parameters extracted in-line from this massive data stream of images with reported solar event metadata records from automated detection modules to create a variety of event-labeled image datasets. These new large-scale datasets can be used for computer vision and machine learning benchmarks as-is, or as the starting point for further data mining research and investigations, the results of which can also aid understanding and knowledge discovery from data (KDD) in the solar science community.

First we present an overview of the dataset creation process, including data collection, analysis, and labeling, which currently spans over two years of data and continues to grow with the ongoing SDO mission. We then highlight a case study to evaluate several data labeling methodologies and provide real-world examples of our dataset benchmarks. Preliminary results show promising capability for the classification of active and quiet regions of the Sun.

### 4.2.1 Introduction

The era of Big Data is here for solar physics. Capturing over 70,000 high-resolution images of the Sun per day, NASA's Solar Dynamics Observatory (SDO) mission produces more data than all previous solar data archives combined [46]. Given this deluge of data that will likely only increase with future missions, it is infeasible to continue traditional human-based analysis and labeling of solar phenomena in every image. In response to this issue, general research in automated detection analysis is becoming increasingly popular in solar physics, utilizing algorithms from computer vision, image

processing, and machine learning. The datasets presented here and available online[3] combine the metadata of several automated detection modules that run continuously in a dedicated data pipeline. We use these metadata catalogs to prune the massive image archive to more specific and useful forms for researchers interested in hassle-free scientific image datasets intended for region-based event (feature) recognition in each individual image over time.

This work builds upon our initial investigation into creating datasets with SDO data products [47]. We revisit the data collection process with more advanced analyses that include enhanced validation and visualization of the data streams that now extend to over two years of data starting from January 1, 2012. We then investigate event-specific recognition tasks to provide a basic performance capability assessment for several event types with varied characteristics. The results show promising recognition capability while providing insights into several solar science-related observations and future case study possibilities.

### 4.2.2 Background

To create this application and achieve our goals, we first had to start at the beginning by curating the dataset. Here we review our works [47, 45] that introduced a new public benchmark dataset of solar image data from the Solar Dynamics Observatory (SDO) mission. It combines region-based event labels from six automated detection modules with ten pre-computed image parameters for each cell over a grid-based segmentation of the full resolution images. Together, these components serve as a standardized, ready-to-use, labeled solar image dataset for general image processing research, without requiring the necessary background knowledge to properly prepare

[3]Available at `http://dmlab.cs.montana.edu/solar/`

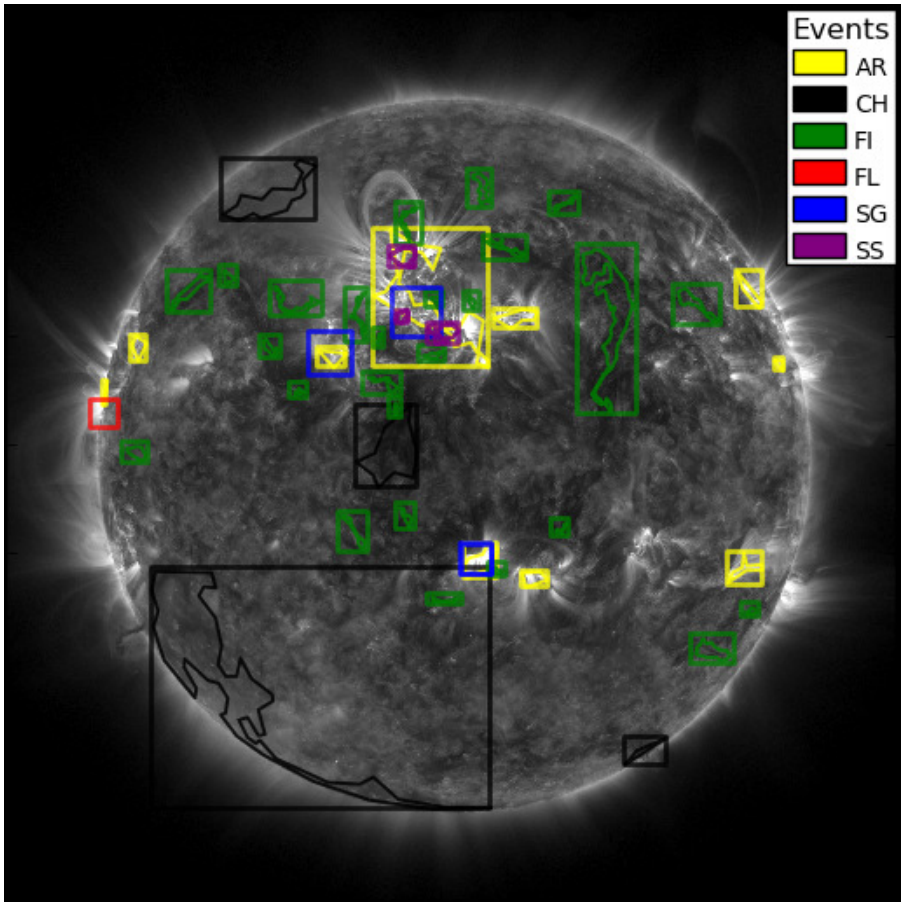



Figure 4.4: An example SDO image with labeled event regions.

it. Current efforts are underway to establish an up-to-date dataset spanning the full length of the SDO mission, which is still ongoing.

4.2.2.1 The Solar Dynamics Observatory. Launched on February 11, 2010, the Solar Dynamics Observatory (SDO) mission is the first mission of NASA's Living With a Star (LWS) program, a long term project dedicated to studying aspects of the Sun that significantly affect human life, with the goal of eventually developing a scientific understanding sufficient for prediction [48]. The purpose of the mission is to gather knowledge about the mechanics of solar magnetic activity, from the generation of the solar magnetic field to the release of magnetic energy in the solar wind, solar

flares, coronal mass ejections (CMEs), and other events [49]. The SDO is a 3-axis stabilized spacecraft in geo-synchronous orbit designed to continuously capture full-disk images of the Sun [49]. It contains three independent instruments, but our image parameter data is only from the Atmospheric Imaging Assembly (AIA) instrument, which captures images in ten separate wavebands across the ultra-violet and extreme ultra-violet spectrum, selected to highlight specific elements of solar activity [50]. The Helioseismic and Magnetic Imager (HMI) instrument is also used to detect and characterize several types of solar events.

An international consortium of independent groups, named the SDO Feature Finding Team (FFT), was selected by NASA to produce a comprehensive set of automated feature recognition modules [46]. The SDO FFT modules[4] operate through the SDO Event Detection System (EDS) at the Joint Science Operations Center (JSOC) of Stanford and Lockheed Martin Solar and Astrophysics Laboratory (LMSAL), as well as the Harvard-Smithsonian Center for Astrophysics (CfA), and NASA's Goddard Space Flight Center (GSFC). Some modules are provided with direct access to the raw data pipeline for stream-like data analysis and event detection. Even though data is made publicly accessible in a timely fashion, because of the overall size, only a small window of data is available for on-demand access, while tapes provide long-term archival storage.

All events are exclusively collected from automated SDO FFT modules, removing any human-in-the-loop limitations or biases in reporting and identification. Events are reported to the Heliophysics Event Knowledgebase (HEK), which is a centralized archive of solar event reports accessible online [51]. While event metadata can be downloaded manually through the official web interface[5], for efficient and automated

[4] `http://solar.physics.montana.edu/sol_phys/fft/`
[5] `http://www.lmsal.com/isolsearch`

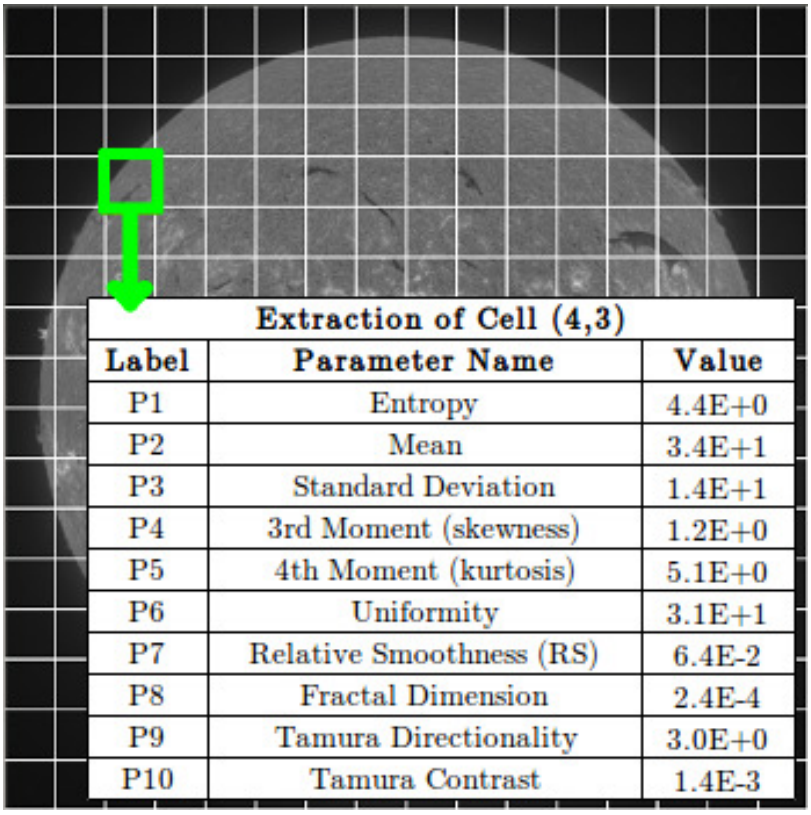
Table 4.1: A summary of solar event types.

| Event | Name | Source | CC | Reports | Module |
|---|---|---|---|---|---|
| AR | Active Region | 193 $\mathring{A}$ | Yes | 30,340 | SPoCA |
| CH | Coronal Hole | 193 $\mathring{A}$ | Yes | 24,885 | SPoCA |
| EF | Emerging Flux | HMI | No | 15,701 | Emerging flux region module |
| FI | Filament | H-alpha | Yes | 15,883 | AAFDCC |
| FL | Flare | 131 $\mathring{A}$ | No | 32,662 | Flare Detective - Trigger Module |
| SG | Sigmoid | 131 $\mathring{A}$ | No | 12,807 | Sigmoid Sniffer |
| SS | Sunspot | HMI | Yes | 6,740 | EGSO_SFC |

large-scale retrieval we developed our own open source and publicly available software application named "Query HEK", or simply QHEK[6]. We retrieve all event reports for seven types of solar events: active region (AR), coronal hole (CH), emerging flux (EF), filament (FI), flare (FL), sigmoid (SG), and sunspot (SS). These specific events were chosen because of their consistent and long-running modules and frequent reporting, leading to larger datasets capable of spanning longer periods of time. In the future, additional modules with similar reporting records could be included for comparison.

A summary of the event types can be found in Table 4.1, which states the primary source (waveband or instrument) the event is reported from, the name of the reporting module, and whether or not it has a detailed polygon-like chain code (CC) boundary in addition to the required MBR-like bounding box. Also provided for comparison are the total number of event reports over the entire two years of 2012 and 2013. We note that the EF, FI, and SS events are reported from entirely different instrumentation, but we include them for the potential of novel knowledge discovery from data and because of their abundant reports and general importance to solar physics. An example event-labeled AIA image is shown in Figure 4.4.

[6] `http://dmlab.cs.montana.edu/qhek`



| Extraction of Cell (4,3) | | |
|---|---|---|
| **Label** | **Parameter Name** | **Value** |
| P1 | Entropy | 4.4E+0 |
| P2 | Mean | 3.4E+1 |
| P3 | Standard Deviation | 1.4E+1 |
| P4 | 3rd Moment (skewness) | 1.2E+0 |
| P5 | 4th Moment (kurtosis) | 5.1E+0 |
| P6 | Uniformity | 3.1E+1 |
| P7 | Relative Smoothness (RS) | 6.4E-2 |
| P8 | Fractal Dimension | 2.4E-4 |
| P9 | Tamura Directionality | 3.0E+0 |
| P10 | Tamura Contrast | 1.4E-3 |

Figure 4.5: Example of parameter extraction over a fixed grid of image cells.

4.2.2.2 Image Parameters. As one of the 16 SDO FFT modules, our interdisciplinary research group at Montana State University (MSU) has built a "Trainable Module" for use in the first ever Content-Based Image Retrieval (CBIR) system for solar images [53]. Each $4096 \times 4096$ pixel image is segmented by a fixed-size $64 \times 64$ grid, which creates 4096 cells per image. As shown in Figure 4.5, we calculate our 10 image parameters for each $64 \times 64$ pixel cell, which are listed in Table 4.2, where $L$ stands for the number of pixels in the cell, $z_i$ is the *i-th* pixel value, $m$ is the mean, and $p(z_i)$ is the grayscale histogram representation of $z$ at $i$. The fractal dimension is calculated based on the box-counting method where $\mathrm{N}(e)$ is the number of boxes of side length $e$ required to cover the image cell. An example $64 \times 64$ pixel image plot

Table 4.2: The MSU FFT image parameters for SDO.

| Label | Name | Equation |
|---|---|---|
| P1 | Entropy | $E = -\sum_{i=0}^{L-1} p(z_i) \log_2 p(z_i)$ |
| P2 | Mean | $m = \frac{1}{L} \sum_{i=0}^{L-1} z_i$ |
| P3 | Standard Deviation | $\sigma = \sqrt{\frac{1}{L} \sum_{i=0}^{L-1} (z_i - m)^2}$ |
| P4 | Fractal Dimensionality | $D_0 = \lim_{e \to 0} \frac{\log N(\epsilon)}{\log \frac{1}{\epsilon}}$ |
| P5 | Skewness | $\mu_3 = \sum_{i=0}^{L-1} (z_i - m)^3 p(z_i)$ |
| P6 | Kurtosis | $\mu_4 = \sum_{i=0}^{L-1} (z_i - m)^4 p(z_i)$ |
| P7 | Uniformity | $U = \sum_{i=0}^{L-1} p^2(z_i)$ |
| P8 | Relative Smoothness | $R = 1 - \frac{1}{1+\sigma^2(z)}$ |
| P9 | T. Contrast | *see Tamura [52] |
| P10 | T. Directionality | *see Tamura [52] |

of each parameter can be seen in Figure 4.6, where the colors range from blue (low values) to red (high values).

In previous work, we evaluated a variety of possible image parameters to extract from the solar images. Given the volume and velocity of the data stream, the best ten parameters were chosen based on not only their classification accuracy, but also their processing time [54, 55]. Preliminary event classification was performed on a limited set of human-labeled partial-disk images from the TRACE mission [56] to determine which image parameters best represented the phenomena [57, 58]. A later investigation of solar filament classification in H-alpha images from the Big Bear Solar Observatory (BBSO) showed similar success, even with noisy region labels and a small subset of our ten image parameters [59].

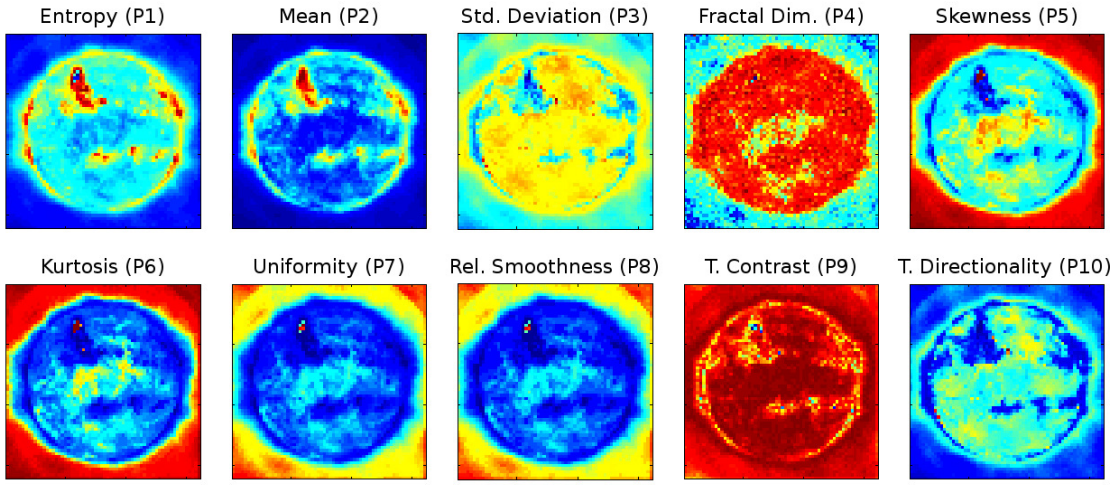



Figure 4.6: Example heatmap plots of an extracted image (in $64 \times 64$ cells) for each of our ten image parameters.

### 4.2.3 Solar Data

Next we discuss the data collection, analysis, and transformation steps required to create our labeled solar datasets. As individual datasets may require different choices in a variety of methodologies throughout these steps, here we emphasize the fundamental processes and initial data source validations, with additional dataset-specific actions provided in the later case studies. We will also highlight the 5 V's of Big Data (Volume, Velocity, Variety, Veracity, and Value) that are all exemplified in the solar data repositories presented here. The presentation is mostly focused on data from the single month of January 2012, with similar (and expanded) results available on our website[7] for all monthly and up-to-date cumulative statistics and datasets.

4.2.3.1 Collection. The Trainable Module parameter data is routinely mirrored from CfA to MSU, where we use it in-house for a variety of research projects and

[7] `http://dmlab.cs.montana.edu/solar/`

services, including our publicly available solar CBIR system[8]. Much work has gone into careful scrutiny of the raw data sources being collected. Some of this is out of necessity, as the entire process is highly automated and still ongoing as new data is constantly generated, so basic statistics and visualizations help inform the human maintainer. More importantly, in the era of Big Data much concern should be placed on the cleanliness of the initial data, which because of the volume is often more noisy than smaller datasets. Large-scale messy data can propagate through models and lead to less valuable (or worse yet, entirely misleading) results.

We operate in the SDO data stream at a static six minute cadence (velocity) on all ten AIA waveband channels (except 4500, which runs every 60 minutes). This totals an expected volume of 2,184 parameter files per day, and 797,160 files per year. We note again that each file contains 4,096 image cells, which results in an expected 8,945,664 cells per day and over 3.2 billion cells per year. Besides the total file and cell counts, we can visualize the time difference between each processed file for each wave, and quickly assess any large outages or cadence issues. An example for January 2012 can be seen in Figure 4.7, where we plot the time difference in minutes between each file, for each separate wave. Note only four small "hiccups" are seen across all waves, and in total we have 98% (66333 / 67704) of the expected files.

In similar fashion, we use QHEK to routinely pull all event instances from the SDO FFT modules for the seven event types previously described (see Table 4.1). The biggest difference with this data is the veracity of multiple independent sources and the non-uniform cadence of data. Again, this can be well visualized with a time difference plot of the time between event reports for each event type. Presented in Figure 4.8, instead of looking for expected smoothness in the plotted time differences

[8] `http://cbsir.cs.montana.edu/sdocbir`

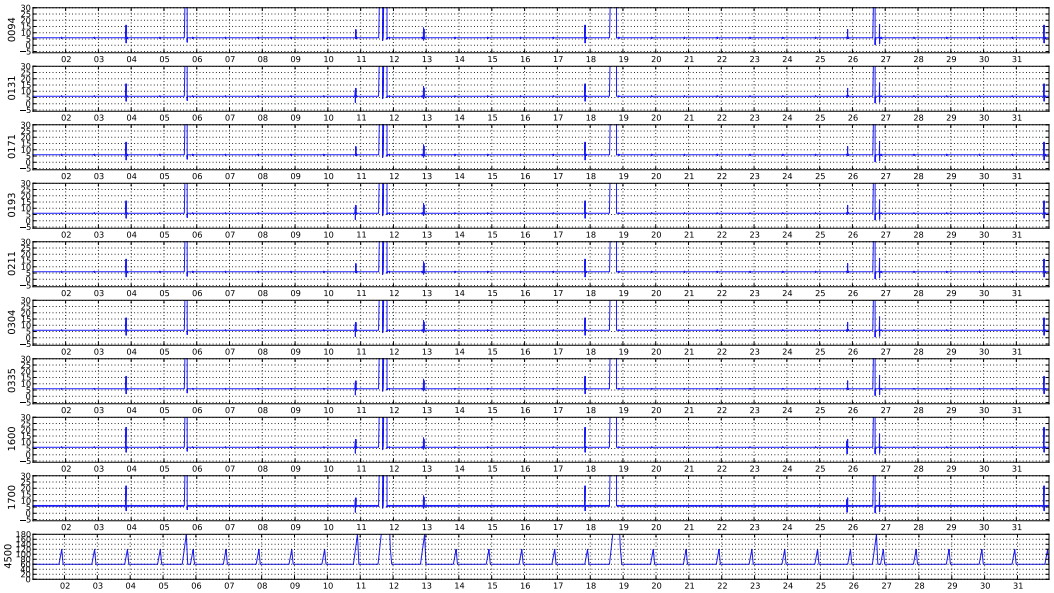



Figure 4.7: The time difference (in minutes) between image parameter files for each of the AIA waveband channels (10 total).

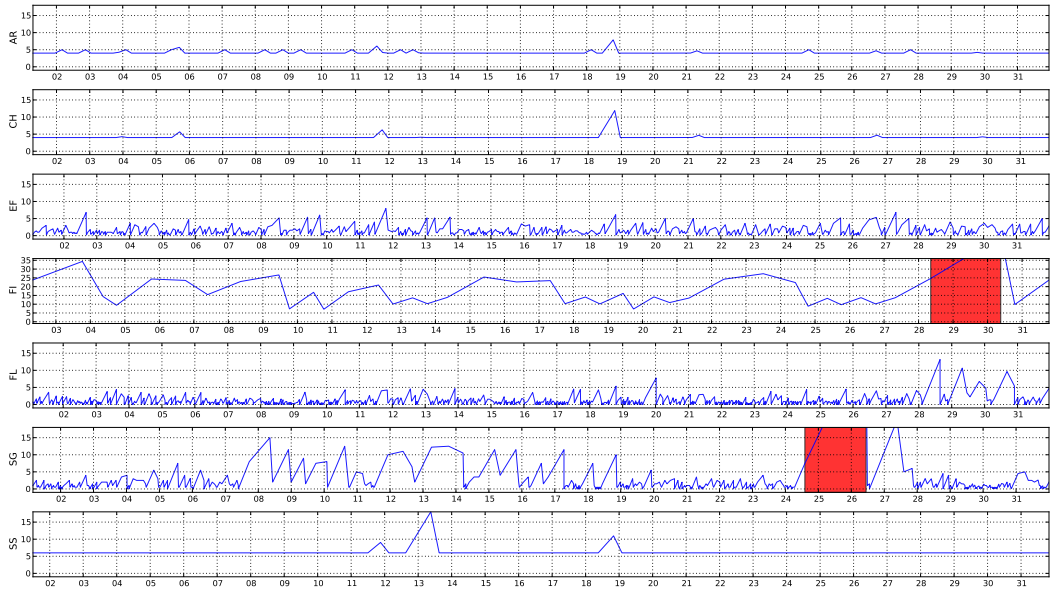



Figure 4.8: The time difference (in hours) between reports to the HEK for each of the event types (7 total).

(in hours), we can quickly identify the variety of reporting velocities for each event type. We also show an example highlight of a possible outage, automatically indicated if the difference between reports exceeds 24 hours (or 12 hours for our parameter data).

We can also visualize the varying volume and velocity of total event reports over time. This can be seen in Figure 4.9, where we plot the total event instances reported for each unique timestamp, again over the entire month of January 2012. Note how consistent the reports and counts for AR and CH events are compared to EF or FL events. This directly conforms with the intrinsic nature of these types of events, *i.e.*, active regions and coronal holes are long lasting and slowly evolving events and emerging flux and flares are relatively small and short-lived events. Also notice how this shows the steady cadence of some modules (AR, CH, SS) vs. the sporadic reporting of other modules (EF, FL, FI), which relates to the occurrence of the specific types of solar events and the data used to identify them. For example, the filament module typically only reports once or twice per day. For a broader scope, total instances for each event type over each month for the two years of data are shown in Figure 4.11.

Lastly, we can verify the general cleanliness of our valuable image parameter data by visualizing a basic 3-statistic (min, max, and avg.) time-series plot of each parameter for all cells in each image. By looking at each parameter over all wavebands, we also gain a sense of how the observational source affects our parameters. For example, in Figure 4.10 we show the mean parameter (P2) over one month of images, where the red, blue, and green lines represent the max, mean, and min values respectively. While this parameter may be less interesting than others, it is the most understandable to a human maintainer interested in sanity checking the data. Since P2 is describing the mean pixel intensity of each cell (ranging from 0 to 255), we can quickly see all the raw images are okay. Consider the obvious counter example of the 3-statistic values a solid black image would produce, which would be immediately identifiable.

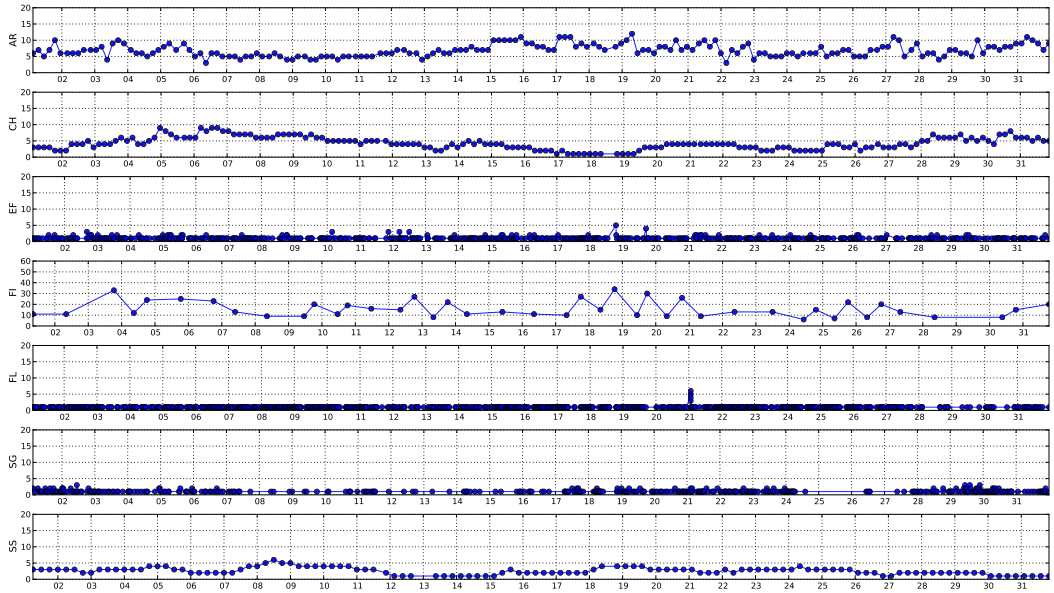



Figure 4.9: Total event reports for each unique timestamp over all event types.

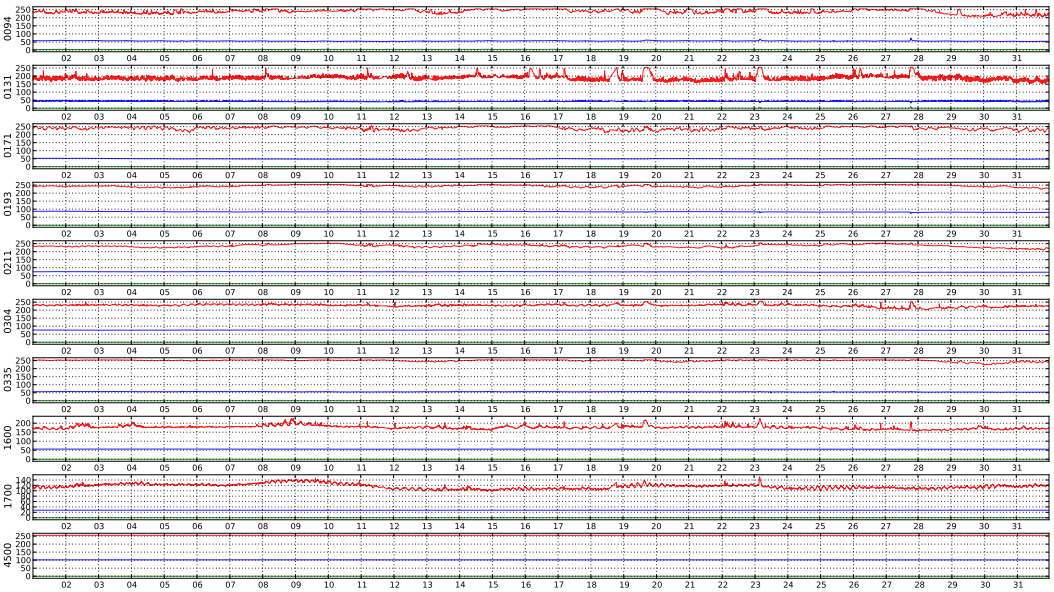



Figure 4.10: The 3-statistic (min, mean, and max) of the parameter P2 (mean) over all AIA waveband channels.

4.2.3.2 Transformation. All event reports are required to have a basic set of information that includes the time, location, and observation origin of the event instance. However, the characteristics of these attributes can vary greatly across different event types and reporting modules. This leads to two important issues: 1) instantaneous labeling of temporal events, 2) cross-origin event labels. For example, if an event has a lengthy duration, where and when should the report be labeled in images? And could an event label be applied across other image sources (wavebands)? Rather than attempt a singular strategy like previously proposed [47], we now present several alternatives and considerations for each case study. The hope is that through an accumulation of empirical results and benchmarks, we can better justify the rationale and validity of our labeling methodologies for different scenarios.

Three spatial attributes define the event location on the solar disk. The center point and minimum bounding rectangle (MBR) are required and an optional detailed polygonal outline (chain code) may also be present. These attributes are given as geometric object strings, encapsulated by the words "POINT()" and "POLYGON()", where point contains a single $(x, y)$ pair of pixel coordinates and polygon contains any number of pairs listed sequentially, *e.g.*, $(x_1, y_1, x_2, y_2, ..., x_n, y_n)$. When the polygon is used for the MBR attribute, it always contains five vertices, where the first and last are identical. We convert all spatial attributes from the helioprojective cartesian (HPC) coordinate system to pixel-based coordinates based on image-specific solar metadata [60, 61]. This process removes the need for any further expert knowledge or spatial positioning transformations.

4.2.3.3 Dataset Creation. Two important decisions impact the creation of a final dataset, 1) defining the classes to be labeled, and 2) defining data instances for such

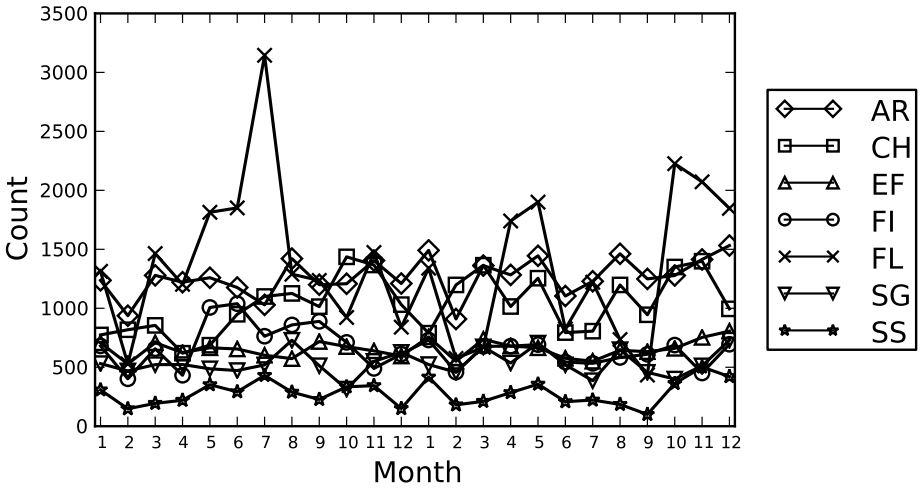



Figure 4.11: Total event counts for each month, starting with Jan 2012.

labels. Classes can represent many things, such as entire event types or subsets of event types (such as flare categories based on magnitude), while a data instance could vary from individual image cells to derived image regions to entire full-disk images. Typically we discard all image cells beyond 110% of the solar disk radius (see the larger pink ring in Figure 4.12) before any region generating or class labeling is performed, as these cells are never in our scope of solar disk-based event recognition.

Through the careful combination of our cell-based image parameters and spatio-temporal event reports, we can produce labeled datasets, which are in the standard, ready-to-use, formats popular to data mining and machine learning communities. Each row of the file represents a single dataset instance (feature vector) by a simple comma-separated list of real values, with a final value representing the class label as an integer. Additionally, images (see Figs. 4.4 and 4.12) and movies can be produced to visualize the dataset instances and quickly view the large-scale datasets.

### 4.2.4 Case Study

We present an example case study that serves the dual purpose of empirically showing recognition capability and comparing dataset creation decisions discussed in the previous section. Since we are focused on producing the best datasets for benchmarking automated event detection, we can use this performance measure as a possible indication of which labeling methods are more accurate. This also helps set precedence for moving forward with a general set of justified decisions for a future set of comprehensive dataset benchmarks.

4.2.4.1 Active Regions and Coronal Holes. We first investigate active region (AR) and coronal hole (CH) events, both of which are reported by the SPoCA module [46] in the AIA 193 Å waveband. We note the AR detection also uses AIA 171, but we chose to ignore this for a straightforward initial comparison. An example of these events are seen in Figure 4.12, which also includes metadata of the solar disk center, disk radius, and our 110% radius cut-off (in pink). We can see the bounding boxes and chain codes of all event instances, as well as the highlighted cells contained within each event chain code. We also show sample "quiet" regions (QR) highlighted in gray to be used as region-based comparison in experiments. Table 4.3 contains the total data instance counts for Jan 2012 of the various datasets presented in this work, where *Remove* cells are beyond our solar radius threshold and *Quiet* cells are non-event data instances.

Since most solar phenomena occur in or near AR events, except for CH events which we also have labeled here, we can also investigate areas of the Sun not covered by either of these labels, which we call "quiet" regions. This leads to the idea of general region-based detection rather than individual cell-based detection, as well as

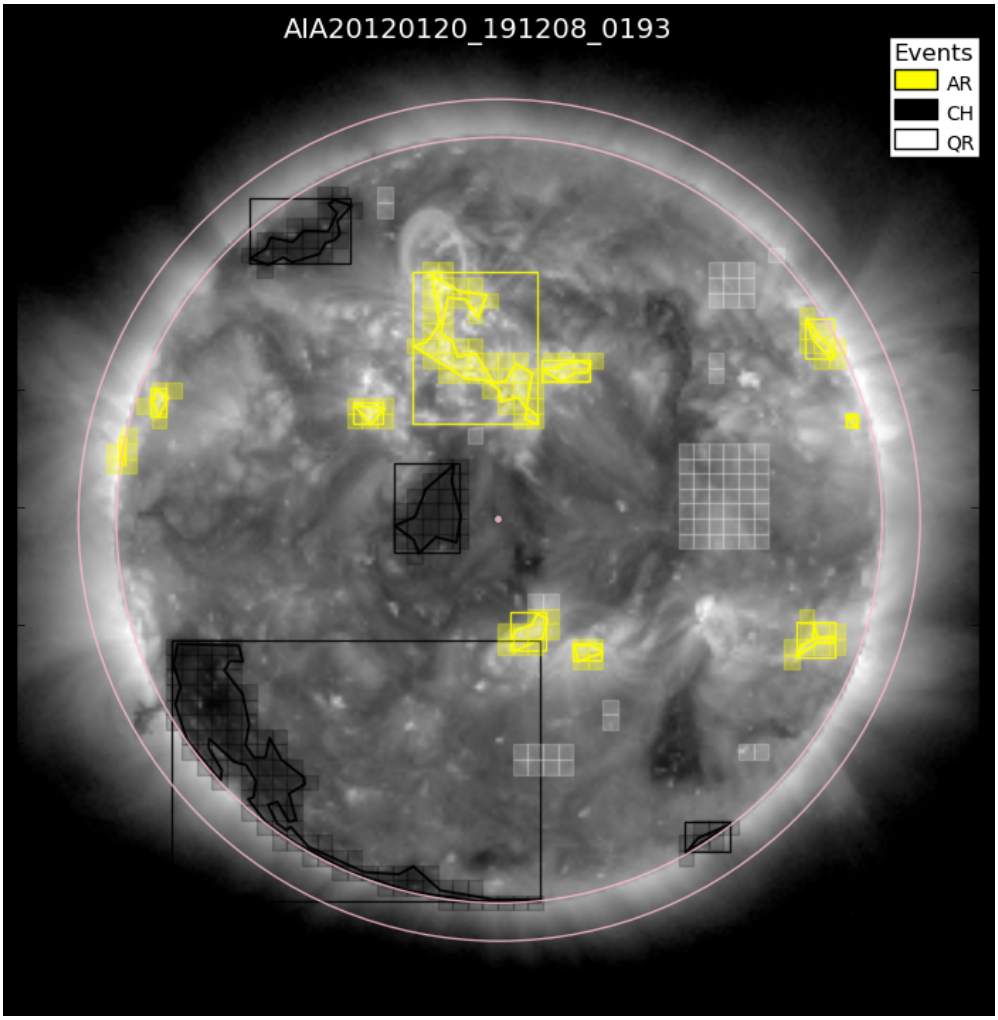



Figure 4.12: An example of event labels on chain code cells.

Table 4.3: A summary of dataset instances.

| Name | Remove | Quiet (QS) | AR | CH |
|---|---|---|---|---|
| ARCH-bbox | 293,612 | 343,147 | 21,664 | 64,201 |
| ARCH-ccode | 293,612 | 383,778 | 13,676 | 31,083 |
| ARCH-region | NA | 1,227 | 1,227 | 756 |

the possibly worthwhile detection of non-event regions that are not explicitly labeled by automated modules. The third dataset we create generates a single data instance (feature vector) for each event label covering any number of cells. We chose to again

use a simple 3-statistic summary of min, mean, and max for each original parameter over all cells covered by the label. Therefore, our new region-based data instances are 30 attributes long with the same class labels at the end. We can see in Table 4.3 that as one would expect, the bounding box labeled cells (*ARCH-bbox*) have the most data instances while the region data instances (*ARCH-region*) have the fewest. Quiet regions are created by replicating each AR event with an equal sized region not already labeled by any other AR or CH event. This is meant to provide a transition from cell-based to region-based event recognition.

4.2.4.2 Machine Learning Classification. The $k$-NN query is often utilized as a simple machine learning technique. Imagine we have a large labeled dataset and are asked to classify an unknown test item. Instead of building and training a model on the entire dataset, we could use a simple majority (or weighted) voting strategy based on the labels of the most similar items found in the dataset through a $k$-NN query.

While conceptually this approach is consistent across all dataset characteristics, there are many practical concerns to consider. Because this is a lazy-learner method, which only evaluates the model locally at the time of query, it pushes most of the computation burden to the testing phase rather than the training phrase. As we have shown, this can be extremely costly in high-dimensional and large-scale datasets, making the $k$-NN query an infeasible tool for an increasingly common environment of modern data.

We use several out-of-the-box machine learning algorithms on the multi-label datasets to provide a broad look at possible performance. These include: Naive Bayes (NB), decision trees (DT), support vector machines (SVM), K-nearest neighbor (KNN), and random forests (RF). Note we perform class balancing before all experiments, so the larger classes are randomly down-sampled to match the size of

the smallest class. We also use all attributes and do no tuning of machine learning algorithms in these initial analyses to present a non-optimized baseline. The DT uses entropy and a max tree height of six levels, while the RF uses 10 DTs with max heights of 4 levels each. All results are the average of 10-fold cross-validation data training and testing practices.

The average accuracy of each machine learning algorithm for each variation of the ARCH datasets is presented in Figure 4.13. Here we show three labeling methods for both bounding box derived label (BB) and chain code derived labels (CC). The first (*cells*) evaluates each individually labeled cell, while the third (*R-fvs*) evaluates each region-based feature vector as a whole. Bridging the gap between these two variations, the second method (*R-cells*) utilizes individually labeled cells, but the cells are only contained in the identified regions. The main difference in choosing cells solely within regions is to assess if the it impacts the QR labeled data instances.

While most algorithms perform similar for all datasets, we are interested in several comparative results here. Specifically, the effects of labeling (bounding box vs chain code) and data instance (cell vs region). One would expect additional noise introduced from the possibly vague bounding box labels, and therefore decreased accuracy, but this looks more minimal than expected. Likewise, we find minimal differences between results where cells were taken explicitly from QR regions versus randomly sampled from any quiet area of the solar disk. We see a great increase in accuracy for decision tree methods (DT and RF) using region representations over individual cells. This is likely due to an easier greedy pick of initial attributes to use, while the SVM sees a decrease in performance because of the increased number of attributes – many of which may be unhelpful and adding needless complexity. It is also worth noting that the SVM was considerably slower than all other methods with even these modest (one month) dataset sizes.

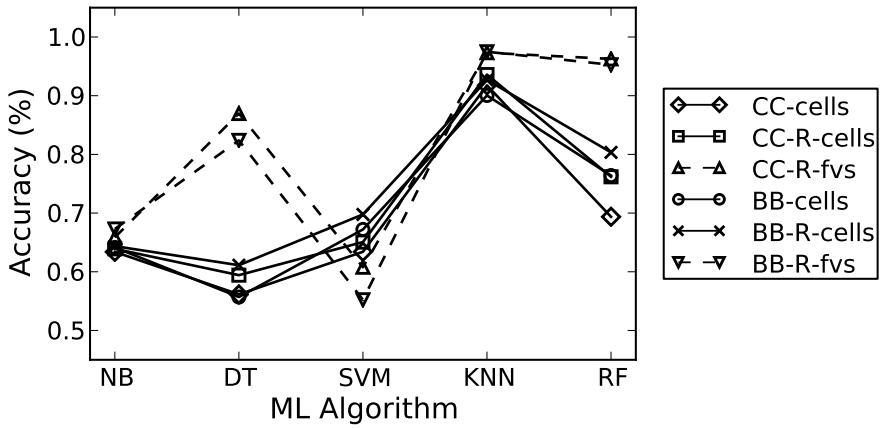
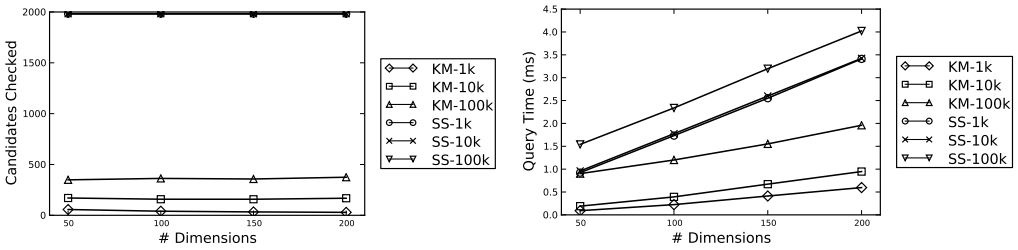



Figure 4.13: Classification results for the solar image ARCH datasets.

4.2.4.3 Data Indexing. We now show the scalability and efficiency of ID* indexing for retrieving the exact $k$-NN results of a query. We randomly select 500 data points and perform $k$-NN queries with $k$ set to 1, 10, and 100. We use proper best practices [39] and set up the ID* index with $2D$ reference points derived from $k$-means (KM) cluster centers, where $D$ is the dimensionality of the space. To show the scalability, we introduce a 50-dimensional feature vector for each of the four wavelengths and aggregate them together to form a 50, 100, 150, and 200-dimensional dataset of feature vectors. Like the last section, these feature vectors represent solar event regions, and we expand to using a 5-statistic of min, mean, median, max, and standard deviation. Note here we are not interested in the labels of the regions or the precision of retrieval, but instead focus solely on the scalability and costs of retrieval over increasing dimensionality.

In Figure 4.14a, we see the average total candidates accessed to satisfy the $k$-NN queries performed. Clearly, sequential scan (SS) must check every data record as a possible candidate, despite the number ($k$) of results returned. However, for ID* (KM) we see significantly fewer candidates required to achieve the same exact




(a) (b)

Figure 4.14: Comparing ID* (KM) to SS with varying sized $k$ on solar data.

results. As expected, the larger $k$ value requires more candidates as the search space is increased. Most importantly, we see negligible effects from increased dimensionality of the region representations.

In Figure 4.14b, we see the average total time (in milliseconds) to perform each $k$-NN query. Notice that SS methods vary in time for various sized $k$, which is due to the additional overhead of maintaining the exact (and sorted) result set list. More importantly though, we see that ID* query time grows sub-linearly in comparison to sequential scan. In fact, notice that ID* can perform $k$-NN queries in 50 dimensional space with $k = 100$ equivalent to that of SS searching for $k = 1$—and it only gets better as dimensionality is increased, well beyond the range where most other traditional indexing algorithms succumb to the curse of dimensionality.

### 4.2.5 Remarks

We presented some of the first benchmarks of event-specific data labeling for automated feature recognition in regions of SDO images. We successfully combine automated event labels with pre-computed grid-based image cell parameters to create a wide range of possible datasets. Through a case study that created a ternary-labeled dataset (AR, CH, QR), we present initial results that indicate random forest and $k$-NN algorithms can quite effectively recognize our solar events. Much future work

revolves around additional and more thorough case studies to build a comprehensive set of benchmarks.

It is clear that our superior ID*index will support any practical number of dimensions we are interested in using to represent our regions. Given the trends we see, there is little reason to suspect that the curse of dimensionality will come into effect any time soon, so the possibilities are quite open-ended. Given the nature of large-scale datasets and CBIR systems, it can be expected that we may perform $k$-NN queries with quite large values of $k$. Here we see the top 100 results in 200 dimensions can be returned in less than 50% of the time required by sequential scan by looking at only about 25% of the data records. Therefore, we can also expect these performance gains will increase as we increase the total size of the datasets.

By providing ready-to-use datasets to the public, we hope to interest more researchers from various backgrounds (computer vision, machine learning, data mining, etc.) in the domain of solar physics, further bridging the gap between many interdisciplinary and mutually-beneficial research domains. In the future, we plan to extend the datasets with: (1) a longer time frame of up-to-date and labeled data, (2) more observations from other instruments on-board SDO and elsewhere, and (3) more types of events and additional event-specific attributes for extended analysis of event "sub-type" characteristics. These labeled datasets will also facilitate better filtered searches in our CBIR systems, and may quite possibly help in the data-driven discovery and cataloging of new and unknown solar phenomena.

# CHAPTER 5

# CONCLUSIONS

This dissertation has covered many aspects of high-dimensional data indexing and $k$-NN retrieval. Through theoretical and experimental analyses we have presented numerous results and insights leading to the creation of our new and state-of-the-art ID* index. Specifically developed to facilitate exact $k$-NN queries in large-scale and high-dimensional numerical datasets, our best efforts enabled us to mitigate the curse of dimensionality and improve indexing and query performance. We have shown that with our novel algorithmic extensions and improvements, the ID* index performs significantly better than current competitors and proves quite beneficial in application.

## 5.1 Future Work

There is much to do surrounding several topics presented herein. Current research seeks to further develop better heuristics for our most recent bucketing extension, as well to merge the many separate extensions together into a more usable and extensible plug-and-play environment. We are also in the process of finishing a new database-centric implementation of ID*, which will require its own benchmarking evaluations in the near future. Lastly, all efforts are in pursuit of the nearing completion of our newest solar CBIR system prototype for region-based image querying of solar data.

### 5.1.1 Clustering for the Sake of Indexing

An important area we plan to return to with greater detail is the topic of dataset pre-processing, which impacts many related aspects of our ID* index. From the crucially necessary cluster-derived reference point locations, to partition-based segmentation splits and outlier detection, data pre-processing enables the discovery of non-trivial dataset characteristics that greatly aid indexing.

In prior work [41], we introduced the concept of *clustering for the sake of indexing* through the creation of a novel EM algorithm to perform a customized variant of $k$-means clustering. By developing our own expectation and maximization criteria, based on insights found from research presented herein, we can create $k$-means clusters specifically tailored to be used as partitions in ID*. While initial results were promising, we now have a much better understanding of the shortcomings of traditional clustering solutions, such as partition overlap and related outlier detection. Future EM algorithm implementations will account for these newfound insights and we hypothesize the overall performance will greatly improve.

### 5.1.2 DBMS Integration

The majority of current data indexing research in academia has very little immediate application in widely-used relational databases with users that could immediately benefit from the new found results. We plan to follow up our initial investigation [62] by implementing the basic functionalities of ID* into PostgreSQL. By using a procedural SQL language (PL/pgSQL), we have the ability to directly access the B-tree record points and can then traverse the tree in the same manner as we currently require during search. While not an explicitly internal implementation, it may prove to be nearly as efficient regardless. This "add-on" software package will be also much

easier for community members to try out and adopt, without having to wait for or rely on an official inclusion into the PostgreSQL repository.

## 5.2 Closing Remarks

Throughout this work, we advanced the state of high-dimensional data indexing and our overall understanding of high-dimensional dataspaces. We produced a highly adaptable and efficient indexing technique that, to the best of our knowledge, is both empirically and theoretically superior to existing comparable algorithms for exact high-dimensional $k$-nearest neighbor search. We hope this research will prove beneficial to the many scientific applications that require processing massive repositories of high-dimensional numerical data. With this thesis, we pushed the boundaries of high-dimensional data indexing for $k$-NN querying, and we hope this work will enable others to make even further contributions to the field some day in the future.

# REFERENCES CITED


[1] R. Bellman. *Dynamic Programming*. Princeton University Press, 1957.

[2] D.G. Lowe. Object recognition from local scale-invariant features. *Proc. of the 7th IEEE Inter. Conf. on Computer Vision*, Volume 2, pages 1150–1157, 1999.

[3] Tony Hey. The fourth paradigm–data-intensive scientific discovery. *E-Science and Information Management*, pages 1–1. Springer, 2012.

[4] Hanan Samet. *Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.

[5] Kevin Beyer, Jonathan Goldstein, Raghu Ramakrishnan, Uri Shaft. When is nearest neighbor meaningful? Catriel Beeri, Peter Buneman, editors, *Database Theory ICDT99*, Volume 1540 series *Lecture Notes in Computer Science*, pages 217–235. Springer Berlin Heidelberg, 1999. doi:10.1007/3-540-49257-7_15.

[6] Roger Weber, Hans-Jörg Schek, Stephen Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. *Proc. of the 24th International Conference on Very Large Data Bases*, VLDB '98, pages 194–205, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[7] Douglas Comer. Ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, June 1979. doi:10.1145/356770.356776.

[8] Rudolf Bayer, Edward M. McCreight. Organization and maintenance of large ordered indices. *Acta Informatica*, 1:173–189, 1972.

[9] Franz Aurenhammer. Voronoi diagrams—a survey of a fundamental geometric data structure. *ACM Comput. Surv.*, 23(3):345–405, September 1991. doi:10.1145/116873.116880.

[10] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. *Proceedings of the 1984 ACM SIGMOD International Conference on Management of Data*, SIGMOD '84, pages 47–57, New York, NY, USA, 1984. ACM. doi:10.1145/602259.602266.

[11] Norbert Beckmann, Hans-Peter Kriegel, Ralf Schneider, Bernhard Seeger. The r*-tree: An efficient and robust access method for points and rectangles. *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, SIGMOD '90, pages 322–331, New York, NY, USA, 1990. ACM. doi:10.1145/93597.98741.

[12] Stefan Berchtold, Christian Böhm, Hans-Peter Kriegal. The pyramid-technique: Towards breaking the curse of dimensionality. *SIGMOD Rec.*, 27:142–153, 1998.

[13] Beng Chin Ooi, Kian-Lee Tan, Cui Yu, Stephane Bressan. Indexing the edges—a simple and yet efficient approach to high-dimensional indexing. *Proceedings of the Nineteenth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '00, pages 166–174, New York, NY, USA, 2000. ACM. doi:10.1145/335168.335219.

[14] King-Ip Lin, H.V. Jagadish, Christos Faloutsos. The tv-tree: An index structure for high-dimensional data. *The VLDB Journal*, 3(4):517–542, 1994. doi:10.1007/BF01231606.

[15] Stefan Berchtold, Daniel A. Keim, Hans-Peter Kriegel. The x-tree: An index structure for high-dimensional data. *Proc. 22nd Int. Conf. on Very Large Databases*, pages 28–39, 1996.

[16] D.A. White, R. Jain. Similarity indexing with the ss-tree. *Data Engineering, 1996. Proceedings of the Twelfth International Conference on*, pages 516–523, Feb 1996. doi:10.1109/ICDE.1996.492202.

[17] R. Zhang, B.C. Ooi, K.-L. Tan. Making the pyramid technique robust to query types and workloads. *Proc. 20th Inter. Conf. on Data Eng.*, pages 313–324, 2004.

[18] Q. Shi, B. Nickerson. Decreasing Radius K-Nearest Neighbor Search Using Mapping-based Indexing Schemes. Technical Report, University of New Brunswick, 2006.

[19] Jeffrey K. Uhlmann. Satisfying general proximity / similarity queries with metric trees. *Information Processing Letters*, 40(4):175 – 179, 1991. doi:http://dx.doi.org/10.1016/0020-0190(91)90074-R.

[20] Paolo Ciaccia, Marco Patella, Pavel Zezula. M-tree: An efficient access method for similarity search in metric spaces. *Proceedings of the 23rd International Conference on Very Large Data Bases*, VLDB '97, pages 426–435, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[21] H. V. Jagadish, Beng Chin Ooi, Kian-Lee Tan, Cui Yu, Rui Zhang. iDistance: An adaptive B+-tree based indexing method for nearest neighbor search. *ACM Trans. Database Syst.*, 30:364–397, June 2005.

[22] Cui Yu, Beng C. Ooi, Kian-Lee Tan, H. V. Jagadish. Indexing the Distance: An Efficient Method to KNN Processing. *VLDB '01: Proc. of the 27th International Conference on Very Large Data Bases*, pages 421–430, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[23] Junqi Zhang, Xiangdong Zhou, Wei Wang, Baile Shi, Jian Pei. Using high dimensional indexes to support relevance feedback based interactive images retrieval. *Proc. of the 32nd VLDB Conf.*, pages 1211–1214. VLDB Endowment, 2006.

[24] Heng Tao Shen, Beng Chin Ooi, Xiaofang Zhou. Towards effective indexing for very large video sequence database. *Proc. of the ACM SIGMOD Inter. Conf. on Mgmt. of Data*, pages 730–741. ACM, 2005.

[25] Sergio Ilarri, Eduardo Mena, Arantza Illarramendi. Location-dependent queries in mobile contexts: Distributed processing using mobile agents. *IEEE Trans. on Mobile Computing*, 5(8):1029–1043, 2006.

[26] Christos Doulkeridis, Akrivi Vlachou, Yannis Kotidis, Michalis Vazirgiannis. Peer-to-peer similarity search in metric spaces. *Proc. of the 33rd VLDB Conf.*, pages 986–997, 2007.

[27] Lin Qu, Yaowu Chen, Xiaofan Yang. iDistance based interactive visual surveillance retrieval algorithm. *Intelligent Computation Technology and Automation (ICICTA)*, Volume 1, pages 71–75. IEEE, Oct 2008.

[28] Yasushi Sakurai, Masatoshi Yoshikawa, Shunsuke Uemura, Haruhiko Kojima. The a-tree: An index structure for high-dimensional spaces using relative approximation. *Proceedings of the 26th International Conference on Very Large Data Bases*, VLDB '00, pages 516–526, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[29] Norio Katayama, Shin'ichi Satoh. The sr-tree: An index structure for high-dimensional nearest neighbor queries. *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, SIGMOD '97, pages 369–380, New York, NY, USA, 1997. ACM. doi:10.1145/253260.253347.

[30] Agma Traina, Agma JM Traina, Christos Faloutsos, i in. Similarity search without tears: the omni-family of all-purpose access methods. *Data Engineering, 2001. Proceedings. 17th International Conference on*, pages 623–630. IEEE, 2001.

[31] Sunil Arya, David M Mount, Nathan S Netanyahu, Ruth Silverman, Angela Y Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *Journal of the ACM (JACM)*, 45(6):891–923, 1998.

[32] Vishwakarma Singh, Ambuj Kumar Singh. Simp: accurate and efficient near neighbor search in high dimensional spaces. *Proceedings of the 15th International Conference on Extending Database Technology (EDBT)*, pages 492–503, 2012.

[33] Piotr Indyk, Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. *Proceedings of the Thirtieth Annual ACM*

*Symposium on Theory of Computing*, STOC '98, pages 604–613, New York, NY, USA, 1998. ACM. doi:10.1145/276698.276876.

[34] Yufei Tao, Ke Yi, Cheng Sheng, Panos Kalnis. Quality and efficiency in high dimensional nearest neighbor search. *Proc. of the ACM SIGMOD Inter. Conf. on Mgmt. of Data*, pages 563–576. ACM, 2009.

[35] K. Fukunaga, Patrenahalli M. Narendra. A branch and bound algorithm for computing k-nearest neighbors. *Computers, IEEE Transactions on*, C-24(7):750–753, July 1975. doi:10.1109/T-C.1975.224297.

[36] James MacQueen, i in. Some methods for classification and analysis of multivariate observations. *Proceedings of the fifth Berkeley symposium on mathematical statistics and probability*, pages 281–297. Oakland, CA, USA., 1967.

[37] Stuart P Lloyd. Least squares quantization in pcm. *Information Theory, IEEE Transactions on*, 28(2):129–137, 1982.

[38] Michael A. Schuh, Tim Wylie, Chang Liu, Rafal A. Angryk. Approximating high-dimensional range queries with kNN indexing techniques. Zhipeng Cai, Alex Zelikovsky, Anu Bourgeois, editors, *Computing and Combinatorics, the Proc. of the 20th International Conference (COCOON '14)*, Volume 8591 series *Lecture Notes in Computer Science*, pages 369–380. Springer International Publishing, Aug. 2014. doi:10.1007/978-3-319-08783-2_32.

[39] Michael A. Schuh, Tim Wylie, Juan M. Banda, Rafal A. Angryk. A comprehensive study of iDistance partitioning strategies for kNN queries and high-dimensional data indexing. Georg Gottlob, Giovanni Grasso, Dan Olteanu, Christian Schallhart, editors, *Proc. of the 29th British National Conference on Databases (BNCOD), Big Data*, Volume 7968 series *Lecture Notes in Computer Science*, pages 238–252. Springer Berlin Heidelberg, July 2013. doi:10.1007/978-3-642-39467-6_22.

[40] T. Zhang, R. Ramakrishnan, M. Livny. BIRCH: an efficient data clustering method for very large databases. *SIGMOD Rec.*, 25(2):103–114, June 1996.

[41] Tim Wylie, Michael A Schuh, John W Sheppard, Rafal A Angryk. Cluster analysis for optimal indexing. *Proc. of the 26th Int. Florida Artificial Intelligence Research Society Conference (FLAIRS)*, pages 166–171. AAAI, May 2013.

[42] Michael A Schuh, Tim Wylie, Rafal A Angryk. Mitigating the curse of dimensionality for exact kNN retrieval. *Proc. of the 27th Int. Florida Artificial Intelligence Research Society Conference (FLAIRS)*, pages 363–368. AAAI, May 2014.

[43] Michael A. Schuh, Tim Wylie, Rafal A. Angryk. Improving the performance of high-dimensional kNN retrieval through localized dataspace segmentation and hybrid indexing. Barbara Catania, Giovanna Guerrini, Jaroslav Pokorný, editors, *Proc. of the 17th East European Conference on Advances in Databases and Information Systems (ADBIS)*, Volume 8133 series *Lecture Notes in Computer Science*, pages 344–357. Springer Berlin Heidelberg, Sept. 2013. doi:10.1007/978-3-642-40683-6_26.

[44] Arthur Zimek, Erich Schubert, Hans-Peter Kriegel. A survey on unsupervised outlier detection in high-dimensional numerical data. *Statistical Analysis and Data Mining*, 5(5):363–387, 2012. doi:10.1002/sam.11161.

[45] Michael A. Schuh, Rafal A. Angryk. Massive labeled solar image data benchmarks for automated feature recognition. *2014 IEEE International Conference on Big Data (Big Data)*, pages 53–60. IEEE, Oct. 2014.

[46] P. C. H. Martens, G. D. R. Attrill, A. R. Davey, A. Engell, S. Farid, P. C. Grigis, i in. Computer vision for the solar dynamics observatory (SDO). *Solar Physics*, Jan 2012.

[47] Michael A Schuh, Rafal A Angryk, Karthik Ganesan Pillai, Juan M Banda, Petrus C Martens. A large-scale solar image dataset with labeled event regions. *Proc. of the 20th IEEE International Conference on Image Processing (ICIP)*, pages 4349–4353. IEEE, Sept. 2013.

[48] G. L. Withbroe. Living With a Star. *AAS/Solar Physics Division Meeting #31*, Volume 32 series *Bulletin of the American Astronomical Society*, page 839, May 2000.

[49] W. Pesnell, B. Thompson, P. Chamberlin. The solar dynamics observatory (sdo). *Solar Physics*, 275:3–15, 2012.

[50] James Lemen, Alan Title, David Akin, Paul Boerner, Catherine Chou, i in. The Atmospheric Imaging Assembly (AIA) on the Solar Dynamics Observatory (SDO). *Solar Physics*, 275:17–40, 2012.

[51] N. Hurlburt, M. Cheung, C. Schrijver, L. Chang, S. Freeland, i in. Heliophysics event knowledgebase for solar dynamics observatory (SDO) and beyond. *Solar Phys.*, 2010.

[52] H. Tamura, S. Mori, T. Yamawaki. Texture features corresponding to visual perception. *IEEE Transactions on Systems, Man, and Cybernetics*, 8(6):460–472, 1978.

[53] M.A. Schuh, J. Banda, R. Angryk, P.C. Martens. Introducing the first publicly available content-based image-retrieval system for the solar dynamics observatory mission. *AAS/SPD Meeting*, Volume 44 series *Solar Physics Division Meeting*, page #100.97, July 2013.

[54] Juan M. Banda, Rafal A. Angryk. Selection of image parameters as the first step towards creating a CBIR system for the solar dynamics observatory. *International Conference on Digital Image Computing: Techniques and Applications (DICTA)*, pages 528–534, 2010.

[55] Juan M. Banda, Rafal A. Angryk. An experimental evaluation of popular image parameters for monochromatic solar image categorization. *Proc. of the 23rd Florida Artificial Intelligence Research Society Conference (FLAIRS)*, pages 380–385, 2010.

[56] B.N. Handy, L.W. Acton, C.C. Kankelborg, C.J. Wolfson, D.J. Akin, i in. The transition region and coronal explorer. *Solar Physics*, 187:229–260, 1999.

[57] J. M. Banda, R. A. Angryk, P. C. H. Martens. On the surprisingly accurate transfer of image parameters between medical and solar images. *18th IEEE Int. Conf. on Image Processing (ICIP)*, pages 3669–3672, 2011.

[58] J. M. Banda, R. A. Angryk, P. C. H. Martens. Steps toward a large-scale solar image data analysis to differentiate solar phenomena. *Solar Physics*, pages 1–28, 2013. doi:10.1007/s11207-013-0304-x.

[59] M. A. Schuh, J. M. Banda, P. N. Bernasconi, R. A. Angryk, P. C. H. Martens. A comparative evaluation of automated solar filament detection. *Solar Physics*, 289(7):2503–2524, July 2014. doi:10.1007/s11207-014-0495-9.

[60] WT Thompson. Coordinate systems for solar image data. *Astronomy and Astrophysics*, 449(2):791–803, 2006.

[61] W. D. Pence. Cfitsio, v2.0: A new full-featured data interface. *Astronomical Data Analysis Software and Systems*, California, 1999.

[62] Michael A Schuh. Google summer of code (gsoc) internship, 2013.

[63] Thierry Bertin-Mahieux, Daniel P.W. Ellis, Brian Whitman, Paul Lamere. The million song dataset. *Proceedings of the 12th International Conference on Music Information Retrieval (ISMIR 2011)*, 2011.

[64] Hervé Jégou, Matthijs Douze, Cordelia Schmid. Product Quantization for Nearest Neighbor Search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33(1):117–128, January 2011. doi:10.1109/TPAMI.2010.57.

[65] DavidG. Lowe. Distinctive image features from scale-invariant keypoints. *International Journal of Computer Vision*, 60(2):91–110, 2004. doi:10.1023/B:VISI.0000029664.99615.94.

# APPENDIX A

## DATASETS

Here we list the characteristics of all the datasets used in this dissertation for easy reference to the reader. We also discuss the pre-process steps required to configure the datasets for standard use.

## A.1 Setup

We first note that all datasets must be pre-processed to adhere to program input standards. Namely, all files are in comma separated values (CSV) format and records such as data points must start with an ID field. If necessary, we also use min-max normalization on each independent dimension to transform the data into the standard unit dataspace.

## A.2 Synthetic

Synthetic datasets are algorithmically generated for testing specific characteristics, such as dataset size ($N$), dimensionality ($D$), cluster density through standard deviation (SD). Here we briefly present example dataset distributions of uniform and clustered datasets in only 8 dimensions for clarity. In Figure A.1, we see, as expected a completely uniform distribution of data points over the entire dataspace range and across all dimensions. In Figure A.2, we can see that the data distribution of each dimension varies based on the exact location of clusters present in the dataspace.

## A.3 Real Data

We use three real world datasets attained from independent sources for a variety of experiments. We also highlight the useage of our own labeled solar image datasets

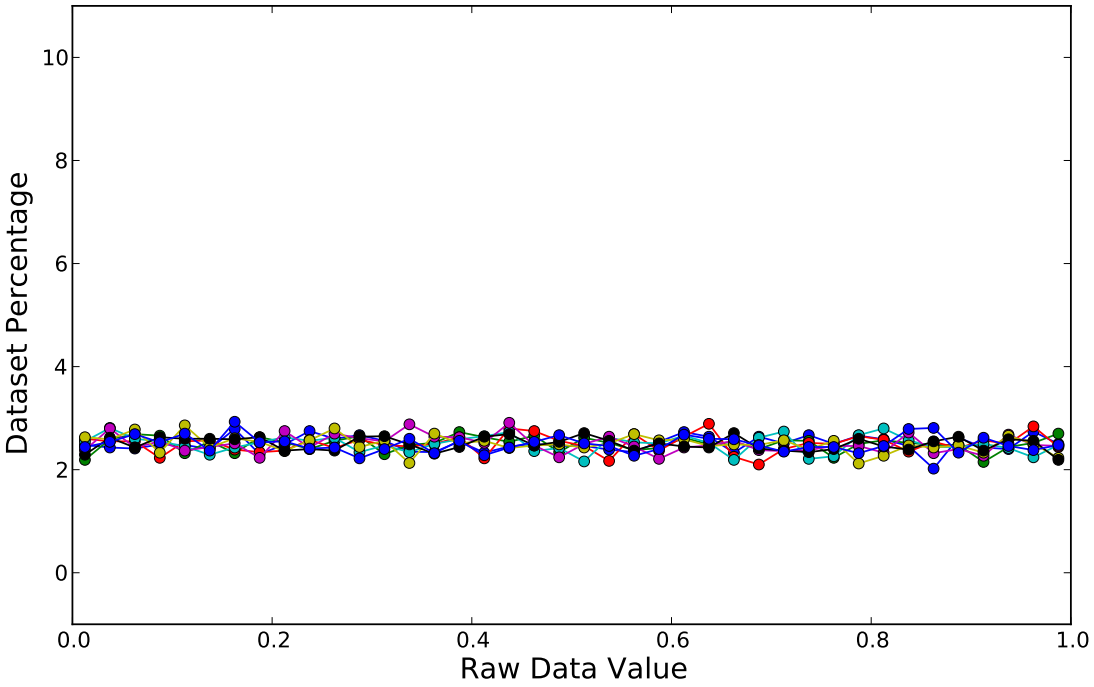



Figure A.1: Uniform dataset distribution.

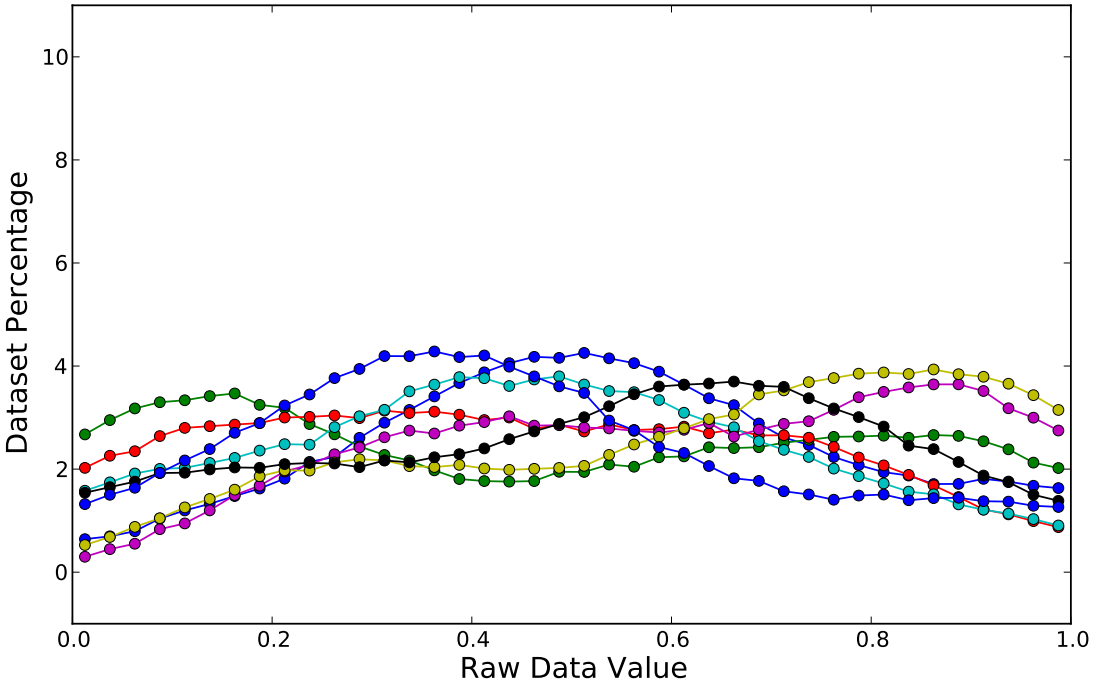



Figure A.2: Clustered dataset distribution.

described in the Applications chapter. Table A.1 contains a quick overview of the dataset characteristics and short-hand names.

Table A.1: Characteristics of real datasets.

| Name | D | N | ID | Label |
|---|---|---|---|---|
| Color | 32 | 68,040 | Yes | No |
| Music | 90 | 515,345 | No | Yes |
| Sift | 128 | 1,000,000 | Yes | No |

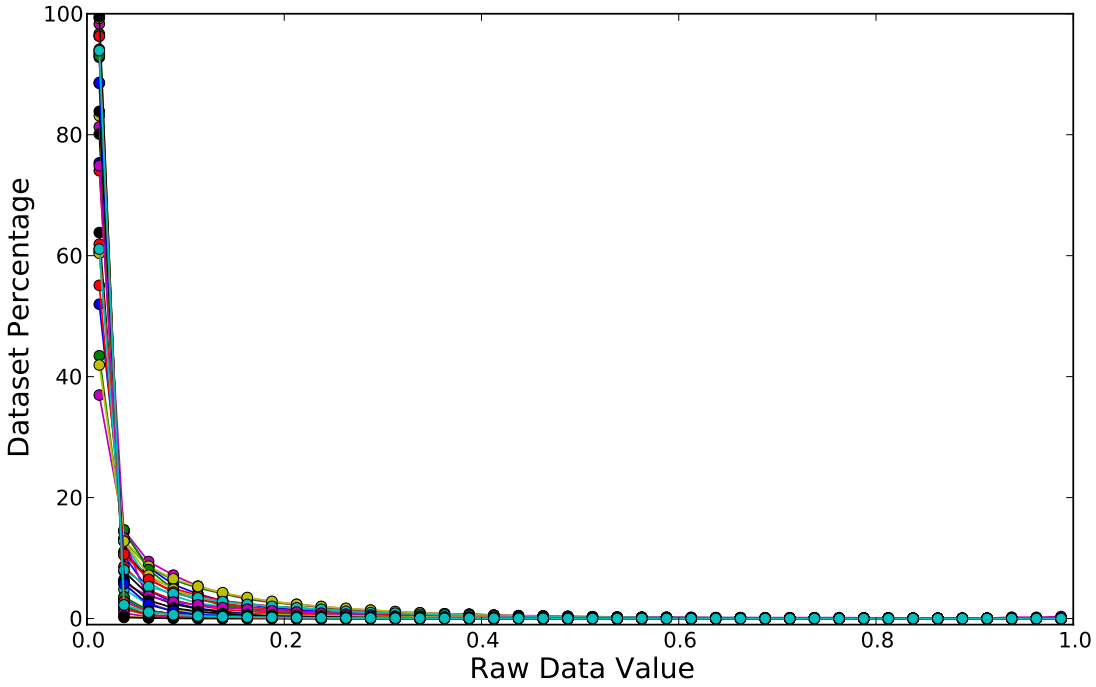
### A.3.1 Color

The Corel Image Features dataset (Color) is publicly available online[1] from the University of California (UC) Irvine's Machine Learning Repository. For our work, we only use the 32-dimensional color histogram attributes, which describes the original image data in the HSV color space using the density value of each color. These 32 colors (dimensions) characterize the 68,040 total data points in the dataset.

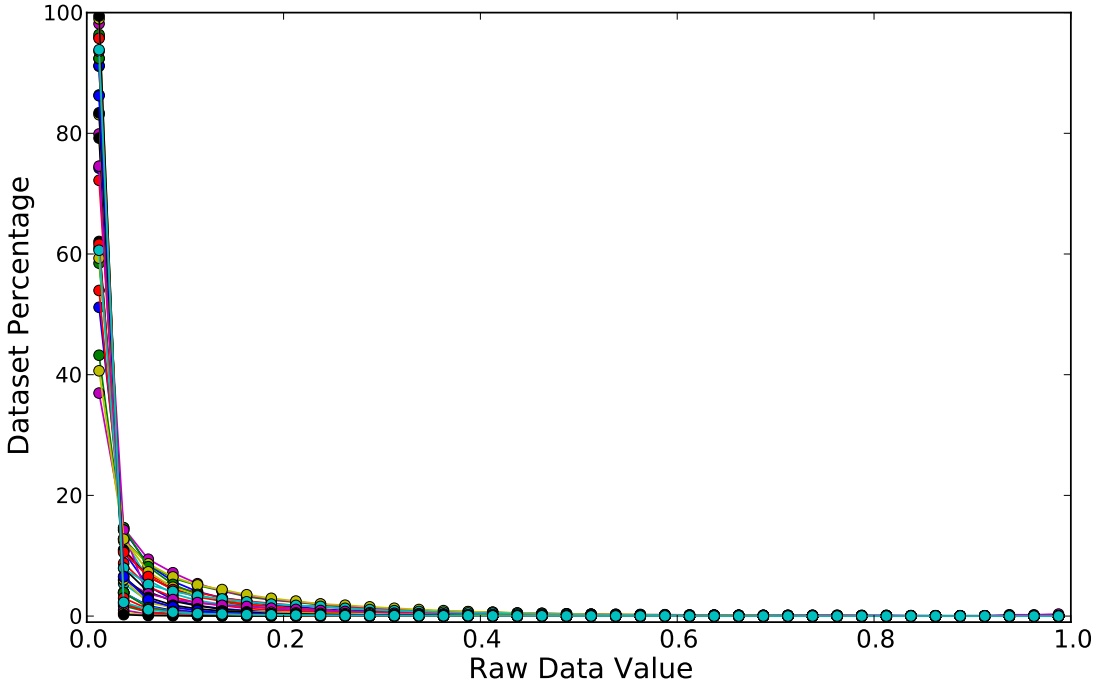
The raw dataset is space-separated with a sequential (1-based) ID at the start of each row, which we preserve regardless. While the dataset is already [0,1] bounded, we note that many dimensions do not extend the entire range, and the overall global minimum and maximum is 0.0 and 1.0001, respectively.

In Figure A.3, we visualize the overall dataset distribution through histogram-based line plots, where each line represents a single dimension and its independent data distribution. In other words, each line plotted is one dimension and the $y$-axis is the percentage of the dataset binned into the bucket at the respective $x$-axis value. We present these distribution plots for both the before (Figure A.3a) and after (Figure A.3b) datasets, using the standard min-max normalization process to transform to the unit hypercube. Note that unless otherwise noted, the $x$-axis always ranges over

[1] `https://archive.ics.uci.edu/ml/datasets/Corel+Image+Features`




(a)




(b)

Figure A.3: Color dataset distribution (a) before and (b) after normalization.

the global min and max values for completeness sake, and we use 40 equal-width bins over this range. Also, the line colors are only for improved readability.

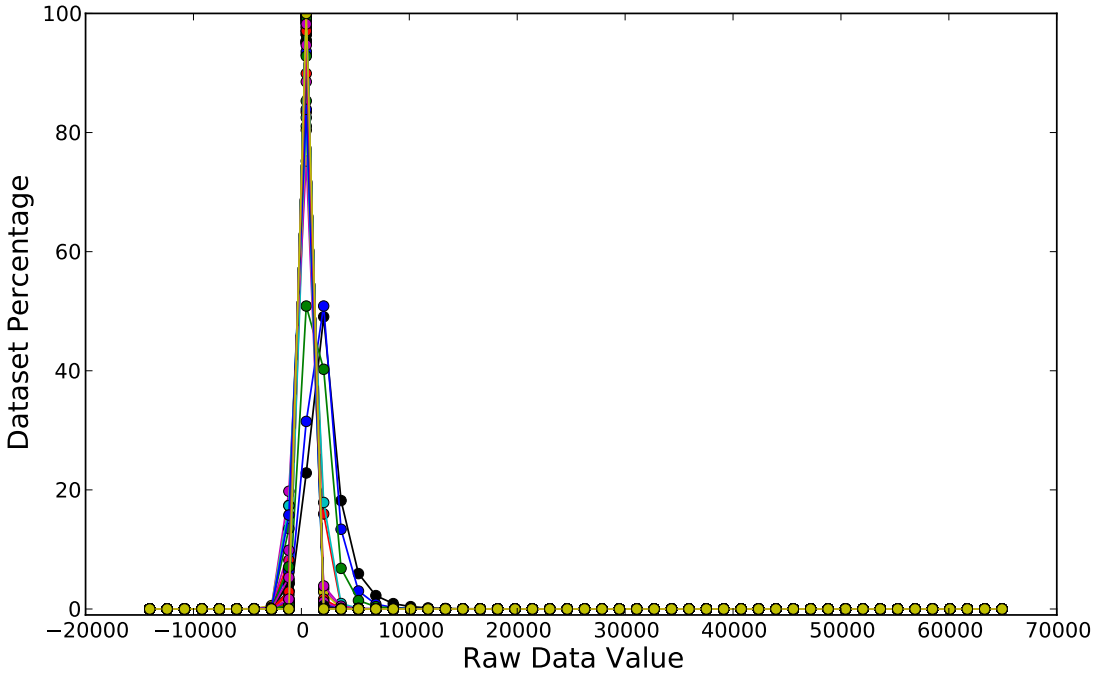
As one would expect, in this case normalization had very little effect on the data. We can see a great majority of the dataset has very low values in all dimensions. While we could hypothesize much of the data is likely grouped in this lower corner (origin) of the dataspace, we cannot gaurantee it without proper correlation between dimensions.

### A.3.2 Music

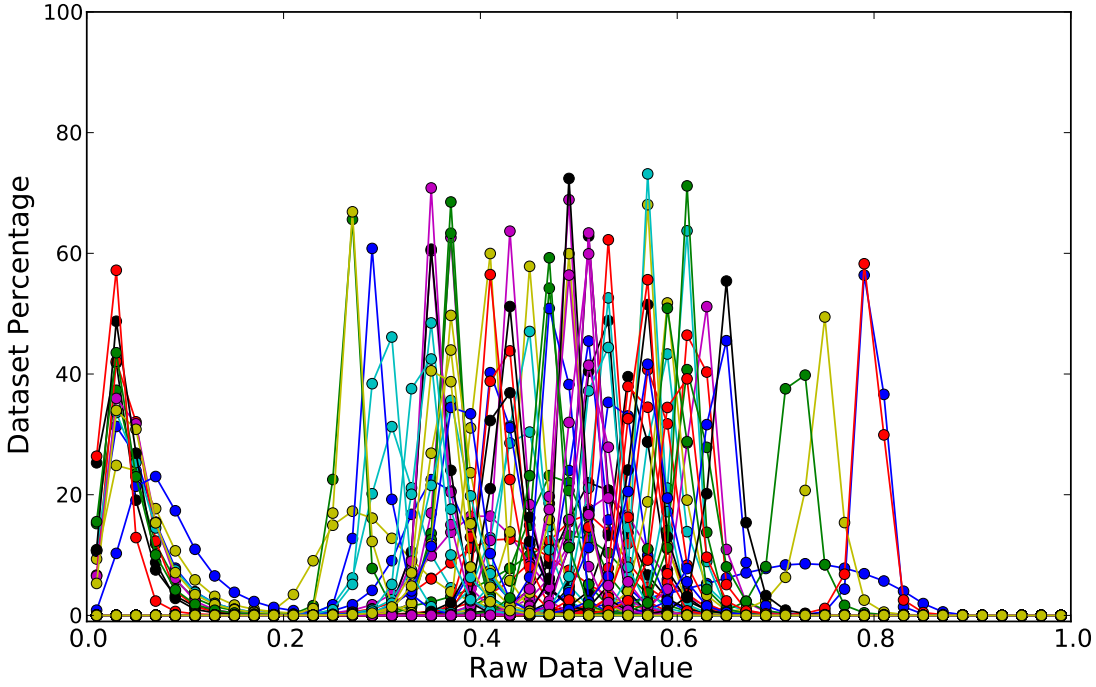
The YearPredictionMSD dataset (Music) is also publicly available online[2] through the UCI repository. It contains 515,345 data points that represent songs based on 90 real-valued audio features (dimensions), such as various timbre metadata. Each data point also contains a label that is the 4-digit year the song was released, ranging from 1922 to 2011. The dataset is a subset of the Million Song Dataset [63] from Columbia University.

In Figure A.4, we again visualize the dataset distributions before and after min-max normalization. While the raw data was already in CSV format, it was not normalized, and contains a global min and max of -14,861.7 and 65,735.8, respectively. Here we see the quite large global range of only a few dimensions dwarfing most of the rest, as well as the downside of our global equal-width histogram buckets. However, despite a poor view of the raw dataset distributions, we know the normalized distributions will always produce a consistent look at the dataset for further analysis.

[2] `http://archive.ics.uci.edu/ml/datasets/YearPredictionMSD`




(a)




(b)

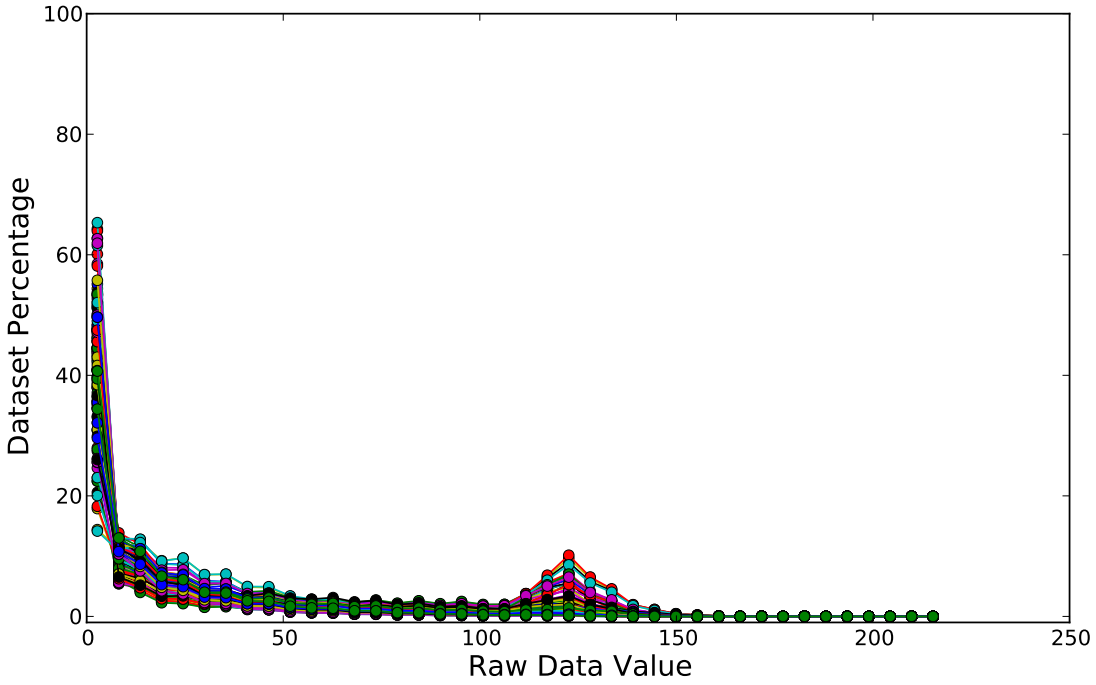
Figure A.4: Music dataset distribution (a) before and (b) after normalization.

### A.3.3 Sift

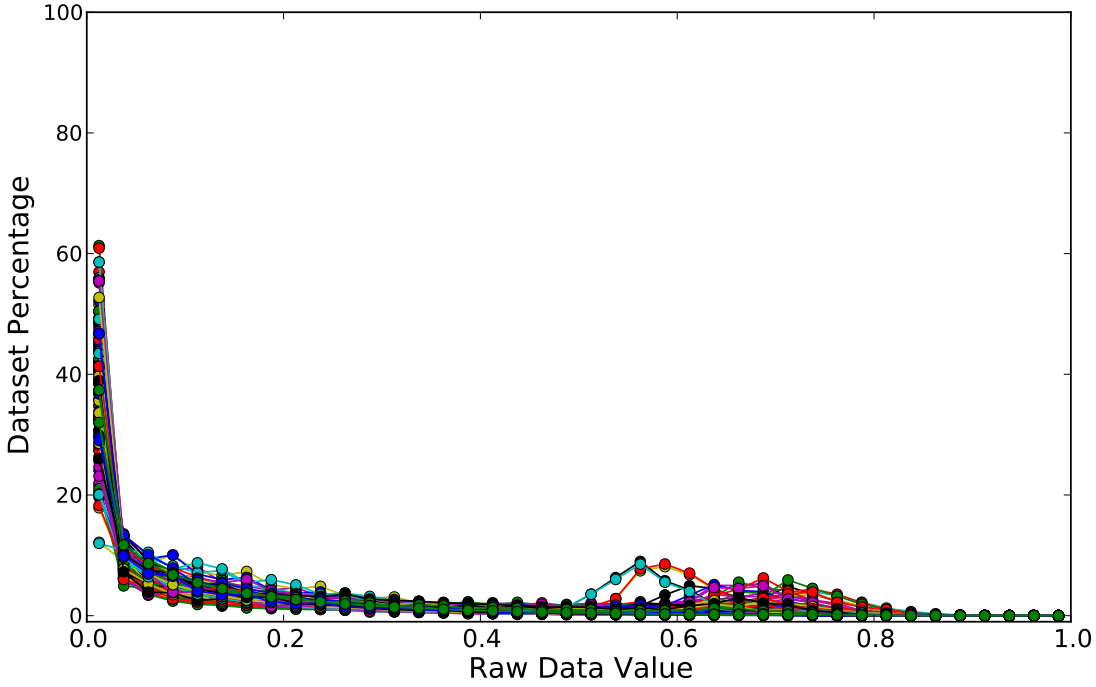
The ANN_SIFT1M (Sift) dataset [64] is publicly available online[3]. It contains 1 million data points over 128-dimensional SIFT [65] attributes.

In Figure A.5, we again visualize the dataset distributions before and after min-max normalization. Here we see similar trends to the Color dataset, however, the general curve is more shallow and there is a notable bump in frequency for multiple dimensions near the end of the global range.

[3] http://corpus-texmex.irisa.fr/




(a)




(b)

Figure A.5: Sift dataset distribution (a) before and (b) after normalization.