Large Scale Nearest Neighbor Search – Theories, Algorithms, and Applications

Junfeng He

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2014

ABSTRACT

Large Scale Nearest Neighbor Search – Theories, Algorithms, and Applications

Junfeng He

We are witnessing a data explosion era, in which huge data sets of billions or more samples represented by high-dimensional feature vectors can be easily found on the Web, enterprise data centers, surveillance sensor systems, and so on. On these large scale data sets, nearest neighbor search is fundamental for lots of applications including content based search/retrieval, recommendation, clustering, graph and social network research, as well as many other machine learning and data mining problems.

Exhaustive search is the simplest and most straightforward way for nearest neighbor search, but it can not scale up to huge data set at the sizes as mentioned above. To make large scale nearest neighbor search practical, we need the online search step to be sublinear in terms of the database size, which means offline indexing is necessary. Moreover, to achieve sublinear search time, we usually need to make some sacrifice on the search accuracy, and hence we can often only obtain approximate nearest neighbor instead of exact nearest neighbor. In other words, by large scale nearest neighbor search, we aim at approximate nearest neighbor search methods with sublinear online search time via offline indexing.

To some extent, indexing a vector dataset for (sublinear time) approximate search can be achieved by partitioning the feature space to different regions, and mapping each point to its closet regions. There are different kinds of partition structures, for example, tree based partition, hashing based partition, clustering/quantization based partition, etc. From the viewpoint of how the data partition function is generated,

the partition methods can be grouped into two main categories: 1. data independent (random) partition such as locality sensitive hashing, randomized trees/forests methods, etc.; 2. data dependent (optimized) partition, such as compact hashing, quantization based indexing methods, and some tree based methods like kd-tree, pca tree, etc.

With the offline indexing/partitioning, online approximate nearest neighbor search usually consists of three steps: locate the query region that the query point falls in, obtain candidates which are the database points in the regions near the query region, and rerank/return candidates. For large scale nearest neighbor search, the key question is: how to design the optimal offline indexing, such that the online search performance is the best, or more specifically, the online search can be as fast as possible, while meeting a required accuracy?

In this thesis, we have studied theories, algorithms, systems and applications for (approximate) nearest neighbor search on large scale data sets, for both indexing with random partition and indexing with learning based partition.

Our specific main contributions are:

1. We unify various nearest neighbor search methods into the data partition framework, and provide a general formulation of optimal data partition, which supports fastest search speed while satisfying a required search accuracy. The formulation is general, and can be used to explain most existing (sublinear) large scale approximate nearest neighbor search methods.

2. For indexing with data-independent partitions, we have developed theories on their lower and upper bounds of time and space complexity, based on the optimal data partition formulation. The bounds are applicable for a general group of methods called Nearest Neighbor Preferred Hashing and Nearest Neighbor Preferred Partition, including, locality sensitive hashing, random forest, and many other random hashing methods, etc. Moreover, we also extend the theory to study how to choose the parameters for indexing methods with random

partitions.

3. For indexing with data-dependent partitions, I have applied the same formulation to develop a joint optimization approach with two important criteria: nearest neighbor preserving and region size balancing. we have applied the joint optimization to different partition structures such as hashing and clustering, and achieved several new nearest neighbor search methods, outperforming (or at least comparable) to state-of-the-art solutions for large scale nearest neighbor search.

4. we have further studied fundamental problems for nearest neighbor search beyond search methods, for example, what is the difficulty of nearest neighbor search on a given data set (independent of search methods)? What data properties affect the difficulty and how? How will the theoretical analysis and algorithm design of large scale nearest neighbor search problem be affected by the data set difficulty?

5. Finally, we have applied our nearest neighbor search methods for practical applications. We focus on the development of large visual search engines using new indexing methods developed in this thesis. The techniques can be applied to other domains with data-intensive applications, and moreover, be extended to other applications beyond visual search engine, such as large scale machine learning, data mining, and social network analysis, etc.

List of Figures

Part I

Introduction

Chapter 1

Introduction and Overview

1.1 Motivation

The advent of Internet brings us to the "Big Data" era, in which huge data sets with billions of samples become quite common. These huge data sets include, for instances, web multimedia, enterprise data centers, mobile/surveillance sensor systems, and network nodes, etc. Taking web multimedia as an example, according to the internet statistics of 2011 [1], Google Youtube has more than 48 hours of videos uploaded every minute, while in February 2012, Facebook announced that it had more than 200 Billion photos, and more than 250 Million new photos are uploaded every day [2].

Lots of these huge data sets consist of high dimension vectors. For example, in multimedia applications, each image can be described by the features of various aspects of visual content like color, shape, objects, etc; in sensor systems, sensor data are usually vectors too. To utilize these huge data sets, one crucial step of many applications is to search nearest neighbors (NN) for a given query vector.

On one hand, lots of applications are essentially large scale nearest neighbor search

[1] http://royal.pingdom.com/2012/01/17/internet-2011-in-numbers/

[2] http://www.popphoto.com/news/2012/02/people-upload-average-250-million-photos-day-to-facebook

problem. First, nearest neighbor search will often directly serve as a content based search engine to return the query's neighbors in the database, which is useful in many different domains such as multimedia, biology, finance, sensor, surveillance, and social network, etc. For example, given a query image, find similar images in a photo database; given a user log profile, find similar users in a user database or social network; given a DNA sequence, find similar DNA sequences; given a stock trend curve, find similar stocks from stock history data; given an event from sensor data, find similar events from sensor network data log; and so on. Take multimedia domain as an example. Finding a query's nearest neighbor will directly help accomplish tasks like multimedia search, duplicate detection, and copyright management. Moreover, from the query's neighbors, we can usually obtain more associated information from meta data, tags, and so on. For instance, we can build an image search engine to answer questions like "what is the product in this image", "who is this guy", "where is this place", and so on, by summarizing and analyzing the meta data, tags, or webpages, associated with the returned images.

On the other hand, many large scale machine learning, data mining and social network problems involve nearest neighbor search as one of the most crucial steps. For instance, the core technique of some classification methods like k-NN and their variations, are basically nearest neighbor search. Moreover, lots of recommendation/collaborative filtering systems rely on finding similar users/objects, which is often a nearest neighbor search problem. Also, plenty of graph or network based learning methods often need a sparse k-NN graph to scale up to huge data sets, or need to propagate one sample's co-efficients/lables to a few other nearest samples, which are actually large scale NN search problems too. Finally when many machine learning problems (classification, regression, clustering, detection, etc.) go to large scale, approximate NN search is usually the key to speed up the algorithms, by approximating the distance, similarity, or inner product operation efficiently.

1.2 Problem Definition

In this thesis, we will focus on the nearest neighbor search problem. Before we start, we first discuss about the assumptions. In this thesis, we assume each data is a vector in a high-dimensional space in which a distance metric is already defined. This seems not to be a weak assumption, since features and distance metric may not be well defined in many applications. However, features and distance metric are domain/application specific, and there are lots of research on them in each domain. Moreover, learning appropriate distance metric from domain data has also been an active research area for a long time. So in this thesis, our focus is to develop general theories and algorithms by assuming the features and distance are already well defined.

Simply speaking, the nearest neighbor search problem can be formulated as follows: given a vector data set and a query vector, how to find the vector(s) in the data set closest to the query. More formally:
suppose there is a data set X with n points $X = \{X_i, i = 1, ..., n\}$, given a query point q, and a distance metric $D(,)$, find the q's nearest neighbor X_{nn} in X, i.e., $D(X_{nn}, q) \leq D(X_i, q), i = 1, ..., n$. Like in most statistics or machine learning research, here $X_i, i = 1, ..., n$ and q are assumed to i.i.d. sampled, from a random vector x. Note that in the discussions of following chapters, sometimes X also represents the data matrix consisting of all data points, and X_i is the i-th data point, which is X's i-th column.

"Linear scan" or "exhaustive search", is the most straight-forward way for nearest neighbor search; however, it can not scale up to huge data sets. In this thesis, we focus on large scale nearest neighbor search, i.e., "approximate" nearest neighbor search on large scale data sets.

There are several possible definitions about "approximate" nearest neighbor search.

1. Find at least one approximate nearest neighbor X_j in X for q, such that

$D(X_j, q) \leq (1+\epsilon)D^q_{nn}$ where $D^q_{nn} = D(X_{nn}, q)$.

2. With a probability of at least $1-\delta$, find q's exact nearest neighbor X_{nn} in X.

3. With a probability at least $1-\delta$, find at least one approximate nearest neighbor X_j in X for q, such that $D(X_j, q) \leq (1+\epsilon)D^q_{nn}$.

The first kind approximation is in the sense of spatial approximation, i.e., we try to find at least one point whose distance to the query is approximately (more specifically smaller than $1+\epsilon$ times of) true nearest neighbor distance. The second kind approximation is probability approximation, i.e., we want to find the true nearest neighbor, but not with 100% probability as in linear scan, instead, we only require a probability guarantee $1-\delta$. And the third kind of approximation is both spatial approximation and probability approximation, i.e., we want to get at least one spatially approximate nearest neighbor with a a probability guarantee.

The second kind of approximation is the most important and popular case in practice, and also the easiest one to analyze, so in this thesis we will mainly discuss the second approximation (and sometimes when applicable the third approximation too).

1.3 Overview on Related Works

There are several overviews on nearest neighbor search techniques [1, 2, 3, 4, 5, 6]. However, in this section, I will review most previous nearest neighbor search methods from the viewpoint of a data partition framework. Briefly speaking, large scale (sublinear) approximate NN indexing/search can be regarded as a data or space partition problem. Given a database which are points in a feature vector space, we can summarize the whole NN search procedure as follows:

1. Offline indexing

(a) Partition the space/data set:
partition the data set into many subsets, or equivalently partition the space into many regions, with some data structure (e.g., trees, hashing functions, grids, quantization functions, etc.)

(b) Construct the indexing (inverted file) structure:
construct the "inverted file" structure to record which points are contained in each region

2. Online search

(a) Compute the query indices:
compute the indices of the "query subset(s)/region(s)" that the query point belongs to

(b) Access the query region(s):
from the indexing structure, use the indices to access the query region(s).

(c) Check by linear scan:
e.g., by sorting the candidates to obtain top retrieved results according to their distances to the query, or by checking whether their distances to the query is smaller than a threshold, etc. Sometimes the linear scan is applied with low dimensional or compressed vectors/bits instead of original vectors to speed up this step.

There are many NN search methods which have designed different data structures to partition the space and construct the indexing structure. Roughly speaking, most of previous works can be categorized into three groups of data structures: tree based, hashing based and clustering/quantizaiton based NN search, which will be introduced in section 1.3.1, 1.3.2, and 1.3.3 respectively. Some discussions about advantages/disadvantages of these methods are provided in Section 1.3.4. Moreover, there are also some methods that do not exactly follow the above framework, which will be discussed in Section 1.3.4 too.

1.3.1 NN Search via Tree Based Partition

Tree based indexing methods include most earliest research on approximate nearest neighbor, to name a few, [7, 8, 9, 10]. However, there are also lots of new works on tree based indexing methods recently [11, 12, 13, 14, 15, 16].

In tree based NN Search methods, at the indexing step, the space is partitioned with hieratical tree structure. More specifically, the whole space will be partitioned into several regions, then each region will be further divided into smaller regions, until some stopping criteria is satisfied.

An example of tree based NN search methods, kd-tree [7], is illustrated in Figure 1.1. In kd-tree, during offline indexing, every internal node is partitioned by an axis-aligned hyperplane, which is the dimension of largest variance for points associated with intermediate node. Usually, an offset (threshold) is chosen for the hyperplane to make sure the partition is balanced, i.e., both sides of the hyperplane contain equal number of points.

The indexing structure is naturally an indexing tree: each intermediate node stores the splitting criteria and each leaf node stores the "inverted file", i.e.,, a list about which data points belong to it. At the search step, we need to traverse from the root to the query leaf node (i.e., the leaf node that the query point belongs to), obtain all candidates in it, and check them. Often, probing the query leaf node alone will only have a low probability to get the true nearest neighbor, and can not give us a satisfying recall, so we need to probe more leaf nodes, e.g., via techniques such as "back tracking" .

Besides kd-tree, there are many other projection based tree indexing methods, such as random (projection) trees/forest[13], PCA tree[15], etc. The main difference among these tree methods are the criteria/methods for generating the projections to partition the database/space at each internal node. For example, in random (projection) trees/forest, at each internal node, a random hyperplane is chosen, so that points in one side of the hyperplane go to left child while points in the other side go

to the right child. And in PCA tree, the projection is chosen as top PCA eigenvectors for the database points.

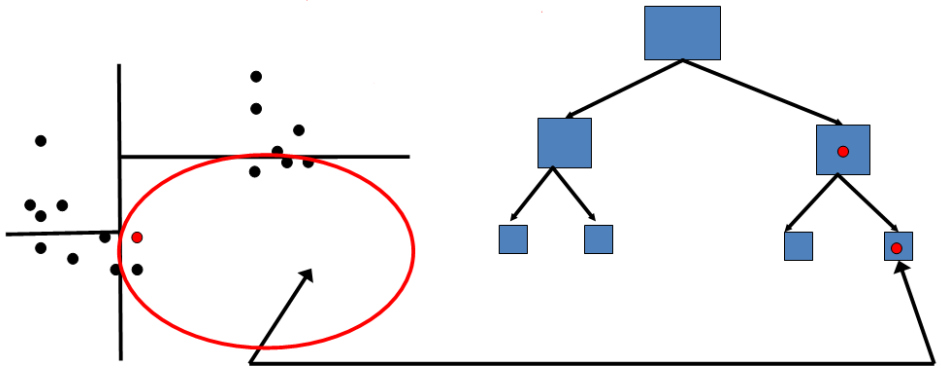


Figure 1.1: kd-tree

Besides projection based tree indexing methods, another important category is metric based tree indexing methods [17, 12, 18], including vantage point tree (vp-tree), cover trees, and MVP trees, etc. For example, in the vp-tree method, we partition data points in each internal node by choosing a point in the data space (i.e., the "vantage point") and then dividing the data points into two parts: those points that have a distance to the vantage point smaller than a threshold, and those points that are not. Equivalently, we define a hyper-sphere with the center as the vantage point and the radius as the threshold, and partition the data points according to whether they are inside or outside the hyper-sphere. An illustration of vp-tree is shown in Figure 1.2.

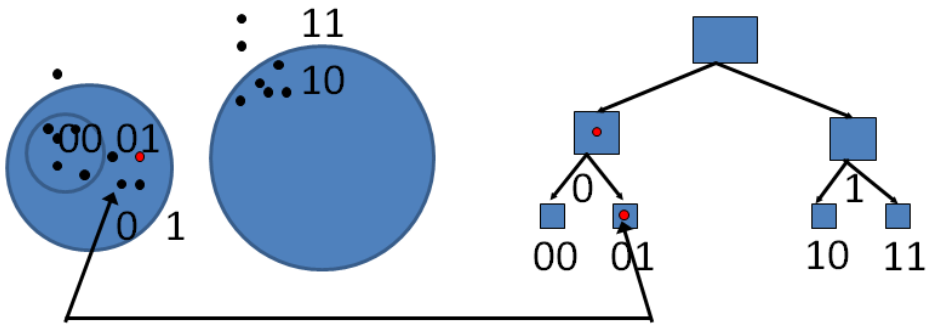


Figure 1.2: vantage point tree (vp-tree)

1.3.2 NN Search via Hashing Based Partition

Tree-based indexing approaches have shown good performance for low dimensional data; however, they are known to degrade significantly when the data dimension is high. So recently, many hash coding based algorithms have been proposed to handle similarity search of high dimensional data, including random based hashing methods such locality sensitive hashing and its variations/extensions [19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29] as well as learning based hashing methods, to name a few, [30, 31, 32, 33, 34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45].

In hashing methods, instead of using tree structures, we use hash functions (usually hyper-planes) to partition the data/space. So the space will be partitioned to many regions, and each region is represented with some hash codes, computed from the hash functions, as illustrated in Figure 1.3. Note that some tree based indexing methods also use hyper planes, but those hyper planes are local, i.e., only working for data points inside one internal node; however, each hash function in hashing based indexing is global, works globally on the whole space/database. This difference causes several important advantages for hashing based methods. First, it only needs $O(\log(n))$ hyper planes to partition the space to n regions while tree based methods

need $O(n)$ hyper planes; Secondly, in hashing, for a given (query) region, it is easy to find its nearby regions, just by permutating the hash bits which only take $O(1)$ time to find each nearby region; while in trees, it is quite difficult to find a nearby region, usually with "backtracking" methods which will take long time especially when the dimension is high. This is the main reason of the "curse of dimensionality" for tree based methods, which makes tree based NN search method sometimes slower than linear scan when the dimension is high.

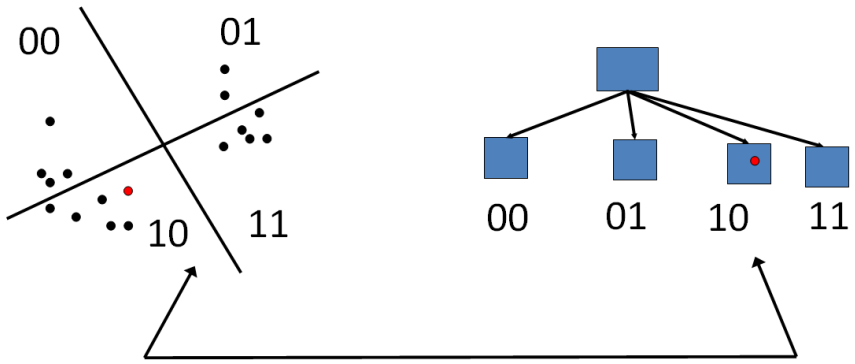


Figure 1.3: Space partition by hash functions

The indexing structure is the hash table, in which each entry represents one region, described by its hash codes [3]. Each entry contains all the IDs for the data points that fall into the region (i.e., data points that have the same hash codes as the region). During the online search, the hash codes for the query point are first computed, and then the hash codes are utilized to access the query entry in the hash table. The

[3]Sometimes when the hash code is too long and there are too many entries in the hash table, a second conventional hashing (e.g., based on prime numbers) will be applied to reduce the number of entries in the table.

process is illustrated in Figure 1.4. Data points in the query entry are obtained as candidates, and usually reranking will be applied on candidates to further ensure retrieving top near neighbors. In practice, to guarantee a high recall, often multiple hash tables (note that each hash table represents one partition of the space) will be generated, and the union of candidates from each hash table will be collected as candidates.

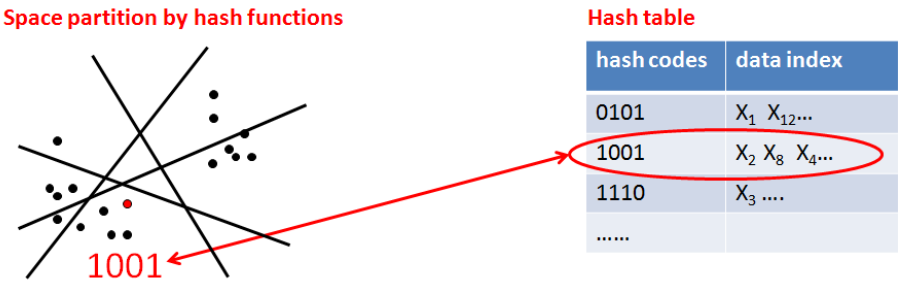


Figure 1.4: Hash table look up

Briefly speaking there are two main groups of hashing methods: random based hashing, and learning based hashing.

Among the popular randomized hash based techniques is the locality-sensitive hashing (LSH) [20]. In LSH, random vectors (with some specific distribution) are used as the projection bases to partition the space. As an important property of LSH, points with high feature similarity are proven to have a high probability of being assigned the same hash code and hence fall into the same region, which guarantees an asymptotic theoretical property of sublinear search time. Variations of LSH algorithms have been proposed in recent literatures to expand its usage to the cases of inner products [21], L_p norms [22], Jaccard Similarity[21], and learned metrics [46], kernel similarity[29], Chi2 distance[29], etc.

Despite of its success, one arguable disadvantage of LSH is the inefficiency of the

hash codes. Since the hash functions in LSH are randomly generated and independent of the data, it is often not very efficient. And hence many hash tables are often needed to get a good recall to keep a high precision. This would heavily increase the requirement of storage, causing problems for very large scale applications. So, many recent research works focus on how to generate high-quality short compact hash codes [30, 31, 32, 33, 34, 35, 36, 38, 39, 40, 41, 42, 43, 44]. These codes are learned from the data set, so each learned bit is more powerful in differentiating the neighbors of random points, compared to those randomly generated bits. And hence one or a few hash tables with short codes, can hopefully still achieve good precision.

The main difference of these methods is about how to learn the hash functions from the data. One group of methods are to generate the hash functions by unsupervised learning, e.g., PCA projections or its variations [47, 32], etc. Another group of methods are via supervised learning (or semi-supervised learning) to make sure nearest neighbors in training set can be preserved by the hash functions, which are learned by neural networks or deep belief networks[30, 31, 33], linear subspace learning techniques and their variations[39, 48, 40], or kernel functions[35, 36, 43].

1.3.3 NN Search via Clustering Based Partition

Besides tree based and hashing based indexing methods, another category of indexing methods[49, 50, 51, 52] are based on clustering or vector quantization. Often in these methods, the space is partitioned by clusters, for instance, obtained from k-means clustering, as shown in Figure 1.5. (Or more specifically, the space is partitioned with the Voronoi cells introduced by clusters.) During the offline indexing, the indexing structure is usually conventional "inverted file", recording the indices of points falling into each cluster. During the online search, we first find the cluster that the query point belongs to, and get all points in the cluster as candidates. Actually, most works on clustering based NN search are related to image/video search with local features, where clusters serve as codebooks, and NN search is used to find matched

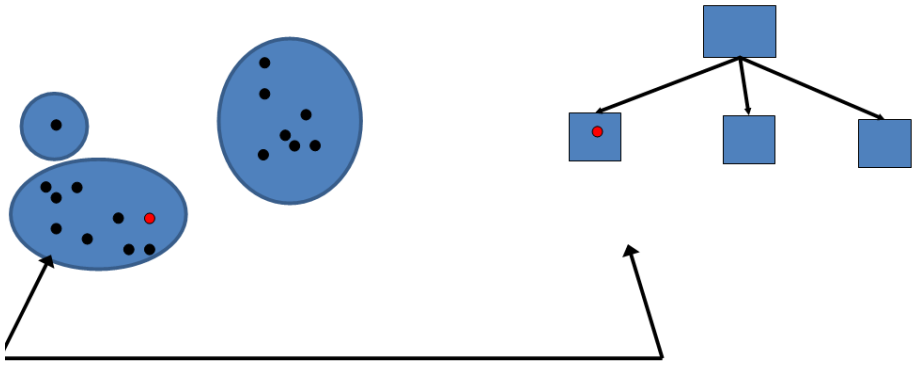


Figure 1.5: Space partition by clusters

local features in images/vides in the database for each query local feature.

The main difference among these clustering based NN search methods is how to cluster/quantize the vectors (data points). The most popular way to cluster data points is via k-means clustering [49], while other methods include clustering via regular lattice [52], supervised or semi-supervised clustering[51]. Moreover, in product quantization method [50], clustering/quantization are done several times, while each quantization is applied to a subset of data dimensions. Moreover, the distance from the query point to the cluster centers is computed or approximated to rerank candidates.

Moreover, we can also build trees by using hieratical clustering methods, where each internal node is quantized to clusters generated from data only in the internal node, as illustrated in 1.6.

1.3.4 Discussions

We have mainly introduced three kinds of approximate NN search methods:tree based methods, hashing based methods, and quantization/clustering based methods. No

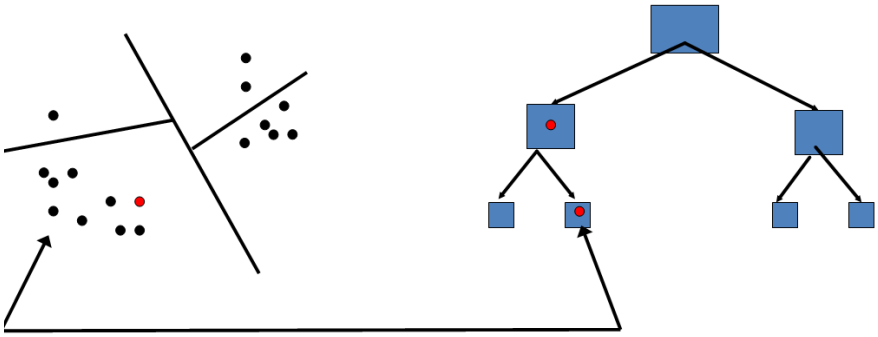


Figure 1.6:

method is superior to others in all scenarios. Each method has its own advantages and disadvantages. For example, one big advantages of tree based methods is the flexibility. Tree based methods can be easily extended to support data with mixed kinds of features (e.g., binary, integer, real numbers), and different distance metric. They are also easy to understand and implement, making them still popular in practice, despite the inferior performance sometimes. However, its main disadvantages are the inefficiency for high dimensional data, and also the requirement to store $O(K)$ partition functions in memory to create K regions. For hashing methods, they overcomes the "curse of dimensionality" in some sense and can usually deal with high dimensional data quite well. Moreover, it only needs $O(\log K)$ partition functions to create K regions, requiring less memory, and making multiple partitions convenient and practical. However, they usually can not support mixed kinds of features or multiple distances. Clustering/quantization methods can usually create high quality partitions, compared to the other two kinds of methods, when the number of region K is fixed. However, it needs to spend expensive training time and store a large number of cluster centers, which prohibits the usage of large K, and hence can not scale up to large data set.

In terms of how the partition functions are generated, the partitions can also be categorized into two main groups: data independent (random) partitions and data dependent (learning based) partitions. The former includes random tree/forest methods, locality sensitive hashing and its variations, and so on, where the partition structures (functions) are randomly generated, independent of the database points. The latter consists of many kinds of tree (e.g., kd-tree, vp-tree), hashing (e.g., PCA hashing, spectral hashing) and quantization based NN search methods, where the partitions are learned or optimized from database points. The main advantage of random partitions is that no training is required, and hence it is very easy to scale up to large data set. However, the partitions may not be optimal/high-quality since they are randomly generated. On the other hand, data dependent methods can usually generate better partitions (when fixing the number of regions K). But they usually need massive computation to learn the partitions. So when computation capacity is the main bottleneck, data independent methods are more appropriate, while when memory is the main bottleneck, data dependent approaches is a better choice.

Finally, it is worth noting that there are some nearest neighbor search methods which may not exactly follow the data partition framework as discussed above. For example, several hashing methods are following the "hamming ranking" paradigm. Basically, after the hash bits are computed, these methods will not build hash tables, and instead they will linear scan all database points by hamming distance between the query point's hash bits and each database point's hash bits. All these methods belong to (or are similar as) "dimension reduction" to some extent. Basically, instead of using original features for linear scan, they apply linear scan on the reduced low dimensional (binary) bits. This method can not achieve sub-linear search time, and hence can not scale up to very large scale data set, like billions or trillions of points by these methods alone. Actually, these methods can be good options as the step 3 ("check by linear scan") in our data partition framework, to scale up to very large scale data sets.

In this thesis, we will mainly discuss the sublinear NN search methods, though which can/may use the linear NN search methods as one step.

1.4 Unified Formulation of Optimal Data Partition for Approximate NN Search

Denote $y = \Psi(x) : \mathbb{R}^d \to \mathbb{N}$ as the partition function, which will map a real vector x in $\mathbb{R}^d$ space to a positive integer y. As discussed in section 1.3, Ψ can be hashing based partition, tree based partition, clustering based partition, etc, and y is actually the index of region that x falls into. For example, for hashing based methods, the index is the hash codes of the bucket; for tree based methods, the index is the codes of the leaf nodes; For clustering based methods, the index is the ID of each cluster.

Sometimes when multiple partitions instead of one partition are needed (for example, multiple hash tables in hashing methods), $\Psi()$ will be $y = \Psi(x) : \mathbb{R}^d \to \mathbb{N}^L$, where L is the number of needed partitions. In other words $\Psi()$ will map a d dimension real vector x to a L dimension integer vector, where y_i, the i-th dimension of y represents the region index of x in the i-th partition.

Under the data partition framework, given a query vector q, the time cost of online search consists of three parts:

1. $T_{indices}(\Psi)$:

   the time cost to compute the index of the query region(s), i.e., $\Psi(q)$.

2. $T_{regions}(\Psi)$:

   the time cost to access the candidate regions. If we only access the region $\Psi(q)$ itself, $T_{regions}(\Psi)$ is $O(1)$ and can often be omitted. However, sometimes we will access not only the region of $\Psi(q)$, but also nearby regions close to $\Psi(q)$. For example, in hashing, we often access all regions whose indices have a small hamming distance to $\Psi(q)$. And other examples include backtracking

techniques in tree based methods. In these cases, we will access multiple regions for one query index, and hence $T_{regions}(\Psi)$ might be much larger.

3. $T_{check}(\Psi)$:

   the time cost to check/rank all retrieved candidates. For a query q, denote $\widehat{P}_{any}(\Psi)$ as the probability for a random database point to be retrieved under the partition Ψ. Then $n\widehat{P}_{any}(\Psi)$ is the number of candidates retrieved. U_{check} is the time cost to check one candidate, which often equals to computing the distance between two d dimensional points. So $T_{check}(\Psi) = \widehat{P}_{any}(\Psi)U_{check}$.

Moreover, suppose $\widehat{P}_{nn}$ is the probability for q's nearest neighbor point to be retrieved under the partition Ψ. $\widehat{P}_{nn}$ is actually the recall of the retrieved points. Suppose δ is the maximum acceptable error probability. In other words, the probability to miss the true nearest neighbor $1 - \widehat{P}_{nn}$ is supposed to be smaller than δ. Here δ is a small positive number satisfying $\delta \leq 1$.

So the optimal partition to minimize search time while guaranteeing search accuracy can be formulated as:

$$\begin{aligned} \min_{\Psi} T(\Psi) = T_{indices}(\Psi) + T_{regions}(\Psi) + n\widehat{P}_{any}(\Psi)U_{check} \\ s.t., \\ 1 - \widehat{P}_{nn}(\Psi) \leq \delta \end{aligned} \tag{1.1}$$

We can also put the constraint into the cost function and obtain another formulation as a joint optimization of search accuracy and time:

$$\min_{\Psi} (1 - \widehat{P}_{nn}(\Psi)) + \lambda T(\Psi) = (1 - \widehat{P}_{nn}(\Psi)) + \lambda[T_{indices}(\Psi) + T_{regions}(\Psi) + n\widehat{P}_{any}(\Psi)U_{check}] \tag{1.2}$$

The above formulations use recall to describe search accuracy. One may argue precision should be better. But note that we have a checking step, e.g.,reranking,

in our process, so high recall will lead to high precision in top returned results after checking.

Often $T_{indices}(\Psi)$ and $T_{regions}(\Psi)$ are much smaller than $T_{check} = n\widehat{P}_{any}(\Psi)U_{check}$, and hence the optimal partition formulation can be simplified as

$$\min_{\Psi}(1 - \widehat{P}_{nn}(\Psi)) + \lambda n\widehat{P}_{any}(\Psi)U_{check} \tag{1.3}$$

So for large scale nearest neighbor search, the key question is: how to design the partition Ψ, such that the above optimization can be achieved.

First, there are some interesting trivial cases, which are obviously not the optimal solutions. For example, one kind of partition Ψ is to put all points into one region (leaving all other regions empty), and hence $P_{nn} = 1$ and $P_{any} = 1$. In this case, this partition will lead us to linear scan. Another strategy of creating Ψ is to putting points into regions arbitrarily without considering their distances. Then for a query point, we can arbitrarily return one point from one region as its nearest neighbor. This will give us the minimal search time, but the worst search accuracy, which is basically "random guess".

Moreover, the optimal partition depends on the scenarios of our nearest neighbor search, including the indexing structure, the way to generate/choose partition functions, etc.

So the unsolved questions now is: under different scenarios, how to formulate Ψ, as well as U_{check}, $T_{regions}$, $T_{indices}$, $\widehat{P}_{nn}(\Psi)$ and $\widehat{P}_{any}(\Psi)$, and moreover how to solve the above optimization problems. We will provide discussion in details in Part II and Part III of this thesis.

1.5 Thesis Outline

For random based partitions, the partitions are independent of the data, and are usually determined by only a few parameters. So the exact formulation and analysis

of Ψ, U_{check}, $T_{regions}$, $T_{indices}$ as well as $\widehat{P}_{nn}(\Psi)$ and $\widehat{P}_{any}(\Psi)$ are not difficult, making both theoretical results and practical methods possible. In Part II of this thesis, following the framework of optimal data partition in Section 1.4, we will provide new bounds on the time/space complexity for LSH and also various kinds of NN search methods based on random partition. Moreover, based on the derivation of the tight bound, we also explore how to choose the parameters of random partition for each particular data set.

For learning based partitions, the partition Ψ depend on the training data, and hence exact formulation and analysis of Ψ, and especially $\widehat{P}_{nn}(\Psi)$ and $\widehat{P}_{any}(\Psi)$, are usually very difficult. So besides theoretical analysis, we will also need approximation, heuristics, and intuition. We will focus on designing algorithms rather than deriving theoretical results like bounds for learning based partitions. In Part III, following the framework of optimal data partition, we will show how to design various NN search methods, which perform better than or at least as good as other state-of-the-art NN search methods.

In Part IV, we will demonstrate examples of applications based on large scale NN search. We will mainly focus on the applications of visual search engine, especially mobile visual search, based on the large scale NN search techniques we discussed in Part II and Part III. Our mobile visual search based on our hashing methods outperforms other visual search methods, and is the first system that can index million scale image object sets and allow search response over low-bandwith networks within 2 seconds.

In Part V, we will investigate more fundamental problems beyond algorithms. For example, what is the difficulty of nearest neighbor search on a given data set, independent of any method? What data properties (e.g., dimension, sparsity, etc.) affect the difficulty and how? How will the difficulty or data property affect the complexity and algorithm design for approximate nearest neighbor search? ... Investigation on these fundamental problems will provide us deep understanding of NN search problems,

and inspiration of better design of NN search methods.

The proofs of all theorems/collaries in this thesis are provided in Chapter 9.

Part II

Nearest Neighbor Search via Random Partitions

In this part of the thesis, we will mainly discuss nearest neighbor search based on random partitions, including methods such as locality sensitive hashing, random (projection) trees/forests, etc. Random partition based methods do not need training/optimization to generate the partition functions, and hence are very practical for very large databases. Moreover, random partition based methods are easy to analyze, and hence we can obtain a very deep understanding of them with solid theoretical results.

In Chapter 2, based on the formulation of optimal data partition in (1.1), we will find the formulation $\widehat{P}_{nn}(\Psi)$, $\widehat{P}_{any}(\Psi)$, $T_{indices}$ and $T_{regions}$ for nearest neighbor search via random partitions. And hence by analyzing the bound of the optimal value of (1.1), we can provide a lower bound of the time complexity for many variations of random partitions based methods. We will first develop the lower bound for locality sensitive hashing, and then extend it to a group of hashing methods called **Nearest Neighbor Preferred Hashing**, a more general group of methods called **Nearest Neighbor Preferred Partition**, including LSH, random hashing, random forests and so on. Our techniques can also be applied to obtain a tighter upper bound for LSH. Moreover, based on the theories, we also provide an approach to choose parameters for LSH and other random partition methods on each particular data set.

Chapter 2

Theories On the Complexity of NN Search via Random Partitions

2.1 Introduction to Previous Works On the Complexity of LSH

Among random partition based nearest neighbor search methods, locality sensitive hashing methods [19, 20, 21, 22], are one of the most popular and successful ones. In this chapter, we will first formulate the time and space complexity, and then develop the tight bound of time complexity for locality sensitive hashing methods (using the LSH proposed in [21] as an example), in Section 2.2 and 2.3. We will generalize the bound to other randomized nearest neighbor search methods in section 2.6 .

Locality sensitive hashing (LSH) was first proposed in [19, 20], with an unpractical method that only works to approximate the hamming distance in the embedded unary bits [1]. It was then extended to approximate the angle distance [21] and L_p distance [22] for high dimensional vectors, making it successful in not only theory but also engineering applications. Later on, variations of LSH are further developed

[1] which is equivalent to L_1 distance of the original feature vector

to approximate advanced distance metrics, like learned metrics [27], kernel similarity [37], complex metrics such as pyramid matching distance [26], etc.

Intuitively speaking, LSH is based on a simple idea: after a linear projection and then assignment of points to a bucket via quantization, points that are nearby are more likely to fall in the same bucket than points that are further away.

Using LSH consists of offline indexing the data and online searching for neighbors of a query point, as discussed in Section 1.3. More specifically,

Step 1: Indexing

- Compute one hash code: Compute one hash code by a random hash function $h(x)$, where $h(x)$ maps a vector x to an integer.
- Multi-line projection: obtain an array of k integers by doing k one-line hash functions. All points that project to the same k values are called members of the same (k-dimensional) bin/bucket. At this stage often a conventional hash is used to reduce the k-dimensional bin identification vector to a location in memory. With a suitable design, this hash produces few collisions and does not affect our analysis.
- Repeat by hashing the dataset to k-dimensional buckets/bins into a total of L times. Thus, every point in the dataset belongs to L tables.

Step 2: Search

1. Compute the L (k-dimensional) buckets/bins for the query point using the hash functions as in the indexing stage.
2. Retrieve all points that belong to these bins (we call them *candidates*), measure their distance to the query point, and return the one that is closest to query point.

There are many variations about locality sensitive hashing. The main difference of each variation is the hash function $h(x)$. For example, in R^d space, one kind of hash

function with $h(x) = sign(v^T x)$, where v follows standard Gaussian distribution, is called **binary LSH**. Another hash function $h(x) = \lfloor \frac{v^T x + b}{w} \rfloor$ is called **p-stable LSH**, where v is a vector and each dimension is i.i.d sampled from p-stable distribution, and b follows the uniform distribution of $[0, w]$.

Intuitively, the definition of locality sensitive hashing is: two points with smaller distance should have higher probability to get the same hash code. More specifically, a (r, cr, p_1, p_2) (where $c > 1$ and $p_1 > p_2$) locality sensitive hashing function family means [19, 20]: for two points with distance smaller than r, they have at least p_1 probability to get the same hash code; while for two points with distance larger than cr, they have at most p_2 probability to get the same hash code. In other words, a (r, cr, p_1, p_2) **sensitive hashing function** [19, 20] is defines as:
For $c > 1$ and $p_1 > p_2$, a family of hash functions H is called (r, cr, p_1, p_2) sensitive for $D(,)$, if for any x and q,
when $D(x, q) \leq r$, $P(h(x) = h(q)) \geq p_1$
when $D(x, q) \geq cr$, $P(h(x) = h(q)) \leq p_2$

Binary LSH, p-stable LSH, and other LSH methods are all (r, cr, p_1, p_2) sensitive, as proved in [21, 22], etc.

One main reason for LSH to become popular and successful is its solid theory foundations. For the first time it provides a theoretical upper bound of sublinear search time that works for high dimensional data [21, 22].

More specifically, for a (r, cr, p_1, p_2) locality sensitive hashing function family, we have the following theory about the upper bound of its time and space complexity [19, 20, 21]:

Theorem 2.1.1. *Consider LSH from a (r, cr, p_1, p_2) hash function family where $c > 1$ and $p_1 > p_2$. For a query q, LSH can solve the c-approximate nearest neighbor problem* [2] *with time complexity $O(d \log \frac{1}{\delta} n^\rho \log_{p_2^{-1}} n)$ and space complexity $O(nd + \log \frac{1}{\delta} n^{1+\rho})$,*

[2] c-approximate nearest neighbor: if the true nearest neighbor has a distance r, LSH will at least

where $\rho = \frac{\log p_1}{\log p_2}$. *The number of needed hash tables is* $O(\log \frac{1}{\delta} n^{\rho})$.

Note that $p_1 > p_2$ is guaranteed in the definition of LSH, so $\rho = \frac{\log p_1}{\log p_2}$ in the above theorem always satisfies $\rho < 1$. And hence the search time $O(d \log \frac{1}{\delta} n^{\rho} \log_{p_2^{-1}} n)$ is sublinear in terms of n. The value of ρ mainly determines how "sublinear" LSH search time will be. In [22], it is shown that $\rho \leq \frac{1}{c}$, and moreover, [53] proves $\rho \geq \frac{0.462}{c^p}$ for LSH to approximate L_p distance.

In this chapter, following the formulation of optimal data partition in (1.1), we formulated the time complexity of LSH as an optimization problem in terms of parameters k and L, where k is the number of hash functions in each hash table and L is the number of hash tables. By analyzing the the optimization problem, we present a lower bound of time and space complexity for LSH. This lower bound can also be applicable to a more general random hashing methods called **Nearest neighbor Preferred Hashing (NPH)**, which include LSH as a special case. Moreover, the tight bound can be further extended to a more general random partition based indexing methods, called **Nearest neighbor Preferred Partitions (NPP)**, which includes LSH, many random hashing, as well as random trees/forests, etc. Moreover, the techniques are also applied to develop a new tighter upper bound for LSH, and also a new approach to choose parameters for LSH and other random partition methods on one particular data set.

The proofs of our theorems/collaries in this section can be found in Section 9.1 in the Appendix.

---

an approximate nearest neighbor point within distance cr

2.2 Formulation of the Time and Space Complexity for LSH

In this section, we will mainly discuss LSH with random hyperplanes to partition the space, such as the one in [21], where the space partitions Ψ consists of L individual partitions (i.e., L hash tables), each of which involves k random projections.

Following the formulation of optimal data partition in (1.1), we will first find the formulation of time and space complexity for locality sensitive hashing with random hyperplanes.

Recall (1.1), which is

$$\begin{aligned} \min_{\Psi} T(\Psi) = T_{indices}(\Psi) + T_{regions}(\Psi) + n\widehat{P}_{any}(\Psi)U_{check} \\ s.t., \\ \widehat{P}_{nn}(\Psi) \geq 1 - \delta \end{aligned} \tag{2.1}$$

To the study the complexity of LSH, we need to study the complexity of $\min_{\Psi} T(\Psi)$ in the scenario of LSH.

For LSH, the partition Ψ is determined by two parameters, k and L. So we need to obtain the exact formulation of $\widehat{P}_{nn}(\Psi)$, $\widehat{P}_{any}(\Psi)$, $T_{indices}$, $T_{regions}$ and finally $T(\Psi)$, in terms of k and L.

Given the fixed query point q, denote p_{q,X_i} as the probability of database point X_i and q to have the same hash code for one hash function. Denote $p_{q,any}$ as the probability of a random database point and query q to have the same hash code for one hash function, in other words, $p_{q,any} = \frac{1}{n}\sum_{i=1,\ldots n} p_{q,X_i}$. Moreover, denote $p_{q,nn}$ as the probability of q and its nearest database point to have the same hash code for one hash function. For simplicity, we will use p_{X_i}, p_{any} and p_{nn} instead of p_{q,X_i}, $p_{q,any}$ and $p_{q,nn}$, when there is no ambiguity. It is easy to see that $p_{any} \leq p_{nn}$ is always true for LSH.

As defined in Section 1.4, $\widehat{P}_{nn}$ is the probability to return a true nearest neighbor point for the query with our partition. Note that the probability to find the nearest neighbor in one hash table $(p_{nn})^k$, the probability to miss the true nearest neighbor in one hash table is $1-(p_{nn})^k$, and so the probability to miss the true nearest neighbor in all L hash tables is $(1-(p_{nn})^k)^L$. So we have

$$\widehat{P}_{nn} = 1 - (1 - (p_{nn})^k)^L \tag{2.2}$$

Moreover, as defined in Section 1.4, $\widehat{P}_{any}$ is the probability to return a random database point for the query with our partition. Note that the probability for a database point X_i to be returned in one hash table is $(p_{X_i})^k$. So the probability for a random database point to be returned in one hash table with k hash functions is $\widehat{P}_{any,1} = \frac{1}{n}\sum_{i=1,...n}(p_{X_i})^k$. Note that according to Jensen's inequality, we have

$$\widehat{P}_{any,1} = \frac{1}{n}\sum_{i=1,...n}(p_{X_i})^k \geq (\frac{1}{n}\sum_{i=1,...n} p_{X_i})^k = (p_{any})^k \tag{2.3}$$

so the probability for a random database point to be returned in L hash tables is $\widehat{P}_{any} = 1 - (1 - \widehat{P}_{any,1})^L >= 1 - (1 - (p_{any})^k)^L$. Note that usually $(p_{any})^k$ is very small, so

$$\widehat{P}_{any} >= 1 - (1 - (p_{any})^k)^L \approx 1 - (1 - L(p_{any})^k) = L(p_{any})^k.$$

Denote U_{bin} as the cost to locate one hash bin (bucket) for each hash table in the memory. It is easy to see $T_{regions} = LU_{bin}$. And $U_{bin} = \Theta(1)$ for conventional locality sensitive hashing with single probe.

Moreover, $T_{indices}(\Psi) = LU_{indices}$, where $U_{indices}$ is the time cost to compute the hash codes in one hash table. Denote U_{hash} is the time cost to compute one hash bit, it is easy to see that $U_{indices} = kU_{hash}$, since in hash table we need to compute k hash bits. $T_{indices}(\Psi) = LkU_{hash}$.

For conventional binary LSH with hperplane like $h(x) = sign(v \cdot x)$, we have $U_{hash} = \Theta(d)$, $U_{check} = \Theta(d)$. For other kinds of binary LSH, U_{hash} and U_{check} may

be different. [3]

Note that usually $U_{bin} \ll U_{hash}$, so the term $U_{bin}L$ can be ignored compared to $U_{hash}kL$, when discussing time complexity.

For LSH, denote $T(\Psi)$ in terms of k and L as $T_{LSH}(k, L)$. Putting everything together, for LSH, we have

$$T_{LSH}(k, L) \geq T(k, L) = U_{hash}kL + U_{check}Ln(p_{any})^k \tag{2.4}$$

Denote T_{min} as follows,

$$T_{min} = \min_{k,L} T(k, L) = \min_{k,L} U_{hash}kL + U_{check}Ln(p_{any})^k \tag{2.5}$$

$$s.t., (1 - (p_{nn})^k)^L \leq \delta \tag{2.6}$$

i.e., T_{min} is optimal $T(k, L)$ with the probability guarantee.

From (2.4), it is easy to see that in the scenario of LSH,

$$\min T_{LSH}(k, L) \geq T_{min}$$

To study the complexity of LSH, i.e., the complexity of $\min T_{LSH}(k, L)$, We will first explore the complexity of T_{min} in Section 2.3, and then show the complexity for LSH in Section 2.4.

We need to store n data points with d dimension each, and L hash tables with n elements each. So the space complexity will be

$$S_{LSH}(k, L) = nL + nd \tag{2.7}$$

Basically, we just need to study the complexity of L for space complexity.

---

[3] For example, for LSH with learned metric [27], $U_{hash} = O(d^2)$. For kernelized LSH [37], $U_{hash} = O(p^2 + p * U_K)$, where p is the number of landmark points to compute the kernelized hash function, and U_K is the time cost to compute one kernel function. Moreover, if we use more complex distances rather than L_p for checking (e.g., reranking) the data, U_{check} may not $\Theta(d)$ any more. Note that, usually we have $U_{bin} \ll U_{hash}$ and $U_{bin} \ll U_{check}$.

2.3 The Complexity of T_{min}

First, it is easy to see the inequality constraint $(1-(p_{nn})^k)^L \leq \delta$ in (2.5) can be changed to equality constraint $(1-(p_{nn})^k)^L = \delta$. [4] In other words,

$$T_{min} = \min_{k,L} T(k,L) = \min_{k,L} U_{hash} kL + U_{check} Ln(p_{any})^k \tag{2.8}$$

$$s.t., (1-(p_{nn})^k)^L = \delta \tag{2.9}$$

Theorem 2.3.1.

$$T_{min} = \Theta(\log\frac{1}{\delta} n^\rho U_{hash} [\frac{U_{check}}{U_{hash}}]^\rho \alpha_0 \alpha^{-\rho})$$

$$S_{min} = \Theta(dn + (\log\frac{1}{\delta}) n^{1+\rho} [\frac{U_{check}}{U_{hash}}]^\rho \alpha^{-\rho})$$

The number of hash tables is $L = \Theta((\log\frac{1}{\delta}) n^\rho [\frac{U_{check}}{U_{hash}}]^\rho \alpha^{-\rho})$. *The number of returned points is* $\Theta(\log\frac{1}{\delta} n^\rho [\frac{U_{check}}{U_{hash}}]^{\rho-1} \alpha^{1-\rho})$.

Here $\alpha_0 = \frac{\log(\tau \frac{U_{check}}{U_{hash}} n)+1}{\tau}$
$\alpha = \frac{\rho \log(\tau \frac{U_{check}}{U_{hash}} n)+1}{\tau}$, $\rho = \frac{\log p_{nn}}{\log p_{any}}$ *and* $\tau = \log(p_{nn}/p_{any})$.

The proofs of the tight bound can be found in section 9.1.2.

When $U_{hash} = U_{check} = \Theta(d)$, $U_{bin} = \Theta(1)$, the following corollary gives us a simplified result.

Corollary 2.3.2. *For conventional binary LSH with* $U_{hash} = U_{check} = \Theta(d)$, $U_{bin} = \Theta(1)$,

$$T_{min} = \Theta(\log\frac{1}{\delta} dn^\rho \beta_0 \beta^{-\rho})$$

$$S_{min} = \Theta(dn + (\log\frac{1}{\delta}) n^{1+\rho} \beta^{-\rho})$$

The number of hash tables is $\Theta((\log\frac{1}{\delta}) n^\rho \beta^{-\rho})$. *Here* $\beta_0 = \frac{\log(\tau n)+1}{\tau}$, $\beta = \frac{\rho \log(\tau n)+1}{\tau}$, $\rho = \frac{\log p_{nn}}{\log p_{any}}$ *and* $\tau = \log(p_{nn}/p_{any})$.

[4] Actually, if the solution k_{min} and L_{min} satisfy $(1-(p_{nn})^{k_{min}})^{L_{min}} < \delta$, we can find L_1 so that $(1-(p_{nn})^{k_{min}})^{L_1} = \delta$ and $L_1 < L_{min}$. Note that $T(k,L)$ is linear with L, so $T(k_{min}, L_1) < T(k_{min}, L_{min})$, which conflicts with the fact that $T(k_{min}, L_{min})$ is the minimal value.

2.4 The Complexity of LSH

2.4.1 Lower Bound of LSH

From Equation (2.4), we know that in the scenario of LSH, the time complexity $\min T_{LSH}(k, L)$ have $\min T_{LSH}(k, L) >= \min T(k, L) = T_{min}$.

Moreover, from Theorem 2.3.1, we know that $T_{min} = \Theta(\log \frac{1}{\delta} n^{\rho} U_{hash} [\frac{U_{check}}{U_{hash}}]^{\rho} \alpha_0 \alpha^{-\rho})$.

So immediately, we can get a lower bound for LSH:

Theorem 2.4.1. *To achieve the exact nearest neighbor with a probability $1 - \delta$, LSH will have a time complexity*

$$\Omega(\log \frac{1}{\delta} n^{\rho} U_{hash} [\frac{U_{check}}{U_{hash}}]^{\rho} \alpha_0 \alpha^{-\rho})$$

and space complexity

$$\Omega(dn + (\log \frac{1}{\delta}) n^{1+\rho} [\frac{U_{check}}{U_{hash}}]^{\rho} \alpha^{-\rho})$$

The number of returned points is $\Omega(\log \frac{1}{\delta} n^{\rho} [\frac{U_{check}}{U_{hash}}]^{\rho-1} \alpha^{1-\rho})$. Here $\alpha_0 = \frac{\log(\tau \frac{U_{check}}{U_{hash}} n)+1}{\tau}$ $\alpha = \frac{\rho \log(\tau \frac{U_{check}}{U_{hash}} n)+1}{\tau}$, $\rho = \frac{\log p_{nn}}{\log p_{any}}$ and $\tau = \log(p_{nn}/p_{any})$.

For conventional binary LSH with $U_{hash} = U_{check} = \Theta(d)$, $U_{bin} = \Theta(1)$, the following corollary gives us a simplified result:

Corollary 2.4.2. *For conventional binary LSH with $U_{hash} = U_{check} = \Theta(d)$, $U_{bin} = \Theta(1)$, the time complexity of LSH is*

$$\Omega(\log \frac{1}{\delta} d n^{\rho} \beta_0 \beta^{-\rho})$$

and space complexity is

$$\Omega(dn + (\log \frac{1}{\delta}) n^{1+\rho} \beta^{-\rho})$$

Here $\beta_0 = \frac{\log(\tau n)+1}{\tau}$, $\beta = \frac{\rho \log(\tau n)+1}{\tau}$, $\rho = \frac{\log p_{nn}}{\log p_{any}}$ and $\tau = \log(p_{nn}/p_{any})$.

In (2.3), If $c_1(\frac{1}{n}\sum_{i=1,...n} p_{X_i})^k \leq \widehat{P}_{any,1} = \frac{1}{n}\sum_{i=1,...n}(p_{X_i})^k \leq c_2(\frac{1}{n}\sum_{i=1,...n} p_{X_i})^k$, for two constant c_1 and c_2, or in other words,

$$\frac{1}{n}\sum_{i=1,...n}(p_{X_i})^k = \Theta((\frac{1}{n}\sum_{i=1,...n} p_{X_i})^k)$$

, then we will have $\widehat{P}_{any,1} = \Theta((p_{any})^k)$ and hence,

$$T_{LSH}(k,L) = \Theta(T(k,L)).$$

In this case, we can actually get a tight bound for the time complexity of LSH:

Theorem 2.4.3. *Given the data set* $\{X_i, i = 1, ..., n\}$*, if* $\frac{1}{n}\sum_{i=1,...n}(p_{X_i})^k = \Theta((\frac{1}{n}\sum_{i=1,...n} p_{X_i})^k)$ *is true, to achieve the exact nearest neighbor with a probability* $1-\delta$*, LSH will have a time complexity*

$$\Theta(\log\frac{1}{\delta} n^\rho U_{hash}[\frac{U_{check}}{U_{hash}}]^\rho \alpha_0 \alpha^{-\rho})$$

and space complexity

$$\Theta(dn + (\log\frac{1}{\delta})n^{1+\rho}[\frac{U_{check}}{U_{hash}}]^\rho \alpha^{-\rho})$$

Corollary 2.4.4. *For conventional binary LSH with* $U_{hash} = U_{check} = \Theta(d)$ *,* $U_{bin} = \Theta(1)$*, the time complexity in Theorem 2.4.3 will become*

$$\Theta(\log\frac{1}{\delta} d n^\rho \beta_0 \beta^{-\rho})$$

and space complexity become

$$\Theta(dn + (\log\frac{1}{\delta})n^{1+\rho}\beta^{-\rho})$$

The assumption that $\frac{1}{n}\sum_{i=1,...n}(p_{X_i})^k = \Theta((\frac{1}{n}\sum_{i=1,...n} p_{X_i})^k)$ seems quite strict at first glance, because it requires that the values of p_{X_i} are not far from each other, in other words p_{X_i} need to be "concentrated". However, as will be discussed in Section 7.2, when the data dimensions is large enough, $D(X_i, q)$, the distance between the query q and data point X_i, will actually concentrate, and hence p_{X_i} will also concentrate too. In other words, the assumption that $\frac{1}{n}\sum_{i=1,...n}(p_{X_i})^k = \Theta((\frac{1}{n}\sum_{i=1,...n} p_{X_i})^k)$ may be practical for high-dimensional data.

2.4.2 New Upper Bounds

Previous theoretical study on the LSH complexity usually focus on the upper bound of the time complexity. For example, for LSH with (r, cr, p_1, p_2) hash functions, in [19, 20, 21, 22], the time complexity of LSH is upper bounded by a function, more specifically,

$$O(dkL + dLn(p_2)^k)$$

, and the constraints is to

$$s.t., (1 - (p_1)^k)^L \leq \delta$$

Basically, the previous works on LSH complexity have studied the following optimization problem,

$$T'_{min} = \min_{k,L} U_{hash} kL + U_{check} Ln(p_2)^k \tag{2.10}$$

$$s.t., (1 - (p_1)^k)^L \leq \delta \tag{2.11}$$

and provide an upper bound for this optimization problem, as shown in to get Theorem 2.1.1, by luckily picking a sub-optimal k and L.

However, in (2.5), if we replace p_{any} with p_2, p_{nn} with p_1, (2.5) will have exactly the same form as (2.10). So all the proofs and conclusions for Theorem 2.3.1 and Collary 2.4.4 are still applicable with the same replacement. In other words,

$$T'_{min} = \Theta(\log \frac{1}{\delta} n^\rho U_{hash} [\frac{U_{check}}{U_{hash}}]^\rho \alpha_0 \alpha^{-\rho})$$

Here $\alpha_0 = \frac{\log(\tau \frac{U_{check}}{U_{hash}} n)+1}{\tau}$

$\alpha = \frac{\rho \log(\tau \frac{U_{check}}{U_{hash}} n)+1}{\tau}$, $\rho = \frac{\log p_1}{\log p_2}$ and $\tau = \log(p_1/p_2)$.

So we can have a new upper bound for LSH:

Theorem 2.4.5. *With (r, cr, p_1, p_2) hash function family, LSH can solve the c-approximate nearest neighbor problem with time complexity $O(\log \frac{1}{\delta} n^\rho U_{hash} [\frac{U_{check}}{U_{hash}}]^\rho \alpha_0 \alpha^{-\rho})$.*

Corollary 2.4.6. *With (r, cr, p_1, p_2) hash function family and moreover, if $U_{hash} = U_{check} = \Theta(d)$, LSH can solve the c-approximate nearest neighbor problem with time complexity $O(\log \frac{1}{\delta} dn^{\rho} \beta_0 \beta^{-\rho})$. Here $\beta_0 = \frac{\log(\tau n)+1}{\tau}$, $\beta = \frac{\rho \log(\tau n)+1}{\tau}$, $\rho = \frac{\log p_1}{\log p_2}$ and $\tau = \log(p_1/p_2)$.*

From Theorem 2.4.5 and Collary 2.4.6, we can see that the previous upper bound in Theorem 2.1.1 might be loose, especially when p_1 close to p_2. For example, an interesting extreme case is when $p_2 = p_1$. In this case, the upper bound in previous time complexity is $O(\log \frac{1}{\delta} dn \log_{p_2^{-1}} n)$, as shown in Theorem 2.1.1, It is even worse than linear scan (i.e.,, $O(dn)$), which is intuitively not very possible. On the contrary, our new upper bound in Collary 2.4.6 shows the time complexity of LSH in this case should be $O(\log \frac{1}{\delta} dn)$, the same as linear scan.

2.5 Parameters for Locality Sensitive Hashing

The current literature does not give a definitive statement about how to find the best parameter values. The previous theoretical results about the LSH parameters are mainly based on [19, 20, 21], which shows $k = O(\log_{p_2^{-1}} n)$. First, since p_2 is independent of the data set, so it will give the same parameters for all data set with the same number of points, without considering the distribution of the data at all, which of course will not reasonable for lots of cases. Moreover, the result is in a range of $O()$, which may be quite loose in practice.

In this Chapter, we will present the analysis of the parameters for LSH for each particular data set, which not only benefits us with algorithms to obtain parameters in practice, but also provides us deeper understanding and better insights for them.

During the discussions of the time and space complexity for LSH, we have actually provided some information about the range for optimal k of LSH (in $O()$ or $\Theta()$terms), for instance, as shown in Theorem 9.1.3 in the Appendix. But in practice we need the actual values of the parameters rather than the ranges.

So in this Chapter, we will go further to find out the actual value for optimal parameters k and L of binary LSH. We will also analyze how the optimal parameters k and L are affected by different factors, like the number of data n, the probability profile p_{nn} and p_{any}, the experiment environment constants U_{hash}, U_{check}, etc.

Recall Theorem 9.1.3 (in the Appendix for the proofs of Chapter 2), we have

$$p_{any}^{-k_{min}} = \frac{\alpha_1 n}{(k_{min} \log p_{nn}^{-1} + 1)}$$

where

$$\alpha_1 = \frac{U_{check} \log(p_{nn}/p_{any})}{U_{hash}}.$$

Moreover, recall that $k_{\min} = \Theta(\frac{\log(\alpha_1 n)}{\log p_{any}^{-1}})$ as shown in Lemma 9.1.4 and 9.1.5, so when n is large, $k_{\min}$ will not be small, and hence $k_{\min} \log p_{nn}^{-1} \geq 1$, unless p_{nn} is almost 1, which is not very possible for real world high dimensional data.

So approximately we have

$$p_{any}^{-k_{\min}} = \frac{\alpha_1 n}{k_{\min} \log p_{nn}^{-1}}$$

We can rewrite it as

$$k_{\min} = \frac{\log(n)}{\log p_{any}^{-1}} + \frac{\log(\eta_0)}{\log p_{any}^{-1}} - \frac{\log(k_{\min})}{\log p_{any}^{-1}} \tag{2.12}$$

where $\eta_0 = \alpha_1 / \log p_{nn}^{-1} = \frac{U_{check} \log \frac{p_{nn}}{p_{any}}}{U_{hash} \log p_{nn}^{-1}}$.

So an approximate solution for the equation (2.12) can be obtained as

$$k_{\min} = k_0 - \frac{\log(k_0)}{\log p_{any}^{-1}} \tag{2.13}$$

where $k_0 = \frac{\log(n)}{\log p_{any}^{-1}} + \frac{\log(\eta_0)}{\log p_{any}^{-1}}$.

Actually, putting $k_{\min} = k_0 - \frac{\log(k_0)}{\log p_{any}^{-1}}$ into the right hand side of (2.12), we get

$$\frac{\log(n)}{\log p_{any}^{-1}} + \frac{\log(\eta_0)}{\log p_{any}^{-1}} - \frac{\log(k_{\min})}{\log p_{any}^{-1}} = k_0 - \frac{\log(k_0 - \frac{\log(k_0)}{\log p_{any}^{-1}})}{\log p_{any}^{-1}}$$

$$= k_0 - \frac{\log(k_0)}{\log p_{any}^{-1}} - \frac{\log(1 - \frac{\log(k_0)}{k_0 \log p_{any}^{-1}})}{\log p_{any}^{-1}} = k_{\min} - \frac{\log(1 - \frac{\log(k_0)}{k_0 \log p_{any}^{-1}})}{\log p_{any}^{-1}}$$

Note that $\frac{\log(1-\frac{\log(k_0)}{k_0 \log p_{any}^{-1}})}{\log p_{any}^{-1}} \approx \frac{\frac{\log(k_0)}{k_0 \log p_{any}^{-1}}}{\log p_{any}^{-1}}$ is quite small. So the approximate solution is quite accurate.

Moreover, with k_{min}, we can obtain L_{min} as follows:

$$L_{min} = \frac{-\log \delta}{p_{nn}^{k_{min}}} \tag{2.14}$$

From (2.13) and (2.14), we can find out how the parameters k and L are affected by factors such as n, p_{nn}, p_{any}, U_{hash}, U_{check}, and δ. [5]. The relationships are shown in Table 2.1, where "↑" means k or L will increase if one of the factors increase; "↓" means k or L will decrease if one of the factors increase; "-" means k or L will not be affected; "x" means how k or L will be affected is unknown.

For example, it is very easy to see that if n increases, k_0 will increase. Note that k_0 will increase much faster than $\log(k_0)$, so k_{min} will increase. Moreover, when U_{check} or p_{nn} increase, η_0 will increase and so will k_0 and k_{min}. Also, when p_{any} increases, k_{min} will increases, because $\frac{\log(n)}{\log p_{any}^{-1}}$ increases faster than $\frac{\log(k_0)}{\log p_{any}^{-1}}$. Finally, it is easy to see δ will not affect k_{min}.

Moreover, when k_{min} increases and p_{nn} is fixed, L will increase. So L is affected by n, U_{check}, U_{hash} or p_{any} the same way as k is. Moreover, we will need less tables L for larger δ.

	n	p_{nn}	p_{any}	U_{hash}	U_{check}	δ
k	↑	↑	↑	↓	↑	-
L	↑	x	↑	↓	↑	↓

Table 2.1: The effect of different factors on the parameters k and L of LSH (with single probe).

[5] n: the database size, p_{nn}: the probability of two nearest neighbor points to have the same code for one hash function $h(x)$, p_{any}: the probability of two random points to have the same code for one hash function $h(x)$, U_{hash}: the unit cost of compute $h(x)$, U_{check}: the unit cost of checking one candidate (e.g., compute the distance), and δ: the error probability to miss the true nearest neighbors

2.6 Other NN Search Methods with Random Partitions

We have formulated the time and space complexity for locality sensitive hashing in Section 2.2, and presented the tight bound in Section 2.3. In this section, we will extend the formulation and the bound to other NN search methods with Random Partitions.

2.6.1 Time and Space Complexity for Nearest Neighbor Preferred Hashing (NPH) Methods

The locality sensitive hashing is formally defined as (r, cr, p_1, p_2)-sensitive hashing, as shown in Section 2.1. Mainly speaking, it requires the collision probability of two points for the hash function $h(x)$ to be inversely monotonic to their distance.

However, for lots of hash functions $h(x)$, it might be too strict to require the monotonic property, or sometimes just too difficult to prove the monotonic property in theory. Actually in our discussions of the lower bound for LSH in above sections, we do not actually need the monotonic property. What we need is a less strict requirement: the probability of two nearest points to have the same hash code is larger than the probability of two random points to have the same hash code. In other words, nearest neighbors are preferred by the hash functions $h(x)$. More specifically, we can formally define Nearest Neighbor Preferred Hashing (NPH) as follows: **Nearest Neighbor Preferred Hashing (NPH)**

Given a data set X and a distance D, a random hash function $h(x)$ is called Nearest Neighbor Preferred Hashing (NPH) in terms of D, if $p_{nn} \geq p_{any}$, where p_{nn} and p_{any} are defined as in Section 2.2.

First of all, it is easy to see all LSH based on hyper-plane are Nearest Neighbor Preferred Hashing, because of the inverse monotonic property of LSH. However, besides hyper-plane based LSH functions, there are also many other possible hash

functions which are NPH.

For example, besides using hyper-planes like in LSH, we can also use other structures to partition the data, e.g., hyper-spheres instead of hyper-planes as hashing functions to partition the data. More specifically, the hash function is defined as $h(x) = sign(D(v_i, x) - b)$, where v_i is a randomly chosen pivot in R^d, and b is a parameter of the radius of the hyper-sphere served as a distance threshold. In other words, $h(x) = 1$, if $D(v_i, x) <= b$, i.e., data point x falls inside the hyper-sphere; $h(x) = 0$, if $D(v_i, x) > b$, i.e., data point x falls outside the hyper-sphere. Moreover, besides hyper-planes and hyper-spheres, we can actually use any (closed) surfaces as the hash function to partition the space. It is intuitive that $p_{nn} \geq p_{any}$ for these kinds of hash functions.

Similarly as in LSH, if we use k hyper-sphere based hash functions $h(x)$ as one partition to build one hash table, and repeat the partition L times to get L hash tables. Then we will get the formulation of the time and space complexity as conventional LSH with hyper-planes, and hence obtain the same bound on time and space complexity:

Theorem 2.6.1. *To achieve the exact nearest neighbor with a probability* $1 - \delta$*, Nearest Neighbor Preferred Hashing (NPH) will have a time complexity*

$$\Omega(\log \frac{1}{\delta} n^{\rho} U_{hash} [\frac{U_{check}}{U_{hash}}]^{\rho} \alpha_0 \alpha^{-\rho})$$

and space complexity

$$\Omega(dn + (\log \frac{1}{\delta}) n^{1+\rho} [\frac{U_{check}}{U_{hash}}]^{\rho} \alpha^{-\rho})$$

The number of hash tables L *is* $\Theta((\log \frac{1}{\delta}) n^{\rho} [\frac{U_{check}}{U_{hash}}]^{\rho} \alpha^{-\rho})$.

Here $\alpha_0 = \frac{\log(\tau \frac{U_{check}}{U_{hash}} n)+1}{\tau}$

$\alpha = \frac{\rho \log(\tau \frac{U_{check}}{U_{hash}} n)+1}{\tau}$, $\rho = \frac{\log p_{nn}}{\log p_{any}}$ *and* $\tau = \log(p_{nn}/p_{any})$.

However, different partition structures will give us different values of p_{any} and p_{nn}, which will affect the time and space complexity.

2.6.2 Time Complexity for Nearest Neighbor Preferred Partition (NPP)

We can further extend the Nearest Neighbor Preferred Hashing to other kinds of partitions besides of hashing. More specifically, **k-level Nearest Neighbor Preferred Partitions (NPP)** is defined as:

For a random partition Ψ on a given data set X, suppose random Ψ is determined by one parameters k. If for one point X_i, its probability to be returned is $\widehat{P}_{X_i}(\Psi)$ has a form like $\widehat{P}_{X_i}(\Psi) = \Theta((p_{X_i})^k)$, and moreover $p_{nn} \geq p_{any}$ where p_{nn} and p_{any} are defined as in Section 2.2, then it is called Nearest Neighbor Preferred Partition.

It is not difficult to see all the analysis and theories for LSH are applicable to NPP too, if we repeat k-level Nearest Neighbor Preferred Partitions (NPP) for L times, where k and L is chosen as discussed in our LSH analysis.

For example, suppose the partition is done by a random forest with L binary trees. In each tree, there are k levels, and hence there are in total 2^k leaf nodes in each tree (note that the root node is level 0). And for each internal node, it has two branches, determined by a single bit, which is computed via some binary function [6]. In other words, if $h(x) = 0$, point x belongs to the left branch; if $h(x) = 1$, point x belongs to the right branch.

Similarly as in LSH, denote p_{X_i} as the probability of point X_i to have the same code as query q for the binary function.

In a binary random tree of k levels as discussed above, we can prove that the probability of any random two points to be in the same leaf node is $(p_{X_i})^k$. Actually, for level 0 and level 1, it is easy to see the conclusion is correct. Now assume the conclusion is correct for level i, we need to prove it is still correct for level $i+1$. In level i, there are 2^i nodes. Suppose $\overline{p}_{X_i,j}$ for $j = 1, 2, ..., 2^i$ is the probability for X_i

[6] for example $h(x) = sign(v \cdot x - b)$, where v is a randomly generated vector and b is a threshold constant.

and the query to both belong to node j. Then we have $\sum_{j=1}^{2^i} \overline{p}_{X_i,j} = (p_{X_i})^i$ from the assumption. In level $i+1$, for two branches of node j, the probability for X_i and the query both belong one of two branches is $p_{X_i}\overline{p}_{X_i,j}$. And hence the probability for two random points to fall into the same node in level $i+1$ is $\sum_{j=1}^{2^i} p_{X_i}\overline{p}_{X_i,j} = (p_{X_i})^{i+1}$.

Denote U_{node} as the time cost to compute the split criteria in one internal node, and $U_{indices}$ as the time cost to compute the index of query leaf node in the tree. Then $U_{indices} = kU_node$, since we need to traverse k nodes to get the leaf. Moreover, denote $U_{regions}$ as the time cost to access one query leaf node in the indexing tree stored in the memory. Moreover, it is easy to see $U_{node} = \Theta(d)$ and $U_{regions} = \Theta(1)$ too, which are the same as U_{hash} and U_{bin} in LSH. And similarly, $U_{regions}$ can be ignored, compared to other terms. So

The optimal partition introduced by random forests is actually to find the optimal parameters k and L as follows,

$$T_{randomtrees}(k,L) \geq T(k,L) = U_{node}kL + U_{check}Ln(p_{any})^k$$
$$s.t.,$$
$$1-(1-(p_{nn})^k)^L \geq 1-\delta \tag{2.15}$$

So all the discussions of time complexity in previous sections about LSH are directly applicable to NN search methods based on random trees/forest.

Theorem 2.6.2. *To achieve the exact nearest neighbor with a probability* $1-\delta$*, Nearest Neighbor Preferred Partition (NPP) will have a time complexity*

$$\Omega(\log\frac{1}{\delta} n^\rho U_{node}[\frac{U_{check}}{U_{node}}]^\rho \alpha_0 \alpha^{-\rho})$$

The number of partitions L *is* $\Theta((\log\frac{1}{\delta})n^\rho[\frac{U_{check}}{U_{node}}]^\rho\alpha^{-\rho})$.
Here $\alpha_0 = \frac{\log(\tau\frac{U_{check}}{U_{node}}n)+1}{\tau}$
$\alpha = \frac{\rho\log(\tau\frac{U_{check}}{U_{node}}n)+1}{\tau}$, $\rho = \frac{\log p_{nn}}{\log p_{any}}$ *and* $\tau = \log(p_{nn}/p_{any})$.

2.6.3 Parameters for NPH and NPP

As discussed in section 2.6, Nearest Neighbor Preferred Hashing (NPH) and Nearest Neighbor Preferred Partitions (NPP) share the same analysis as LSH. So it is easy to see the optimal parameters in Section 2.5 is applicable to NPH and NPP too. For example, for random forests method, k here is number of levels in each tree, and L is the number of trees. And we can find the parameters of k as (2.13) and of L as (2.14) for random forest methods.

Part III

Nearest Neighbor Search via Learning Based Partitions

One main disadvantage for NN methods with random partitions is that the partition function is randomly generated, and hence may not be very efficient. LSH and other random partition methods resolve this problem by utilizing many partitions, e.g., many hash tables in LSH method, or many trees in random forest methods. However, when the memory budget are limited, the number of partitions can only be small. In this case, random partition methods may perform poorly. However, if we can improve each partition via learning from data (rather than generating randomly), we may still get satisfying performance with few partitions, when the memory budget is limited.

So in this part of the thesis, we will discuss about nearest neighbor search with learning based partitions, including indexing with learning based hashing and indexing with clustering. To make our discussions easier, we assume to learn only one partition ($L = 1$).

Unlike random partitions, for learning based partitions, the partitions Ψ depend on the training data, and hence Ψ and especially $\widehat{P}_{nn}(\Psi)$ and $\widehat{P}_{any}(\Psi)$ are usually very difficult to formulate. So in the discussion of this part, we will need lots of approximation, heuristics, and intuition, besides theoretical analysis.

More specifically,

1. In Chapter 3, based on the formulation of optimal data partition in (1.1), we will obtain the two criteria for learning based hashing, i.e., balancing buckets and preserve nearest neighbors, and moreover formulate a joint optimization to achieve the two criteria.

2. In Chapter 4, based on the formulation of optimal data partition in (1.2), we will extend the conventional K-means clustering algorithm to balanced K-means clustering, which is more suitable to be utilized for indexing applications.

The proofs for theorems in Chapter 3 and Chapter 4 can be found in Section 9.2 and 9.3 in the appendix respectively.

Chapter 3

Algorithms of Optimal Partitions for Hashing Based NN Search

3.1 Optimal Partition Criteria for Hashing

Now we consider to learn hash functions instead of using random hash functions for partition the data. To make things easier, we assume we want to learn one partition ($L = 1$) with k hash bits by hash functions $H(x)$, where the m-th bit is computed by hash function $H_m(x)$. We can start by using linear hash functions, i.e., $H(x) = T^T x - b$, or equivalently, $H_m(x) = T_m^T x - b_m$, where T is a matrix of m by d, T_m is the m-th column of T, and b is vector, and b_m is the m-th dimension of b.

Recall the optimal partition formulation in (1.1), i.e.,

$$\begin{aligned} \min_{\Psi} T(\Psi) = T_{indices}(\Psi) + T_{regions}(\Psi) + n\widehat{P}_{any}(\Psi)U_{check} \\ s.t., \\ \widehat{P}_{nn}(\Psi) \geq 1 - \delta \end{aligned} \tag{3.1}$$

In the case of using k hash bits for NN search, $T_{indices} = kU_{hash}$, where U_{hash} is the time cost to compute one inner product between the query point and one projection

vector T_m, and hence $U_{hash} = \Theta(d)$.

Moreover, if we only probe one bucket in the hash table, $T_{regions} = \Theta(1)$. If we probe more buckets, like all buckets within hamming distance r to the query bucket, then $T_{regions} = \Theta(C_k^r)$.

For a fixed k, the partition Ψ is determined by T_m and b_m, $m = 1, ..., k$. However, $T_{indices}(\Psi)$, $T_{regions}(\Psi)$, U_{check} and n are all independent of T_m and b_m. So in the case of learning k hash functions for optimal partition of NN search, we can simply (1.1) as:

$$\begin{aligned} &\min_{T,b} \widehat{P}_{any}(T, b) \\ &s.t., \\ &\widehat{P}_{nn}(T, b) \geq 1 - \delta \end{aligned} \tag{3.2}$$

The above formulation of optimal partition for learning k hash functions actually tries to find T_m and b_m, $m = 1, ..., k$ such that $\widehat{P}_{nn}(\Psi))$ is as large as possible, and $\widehat{P}_{any}(\Psi)$ is as small as possible. Moreover, note that decreasing $\widehat{P}_{any}(\Psi)$ will lead to less search time, and increasing $\widehat{P}_{nn}(\Psi))$ will provide higher search accuracy.

However, the unsolved problems are how to formulate $\widehat{P}_{any}(T, b)$ and $\widehat{P}_{nn}(T, b)$, and how to solve the optimization problem. We will answer these questions in following sections.

3.1.1 Bucket Balancing for Search Time ($\widehat{P}_{any}(\Psi)$)

From (3.2), we want to find partition to decrease $\widehat{P}_{any}(\Psi)$.

Suppose there are in total K regions (buckets) after partition. (In the case of hashing with k bits, $K = 2^k$.) Suppose there are n_i points in bucket i, for $i = 1, ..., K$. The following theorem shows that $\widehat{P}_{any}(\Psi)$ is minimized when all n_i are equal.

Theorem 3.1.1. $\widehat{P}_{any}(\Psi)$ ***can be minimized if all buckets (regions) are perfectly balanced, i.e., every bucket contains the same number of samples. In other words,*** $n_i = n/K$, $i = 1, ..., K$.

Proof:

Denote p_i as the probability for one random point to fall into cluster i. Then when n is large enough, the probability for a random query and a random database point both falls into the cluster i is p_i^2. Then

$$\widehat{P}_{any}(\Psi) = \sum_{i=1}^{K} p_i^2 = \frac{1}{n^2} \sum_{i=1}^{K} n_i^2$$

Note that $\sum_{i=1,...,K} n_i = n$, $\sum_{i=1}^{K} (n_i)^2$ *will minimized if n_i are equal, i.e., $n_i = n/K$, $i = 1, ..., K$.*

The following theorem provides a maximum entropy and moreover a minimum mutual information criteria to make all n_i equal.

Denote y as a k-dimension random binary vector. y_m is the m-th dimension of y, which is a binary random variable generated by $H_m(x)$.

Theorem 3.1.2. ***Suppose*** $y = H(x)$***, i.e.,*** y ***is the*** k ***hash bits for a random vector*** x***. The regions created by the partitions from hash functions*** H***, are perfectly balanced, i.e., all*** n_i ***are equal, if and only if*** $Entropy(y)$ ***is maximized, or equivalently, mathematical expectation*** $E(y_m) = 0$ ***for*** $m = 1, ..., k$ ***and the mutual information*** $I(y_1, ..., y_m, ..., y_k)$ ***is minimized.***

Proof:

Note that $Entropy(y) = \{-\sum_{i=1}^{2^k} P(y = a_i) \log P(y = a_i)\} = \{-\sum_{i=1}^{2^k} \frac{n_i}{n} \log(\frac{n_i}{n})\}$. *It is easy to see that $n_i = \frac{n}{2^k}$ for $i = 1, ..., 2^k$, if and only if $Entropy(y)$ gets its maximum value.*

As shown in [54, 55],

$$Entropy(y) = \sum_{m=1}^{k} Entropy(y_m) - I(y_1, ..., y_m, ..., y_k) \tag{3.3}$$

where $I()$ is the mutual information.

So Entropy(y) would be maximized, if $\sum_{m=1}^{k}$ Entropy(y_m) is maximized and $I(y_1, ..., y_m, ..., y_k)$ is minimized. Moreover, note that y_m is a binary random variable. If the mathematical expectation $E(y_m) = 0$, half samples would have bit $+1$ and the other half would have bit -1 for y_m, which means Entropy$(y_m) = 1$, and is maximized.

In conclusion, if $E(y_m) = 0, m = 1, ..., k$ and $I(y_1, ..., y_m, ..., y_k)$ is minimized, Entropy(y) would be maximized, and the search time would be minimized. This completes the proof of Proposition 1.

Note that if $I(y_1, ..., y_m, ..., y_k)$ is minimized, it means $y_1, ..., y_m, ..., y_k$ are independent [1]. So minimizing mutual information criterion is also to provide the most compact and least redundant hash codes.

3.1.2 Preserve Nearest Neighbors for Search Accuracy ($\widehat{P}_{nn}(\Psi)$)

From (3.2), we know that to obtain a good partition, we need to increase $\widehat{P}_{nn}(\Psi))$ to improve search accuracy.

The exact formulation for $\widehat{P}_{nn}(\Psi))$ in the case of learning based hashing is unfortunately very difficult. However, it is intuitive that large $\widehat{P}_{nn}(\Psi))$ means nearest neighbor preserving, i.e., to keep nearest neighbors in the same region (bucket) or nearby regions. Lots of approaches are proposed to preserve nearest neighbors in different index methods. In hashing methods like [31, 32], there are usually a nearest neighbor preserving term. We will follow the nearest neighbor preserving term in [32].

More specifically, denote $Y_i = H(X_i)$, the hash bits for X_i, which is a k-dimensional vector. Since $\{X_i, i = 1, ..., n\}$ are i.i.d. sampled from a random vector x, $\{Y_i =$

[1] Independence among hash bits are mentioned in spectral hashing [32], but it does not relate independence to search time, and moreover, there is no actual formulation, derivation or algorithm to achieve the independence.

$H(X_i), i = 1, ..., n\}$ are i.i.d. samples from $y = H(x)$. For two data samples X_i and X_j in the training set, suppose W_{ij} is the similarity between X_i and X_j. Similarity W_{ij} can come from feature similarity or label consistency, etc., depending on the applications. The only requirement for W is the symmetry. The nearest neighbor term in [32] is:

$$D(Y) = \sum_{i=1,...,n} \sum_{j=1,...,n} W_{ij} ||Y_i - Y_j||^2 \tag{3.4}$$

where Y is the set of all Y_i. With this criterion, samples with high similarity, i.e., larger W_{ij} , are supposed to have similar hash codes, i.e., smaller $||Y_i - Y_j||^2$. Here $\sum_{i=1,...,n} \sum_{j=1,...,n} W_{ij} ||Y_i - Y_j||^2$ tries to preserve feature similarity between original data points. On average, samples with high similarity, i.e., larger W_{ij} , should have similar hash codes, i.e., smaller $||Y_i - Y_j||^2$.

However, W in our algorithm does not need to be fixed as $W_{ij} = \exp(-||X_i - X_j||^2/\sigma^2)$ in spectral hashing. Furthermore, the common requirements for similarity matrix, like positive semi-definite, or non-negative elements, are unnecessary here either. Actually, any symmetric W can be applied in our method. So, besides the usual feature similarities, other kinds of similarity, e.g., those based on class label consistency, can also be used. In other words, our method supports supervised, unsupervised, and semi-supervised hashing, with W respectively defined as label similarity only, feature similarity only, or combination of label similarity and feature similarity.

3.1.3 Intuition

On one hand, it is easy to see that nearest neighbor preserving alone does not guarantee a good hash/serach method. For example, as an extreme case, one can always assign every data point to the same region, and hence nearest neighbors are perfectly preserved. However, in this case, for every query, all the data would be returned, which is the worst case of search time, and actually equals linear scan. Moreover, the search precision will be very low too.

On the other hand, bucket balancing alone is not sufficient either. As an extreme case, we can randomly assign data points to different regions to make sure every region contains exactly the same number of points. However, this kind of partition will cause very bad search accuracy, which is actually the same as returning random results.

So a good partition should not focus on nearest neighbor preserving only, or bucket balancing alone only, but aim at a good tradeoff between search accuracy and search time, by jointly optimize nearest neighbor preserving and bucket balancing.

3.2 Hashing with Joint Optimization

3.2.1 Formulation of Hashing with Joint Optimization

Note that $E(y_m) = 0, m = 1, ..., k$ means $E(y) = 0$, which can further be rewritten as $\sum_{i=1}^{n} Y_i = 0$ with samples $\{Y_i, i = 1, ..., n\}$. By incorporating the similarity preserving term $D(Y)$ for search accuracy and the mutual information criterion for search time, the problem of joint optimization for $\widehat{P}_{any}(\Psi)$ and $\widehat{P}_{nn}(\Psi)$. together can now be formulated as:

$$\begin{aligned} &\min_{H} I(y_1, ..., y_m, ..., y_k) \\ &s.t., \sum_{i=1}^{n} Y_i = 0 \\ &D(Y) \leq \eta \\ &Y_i = H(X_i) \end{aligned} \tag{3.5}$$

We first parameterize H, so that it can be optimized more easily. For simplicity, we first assume data are in vector format, and H is a linear function with a sign threshold, i.e.,

$$H(x) = sign(T^T x - b) \tag{3.6}$$

and later on, we will provide a generalized version in section 3.3.1.1 to handle data with general format. Here T is a projection matrix of $d \times k$ and b is a vector.

Even with the parameterizations of H, the problem in (3.5) is still difficult to optimize (e.g., non-differential), and hence relaxation is needed. In the following discussion, we will show how to relax equation (3.5) with the parameterized H.

3.2.2 Relaxation for $D(Y)$

First of all, recall that $Y_i = H(X_i) = sign(T^T X_i - b)$. A traditional relaxation as in many algorithms (e.g., [32]), is to remove the binary constraint by ignoring the $sign()$ function so that $D(Y)$ is differentiable. In other words, $D(Y)$ is relaxed as:

$$\sum_{i,j=1,...,n} W_{ij}||(T^T X_i - b) - (T^T X_j - b)||^2 = \sum_{m=1}^{k} T_m^T C T_m \tag{3.7}$$

where $C = XLX^T$. Here L is the Laplacian matrix $L = D - W$. D is a diagonal matrix, $D_{i,i} = \sum_{j=1}^{n} W_{i,j}$.

Moreover, $\sum_{i=1}^{n} Y_i = 0 \Rightarrow \sum_{i=1}^{n} (T^T X_i - b) = 0$, so $b = \frac{1}{n} T^T \sum_{i=1}^{n} X_i$.

3.2.3 Relaxation for minimizing $I(y_1, ..., y_m, ..., y_k)$

Denote $z_m = T_m^T x$, where T_m is the m-th column of T. So $y_m = sign(z_m - b_m)$.

It is easy to see that if z_m are independent, y_m would be independent too. Hence if $I(z_1, ..., z_m, ..., z_k) = I(T_1^T x, ..., T_m^T x, ..., T_k^T x)$ is minimized, $I(y_1, ..., y_m, ..., y_k)$ would be minimized. So we will minimize $I(z_1, ..., z_m, ..., z_k) = I(T_1^T x, ..., T_m^T x, ..., T_k^T x)$ instead of $I(y_1, ..., y_m, ..., y_k)$ in equation (3.5).

In the field of ICA, independence or mutual information is well studied for a long time. As discussed in [54], minimizing $I(z_1, ..., z_m, ..., z_k)$ can be well approximated as minimizing $C_0 - \sum_{m=1}^{k} ||g_0 - E(G(z_m))||^2$, which equals maximizing

$$\sum_{m=1}^{k} ||g_0 - \frac{1}{n} \sum_{i=1}^{n} G(T_m^T X_i)||^2 \tag{3.8}$$

under the constraint of whiten condition, i.e.,

$$\begin{aligned} & E\{z_m z_j\} = \delta_{mj} \\ & \Rightarrow E\{(T_m^T x)(T_j^T x)\} = T_m^T E\{xx^T\} T_j = T_m^T \Sigma T_j = \delta_{mj} \end{aligned} \tag{3.9}$$

for $1 \leq m, j \leq k$.

Here C_0 is a constant, $E()$ means the expectation, $G()$ is some non-quadratic function such as $G(u) = -e^{-u^2/2}$, or $G(u) = \log\cosh(u)$, etc., and g_0 is a constant. $\delta_{mj} = 1$, if $m = j$; $\delta_{mj} = 0$, if $m \neq j$. $\Sigma = E(xx^T)$.

3.2.4 Similarity Preserving Independent Component Analysis (SPICA)

In sum, after relaxation with equation (3.7), (3.8), and (3.9), the problem in equation (3.5) can now be formulated as:

$$\begin{aligned} & \max_{T_m, m=1...k} \sum_{m=1}^{k} ||g_0 - \tfrac{1}{n} \sum_{i=1}^{n} (G(T_m^T X_i))||^2 \\ & s.t., T_m^T \Sigma T_j = \delta_{mj}, 1 \leq m, j \leq k \\ & \sum_{m=1}^{k} T_m^T C T_m \leq \eta \end{aligned} \tag{3.10}$$

where $C = XLX^T$ and $\Sigma = E(xx^T)$. The hash bits for X_i can be computed as $Y_i = sign(T^T X_i - b)$, for $i = 1, ..., n$, where $b = \frac{1}{n} T^T \sum_{i=1}^{n} X_i$.

Surprisingly, after relaxation steps mentioned above the solution becomes quite intuitive. We call this method SPICA (Similarity Preserving Independent Component Analysis), because it incorporates a similarity preserving term into the Fast-ICA formulation [54].

3.3 Optimization

3.3.1 Optimization Algorithm

The optimization problem in both equations (3.10) and (3.12) is nonconvex. It is not trivial to obtain a fast algorithm to solve them efficiently, especially when the data set is very large. Inspired by the work in [54, 56], here we provide a fast and efficient approximate method to solve the problems of equation (3.10) and (3.12). The workflow of the optimization is described in Algorithm 1 with details. The method is shown to converge quite fast and perform well in the extensive experiments to be described later.

Note that γ in algorithm 1 is equivalent to parameter η in (3.10) and (3.12). Actually, $\gamma = 0$ means $\eta = \infty$. Larger γ is equivalent to smaller η.

The details of derivation for the algorithm are shown in Section 9.2 in the appendix.

3.3.1.1 Generalization–GSPICA

In practice, many applications involve structured data in the forms of graphs, trees, sequences, sets, or other formats. For such general data types, usually certain kernel functions are defined to compute the data similarities, e.g., [57, 58]. Moreover, even if the data are stored in the vector format, many machine learning solutions benefit from the use of domain-specific kernels, for which the underlying data embedding to the high-dimensional space is not known explicitly, namely only the pair-wise kernel function is computable. We can obtain the kernelized version of SPICA to deal with data of general format by parameterizing the hash functions as:

$$y = H(x) = sign(T^T K_x - b) \tag{3.11}$$

where $\mathrm{K}_x = [K(x, Z_1), ..., K(x, Z_i), ..., K(x, Z_p)]^T$. Here K is the kernel function, and $Z_i, i = 1, ..., p$ are some landmark samples, which for example can be a subset chosen from the original n samples. Usually $p \ll n$.

Algorithm 1 Workflow for optimization of SPICA.

Input: data $X_i, i = 1, ..., n$, similarity matrix W, the number of required bits: k . (By replacing X_i with K_{X_i} and X with $K_{p\times n}$, we can obtain the optimization for GSPICA.)

Output: hash functions to generate k bits for each sample, i.e., $Y_i = H(X_i) = sign(T^T X_i - b)$

Workflow:

1. Compute $\Sigma = \frac{1}{n}\sum_{i=1}^{n} X_i {X_i}^T$; Apply SVD to Σ, $\Sigma = \Omega\Lambda\Omega^T$;

2. $Q = \Omega_k \Lambda_k^{-\frac{1}{2}}$, where Λ_k is a diagonal matrix consisting of k largest eigen values of Λ, Ω_k is the corresponding column of Ω

3. Compute $C = XLX^T = X(D - W)X^T$, Compute $\widetilde{\mathrm{C}}$=$Q^T CQ$

4.

for $m = 1, ..., k$ **do**

if $m = 1$ **then**

$B = I$

else

apply QR decomposition to matrix $[\widetilde{T}_1, ...\widetilde{T}_{m-1}]$ to get matrix B, such that $[\widetilde{T}_1, ...\widetilde{T}_{m-1}, B]$ is a full-rank orthogonal matrix.

end if

$A = B^T \widetilde{C} B, \widehat{X}_i = B^T \widetilde{X}_i = B^T Q^T X_i$

Random intialize w

repeat

$\beta = \frac{1}{n}\sum_{i=1}^{n} (w^T \widehat{X}_i G'(w^T \widehat{X}_i)) - \gamma w^T A w.$

$w^+ = w - [\frac{1}{n}\sum_{i=1}^{n} (G''(w^T \widehat{X}_i))I - \beta I - \gamma A]^{-1}[\frac{1}{n}\sum_{i=1}^{n} (\widehat{X}_p G'(w^T \widehat{X}_i)) - \beta w - \gamma A w].$

$w = w^+/||w^+||.$

until converge

$\widetilde{T}_m = Bw$

end for

5. $T_m = Q\widetilde{T}_m$ and $b = \frac{1}{n} T^T \sum_{i=1}^{n} X_i$

6. For any sample X_i, compute its k hash bits $Y_i = sign(T^T X_i - b)$

Denote $z_m = T_m^T K_x$. With similar relaxation and derivation, we can get

$$\begin{aligned} &\max_{T_m, m=1...k} \sum_{m=1}^{k} ||g_0 - \tfrac{1}{n} \sum_{i=1}^{n} (G(T_m^T K_{X_i}))||^2 \\ &s.t., \sum_{m=1}^{k} T_m^T C T_m \leq \eta \\ &T_m^T \Sigma T_j = \delta_{mj}, 1 \leq m, j \leq k \end{aligned} \tag{3.12}$$

where

$$\mathrm{K}_{X_i} = [K(X_i, Z_1), ..., K(X_i, Z_p)]^T$$

Moreover $b = \frac{1}{n} T^T \sum_{i=1}^{n} K_{X_i}$ and $Y_i = sign(T^T K_{X_i} - b)$, for $i = 1, ..., n$. Here, $C = K_{p\times n}(D - W)K_{p\times n}^T$. $K_{p\times n}$ is defined as

$$(K_{p\times n})_{i,j} = K(Z_i, X_j),\ i = 1, \cdots, p,\ j = 1, \cdots, n. \tag{3.13}$$

And $\Sigma = E\{K_x {K_x}^T\} = \frac{1}{n} \sum_{i=1}^{n} K_{X_i} K_{X_i}^T = \frac{1}{n} K_{p\times n} K_{p\times n}^T$.

In Equation (3.12), one can see that Mercer condition is unnecessary for function K. Actually, any similarity function is applicable. We call the method in equation (3.12) Generalized SPICA (GSPICA), which can handle both vector data and structured data with any kernel function or similarity/proximity function K defined.

3.3.2 Complexity and Scalability

In algorithm 1 for SPICA and GSPICA, the bottleneck of time complexity is the computation of XWX^T or $K_{p\times n} W K_{p\times n}^T$ in step 3. When we have a large scale data set, what may consist of millions of samples, it would be very expensive to compute XWX^T or $K_{p\times n} W K_{p\times n}^T$, with a time complexity of $O(dn^2)$ and $O(pn^2)$ respectively.

One way to overcome the computation complexity is to use a sparse W. Generally speaking, one can always sample a small subset of training samples to compute similarity matrix, so that W is sparse, and hence the time complexity of XWX^T or $K_{p\times n} W K_{p\times n}^T$ is acceptable.

Another approach is by low rank representation/approximation for W such that $W = RQR^T$, where R and Q are low rank matrix. In this case, $K_{p\times n} W K_{p\times n}^T$ can

be computed as: $K_{p\times n}WK_{p\times n}^T = (K_{p\times n}R)Q(K_{p\times n}R)^T$ which involves small matrices only. There are several ways to obtain the low rank approximation, for example, we can choose W as $W = XX^T$ if X are normalized data, or apply Nyström algorithm [59] to get a low rank approximation for W. Moreover, when the W is defined by some "shift-invariant" kernel functions such as $W(i,j) = e^{-||X_i - X_j||^2/\sigma^2}$, we can apply the kernel linearization technique [60] to approximate W as $W = ZZ^T$, where Z are the random Fourier Features as in [60]. Note that we don't need to compute or store W, but only compute or store the low rank matrix.

With the speed up, algorithm 1 would take about $O(d^2n)$ or $O(p^2n)$ for SPICA or GSPICA respectively, which is close to state-of-the-art methods like spectral hashing [32] ($O(d^2n)$) or OKH [61] ($O(p^2n)$).

3.4 Degenerated Case with a Simple Solution

3.4.1 Formulation

As a variation of (3.5), we can put the mutual information term into constraints:

$$\begin{aligned} &\min_H D(Y) \\ &s.t., I(y_1, ..., y_m, ..., y_k) = 0 \\ &\sum_{i=1}^{n} Y_i = 0 \\ &Y_i = H(X_i) \end{aligned} \tag{3.14}$$

However, if we only require the uncorrelation instead of independence among $y_1, ..., y_m, ..., y_k$, the constraint $I(y_1, ..., y_m, ..., y_k) = 0$ would become $\frac{1}{n}\sum_{i=1}^{n} Y_i Y_i^T = \mathrm{I}$, where I is the identity matrix. Moreover, suppose the hash function is the kernel based hash function similar as in (3.11):

$$y = H(x) = sign(A^T K_x - b) \tag{3.15}$$

In this case, we will get a degenerated case of (3.14) as follows:

$$\begin{aligned} &\min_{H} D(Y) \\ &s.t., \tfrac{1}{n}\sum_{i=1}^{n} Y_i {Y_i}^T = \mathrm{I} \\ &\sum_{i=1}^{n} Y_i = 0 \\ &Y_i = sign(A^T K_i - b) \end{aligned} \tag{3.16}$$

where

$$\mathrm{K}_i = [K(X_i, Z_1), ..., K(X_i, Z_p)]^T$$

In the following, we will derive the analytical solutions of the above optimization problem and analyze the complexity of the method. Specifically, we will show the optimal kernel hash functions can be found elegantly by solving an eigenvector problem.

3.4.2 Derivation

Theorem 3.4.1. ***With the same relaxation as in spectral hashing by ignoring the constraint of*** $Y_i \in \{-1, 1\}^k$***, the above optimization problem is equivalent to the following:***

$$\begin{aligned} \min_{A} \quad & tr(A^T \frac{(C + C^T)}{2} A) \\ s.t. \quad & A^T G A = I \end{aligned} \tag{3.17}$$

with

$$b = A^T \bar{a}.$$

where

$$C = K_{p\times n}(D - W)K_{p\times n}^T \tag{3.18}$$

and

$$G = \frac{1}{n} K_{p\times n} K_{p\times n}^T - \bar{a}\bar{a}^T \tag{3.19}$$

Here $K_{p\times n}$ is the kernel matrix between p landmarks and n samples. More specifically the element of i-th row and j-th column for $K_{p\times n}$ is defined as

$$(K_{p\times n})_{i,j} = K(Z_i, X_j),\ i = 1, \cdots, p,\ j = 1, \cdots, n. \tag{3.20}$$

K_i is the ith column of $K_{p\times n}$, and

$$\bar{a} = (\sum_{i=1}^{n} K_i)/n. \tag{3.21}$$

$K_{p\times p}$ is the kernel matrix among p landmarks. More specifically the element of ith row and jth column for $K_{p\times p}$ is defined as

$$(K_{p\times p})_{i,j} = K(Z_i, Z_j),\ i = 1, \cdots, p,\ j = 1, \cdots, p \tag{3.22}$$

and D is a diagonal matrix with $D_{ii} = (\sum_{j=1}^{n} W_{ij} + \sum_{j=1}^{n} W_{ji})/2$, $(i = 1, \cdots, n)$.

Note here C and G are both $p \times p$ matrix.

3.4.3 Implementation

The above optimization problem in (3.17) can be further rewritten into an eigen vector problem for simpler implementation.

More specifically, suppose the SVD decomposition of G is

$$G = T_0 \Lambda_0 {T_0}^T \tag{3.23}$$

and denote $\tilde{A}$ as

$$A = T\Lambda^{-\frac{1}{2}}\tilde{A} \tag{3.24}$$

where Λ is a diagonal matrix consisting of k largest elements of Λ_0, while T is the corresponding columns of T_0.

The problem in (3.17) equals to

$$\begin{aligned} \min_{\tilde{A}} \quad & tr\left(\tilde{A}^T\Lambda^{-\frac{1}{2}}T^T\frac{(C+C^T)}{2}T\Lambda^{-\frac{1}{2}}\tilde{A}\right) \\ s.t. \quad & \tilde{A}^T\tilde{A} = I \end{aligned} \tag{3.25}$$

The solution $\tilde{A}$ is a $k \times k$ matrix, which is the k eigen vectors for matrix

$$\tilde{C} = \Lambda^{-\frac{1}{2}} T^T \frac{(C + C^T)}{2} T \Lambda^{-\frac{1}{2}}. \tag{3.26}$$

Given $\tilde{A}$, A can be obtained from equation (3.24). For a novel sample x, its mth bit code y_m can be computed as

$$y_m = H_m(x) = sign(A_m^T k_x - b_m) \tag{3.27}$$

where

$$k_x = [K(x, Z_1), ..., K(x, Z_p)]^T \tag{3.28}$$

namely, the kernel values between x and the landmark points. Equally, $y = sign(A^T k_x - b)$.

As shown in the above, kernel based hash functions $\{H_m, m = 1, ..., k\}$ can be optimized by solving an eigen vector problem on a matrix with a size around $k \times k$. (Recall (3.27), (3.26) and (3.24)). After $\{H_m, m = 1, ..., k\}$ are learned via optimization, they can directly hash new samples of any data format using properly defined kernel function, as shown in (3.27) and (3.28).

3.5 Experiments

3.5.1 Experiment Setup

We compare our GSPICA algorithm with several state-of-the-art methods, including spectral hashing (SH) [32], locality sensitive hashing(LSH)[21] and kernelized locality sensitive hashing (KLSH) [37].

All algorithms are compared using the same number of hash bits. For a fair comparison, we always use the same kernel function with the same parameters (if any), when kernels are used in methods including GSPICA and KLSH. The same number of landmark samples are used for GSPICA and KLSH. And moreover, the

number of landmark points are set close to the number of feature dimensions, so that GSPICA and SH would have almost the same indexing time. The parameter γ is chosen with a validation set, which is independent of the training set or the query set.

3.5.2 Evaluation Metrics

For evaluation, we will first compare the precision-recall curve, which is one of the most papular evaluation methods in large scale retrieval research.

We also report comparison of accuracy-time tradeoff for different hashing methods. Search accuracy is represented by recall rate, i.e., percentage of groundtruth neighbors found. However, direct comparison of machine time for each algorithm is not practical, since different implementation (e.g., different programming languages) may result in varied search times of the same method. So in our experiments, search time is represented via the number of retrieved samples in the selected buckets. By this, we try to provide an unbiased comparison of search time.

3.5.3 Experiment Results

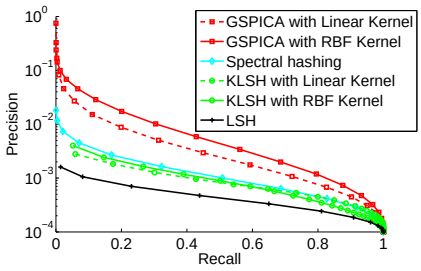
1 million web image data set

This is a data set consisting of $1M$ web images downloaded from flickr web site: www.flickr.com. 512 dimension gist features [62] are extracted for each image. RBF kernel is used for this data set. 32 hash bits are used for all the hashing methods.

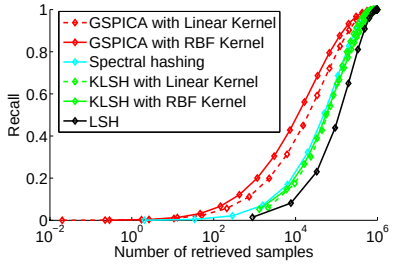
To get the precision or recall, we need to obtain groundtruth of the true nearest neighbors for each query sample. Similar to the previous works [32, 37], we establish the groundtruth by choosing the top samples (e.g., top 100 samples) found via linear scan.

In Figure 3.1, we first show the comparison of precision-recall curves. GSPICA performs significantly better than other methods, confirming its superiority on search

performance. Then we report accuracy-time comparison for different hashing methods. GSPICA also achieves a much better tradeoff between search accuracy and time. For instance, at the same search time (1 m retrieval samples), the recall of our method is several times higher than other methods.



(a) Precision-recall curve on 1M web image data



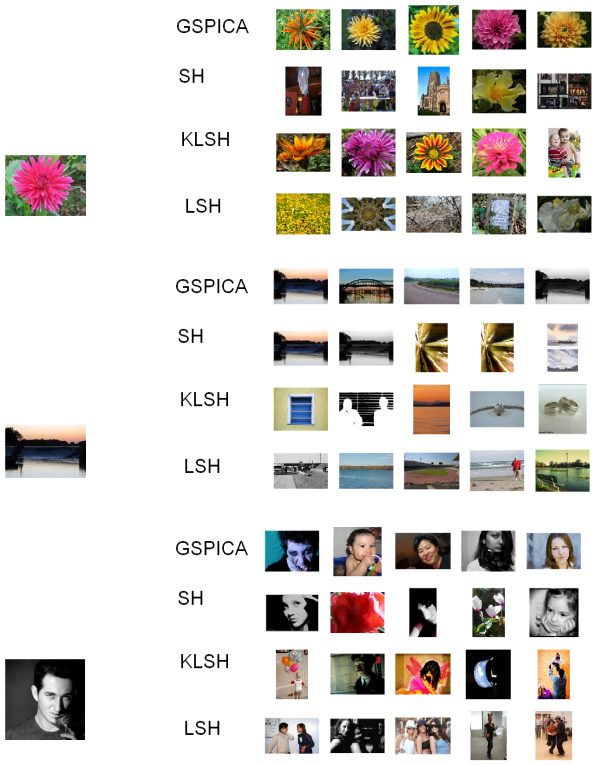
(b) Accuracy-time comparison on 1M web image data

Figure 3.1: Search results on 1M web image data set with 32 hash bits. In (a), the comparison of precision-recall curve is provided. In (b), comparison of accuracy-time curve is shown where recall represents search accuracy, and the number of retrieved samples in selected buckets represents search time. Graphs are best viewed in color.

In Figure 3.2, some example query images and the top 5 search results are also provided. The results of the proposed method are confirmed to be much better than others.

100K Photo Tourism image patch data set with multi hash tables

One key property of LSH and its variations like KLSH is the capacity to create multiple hash tables to improve the recall. Though only GSPICA with single hash table is discussed above, it can be easily extended to use multiple hash tables, for



(a) Queries (b) Top 5 search results.

Figure 3.2: On 1M web image data set, example query images and top 5 search results of GSPICA, SH, KLSH, and LSH ranked by Hamming distance with 32 hash bits are shown. Note that we are using gist features, so color information is not considered.

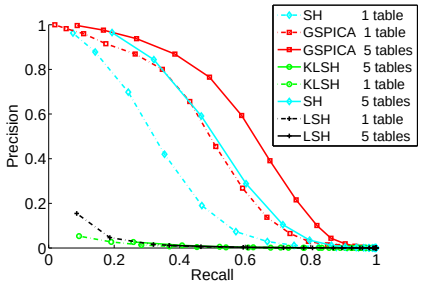
example, by using a subset of training set for each table.

We test our GSPICA method with multi hash tables on Photo Tourism image patch set [63].

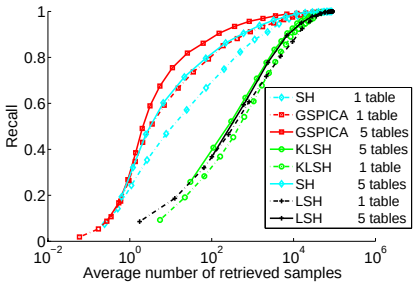
In our experiments, we use 100K patches, which are extracted from a collection of Notre Dame pictures. $1K$ patches are randomly chosen as queries, $90K$ are used as training set to learn the hashing function. For each patch, 512 dimension gist features [62] are extracted. The task is to identify the neighbors, i.e., near-duplicate patches, in the training set for each query patch. For each patch, its near-duplicates are used as the groundtruth. In our experiments, we randomly sample 10,000 training samples to compute 48 bits for each hash table. The same procedure is also done to create multi tables for spectral hashing.

The results are shown in Figure 3.3. By using multi hash tables, the recall of our GSPICA method is improved. Moreover, GSPICA works significantly better than LSH, KLSH or spectral hashing, no matter with a single hash table or multi hash tables.

We also explore how the change of experiment setting, e.g., the number of landmark samples P, or the parameter γ in Algorithm 1, would affect our results. As shown in Figure 3.4, the proposed method is quite insensitive and stable to reasonable change of P and γ. And not surprisingly, the performance increases slowly with P.

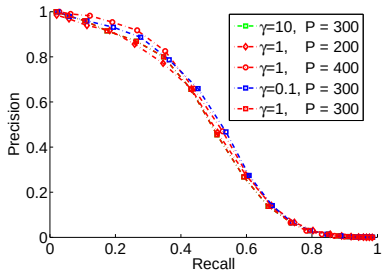


(a) Precision-recall curve on Photo Tourism data



(b) Accuracy-time comparison on Photo Tourism data

Figure 3.3: Search results on 100K Photo Tourism image patch data with 48 hash bits for each hash table. In (a), the comparison of precision-recall curve is provided. In (b), comparison of accuracy-time curve is shown where recall represents search accuracy, and the number of samples in selected buckets represents search time. Graphs are best viewed in color.



(a) Precision-recall curve for different experiment setting

Figure 3.4: Search results on 100K Photo Tourism image patch data with 48 hash bits for one hash table with different number of landmark samples P and parameter γ in Algorithm 1. Note that the green curve with square marks are covered by other curves and can not be seen. As shown, the proposed algorithm is quite stable and not sensitive to reasonable change of P and parameter γ. Graphs are best viewed in color.

Chapter 4

Algorithms of Optimal Partition for Clustering based NN Search

In this chapter, we consider clustering based indexing for nearest neighbor search. We will focus on the most popular clustering method: K-means. Following the optimal data partition framework, a clustering method more suitable for nearest neighbor search, balanced K-means, will be discussed in this Chapter.

We start with the background of conventional K-means Clustering methods.

4.1 Background of K-means Clustering

Simply speaking, given a data set $\{X_i, i = 1, ...n\}$, the objective of K-means clustering is to partition the data into K subsets $S_j, j = 1, ..., K$ to minimize the within-cluster distance, i.e., mean squared error (mse). A formal formulation of K-means is as follows:

$$\min_{S_j, j=1,...,K} \sum_{j=1}^{K} \sum_{i \in S_j} (X_i - C_j)^2 \tag{4.1}$$

where C_j is the center of S_j, or more specifically $C_j = \sum_{i \in S_j} X_i / |S_j|$. Here $|S_j|$ is the number of data points in S_j.

The above optimization problem is known to be NP-hard. So in practice heuristic algorithms have been developed to allow for a quick convergence to a local optimum. Furthermore a large number of variations have been proposed on how to initialize the starting set of centroids, or when to update the cluster centers. Algorithm 2 presents a sequential version of the algorithm that is frequently used [64]

Algorithm 2 K-means Clustering Algorithm.

Initialization by randomly assigning data points to S_j and then compute initial cluster center C_j.

Denote y_i as the cluster label for X_i, i.e., if $X_i \in S_j$, then $y_i = j$.

for iteration $t = 1, ..., T$ **do**

  for data sample $i = 1, ..., N$ **do**

    $y_i^{'} = y_i$, $y_{min} = y_i$, $d_{min} = +\infty$

    for $j = 1, ..., K$ **do**

      if $||X_i - C_j|| < d_{min}$ **then**

        $y_{min} = j, d_{min} = ||X_i - C_j||$

      end if

    end for

    if $y_i \neq y_{min}$ **then**

      $y_i = y_{min}$, Move i from $S_{y_i^{'}}$ to S_{y_i}

      update the two centers $C_{y_i^{'}}$ and C_{y_i}.

    end if

  end for

end for

4.2 Optimal Clustering for NN Search–Balanced K-Means

Suppose we are going to use K clusters for approximate nearest neighbor search. More specifically, we will return all the points in the query cluster (which the query belongs to) as candidates.

Consider the optimal partition formulated in (1.2), which is

$$\min_{\Psi} (1 - \widehat{P}_{nn}(\Psi)) + \lambda[T_{indices}(\Psi) + T_{regions}(\Psi) + n\widehat{P}_{any}(\Psi)U_{check}]$$

In the case of using clustering for NN search, $T_{indices}(\Psi) = KU_{indices}$, where $U_{indices}$ is the time cost to compute the distance between the query point and one cluster center, and hence $U_{indices} = \Theta(d)$. Moreover, $T_{regions}$ is the time to access one cluster and hence $T_{regions}(\Psi) = \Theta(1)$.

Suppose there are n_i points in cluster i, denote p_i as the probability for one random point to fall into cluster i. Similarly as discussed in Seciton 3.1, when n is large enough, the probability for a random query and a random database point both falls into the cluster i is p_i^2. Then

$$\widehat{P}_{any}(\Psi) = \sum_{i=1}^{K} p_i^2 = \frac{1}{n^2} \sum_{i=1}^{K} n_i^2$$

$\widehat{P}_{nn}(\Psi)$ is the probability for two nearest neighbor points to fall into the same cluster. In clustering based indexing methods, the clusters are obtained such that the within-cluster distances are minimized, which intuitively pushes nearest neighbors into the same cluster, and hence larger $\widehat{P}_{nn}(\Psi)$. In Section 9.3 in the Appendix, we show that, to some extend, minimizing $1 - \widehat{P}_{nn}(\Psi)$ will lead to the original cost function of K-Means, i.e., $\min \sum_{S_j, j=1,\ldots,K} \sum_{j=1}^{K} \sum_{i \in S_j} (X_i - C_j)^2$.

Following the optimal partition framework in (1.2), putting $T_{indices}(\Psi) = KU_{indices}$, $\widehat{P}_{any}(\Psi) = \sum_{i=1}^{K} p_i^2 = \frac{1}{n^2} \sum_{i=1}^{K} n_i^2$ into $\min_{\Psi} (1 - \widehat{P}_{nn}(\Psi)) + \lambda[T_{indices}(\Psi) + T_{regions}(\Psi) +$

$n\widehat{P}_{any}(\Psi)U_{check}]$, and moreover, replacing $(1-\widehat{P}_{nn}(\Psi))$ by $\sum_{j=1}^{K}\sum_{i\in S_j}(X_i-C_j)^2$, as well as omitting the constant $T_{regions}(\Psi)$, we get the optimal clustering for NN Search formulated as follows:

$$\min_{S_j,j=1,...,K}\sum_{j=1}^{K}\sum_{i\in S_j}(X_i-C_j)^2+\lambda[KU_{indices}+\frac{1}{n}\sum_{i=1}^{K}n_i^2U_{check}] \tag{4.2}$$

Note that the partition Ψ is determined by the cluster sets $S_j, j=1,...,K$.

If the number of clusters K is fixed, we can further remove the term $KU_{indices}$. Moreover, $U_{check}n$ is a constant for a given data set. Let us denote $U_{check}n\lambda$ as a new λ, then (4.2) can be simplified as the cost function :

$$\min_{S_j,j=1,...,K}\sum_{j=1}^{K}\sum_{i\in S_j}(X_i-C_j)^2+\lambda\sum_{j=1}^{K}(n_j)^2 \tag{4.3}$$

where C_j is the center of S_j and n_j is the number of data samples in S_j.

This cost function is denoted as $CF0$ as in later discussions in this section. Note that $\sum_{j=1,...,K}n_j=n$, $\sum_{j=1}^{K}(n_j)^2$ will be minimized if n_j are equal. Actually, this term can be other forms like $\sum_{j=1}^{K}(n_j)^3$, or functions like $\sum_p(n_j)^{m_p}$ or $n_j\log(n_j)$, where m_p is positive. We choose $(n_j)^2$ and $(n_j)^3$ because of their simplicity and their good performance in experiments.

We call the cost function $CF1$ when $(n_j)^3$ is used:

$$\min_{S_j,j=1,...,K}\sum_{j=1}^{K}\sum_{i\in S_j}(X_i-C_j)^2+\lambda\sum_{j=1}^{K}(n_j)^3 \tag{4.4}$$

Basically speaking, the optimal partition leads us to a balanced K-means clustering, which not only minimizes the within-cluster distance but also balances the number of points in each cluster simultaneously.

4.3 Iteration Algorithms for Balanced K-Means Clustering

We can alter Algorithm 2 to provide the iteration algorithm for balanced K-means clustering as shown in Algorithm 3. Intuitively speaking, in the case of conventional K-means a point is moved from the current cluster to a new cluster if the distance between the point and the new cluster center can be decreased. However, in balanced K-means, this decision is made based on whether the cost, which combines both the cluster distance term and balancing term, can be decreased.

Note that by setting $\lambda = 0$ the balanced K-means algorithm de-generates to conventional K-means. Moreover, it is easy to see this balanced K-means algorithm has the same time complexity as conventional K-means. And it is very easy to implement, by just modifying several lines of code in the iteration step.

4.4 Experiments

4.4.1 Data Sets

For the analysis and results presented in this paper, we have used 2 data sets, that we will briefly introduce in this section.

2D Synthetic Data Set We start with a 2 dimensional synthetic data set of 500 data points sampled at random, using a Gaussian distribution (x-axis: $\mu = 0$, $\sigma = 0.3$; y-axis: $\mu = 0$, $\sigma = 0.2$). We use this data set to analyze the trade off between the minimization of the mean squared error and the balancing the cluster assignment. It also allows us to visualize the effect of the distribution of the centroids over the feature space.

European Cities 1M Collection This collection of geo-tagged Flickr images is proposed by Avrithis et al. [65] and consists of approximately 1 million images [1].

[1]For our experiments we only used the list of distractor images, resulting in a collection of 860.500

Algorithm 3 Balanced K-means Clustering Algorithm.

Initialization by randomly assigning data points to S_j and then compute initial cluster center C_j.

Denote y_i as the cluster label for X_i, i.e., if $X_i \in S_j$, then $y_i = j$.

for iteration $t = 1, ..., T$ **do**

for data sample $i = 1, ...n$ **do**

%% If stay in the current cluster, the cost is 0

$y_i^{'} = y_i$, $y_{min} = y_i$, $cost_{min} = 0$

for $j = 1, ..., K, j \neq y_i$ **do**

%% If move to cluster j, the cost is computed as $cost_j$

$$\begin{aligned} cost_j = (||X_i - C_j||^2 - ||X_i - C_{y_i'}||^2) + \\ \lambda[(n_j + 1)^2 + (n_{y_i'} - 1)^2 - (n_j)^2 - (n_{y_i'})^2] \end{aligned} \tag{4.5}$$

or

$$\begin{aligned} cost_j = (||X_i - C_j||^2 - ||X_i - C_{y_i'}||^2) \\ +\lambda[(n_j + 1)^3 + (n_{y_i'} - 1)^3 - (n_j)^3 - (n_{y_i'})^3] \end{aligned} \tag{4.6}$$

if $cost_j < cost_{min}$ **then**

$y_{min} = j, cost_{min} = cost_j$

end if

end for

if $y_i \neq y_{min}$ **then**

%% move to cluster y_{min}

$y_i = y_{min}$, Move i from $S_{y_i'}$ to S_{y_i}

update $n_{y_i'}$ and n_{y_i}, and the two centers $C_{y_i'}$ and C_{y_i}.

end if

end for

end for

Using the SURF descriptor [66], as proposed by Bay et al. we extracted a total of approximately 350 million descriptors. The (balanced) K-means models presented in this paper are based on a random set of 1 million descriptors that have been used as the training data set. For the evaluation of the retrieval performance, we used an independent query image set and their binary relevance judgements as introduced in [67] and provided by Yahoo! through the Webscope program [2]

4.4.2 Experiments of Balanced K-means Clustering

In this section we analyze the effect of balancing the K-means quantization in terms of mean squared error and cluster balance. The mean squared error (mse), or "within cluster distance" is typically computed as

$$mse = \sum_{j=1}^{K} \sum_{i \in S_j} (X_i - C_j)^2 \tag{4.7}$$

where K indicates the number of clusters, C_j denotes the cluster centroid, S_j contains the indices of all features in cluster j. The balance (bal) of the K-means model is then measured using

$$bal = \frac{1}{K} \sum_{j=1}^{K} \frac{|n_j|^2}{(n/K)^2} \tag{4.8}$$

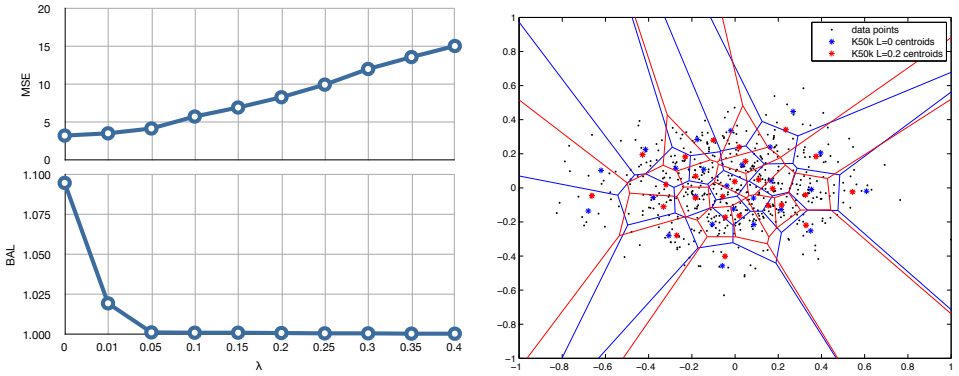
where n is the number of all features, and n_j denotes the number of features assigned to cluster j. If the balance $bal = 1$, then every cluster is perfectly balanced, and has the same number of points.

Clustering on 2D Synthetic Data Set

To see if the intuition behind the theory is correct, we start the analysis of clustering with the 2D synthetic data set with number of clusters $K = 25$. We train various

images

[2]Yahoo! Webscope Program provides a reference library of datasets for non-commercial use. See: `http://webscope.sandbox.yahoo.com/` for details.



(a) (b)

Figure 4.1: (a)Trading off *mse* and *bal* on the 2D synthetic data set. Note that when $\lambda = 0$, it is actually conventional K-means. (b)Visualization of cluster assignments on the 2D synthetic data set. Comparing K-means (blue) and balanced K-means with $\lambda = 0.1$ (red). Centroids of the balanced K-means quantization model are more oriented to the center of gravity of the point set. Graphs are best viewed with colors.

models of balanced K-means by sweeping the only model parameter λ, using number of iteration $I = 100$. Figure 4.1(a) shows how the mean squared error is traded off for a balanced cluster assignment. Note that we get conventional when $\lambda = 0$. We observe that for $\lambda > 0.05$ the balance (*bal*) quickly converges to around 1 and the within cluster distances (*mse*) are about the same as K-means. When *bal* is close to 1, we have derived an almost perfectly balanced model. Further increasing λ will increase the mean squared error (mse), which is not desirable.

Figure 4.1(b) depicts the effect of balancing the K-means quantization on the distribution of the centroids over the feature space. As expected, the centroids of the balanced K-means model are more oriented towards the dense areas in the feature space, resulting in smaller Voronoi cells for the dense areas.

Clustering of 1M SURF Features

The initial set of centroids has been set randomly from the sample features. The clusters haven been trained with $I = 100$ iterations, with cost function CF1 as in (4.4), and the size of the visual vocabulary is set to $K = 50,000$. Please note that if

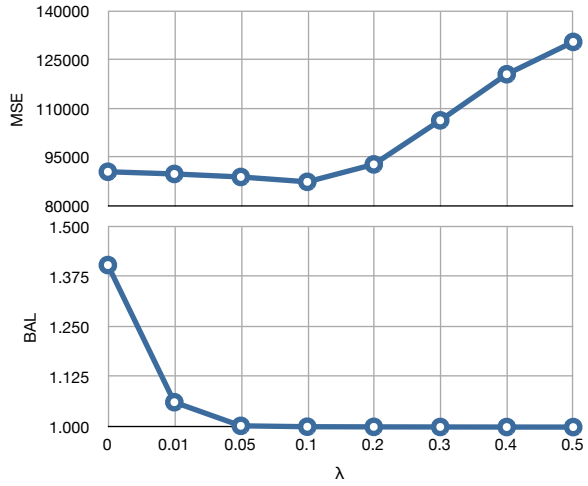


Figure 4.2: Trading off *mse* and *bal* on the 1M SURF feature collection. Note that when $\lambda = 0$, it is actually conventional K-means.

$\lambda = 0$ a regular K-means quantization model is trained.

As shown in Figure 4.2 the trade-off between the *mse* and *bal* for the models follows a similar trend as observed for the 2D synthetic example. We can maintain *mse* about the same as conventional K-means while obtain almost perfectly balanced clusters for $\lambda = 0.05 - 0.2$. Further increasing λ will increase the mean squared error (mse).

We also observe that the *mse* is initially declining when increasing λ, but for larger values of λ, one can see that the *mse* increases rapidly. Intuitively, $\lambda = 0.1$ would provide the best trade-off between *mse* and *bal*.

4.4.3 Experiments on Image Retrieval with Local Feature Quantization via Balanced K-means

To put our theory to the ultimate test, we deploy an image similarity retrieval system, similar in spirit as proposed by Sivic et al. [49]. Using the vector space model [68] we index the images using the quantized descriptors as a visual-bag of words. We use $1M$ randomly sampled SURF features to train a quantization model with $50K$ clusters. This results in an index with approximately 100K posting lists, as we multiply

the quantization id with the sign of the laplacian [66]. Finally, to improve the system performance, we deploy the approximate query evaluation method that was first introduced by Bröder et al. [69], and which is optimized to handle long queries efficiently [69]. We do not deploy a post-retrieval filter such as "RANSAC" or geometry verification methods as is commonly used for image object retrieval systems, because this would obscure the actual retrieval performance with the different quantization models, e.g., conventional K-Means and our proposed balanced K-Means.

One may argue that a simple trick to improve the balancing for K-means is to remove stop words, i.e., discard those codewords in which the number of quantized local features is larger than a threshold. However, as discussed in [70] and observed in our experiments, this will usually degrade the system accuracy. So we will use K-means without removing stop words as our baseline.

Search Accuracy

For the evaluation of the retrieval performance, we have adopted the TREC methodology and evaluation metrics [71]. In Table 4.1 the best performing runs for the two cost functions CF0 (Eq. 4.3) and CF1 (Eq. 4.4) of balanced K-means is presented, and compared against the performance using the standard K-means algorithm. We observe that both balanced K-means models ($CF0,\ \lambda = 1$ and $CF1,\ \lambda = 0.2$) outperform the standard K-means algorithm ($\lambda = 0$) on the EC1M collection. For example, we can improve the precision from 0.8 to 0.92 on the first returned image, for the balanced K-means model, using $CF1$, compared to the conventional K-means model. Some examples of query images and top 5 retrieved images with K-means and balanced K-means (using $CF1$) quantization model are shown in Figure 4.3.

Focusing on the $CF1$ cost function, Table 4.2 shows the impact on the retrieval performance when sweeping the λ parameter. On the EC1M collection we find the optimal value of $\lambda = 0.2$, choosing larger values of λ would positively impact the balance of the quantization, but negatively affect the quantization quality and as shown here the retrieval performance.

K-means configuration			
cost function	-	CF0	CF1
λ	K-means	1	0.2
retrieval performance			
relevant retrieved	547	588	**601**
p@1	0.8	0.88	**0.92**
p@5	0.784	0.792	**0.84**
p@10	0.7	0.74	**0.78**

Table 4.1: Retrieval performance on EC1M collection, comparing cost functions CF0 and CF1.

K-means configuration (using CF1)								
λ	K-means	0.01	0.05	0.1	0.2	0.3	0.4	0.5
retrieval performance								
relevant retrieved	547	584	540	590	**601**	587	563	540
p@1	0.8	0.8	0.76	**0.92**	**0.92**	0.88	0.76	0.76
p@5	0.784	0.784	0.776	0.816	**0.84**	0.808	0.76	0.776
p@10	0.7	0.716	0.716	0.728	**0.78**	0.768	0.7	0.716

Table 4.2: Retrieval performance on EC1M collection

Search Time

Proceeding with our best performing configurations, we finally present the impact of balanced K-means in terms of retrieval time in milliseconds on the EC1M collections. Note that the actual retrieval time depends on the size of the collection, the query length, e.g. the number of local features extracted from the image, among other factors. We have provided information about the query length in bottom part of Table 4.3. As can be observed in the top part of Table 4.3 balancing K-means quantization significantly and consistently reduces the average retrieval time. On the EC1M collection search time is cut with **28.8%** when compared to a conventional

Figure 4.3: Examples of queries and top 5 retrieved images, with K-means and balanced K-means quantization model. "b K-means" represents the proposed "balanced K-means".

	K-Means	Balanced K-Means
mean of search time	283.00	**204.42**
variance of search time	279.63	260.10

Table 4.3: System performance evaluation

K-means quantization with the same characteristics.

Part IV

Systems and Applications

In this part of the thesis, we will apply our proposed large scale NN search methods to real world applications, mainly focusing on visual search applications.

In Section 5, we will demonstrate the applications of bookcover search, i.e., searching a book via taking a picture of its cover. We will follow the traditional "Bag of Words" visual search paradigm, with the proposed "balanced K-Means" method for quantization. We show that "Bag of Words" with our proposed "balanced K-Means" method as discussed in Chapter 4 outperforms "Bag of Words" with conventional K-Means. And more importantly, we justify that visual search for these kinds of 2D planar objects is very mature, in fact ready for commercial applications.

Furthermore, in Section 6, we will provide the application of a mobile product search system, i.e., searching a product such as shoes, furniture, etc., by taking a picture with smart phones. We will provide the details about how to build an end-to-end mobile visual search system efficiently, to overcome the unique challenges including constrained memory, computation, and bandwidth, based on our proposed hashing methods as discussed in Chapter 3. The system will also involve other techniques like object segmentation and object boundary re-ranking to further improve the system performance. Our mobile visual system is the first system that can index large scale image data sets with local features, and allow object-level search with a response within 1 or 2 seconds over low-bandwith networks.

Chapter 5

Bookcover Search with Bag of Words

5.1 Data and System Outline

Open Library Book Covers Collection The open library book cover[1] data set consist of a collection of 4,283,246 book cover images, after removing cover images smaller than 10kb in size or a height smaller than 200 pixels. Using SURF features, we extracted 1.76 billion SURF descriptors. A random set of 1 million descriptors has been used as the training data set for clustering for Bag of Words. For the retrieval experiments, we have complemented the collection with a set of 117 query images of book covers from our personal book collection and collected the binary relevance judgements for the top 10 images retrieved by the various system configurations that we have evaluated. To collect the judgements, we deployed a blind-review pooling method as is commonly used for retrieval performance evaluation experiments [71].

Our search system is the same as described in Section 4.4.3, following the spirit of "Bag of Words" as proposed by Sivic et al [49], with our proposed balanced K-Means quantization method. Using the vector space model [68] we index the images using

[1] See `http://openlibrary.org/dev/docs/api/covers` for details.

the quantized descriptors as a visual-bag of words. We do not deploy a post-retrieval filter as is commonly used for image object retrieval systems, as this would obscure the actual retrieval performance with the different quantization models.

5.2 Experiment Results

Search Accuracy

In Table 5.1 we present the results of the balanced K-means model ($CF1,\ \lambda = 0.3$) in comparison to the standard K-means model as trained on the BOOKS collection. In this case, we optimized for early precision ($P@1$) as a typical application for this collection would be to identify the author and title of the book in the query image.

retrieval performance		
	K-means	Balanced K-means
relevant retrieved	318	**342**
p@1	0.85	**0.88**
map	0.760	**0.783**

Table 5.1: Retrieval performance on Open Library book covers collection

Search Time

Proceeding with our best performing configurations, we finally present the impact of balanced K-means in terms of retrieval time in milliseconds on BOOKS collections. As can be observed in the top part of Table 5.2 balancing K-means quantization significantly reduces the average retrieval time. On the BOOKS collection the average search time is reduced with **24.8%**.

In conclusion, our proposed balanced K-Means method significantly outperform conventional K-Means as a quantization for NN search, in terms of both search accuracy and time.

Moreover, Figure 5.1 shows a distribution of the number of descriptors per quanti-

	BoW with K-means	BoW with balanced K-means
mean of search time	715.70	**538.43**
variance of search time	313.30	287.41

Table 5.2: Search time (ms)

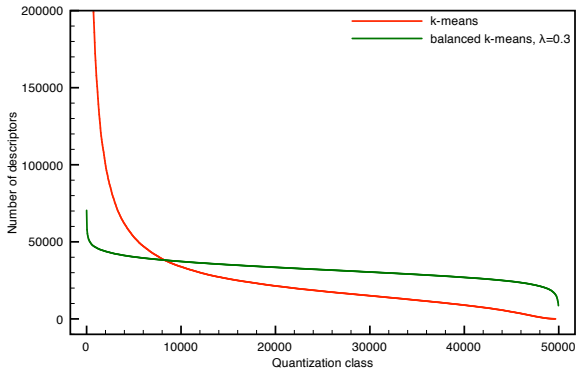


Figure 5.1: Distribution of the number of descriptors over the 50,000 quantization classes for the BOOKS collection. The figure depicts a comparison of K-means clustering vs. balanced K-means clustering ($\lambda = 0.3$). Graphs are best viewed with colors.

zation class for the (balanced) K-means models using all 1.76 billion descriptors in the BOOKS collection. The quantization classes have been sorted in descending order of number of descriptors assigned. It clearly illustrates how the regular K-means quantization model will lead to a highly unbalanced cluster assignment, which is bound to hurt both retrieval precision and time. The balanced K-means quantization model distributes the number of descriptors more evenly over the quantization classes, but is also not perfect. This is explained by the fact that we have used a relatively small sample of 1million descriptors to train the quantization models. Increasing the number of training samples will cause the distribution of descriptors to become more balanced, and eventually flat.

Finally, we show example queries and results from our book cover search in Figure 5.2 and 5.3. We can see that the query images are very challenging, taken with

variations of size, rotation, lighting, camera viewpoints, etc. For most of cases, we can get the correct search results even just for the first candidate. Actually, as shown in Table 5.1, our precision at top first candidate is 0.88, which is satisfying even for real-world commercial applications. In other words, large scale NN search is ready for image search applications like bookcover, CD/DVD cover, and other 2-D planar objects.

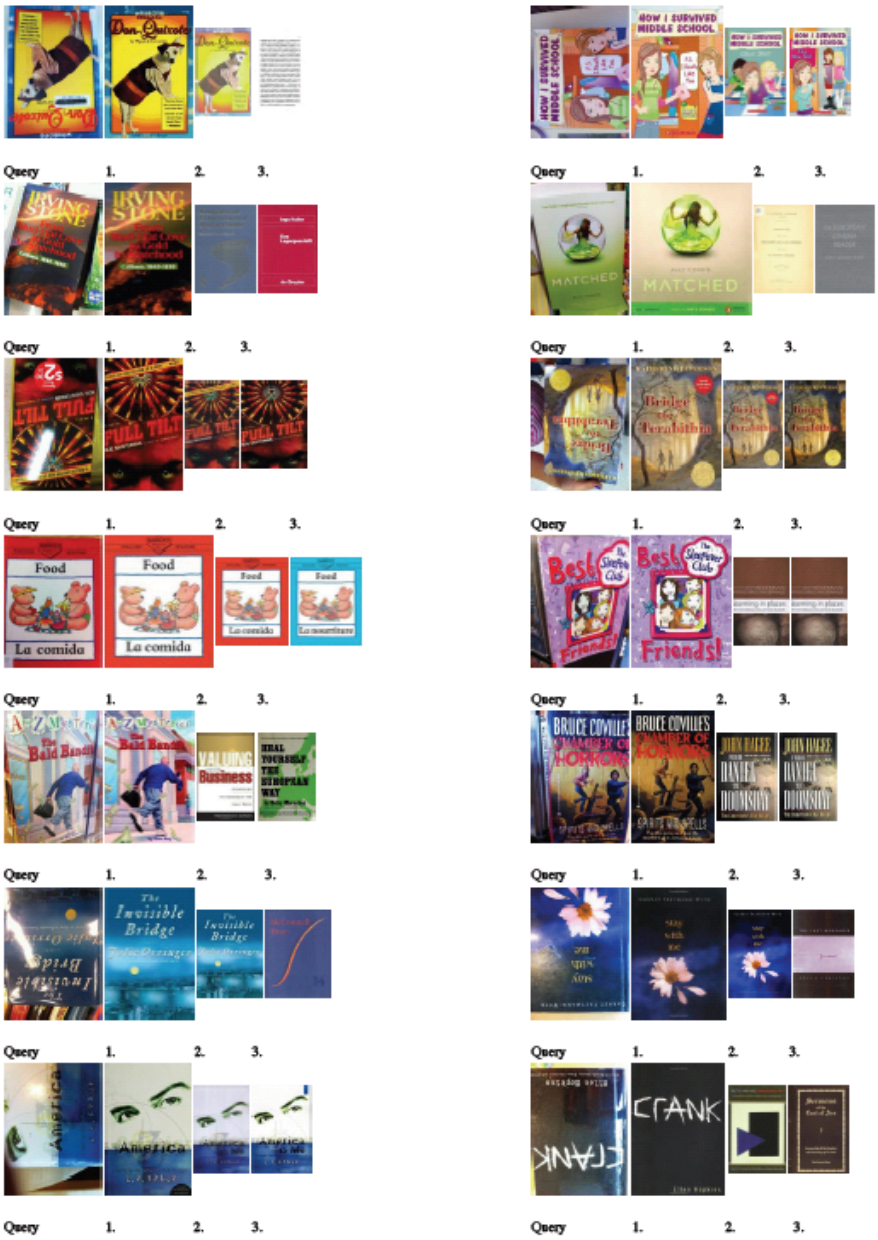


Figure 5.2: Example queries and results from our book cover search. Graphs are best viewed with colors.

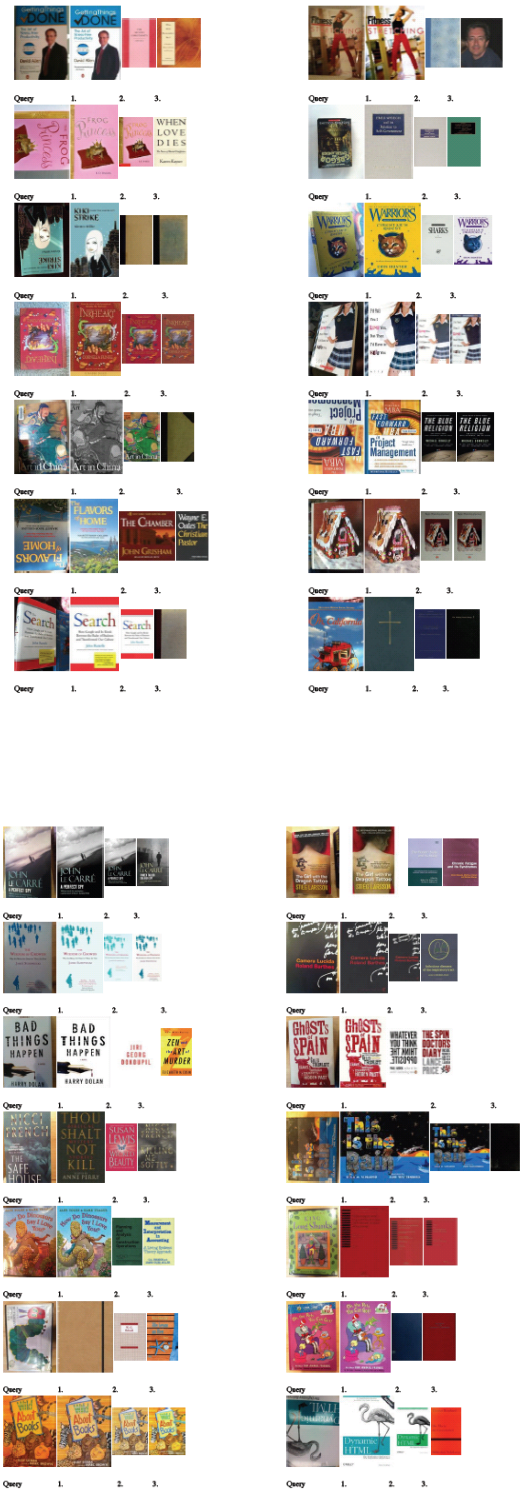


Figure 5.3: More example queries and results from our book cover search. Graphs are best viewed with colors.

Chapter 6

Mobile Product Search with Bag of Hash Bits

6.1 Introduction

The advent of smartphones provides a perfect platform for mobile visual search, in which many interesting applications have been developed [72, 73, 74, 75], such as location search, product search, augmented reality, etc. Among them mobile product search is one of the most popular, because of the commercial importance and wide user demands. There are several preliminary commercial systems on mobile product search such as Google " Goggles", Amazon "Snaptell", and Nokia "Point and Find". For mobile product search, local feature (LF) like SIFT[76] or SURF[66] is a popular choice, since global features usually cannot support object-level matching, which is crucial for product search.

Similar as conventional desktop visual search problems, mobile visual search has the same requirement of efficient indexing and fast search. However, besides that, mobile visual search has some unique challenges, e.g.,reducing the amount of data sent from the mobile to the server[1], having low computation and cheap memory on

[1]Not only because the bandwidth and speed of networks are still limited, but also because sending

the mobile side, etc.

Early mobile visual search systems only use the mobile as a capture and display device. They usually send the query image in a compressed format like JPEG to the server, and apply all other steps, like local feature extraction and searching, on the server side. As the computation capacity of smartphones becomes more and more powerful, extracting local features on the mobile client can be done very fast now, almost in real time. So recent mobile visual systems tend to extract local features on the mobile side. However, such local features need to be compressed before transmission; otherwise, sending the raw local features may cost more than sending the image. As shown in [77, 78, 79], if we compress each local feature to tens of bits and send the compressed bits, the transmission cost/time will be reduced by many times, compared to sending the JPEG images.

So in this chapter, we will only focus on the paradigm that transmits compressed local features instead of JPEG images. So the main challenge is: how can we compress local features to a few bits, while keeping the nearest neighbor search accuracy?

One straightforward approach for compressing local features is to quantize each local feature to a visual word on the mobile side, and then send the visual words to the server. However, most quantization methods with large vocabulary (which is important for good search results) such as vocabulary tree [80], are not applicable on current mobile devices, due to the limited memory and computation capacity. Similarly, some promising fast search methods like [81] are not suitable for mobile visual search either, because they usually need large memory and/or heavy computation on the mobile side.

The most popular way for mobile visual search nowadays is to compress the local features on the mobile side by some coding method, for instance CHoG [82], in which the raw features is encoded by using entropy based coding method, and can be decoded to approximately recover features at the server. The server will then quantize

---

more data will cost users more money and consume more power.

the recovered features to visual codewords and following the standard model of "bag of words" (BoW), which represents one image as collections of visual words contained in the image.

In this Chapter, we present a new mobile visual search system based on Bag of Hash Bits (BoHB) instead of conventional Bag of Words. More specifically, in the proposed BoHB approach, each local feature is encoded to tens of hash bits using similarity preserving hashing functions as discussed in Chapter 3, and each image is then represented as a bag of hash bits instead of bag of words.

First, the proposed BoHB method meets the unique requirements of mobile visual search: for instance, the mobile side only needs very little memory (i.e., storing tens of vectors with the same dimension of the local feature) and cheap computation (i.e., tens of inner products for each local feature). And moreover, the data sent from mobile to server is also very compact, about the same as the state-of-the-art mobile visual search method like CHoG [72, 82], much smaller than sending JPEG images.

Moreover, in terms of efficient searching, roughly speaking, the main difference between bag of words representation and bag of hash bits representation is how to search the database and find matched local features for each local feature contained in the query. In the bag of words model, this step is done by quantizing each local feature to a visual word, and then all local features with the same word index are considered as "matched" ones. To some extend, the hash bits of each local feature can also be viewed as the visual word index, however, the advantages of using hash bits are: 1) In "bag of words" representation, the distance between "word index" of local features is meaningless. For example, word index 4 is not "meaningfully" closer to word index 5 than word index 200, since word index are just clustering labels; however, the hamming distance between the hash bits is actually meaningful when we use similarity preserving hash functions, like PCA hashing, SPICA hashing [83] or LSH [21]. The hamming distance among hash bits is often designed to approximate the original feature distance, and hence is helpful for matching local features much

more accurately. 2) the hash bits allow us to create the indexing structure in a very flexible manner, eg., by building multiple hash tables. Hence, searching matched local features can be done by checking multiple buckets in multiple hash tables. These advantages are important in developing successful search systems that maintain high retrieval accuracy while reducing the number of the candidate results and hence the search time.

Some previous works [78, 72] have reported that hashing may not be a good choice to compress local features, and hence not suitable for mobile visual search. We believe such preliminary conclusions are drawn based on implementations that did not fully explore the potentials of the hash techniques. In this Chapter, we will present a different finding and develop a hash based system for mobile visual search that significantly outperforms prior solutions. The key ideas underlying our approaches are:

1. Use compact hashing (e.g., SPICA hashing[83] or its degenerated case: PCA hashing (PCH), instead of random hash functions like Locality Sensitive Hashing (LSH) [21].

2. Build multiple hash tables and apply hash table lookup when searching for the nearest neighbor ("matched") local features, instead of just using the linear scan over the hash bits .

3. Apply multi-probe within certain Hamming distance thresholds in each hash table, to reduce the number of needed hash tables.

4. Generate multiple hash tables through hash bit reuse, which further helps reduce the number of transmitted data to tens of bits per local feature.

The other focus of the Chapter is to develop effective and efficient features suitable for mobile product search. Boundary features are especially suitable for this purpose, since the object boundary can be represented in a very compact way, without further compression. Moreover, boundary feature is complementary with local features

that have been used in typical systems. Local features can capture unique content details of the objects very well. However, it lacks adequate descriptions about the object shape, which can actually be provided by the boundary information. There are some works on boundary feature [84, 85]. However, the combination of boundary features with the local features has not been explored for mobile visual search. To the best of our knowledge, our work is the first one to fuse local feature and boundary feature together for mobile product search. One of the main difficulties to use boundary features is to automatically segment the objects in the database and obtain the boundaries. However, for product image databases, this is usually not a major concern because of the relatively clean background in the images crawled from the sites like Amazon and Zappos. Even for the images captured and submitted by users, usually the product object is located in the center of the picture with a high contrast to the background. By applying automatic saliency cut techniques like those proposed in [86], we will demonstrate the abilities to automatically extract boundaries for product images. Finally for images with complicated backgrounds, we also provide interactive segmentation tools like Grabcut [87] on the mobile side to further improve the boundary accuracy extracted from the query images.

The outline of this Chapter is:

1. In Section 6.2, we present the overview of our mobile visual system .

2. In Section 6.3, we discuss our proposed mobile visual scheme based on Bag of Hash Bits, which significantly outperforms the-state-of-art visual search methods, including not only mobile ones [72, 82] but also conventional (desktop) search systems [81, 80].

3. In Section 6.4, we incorporate boundary reranking to improve the accuracy of mobile product search, especially at the category level.

4. In Section 6.5, we have collected a large scale challenging product search data sets with diverse categories. These product data sets will be released to facilitate

further research in this exciting research area. Moreover, we have implemented a fully functional mobile product search system including all the functions and the large product data set described in the paper.

6.2 An Overview for the Proposed Approach

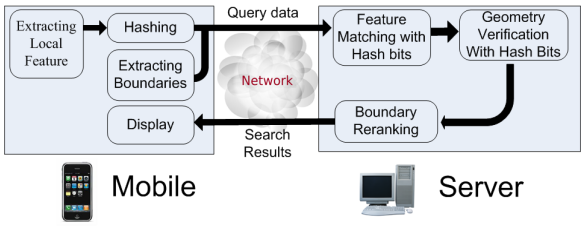


Figure 6.1: Architecture of the proposed mobile product search approach based on bag of hash bits and boundary reranking.

Figure 6.1 shows the overall workflow of the proposed system. In our work, we choose SURF as the local feature, because of its proven performance in accuracy and speed in several existing systems. For database indexing, each local feature in the database is encoded into M bits (M is tens in our case) by using similarity preserving hashing functions. Multiple hash tables are built, each of which uses a subset of the k bits.

For online searching, first, on the client (mobile) side, we compress each local feature in the query image to M bits by the same similarity preserving hashing function. We also encode the (x,y) coordinates of each local feature, using less than 10 extra bits in a way similar to [72], in order to use the spatial layout information for reranking. We then send the hash bits (M bits) and the coordinate bits (less than 10 bits) for all the local features, together with the boundary curve of the whole object to the server. Note that only one boundary curve is needed and it is usually very compact, for example, less than 200 bytes, and hence will increase the transmission cost very little.

For a local feature in the query image and a local feature in the database, if they fall into two buckets within a Hamming distance r in any hash table, the database local feature is considered a "matched" feature for the query local feature. The search process starts with finding all matched local features to each query local feature by probing all buckets within hamming distance r in all hash tables. Once the matched features are found, we collect candidate images whose "matched" local features exceed a certain threshold. We then apply an image-level geometry verification to compute the geometry similarity scores between the query image and candidate images. The geometry verification utilizes both the hash bits and the locations of local features in the image. Finally, we integrates object boundary features into our reranking process. By fusing the geometry similarity score and boundary similarity score, we rerank the candidate images and return the final top K retrieved images.

Our bag of hash bit approach requires similar bit budgets (e.g., 60-100 bits per local feature) as the state-of-art mobile visual search works like CHoG [82, 72], but a much higher search accuracy and faster search speed as shown in the experiments. For example, as shown in the experiments over a large dataset of product images, compared to CHoG, our approach can obtain about the same accuracy (in terms of recall for top K results) but tens of times speedup in the search step, or perform significantly better in both search accuracy (30% improvement) and search time (several times speedup). The BoHB method also (significantly) outperforms popular conventional visual search systems, such as bag of words via vocabulary tree [80], or product quantization [81]. Moreover, hashing based approach is very fast to compute (only requiring tens of inner products), and very easy to implement on mobile devices, and applicable for different types of local features, hashing algorithms and image databases.

Moreover, from the boundary curve, we extract a boundary feature called "central distance", which is translation, scale, and rotation invariant to boundary changes. By incorporating the boundary feature into the reranking step, the overall search perfor-

mance is further improved, especially in retrieving products of the same categories (shoe, furniture, etc).

6.3 Mobile Visual Search with Bag of Hash Bits

6.3.1 Hash Local Features into Bits

We will apply linear hashing methods for encoding each local feature to hash bits. More specifically, for any local feature x, where x is a 128 dimension vector when using SURF in our case, we can get one hash bit for x by

$$b = sign(v^T x - t) \tag{6.1}$$

Here v is a projection vector in the feature space, and t is a threshold scalar. Though v can be a projection vector from any linear similarity preserving hashing method, we have found randomly generated v like in LSH [21] performs quite poorly, because a small number of hash tables is utilized, due to the memory limit. On the other hand, projections from compact hashing like SPICA hashing [83] or its degenerated case, PCA hashing, will provide much better search performance. So in the rest of this paper, we assume v comes from SPICA [83] or PCA projections. Moreover, as well known in hashing research area, balancing hash bit will usually improve the search performance, so we choose t as the median value such that half of each bit are +1, and the other are -1.

Following the considerations in [72], constrained by transmission speed, we limit the number of hash bits for each local feature to less than 100.

Matching Local Features with Multiple Hash Tables

One popular technique to achieve fast sublinear search rather than linear scan, is to utilize multiple hash tables.

For one hash bit/function, denote p_{NN} as the probability of one query point (local feature in our case) and one of its neighbor points to have the same bit value, and p_{any}

as one query point and a random database point to have the same bit value. When using similarity persevering hash functions, p_{NN} will be larger than p_{any}. Suppose we use k bits in one table. For the simplicity of our discussion, we assume p_{NN} is the same for every bit, and so is and p_{any}. And moreover, bits are independent. (Violation of these assumptions will make the discussion much more complex but lead to similar conclusions.) Denote $P_{NN}(k, r)$ as the probability of one query point and one of its nearest neighbors to fall into two buckets whose hamming distance is no larger than r, where $r \ll k$. We have:

$$P_{NN}(k, r) = \sum_{i=0,...r} C_k^i (p_{NN})^{k-i} (1 - p_{NN})^i$$

Similarly, denote $P_{any}(k, r)$ as the probability of one query point and a random database point to fall into two buckets whose hamming distance is no larger to r. Similarly,

$$P_{any}(k, r) = \sum_{i=0,...r} C_k^i (p_{any})^{k-i} (1 - p_{any})^i$$

Note that $p_{any} < p_{NN}$ and $r \ll k$, so $P_{any}(k, r)$ will decrease much faster than $P_{NN}(k, r)$ when k increases. Note that the expected number of returned nearest neighbors is $N_{NN} P_{NN}(k, r)$ where N_{NN} is the number of total nearest neighbor points for the query point. Moreover, the expected number of all returned points is $N P_{any}(k, r)$, where N is the number of points in the database. So the precision of nearest neighbors in returned points is

$$\frac{N_{NN} P_{NN}(k, r)}{N P_{any}(k, r)},$$

If k becomes larger, $\frac{N_{NN} * P_{NN}(k,r)}{N * P_{any}(k,r)}$ will becomes larger too, and hence the precision of finding matched local features will be high. However, when k is large, $N_{NN} * P_{NN}(k, r)$ itself may be too small, and hence we cannot obtain enough nearest neighbors. So we can increase the number of hash tables L to improve the chance of obtaining nearest neighbors, while still keep the high precision.

If we only check one bucket (i.e., $r = 0$) in each hash table, L often has to be very large like hundreds to get a reasonable recall for finding nearest neighbors.

One popular solution to reduce the number of tables is to probe multiple buckets in one hash table. For example, if we set r as 2 or 3, and check all buckets within hamming distance r in each hash table, the number of needed tables can then be reduced significantly, to about ten for example, because $P_{NN}(k, r)$ will increase when r becomes larger, for a fixed k.

In practice, hamming distance r is usually a small number, e.g., ≤ 3, the number of bits k in each hash table is about $20 - 40$, and L is $5 - 20$.

Multiple Hash Tables Proliferation

We adopt the idea of using multi hashing table to find nearest neighbor local features for the query ones. However, the number of bits to build L hash tables is Lk, usually hundreds or thousands bits. For mobile visual search, we only have a budget of tens of bits per local feature. So instead of sending hundreds/thousands of hash bits for each local feature over the mobile network, we only send tens of bits per local feature, but generate multiple hash tables by reusing the bits. More specifically, suppose we have M bits for each local feature, we will build each hash table by randomly choosing a subset of k bits from M bits. We have observed if M is more than 2 or 3 times larger than k, constructing tables by reusing bits does not largely affect the search result, especially when the number of hash tables is not large, e.g., about ten in our case. We can thus construct multiple tables without increasing the total amount of bits needed for each local feature.

6.3.2 Geometry Verification with Hash Bits

Suppose one database image contains several matched local features, one would like to check if these matches are geometrically consistent, i.e., whether a valid geometric transformation can be established between the feature positions in the query image and the positions in the database image. The existence of a consistent geometric transformation between two images strongly indicates that the image indeed contains similar object(s). Considering the popular geometric verification method, RANSAC,

is too slow, we apply a fast geometry verification method based on length ratio, inspired by [72]. The method is about tens/hundreds of times faster than the RANSAC algorithm and was shown to perform well on real-world databases. Intuitively, it estimates the portion of matched features between query and reference images that share a consistent scale change. The higher value this is, the higher score the candidate reference image receives.

Length Ratio Similarity with Hash Bits

More specifically, for a the query image I_q and a database image I_{db}, suppose they have $m \geq 2$ matched local features. For two features p and q in I_q, suppose their matched features are p' and q' in I_{db} respectively. Denote x_p and y_p as the (x, y) coordinate of local feature p in the image. The length ratio is defined as the ratio of distance between p and q and distance between p' and q': $\frac{||(x_{p'}-x_{q'})^2+(y_{p'}-y_{q'})^2||^{\frac{1}{2}}}{||(x_p-x_q)^2+(y_p-y_q)^2||^{\frac{1}{2}}}$. There are C_m^2 ratio values between I_q and I_{db}, since there are m matched local features, and C_m^2 matched feature pairs. Each ratio value i will be quantized to some bin a in the ratio value histogram. Suppose, ration value i is computed with local feature p, p' and q, q' , then i's vote to bin j is computed as

$$v_{i,j} = \alpha^{(d_{pp'}+d_{qq'})}, \text{if } j == a,$$

$$v_{i,j} = 0, \text{ otherwise.}$$

Here α is a constant which is smaller than but close to 1, and $d_{pp'}$ and $d_{qq'}$ are the hamming distances between p, p' and q, q' respectively. Then the maximum value in the histogram is taken as the geometry similarity score between two images, or more specifically,

$$S_g(I_q, I_{db}) = \max_j \sum_{i=1,\ldots,C_m^2} v_{i,j}. \tag{6.2}$$

However, one observation is: the similarity score as above results in an approximately quadratic growth versus the number of matched points. In our experiment, we have found this put too much weight on the number of matches while ignoring the quality of matches themselves. One distracter image with complex background

can, e.g., have higher score than the correct one just because it has more matches to the background clutter of query image. We thus divide the maximum value in the histogram by the total number of matches to further improve the robustness to noisy matches.

6.4 Boundary Reranking

To obtain boundary features, we need to extract the boundaries first. Since product objects are usually the most salient regions in images, we applied the SaliencyCut algorithm to extract the boundaries automatically [86].

We also implement the interactive segmentation Grabcut [87] on the mobile device, to further improve the segmentation accuracy for the query.

Some example results of the automatic SaliencyCut are shown in figure 6.2.

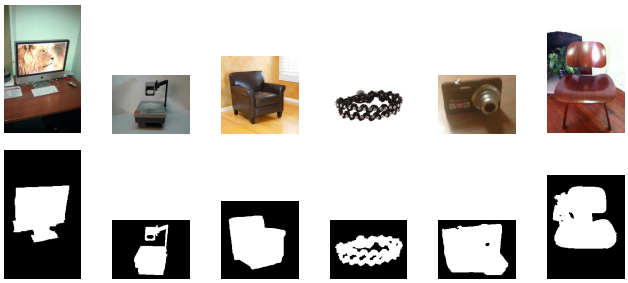


Figure 6.2: Examples of automatic SaliencyCut results. The first 4 are segmented correctly, while the last two do not find perfect cut due to shadow, lighting, and distracting background. Pictures are best viewed in color.

There are different boundary features proposed [84, 88], however, in our system, we utilize a very straightforward boundary feature, central distance, because of its simplicity and robustness.

Before feature extraction, we first smooth the boundary by using a moving average filter to eliminate noises on the boundary. Then the feature is expressed by the distances between sampled points $\mathbf{p}(n)$ along the shape boundary and the shape

center $\mathbf{c} = (c_x, c_y)$:

$$\mathbf{D}(n) = \frac{\|\mathbf{p}(n) - \mathbf{c}\|_2}{\max_n \|\mathbf{p}(n) - \mathbf{c}\|_2}, n = 1, 2, \ldots, N$$

The points $\mathbf{p}(n)$ are sampled with a fixed length along the boundary. For a desired feature length N, the sampling step can be set to L/N, where L is the boundary length (the total number of pixels in the boundary curve). It is easy to see the central distance feature is invariant to translation. In addition, the feature $\mathbf{D}$ is normalized by its maximum element, and hence will be scale invariant.

Moreover, to make it rotation invariant and start point independent, the discrete Fourier transform (DFT) is applied:

$$\mathbf{F}(n) = |f[\mathbf{D}(n)]|, n = 1, 2, \ldots, N \tag{6.3}$$

where $f[\cdot]$ is the discrete Fourier transform and can be computed efficiently by fast Fourier transform (FFT) in linear time. Any circular shift of the feature vector $\mathbf{D}(n)$ only affects the phases of its frequency signal, while $\mathbf{F}(n)$, the magnitude of frequency signal, keeps unchanged. In sum, $\mathbf{F}(n)$ will be translation, scale, and rotation invariant to boundary changes.

For one query image I_q and one database image I_{db}, their boundary similarity $S_b(I_q, I_{db})$ is defined as $S_b(I_q, I_{db}) = e^{-||F_q - F_{db}||}$, where F_q and F_{db} are the frequency magnitude for I_q and I_{db}, as defined in Equation (6.3).

We fuse the two similarity, i.e., geometry similarity S_g as computed in (6.2) and boundary similarity S_b, with a linear combination: $S(I_q, I_{db}) = S_g(I_q, I_{db}) + \lambda S_b(I_q, I_{db})$. The combine similarity s are used to rerank the top results.

6.5 Experiments

We have provided some video demos at `http://www.ee.columbia.edu/~jh2700`, to demonstrate the end-to-end mobile product search system that has been operational over actual mobile networks.

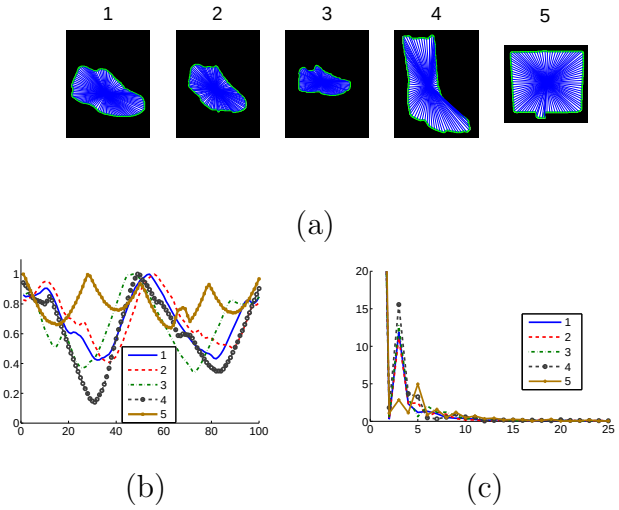


Figure 6.3: (a) 5 example object boundaries. (b) The central distance features for 5 samples. (c) The central distance features in frequency domain. The boundaries are extracted by the automatic SaliencyCut method. As shown in (b) and (c), the similarity among the central distance features or their frequencies capture the similarity among boundaries quite well. Graphs are best viewed in color.

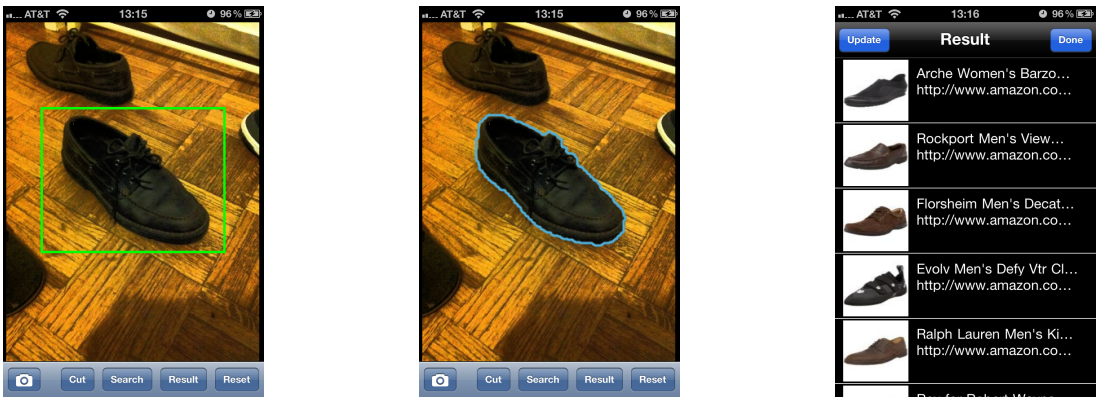


Figure 6.4: Example user interfaces on iphone. Users may select the whole image, or a subwindow as query input, which can further be refined by automatic SalientCut or interactive object segmentation tools like Grabcut. Matched products are shown on the right.

Two snapshots from our video demo are shown in Figure 6.4 to illustrates the UI of our system [2]. Our system supports search with the whole image or a subwindow as the query.

Our system will have the same speed in most of the steps as other mobile visual search system, e.g, local feature extraction on mobile (< 1s), transmitting query data through network (usually < 1s). The step of compressing each local feature to hash bits is very fast, and can almost be ignored, compared to other steps.

The main difference between our approach and other mobile product search system is the searching step: searching with bag of hash bits v.s. searching with bag of words. To further justify our approach, we conduct experiments to provide quantize analysis on this searching step. Since the whole searching step only involves computation on the server and does not involve mobile, our experiments are conducted on a workstation. We will report both search accuracy and search time in our experiments. Note that the time of other steps (e.g., feature extraction, transmission) is independent of the database size. The database size will only affect this searching step. So the searching time represents the scalability of the system.

6.5.1 Data Sets

The existing product (or object) image sets, e.g., "Stanford Mobile Visual Data Set" [89], or "UKBench" object data set [80], usually have a small scale like thousands of images. or contain mainly 2D objects like CD/DVD/Book covers.

For mobile product search applications, the most meaningful data set may actually come from online shopping companies like Amazon, Ebay, etc. So we have collected two large scale product sets from Amazon, Zappos, and Ebay, with $300k-400k$ images of product objects, from diverse categories like shoes, clothes, groceries, electrical

[2] We have implemented the UI in iPhone platform with Objective C. Moreover, the implementation of extracting SURF features and some image processing steps are accomplished by adapting OpenCV into iPhone platform

devices, etc. These two product image sets are the largest and most challenging benchmark product datasets to date.

Product Data Set 1

The first data set includes 360K product images crawled from Amazon.com. It contains about 15 categories of products. For this data set we have created a query set of 135 product images. Each query image will have one groundtruth image in the database, which contains exactly the same product as the query, but will differ in object sizes, orientations, backgrounds, lightings, camera viewpoints, etc.

Product Data Set 2

The second data set contains 400K product images collected from multiple sources, such as Ebay.com, Zappos.com, Amazon.com, etc, and with hundreds of categories. For this data set we have collected a query set of 205 product images.

The image sizes for both sets are usually 200-400 by 200-400. And each image usually contains about 50-500 SURF features. No subwindow is provided for each query image. And moreover, the boundaries for product objects in both database and queries are extracted by automatic SaliencyCut.

Some examples of query images and their groundtruths from our data sets are shown in Figure 6.5.

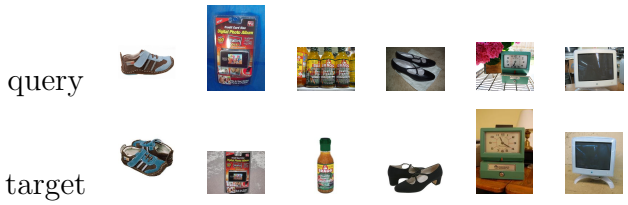


Figure 6.5: Some example queries and their groundtruths in Product Dataset 2. The first row are queries, and the second row are corresponding groundtruths.

6.5.2 Performance of Bag of Hash Bits

First, we compare our "bag of hash bits" approach to CHoG [82, 72] approach, which is the state-of-the-art in mobile visual search area, on Product Data Set 1.

In CHoG implementation, each CHoG feature is about 80 bits. The CHoG features will be quantized with vocabulary tree [80], which is the state-of-the-art method to quantize local features. Since the quantization step for CHoG approach is done on the server side, using large scale codebook is possible. In our implementation, we use a codebook with 1M codewords.

For our "Bag of Hash Bits", we use 80 bits for each local feature by using SPICA hashing [83] or LSH [21]. And then we build 12 hash tables, each of which is constructed by randomly choosing 32 bits from the 80 bits. We will check buckets within hamming distance r in each hash table. r is set to 1-3 in our experiments.

As shown in Figure 6.6, with bits generated from SPICA hashing and hamming distance $r = 2$, our approach of "Bag of Hash Bits" on surf features can obtain about the same recall as BoW on "CHoG" features, but the search speed is improved by orders of magnitude. And if we set $r = 3$, we can improve the accuracy significantly, e.g., about 30% improvement for the recall at top 10 retrieved results, while be several times faster.

However, if we use bits from LSH, the search time of our BoHB approach will be increased by tens/hundreds of times. The main reason is: LSH bits are randomly generated and hence are quite redundant. That actually explains why previous hashing based systems (usually utilizing LSH bits) perform quite poorly.

We also compare our "bag of hash bits" approach to other popular visual search systems, such as BoW via vocabulary tree [80] or product quantization [81], with SURF features, even though they may not be applicable to mobile visual search scenarios.

For vocabulary tree implementation, we follow the paradigm in vocabulary tree in [80]. The codebook size on SURF features is up to 1M, which is the largest codebook

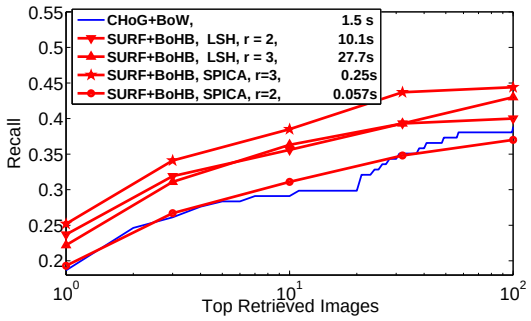


Figure 6.6: Performance comparison between CHoG based approach and our BoHB approach on Product DataSet 1. For BoHB, we have tried bits from two different hash methods, SPICA hashing [83] and LSH. Search time of different approaches are included in the legends. Graph is best viewed in color.

in the current literatures. As shown in Figure 6.7, with bits generated from PCA hashing (PCH), and hamming distance $r = 1$ or 2, the proposed BoHB approach can achieve 2-3 fold increase in terms of recall. For product quantization approach, we utilize the product quantization method to match top K nearest neighbor local features for each query local feature, and then rerank the candidate database images. In our current implementation, $K = 1000$, and reranking is based on the counts of matched local features in each candidate images. [3] We can see that product quantization method achieves relatively good accuracy (slightly lower than our top results), but (much) slower search speed. However, if LSH or ITQ [40] hash bits are utilized, we will see the search time of the BoHB approach will be quite long. This confirms the merit to combine compact hashing of low redundance (like PCA hashing or SPICA hashing) with the proposed Bags of Hash Bits idea.

[3]We have tried other K and different reranking methods e.g., build BoW histograms with matched local features and compute cosine similarity between histograms, or sum the distances between query local features and matched local features in each candidate image. We choose the current K and ranking method, because they provides the best accuracy.

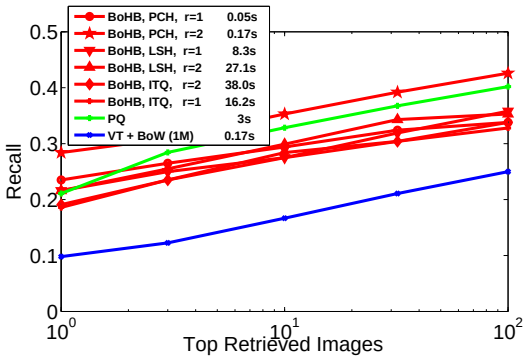


Figure 6.7: Performance comparison between the proposed BoHB approach and other visual search methods including BoW (with Vocabulary Tree) approach and product quantization approach, on Product DataSet 2. For BoHB, we have tried bits from different hash methods, such ass PCA hashing (PCH), ITQ hashing [40] and LSH. Search time of different approaches are included in the legends. Graph is best viewed in color.

6.5.3 Performance of Boundary Reranking

If we repeat our BoHB approach with SPICA or PCH bits and r = 3 as in Figure 6.6, but include boundary reranking, the recall of top 100 results will have about relatively 8%-10% improvement. The improvement caused by boundary reranking seems good, but not exciting. However, that is mainly due to the strict definition of our groundtruth. For each query, we only define one groundtruth, which is exactly the same product as the query. Some examples of search results without/with boundary reranking (BR) are shown in Figure 6.8. As shown, our boundary reranking is very helpful to filter out noisy results, and improve the search quality, especially increase the number of relevant products (in the sense of the same category). But this is not represented in recall, under the strict definition of our groundtruth. However, the advantage of boundary reranking will be very helpful in practice, especially when the query product does not exist in the database, and hence relevant/similar products will be the best we can return.

Figure 6.8: Example queries and their top 5 search results, without or with boundary reranking (BR). Pictures are best viewed in color.

Part V

Additional Discussions on Nearest Neighbor Search

Chapter 7

Theories on the Difficulty of Nearest Neighbor Search

7.1 Introduction

Following (1.1) and (1.2), we have discussed different kinds of approximate Nearest Neighbor (NN) search techniques. However, no matter our proposed methods in Part II and Part III, or previous related methods discussed in Section 1.3, the performance of all these techniques depends heavily on the data set characteristics.

For example, we have discussed a lot about how to optimize the partition Ψ in (1.1), i.e.,

$$\begin{aligned} \min_{\Psi} T(\Psi) = T_{indices}(\Psi) + T_{regions}(\Psi) + n\widehat{P}_{any}(\Psi)U_{check} \\ s.t., \\ 1 - \widehat{P}_{nn}(\Psi) \leq \delta \end{aligned} \tag{7.1}$$

especially, how to optimize $\widehat{P}_{any}(\Psi)$ and $\widehat{P}_{nn}(\Psi)$ by choosing optimal parameters for random partitions or designing optimal partition functions for learning based partitions.

However, $\widehat{P}_{any}(\Psi)$ and $\widehat{P}_{nn}(\Psi)$, and hence the optimal value for (1.1), not only depend on the partition methods, but also the data set itself.

For example, could one data set be just too difficult for NN search? And hence, no NN search methods can achieve meaningful performance, i.e., can get a better time complexity than linear scan? Or is it possible that one data set A is more difficult than another data set B, and hence the possible optimal value for (1.1) on A is thus larger than B? If one data set A is too difficult or more difficult than B, why is that? or more specifically, how is the difficulty of a data set related to the characteristics, such as dimension, sparsity or metric definition, etc.?

In sum, there are three fundamental questions here:

1. How to measure the difficulty of a given data set for NN search (independent of any NN search methods)?

2. How will the data characteristics, such as dimension, sparsity or metric definition on the dataset, related to the "difficulty" of the data set?

3. How will the performance of NN search methods, for example, the optimal value in (1.1), be affected by the difficulty of a given data set?

We will introduce a new concrete measure *Relative Contrast* for the difficulty of nearest neighbor search problem in a given data set (independent of indexing methods). Unlike previous works that only provide asymptotic discussions for one or two data properties, we derive an explicitly computable function to estimate relative contrast in non-asymptotic case. It for the first time enables us to analyze how the difficulty of nearest neighbor search is affected by different data properties simultaneously, such as dimensionality, sparsity, database size, along with the norm of L_p distance metric , for a given data set, as shown in Sec. 7.2. [1] In Section 7.3, we will discuss how the difficulty of the data set will affect the performance of NN search methods. We

[1] As a comparison, most of the existing works on analyzing NN search have focused on the effect of one data property: dimensionality, that too in an asymptotic sense, showing that NN search will

provide a theoretical analysis and experiment justification, in a non-asymptotic quantitative sense, on how the dataset difficulty and hence data properties affect the time complexity of LSH, a special case of (1.1). We provide the proofs of the theorems of this chapter in Section 9.4 in the Appendix. Moreover, we also show many previous works on NN search analysis are special cases of ours, in Section 9.4.

7.2 The Difficulty of Nearest Neighbor Search for a Given Data Set

7.2.1 Relative Contrast (C_r) – Measure the Difficulty of Nearest Neighbor Search

Suppose we are given a data set X containing n d-dim points, $X = \{X_i, i = 1, \ldots, n\}$, where $X_i \in R^d$ are i.i.d samples from a random vector x with an unknown distribution $p(x)$. Denote x^j as the j-th dimension of x. Suppose a query q is also a random vector in R^d, following the same distribution of $p(x)$. Denote q^j as the j-th dimension of q.

Further, let $D(\cdot, \cdot)$ be the distance function for the d-dimensional data. We focus on L_p distances in this analysis: $D(x, q) = (\sum_j |x^j - q^j|^p)^{1/p}$.

Suppose $D^q_{min} = \min_{i=1,\ldots n} D(X_i, q)$ is the distance to the nearest database sample[2], and $D^q_{mean} = E_x[D(x, q)]$ is the expected distance of a random database sample from the query q. We define the relative contrast for the data set X for a query q as :

be meaningless when the number of dimensions goes to infinity ([90, 91, 92]). First, non-asymptotic analysis has not been discussed, i.e., when the number of dimensions is finite. Moreover, the effect of other crucial properties has not been studied, for instance, the *sparsity* of data vectors. Since in many applications, high-dimensional vectors tend to be sparse, it is important to study the two data properties e.g., dimensionality and sparsity together, along with other factors such as database size and distance metric.

[2]Without loss of generality, we assume that the query is distinct from the database samples, i.e., $D^q_{min} \neq 0$.

$C_r^q = \frac{D_{mean}^q}{D_{min}^q}$. It is a very intuitive measure of *separability* of the nearest neighbor of q from the rest of the database points. Now, taking expectations with respect to queries, the relative contrast for the dataset X is given as,

$$C_r = \frac{E_q[D_{mean}^q]}{E_q[D_{min}^q]} = \frac{D_{mean}}{D_{min}} \tag{7.2}$$

Intuitively, C_r captures the notion of difficulty of NN search in X. Smaller the C_r, more difficult the search. If C_r is close to 1, then on average a query q will have almost the same distance to its nearest neighbor as that to a random point in X. This will imply that NN search in database X is not very meaningful.

In the following sections, we derive relative contrast as a function of various important data characteristics.

7.2.2 Estimation of Relative Contrast

Suppose x^j and q^j are the j-th dimensions of vectors x and q. Let's define,

$$R_j = E_q[|x^j - q^j|^p], R = \sum_{j=1}^{d} R_j. \tag{7.3}$$

Both R_j and R are random variables (because x^j is a random variable). Suppose each R_j has finite mean and variance denoted as $\mu_j = E[R_j], \quad \sigma_j^2 = var[R_j]$. Then, the mean and variance of R are given as,

$$\mu = \sum_{j=1}^{d} \mu_j, \quad \sigma^2 \leq \sum_{j=1}^{d} \sigma_j^2.$$

Here, if the dimensions are independent then $\sigma^2 = \sum_j \sigma_j^2$. Without the loss of generality, we can scale the data such that the new mean μ' is 1. The variance of the scaled data, called normalized variance will be:

$$\sigma'^2 = \frac{\sigma^2}{\mu^2}. \tag{7.4}$$

The normalized variance gives the spread of the distances from query to random points in the database with the mean distance fixed at 1. If the spread is small, it is

harder to separate the nearest neighbor from the rest of the points. Next, we estimate the relative contrast for a given dataset as follows.

Theorem 7.2.1. *If $\{R_j, j=1,...d\}$ are independent and satisfy Lindeberg's condition*[3]*, the relative contrast can be approximated as,*

$$C_r = \frac{D_{mean}}{D_{min}} \approx \frac{1}{[1 + \phi^{-1}(\frac{1}{n} + \phi(\frac{-1}{\sigma'}))\sigma']^{\frac{1}{p}}} \tag{7.5}$$

where ϕ is the c.d.f of standard Gaussian, n is the number of database samples, σ' is normalized standard deviation, and p is the distance metric norm.

Range of C_r: Note that when n is large enough $\phi(\frac{-1}{\sigma'}) \leq \frac{1}{n} + \phi(\frac{-1}{\sigma'}) \leq \frac{1}{2}$, so $0 \leq 1 + \phi^{-1}(\frac{1}{n} + \phi(\frac{-1}{\sigma'}))\sigma' \leq 1$ and hence C_r is always ≥ 1. And moreover, when $\sigma' \to 0$, $\phi(\frac{-1}{\sigma'}) \to 0$, and $C_r \to 1$.

Generalization 1: The concept of relative contrast can be extended easily to the k-nearest neighbor setting by defining $C_r^k = \frac{D_{mean}}{D_{knn}}$, where D_{knn} is the expected distance to the k-th nearest neighbor. Using $\bar{n}(D_{knn}^p) \approx \bar{n}(R_{knn}) = k$, and following similar arguments as above, one can easily show that

$$C_r^k = \frac{D_{mean}}{D_{knn}} \approx \frac{1}{[1 + \phi^{-1}(\frac{k}{n} + \phi(\frac{-1}{\sigma'}))\sigma']^{\frac{1}{p}}} \tag{7.6}$$

7.2.3 What Data Properties Affect the Relative Contrast and How?

7.2.3.1 Effect of normalized variance σ' on Relative Contrast C_r

From (7.5), relative contrast is a function of database size n, normalized variance σ'^2, and distance metric norm p. Here, σ' is a function of data characteristics such as dimensionality and sparsity. Figure 7.1 shows how C_r changes with σ' according

[3]Lindeberg's condition is a sufficient condition for central limit theorem to be applicable even when variables are not identically distributed. Intuitively speaking, the Linderberg condition guarantees that no R_j dominates R.

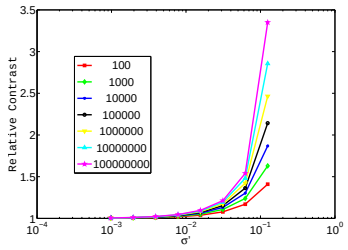


Figure 7.1: Change in relative contrast with respect to normalized data variance σ' as in (7.5). The database size n varies from 100 to $100M$ and $p = 1$. Graph is best viewed with color.

to (7.5) when n is varied from 100 to $100M$, and $0 < \sigma' < 0.2$ (Note that σ' is usually very small for high dimensional data, e.g., far smaller than 0.1). It is clear that smaller σ' leads to smaller relative contrast, i.e., more difficult nearest neighbor search.

In the above plots, p is fixed to be 1 but other values yield similar results. An interesting thing to note is that as the database size n increases, relative contrast increases. In other words, nearest neighbor search is more meaningful for a larger database.[4] However, this effect is not very pronounced for smaller values of σ'.

7.2.3.2 Data Properties vs σ'

Since we already know the relationship between C_r and σ', by analyzing how data properties affect σ', we will find out how data properties affect C_r, i.e., the difficulty of NN search. Though many data properties can be studied, in this section we focus on sparsity (a very important property in many domains involving, say, text, images and videos), together with other properties like data dimension and metric.

Suppose, the j^{th} dimensions of vectors x and q are distributed the same way as a random variable V_j. But each dimension has only s_j probability of having a non-zero value where $0 < s_j \leq 1$. Denote $m_{j,p}$ as the p-th moment of $|V_j|$, and $m'_{j,p}$ as the p-th

[4]It should not be confused with computational ease since computationally search costs more in larger databases.

moment of $|V_{j1} - V_{j2}|$, where V_{j1} and V_{j2} are independently distributed as V_j.

Theorem 7.2.2. *If dimensions are independent,*
$\sigma'^2 = \frac{\sum_{j=1}^{d} s_j^2 m'_{j,2p} + 2(1-s_j)s_j m_{j,2p} - \mu_j^2}{(\sum_{j=1}^{d} \mu_j)^2}$
where $\mu_j = s_j^2 m'_{j,p} + 2(1-s_j)s_j m_{j,p}$. *Moreover, if dimensions are i.i.d.,*

$$\sigma' = \frac{1}{d^{1/2}} \sqrt{\frac{s[(m'_{2p} - 2m_{2p})s + 2m_{2p}]}{s^2[(m'_p - 2m_p)s + 2m_p]^2} - 1}. \tag{7.7}$$

For some distributions, m_p and m'_p have a closed form representation. For example, if every dimension follows uniform distribution $U(0,1)$, then p^{th} moment is quite easy in this case: $m_p = \frac{1}{(p+1)}, m'_p = \frac{2}{p+1} - \frac{2}{p+2}$. However, if m_p and m'_p do not have a closed form representation, one can always generate samples according to the distribution, and estimate m_p and m'_p empirically.

7.2.3.3 Data Properties vs Relative Contrast C_r

We now summarize how different database properties and distance metric affect relative contrast.

Data Dimensionality (d): From (7.7), it is easy to see that larger d will lead to smaller σ'. Moreover, from (7.5), smaller σ' implies smaller relative contrast C_r, making NN search less meaningful. This indicates the well-known phenomenon of distance concentration in high dimensional spaces. However, when dimensions are not independent, thankfully, the rate at which distances start concentrating slows down.

Data Sparsity (s): From (7.7), we can see that $\sigma' = \frac{1}{d^{1/2}} \sqrt{\frac{(m'_{2p} - 2m_{2p}) + \frac{2m_{2p}}{s}}{[(m'_p - 2m_p)s + 2m_p]^2} - 1}$. If $m'_p - 2m_p \geq 0$, when s becomes smaller (i.e., data vectors have fewer non-zero elements), σ' gets larger, and so does the relative contrast. Another interesting case is when $p \to 0_+$, i.e., L_0 or zero-one distance. In this case, $m_p = m'_p = 1$, and from (7.7) $\sigma' = \frac{1}{d^{1/2}} \sqrt{\frac{(1-s)^2}{1-(1-s)^2}}$, which increases monotonically as s decreases. However, for general cases, it is not easy to theoretically prove how σ' will change when s gets smaller. But in experiments, we have always found that smaller s will lead to

larger σ'. In other words, when data vectors become more sparse, NN search becomes easier. That raises another interesting question: What is the effective dimensionality of sparse vectors? One may be tempted to use $d \cdot s$ as the intrinsic dimensionality. But as we will show in the experimental section, this is generally not the case and relative contrast provides an empirical approach to finding intrinsic dimensionality of high-dimensional sparse vectors.

Database Size (n): From (7.5), keeping σ' fixed, C_r increases monotonically with n. Hence, NN search is more meaningful in larger databases. Actually, when $n \to \infty$, irrespective of σ', $1+\phi^{-1}(\frac{1}{n}+\phi(\frac{-1}{\sigma'}))\sigma' \to 0$, and $C_r \to \infty$. Thus, when the database size is large enough, one doesn't need to worry about the meaningfulness of NN search irrespective of the dimensionality. However, unfortunately when dimensionality is high, C_r increases very slowly with n, making the gains not very pronounced in practice. This is the same phenomenon noticed in Fig. 7.1 for small values of σ'.

Distance Metric Norm (p): Since p appears in both (7.5) and (7.7), it makes analysis of relative contrast with respect to p not as straightforward. In the special case when data vectors are dense (i.e., $s = 1$), and each dimension is i.i.d with uniform distribution, one can show that smaller p leads to bigger contrast.

7.2.4 Validation of Relative Contrast

To verify the form of relative contrast derived in Sec. 7.2.2, we conducted experiments with both synthetic and real-world datasets, which are summarized below.

7.2.4.1 Synthetic Data

We generated synthetic data by assuming each dimension to be i.i.d from uniform distribution $U[0, 1]$. Fig. 7.2 compares the predicted (theoretical) relative contrast with the empirical one. The solid curves show the predicted contrast computed using (7.5), where the normalized variance σ' is estimated using (7.7). The dotted curves show the empirical contrast, directly computed according to the definition in (7.2)

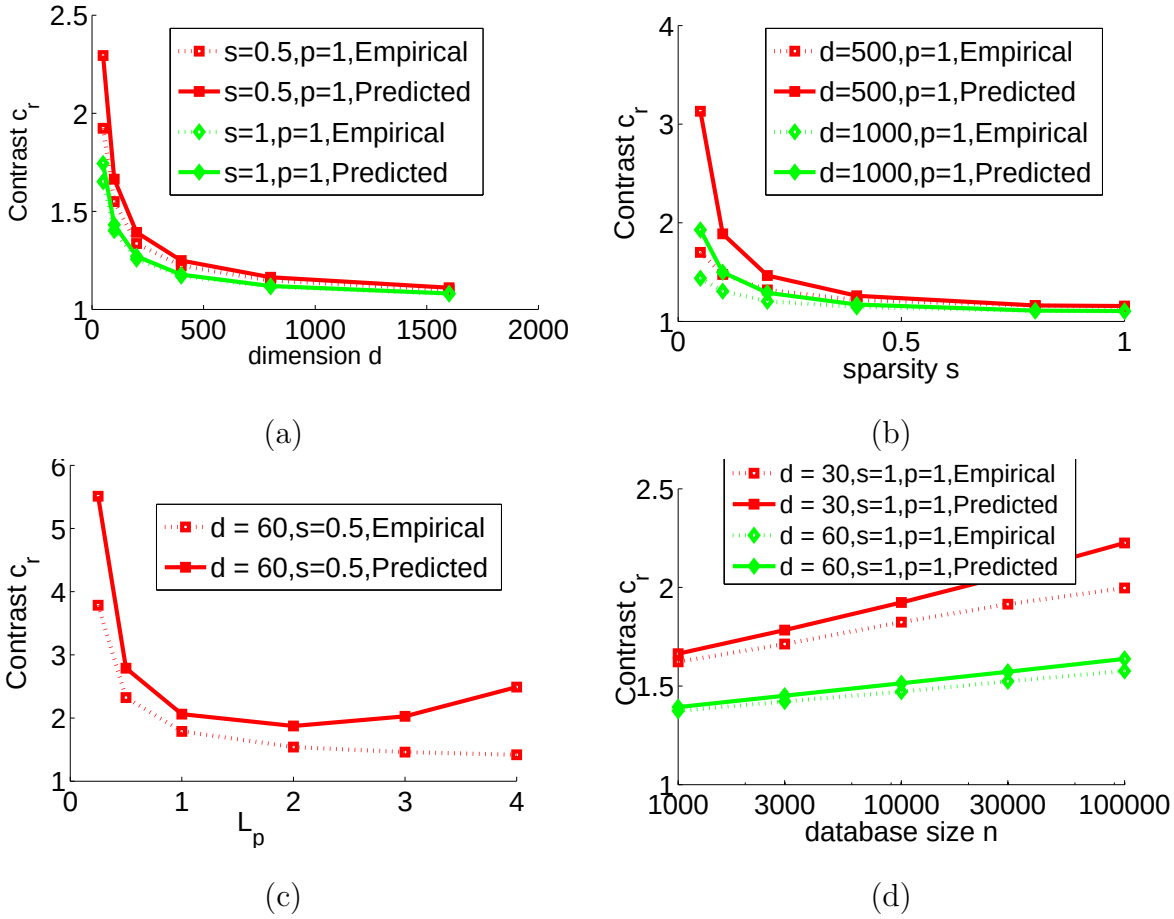


Figure 7.2: Experiments with synthetic data on how relative contrast changes with different database characteristics. Graphs are best viewed with color.

from the data by averaging the results over one hundred queries. For most of the cases, the predicted and empirical contrasts have similar values.

Fig. 7.2 (a) confirms that as dimensionality increases, relative contrast decreases, thus making the nearest neighbor search harder. Moreover, except for very small d, the prediction is close to the empirical contrast verifying the theory. It is not surprising that predictions are not very accurate for small d since the central limit theorem(CLT) is not applicable in that case. It is interesting to note that (7.5) also predicts the rate at which contrast changes with d, unlike the previous works ([90, 91]) which only show that NN search becomes impossible when dimensionality

goes to infinity.

Fig. 7.2 (b) shows how data sparsity affects the contrast for two different choices of d. The main observation is that as s increases (denser vectors), contrast decreases, making nearest neighbor search harder. In other words, lesser the number of non-zero dimensions for a fixed d, easier the search. In fact, the search remains well-behaved even in high-dimensional datasets if data is sparse. The prediction is quite accurate in comparison to the empirical one except when $s.d$ is small and hence CLT does not apply any more. As a note of caution, one should not regard $s.d$ as the intrinsic dimensionality of the data, since a dataset with dense vectors of dimension $s.d$ usually has different contrast than the d-dim s-sparse data set.

The effects of two other characteristics i.e., L_p distance metric for different p and database size n are shown in Figs. 7.2 (c) and (d), respectively. The effect of these parameters on relative contrast is milder than that of d and s. For large d, the contrast drops quickly and it becomes hard to visualize the effects of p and n. So, here we show these plots for smaller values of d. From Fig. 7.2 (c) it is clear that for norms less than 1, contrast is the highest (Note that we have an approximation for $p > 1$ in Theorem 7.2.1, which causes the bias of predicted C_r for $p = 3, 4$). This observation matches the conclusion from ([91]) for dense vectors. Fig. 7.2 (d) shows that as the database size increases, it becomes more meaningful to do nearest neighbor search. But as the dimensionality is increased (from 30 to 60 in the plot), the rate of increase of contrast with n decreases. For very high dimensional data, the effect of n is very small.

7.2.4.2 Real-world Data

Next, we conducted experiments with four real-world datasets commonly used in computer vision applications: *sift, gist, color* and *image*. The details of these sets are given in Table 7.1. The sift and gist sets contain 128-dim and 384-dim vectors, which are mostly dense. On the other hand, both color and image datasets are very high

Table 7.1: Description of the real-world datasets. n - database size, d - dimensionality, s - sparsity (fraction of nonzero dimensions), d_e - effective dimensionality containing 85% of data variance.

	n	d	s	d_e
gist	95000	384	1	71
sift	95000	128	0.89	40
color (histograms)	95000	1382	0.027	22
image (bag-of-words)	95000	10000	0.024	71

dimensional as well as sparse. Color data set contains color histogram of images while the image data set contains bag-of-words representation of local features in images.

While deriving the form of relative contrast in Sec. 7.2.2, we assumed that dimensions were independent. However, this assumption may not be true for real-world data. One way to adress this problem would be to assume that the dimensions become independent after embedding the data in an appropriate low-dimensional space. In these experiments, we define effective dimensionality d_e as the number of dimensions necessary to preserve 85% variance of the data[5]. The effective dimensionality for different datasets is shown in Table 7.1. Table 7.2 compares the empirical and predicted relative contrasts for different datasets. Since our theory is based on the law of large numbers, the prediction is more accurate on image and gist data sets as their effective dimensions are large enough. For the color data, d_e is too small (just 22) and hence the prediction of relative contrast shows more bias for this set.

One interesting outcome of these experiments is that our analysis provides an alternative way of finding intrinsic dimensionality of the data which can be further used by various nearest neighbor search methods. The traditional method of finding intrinsic dimensionality using data variance suffers from the assumption of linearity of

[5]For large databases, one can use a small subset to estimate the covariance matrix.

Table 7.2: Experiments with four real-world datasets. Here, predicted contrast is computed using the effective dimensionality containing 85% of data variance.

	p=1	p=2
gist empirical contrast	1.83	1.78
gist predicted contrast	1.62	1.87
sift empirical contrast	4.78	4.23
sift predicted contrast	2.03	3.94
color emprical contrast	3.19	4.81
color predicted contrast	2.78	8.10
image empirical contrast	1.90	1.66
image predicted contrast	1.62	1.87

the low-dimensional space and the arbitrary choice of threshold on variance. On the other hand, nonlinear methods are computationally prohibitive for large datasets. In the relative contrast based method, for a given dataset, one can sweep over different values of d' where $0 < d' < d$, and find the one which gives the least discrepancy between the predicted and empirical contrasts averaged over different p. For large datasets, one can use a smaller sample and a few queries to estimate the empirical contrast. Using this procedure, the intrinsic dimensionality for the four datasets turns out to be: sift - 41, gist - 75, color - 41, image - 70. For the two sparse datasets (color and image), it indicates the dimensionality of equivalent low-dimensional *dense* vector space. It is interesting to note that intrinsic dimensionality is not equal to $d \cdot s$ for the two sparse datasets as discussed before. For image dataset, it is much smaller than $d \cdot s$ indicating high correlations in non-zero entries of the data vectors.

7.3 How Will the Difficulty Affect the Performance NN Search Methods

7.3.1 How Will the Difficulty Affect LSH

In this section, we will discuss how the difficulty measure, i.e., relative contrast, will affect the complexity of LSH. We will also see how relative contrast will affect the parameter design for LSH.

We will mainly discuss p-stable locality sensitive hashing ([22]) with a hash function $h(x) = \lfloor \frac{v^T x+b}{w} \rfloor$, where v is a vector with entries sampled from a p-stable distribution, and b is uniformly distributed as $U[0, w]$.

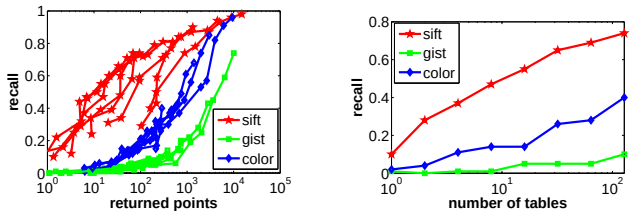
We provide the following theorems to show how relative contrast (C_r) affects the complexity of LSH.

Theorem 7.3.1. *To find the exact nearest neighbor with probability $1-\delta$, the returning candidate points, the time complexity, space complexity and the number of hash tables needed are monotonically decreasing with C_r.*

Thus, among the datasets of same size, to get the same recall of the true nearest neighbor, the dataset with higher relative contrast C_r will have better upper bound on the time and space complexity, the number of candidates for reranking, and the number of hash tables, or in one word, be easier for approximate NN search with LSH.

To verify the effect of relative contrast on LSH, we conducted experiments on three real-world datasets.

In Fig. 7.3, performance of LSH for L_1 distance (i.e., $p = 1$) is given on three datasets: sift, gist and color. From Table 7.2, for $p = 1$, C_r for the three datasets is in this order: sift(4.78) > color(3.19) > gist (1.83). From Fig. 7.3 (a), we can see that for several settings of number of bits and number of tables, the number of returned points needed to get the same nearest neighbor recall for the three sets follows sift <

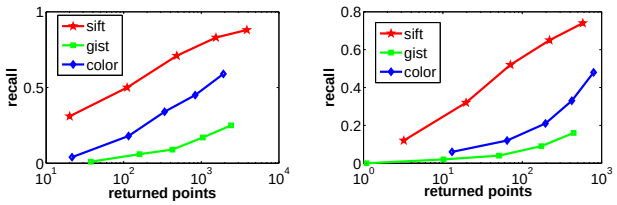


(a) (b)

Figure 7.3: Performance of LSH on three datasets: sift, gist, and color. (a) Recall of the nearest neighbor. Each curve represents different number of bits, e.g., $k = 12, 16, ...40$. Each marker on the curve represents different number of hash tables l, e.g., $l = 1, 2, ...128$. (b) Recall of the nearest neighbor for different number of hash tables for $k = 32$. Graphs are best viewed with color.

color < gist, as predicted by Theorem 7.3.1. Moreover, from Fig. 7.3 (b), the number of hash tables needed to get the same recall follows sift < color < gist. We have tried experiments with $k = 12, 16..., 40$ and observe the same trend, but only show results for $k = 32$ due to space limit.

The above experiments used the typical framework of hash table lookup. Another popular way to retrieve neighbors in code space is via hamming ranking. When using a k-bit code, points that are within hamming distance r to the query are returned as candidates. In Figure 7.4, we show the recall of nearest neighbor for two different values of k. Similar to the case of hash table lookup experiments, the number of returned points needed to get the same recall follows sift < color < gist. This follows the same order as suggested by relative contrast. The interesting thing is that color has much higher dimensionality than gist, but its sparsity helps in achieving better relative contrast and hence better search performance.



(a) k =20 (b) k = 28

Figure 7.4: Recall vs the number of returned points when using hamming ranking. Number of bits $k = 20$ for (a) and $k = 28$ for (b). Graphs are best viewed with color.

Part VI

Conclusions

Chapter 8

Summary and Future Works

8.1 Summary of Contributions

In this thesis, we have studied the problem of large scale (approximate) nearest neighbor search. Our contributions are three folds: theories, algorithms, and applications.

1. Theories

   (a) Data partition framework and optimal data partition criteria for NN search:
   We have unified NN search methods into the data partition framework, and furthermore proposed the general formulation of optimal data partition for NN search, which can be applied to explain and improve most existing NN search methods.

   (b) Theoretical bound for indexing/search with random partitions:
   With the optimal data partition formulation, we have developed new bounds for locality sensitive hashing, and also other search methods via random partitions, like nearest neighbor preferred hashing, or random trees/forests, and so on. Based on the theoretical derivation of the bounds, we also have developed new methods to design parameters for search with random partitions, for one particular data set.

(c) The Difficulty of NN search:

We have studied some fundamental theoretical problems for nearest neighbor search, for example, how to measure the difficulty of a given data set for nearest neighbor search, and how data properties will affect the difficulty of nearest neighbor search. This fundamental theory is not only helpful to understand the nearest neighbor problems, but also useful to design the nearest neighbor search algorithms.

2. Algorithms

(a) Indexing/search with learning based partition:

Following the optimal data partition formulation, we demonstrated that optimal partition functions should have two criteria simultaneously: p-reserve nearest neighbors, i.e., guarantee that nearest neighbors fall into the same or close buckets, and balance regions/buckets, i.e., make sure each bucket contains about the same number of points. We have designed joint optimization methods to satisfy/tradeoff the above two criteria simultaneously, for various indexing methods, like indexing via hashing, or quantization.

3. Applications – Visual Search Engines via Large Scale Nearest Neighbor Search

(a) Book cover search via bag of words:

We have developed image search engine on millions of book cover data set, with our proposed balanced K-means to search/match local features. We show that our approach is promising enough even for practical applications, with 88% recall for the first return result.

(b) Mobile visual search via bag of hash bits:

We have developed an end-to-end mobile product search system on iphone platform, which is based on our Bag of Hash Bits indexing/search techniques. Our system can search on the 100M local feature database within

0.01 second on a single desktop. We also applied interactive/automatic segmentation to obtain product boundaries to further improve the product search accuracy.

8.2 Future Works

There are still lots of interesting topics to be studied in the area of large scale nearest neighbor search. Some examples are listed as follows:

1. Develop theories for NN search with learning based partitions:
   To some extent, most current NN search methods with learning based partitions (including ours) are heuristic. Can we develop theories (like those for random partitions), to obtain some theoretical bounds for learning based partitions, and moreover, design practical methods to approximate the theoretical bounds? This direction will deepen our understanding of NN search problem with learning based partitions, and may also discover more practical NN search methods.

2. Apply our large scale NN search methods to data in other domains besides multimedia:
   Currently our applications focus on multimedia data sets. It will be interesting to see how large scale NN search will perform in other domains, especially those domains where "semantic gap" is less an issue, namely, nearest neighbors based in low level feature retrieval meet the true "application" or "semantic" needs.. We may find more practical and meaningful applications in those domains.

3. Parallelize our indexing/search algorithms to deal with web scale data:
   Currently, all our algorithms and experiments are done on a single machine. To deal with web scale data, like web multimedia, thousands of machines or even more will be necessary. How to parallelize our methods in this case? How to

design new systems based on our methods for this distributed environment? All these problems are crucial to scale up our methods/systems for the web scale data.

4. Extend our works to large scale machine learning and data mining area:
   Large scale nearest neighbor search is also a crucial step for many algorithms in large scale machine learning and data mining (e.g., many graph based methods or non-parametric methods). We would like to apply and generalize our theories and algorithms to help understand and solve problems from large scale machine learning/data mining etc.

Part VII

Bibliography

Bibliography

[1] N. Bhatia and et.al. Survey of nearest neighbor techniques. *arXiv preprint arXiv:1007.0085*, 2010.

[2] K.L. Clarkson. Nearest-neighbor searching and metric space dimensions. *Nearest-Neighbor Methods for Learning and Vision: Theory and Practice*, pages 15–59, 2006.

[3] A. Andoni. *Nearest neighbor search: the old, the new, and the impossible*. PhD thesis, Citeseer, 2009.

[4] L. Cayton and S. Dasgupta. A learning framework for nearest neighbor search. *Advances in Neural Information Processing Systems*, 20, 2007.

[5] P.M. Riegger. Literature survey on nearest neighbor search and search in graphs. 2011.

[6] K. Yamaguchi, E. Tanaka, and Y. Shirasaka. Data structure and algorithm for nearest neighbour search. *IEIC Technical Report (Institute of Electronics, Information and Communication Engineers)*, 98(562):73–80, 1999.

[7] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 1975.

[8] J.H. Friedman, J.L. Bentley, and R.A. Finkel. An algorithm for finding best

matches in logarithmic expected time. *ACM Transactions on Mathematical Software (TOMS)*, 3(3):209–226, 1977.

[9] S.M. Omohundro. *Five balltree construction algorithms*. International Computer Science Institute, 1989.

[10] J.K. Uhlmann. Satisfying general proximity/similarity queries with metric trees. *Information Processing Letters*, 40(4):175–179, 1991.

[11] T. Liu, A.W. Moore, A. Gray, and K. Yang. An investigation of practical approximate nearest neighbor algorithms. *Advances in neural information processing systems*, 17:825–832, 2004.

[12] A. Beygelzimer, S. Kakade, and J. Langford. Cover trees for nearest neighbor. In *Proceedings of the 23rd international conference on Machine learning*, pages 97–104. ACM, 2006.

[13] S. Dasgupta and Y. Freund. Random projection trees and low dimensional manifolds. In *Proceedings of the 40th annual ACM symposium on Theory of computing*, pages 537–546. ACM, 2008.

[14] M. Muja and D.G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *International Conference on Computer Vision Theory and Applications (VISSAPPâ 09), pages 331â*, volume 340. Citeseer, 2009.

[15] N. Verma, S. Kpotufe, and S. Dasgupta. Which spatial partition trees are adaptive to intrinsic dimension? In *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence*, pages 565–574. AUAI Press, 2009.

[16] M. McCartin-Lim, A. McGregor, and R. Wang. Approximate principal direction trees. *arXiv preprint arXiv:1206.4668*, 2012.

[17] P.N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, pages 311–321. Society for Industrial and Applied Mathematics, 1993.

[18] T. Bozkaya and M. Ozsoyoglu. Distance-based indexing for high-dimensional metric spaces. In *ACM SIGMOD Record*, pages 357–368. ACM, 1997.

[19] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proceedings of the 25th International Conference on Very Large Data Bases*, pages 518–529. Morgan Kaufmann Publishers Inc., 1999.

[20] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*, pages 604–613. ACM, 1998.

[21] M.S. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*, pages 380–388. ACM, 2002.

[22] M. Datar, N. Immorlica, P. Indyk, and V.S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*, pages 253–262. ACM, 2004.

[23] Rina Panigrahy. Entropy based nearest neighbor search in high dimensions. In *In Proceedings of SODA*, 2006.

[24] Zhe Wang Moses Charikar Kai Li Qin Lv, William Josephson. Multiprobe lsh: Efficient indexing for high-dimensional similarity search. In *In Proceedings of VLDB*, 2007.

[25] A. Joly and O. Buisson. A posteriori multi-probe locality sensitive hashing. In *Proceedings of ACM MM*, 2008.

[26] K. Grauman and T. Darrell. Pyramid match hashing: sub-linear time indexing over partial correspondences. In *CVPR*, 2007.

[27] Prateek Jain, Brian Kulis, and Kristen Grauman. Fast image search for learned metrics. In *CVPR*, 2008.

[28] Hervé Jégou Loïc Paulevé and Laurent Amsaleg. Locality sensitive hashing: A comparison of hash function types and querying mechanisms. In *Pattern Recogn. Lett.*, 2010.

[29] D. Gorisse, M. Cord, and F. Precioso. Locality-sensitive hashing for chi2 distance. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 34(2):402–409, 2012.

[30] R. Salakhutdinov and G. Hinton. Semantic hashing. In *In Proceedings of SIGIR*, 2007.

[31] R. Salakhutdinov and G. Hinton. Learning a nonlinear embedding by preserving class neighbourhood structure. In *AI and Statistics*, 2007.

[32] Y. Weiss, A. Torralba, and R. Fergus. Spectral hashing. In *NIPS*, 2008.

[33] R. Fergus A. Torralba and Y.Weiss. Small codes and large image databases for recognition. In *CVPR*, 2008.

[34] Shumeet Baluja and Michele Covell. Learning to hash: forgiving hash functions and applications. 2008.

[35] Brian Kulis and Trevor Darrell. Learning to hash with binary reconstructive embeddings. In *NIPS*, 2009.

[36] M. Raginsky and S. Lazebnik. Locality sensitive binary codes from shift-invariant kernels. In *NIPS*, 2009.

[37] Brian Kulis and Kristen Grauman. Kernelized locality-sensitive hashing for scalable image search. In *ICCV*, 2009.

[38] Mohammad Norouzi and David Fleet. Minimal loss hashing for compact binary codes. In *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, ICML '11, 2011.

[39] J. Wang, S. Kumar, and S.F. Chang. Sequential projection learning for hashing with compact codes. In *Proceedings of International Conference on Machine Learning*, 2010.

[40] Y. Gong and S. Lazebnik. Iterative quantization: A procrustean approach to learning binary codes. In *Computer Vision and Pattern Recognition (CVPR), 2011 IEEE Conference on*, pages 817–824. IEEE, 2011.

[41] A. Bergamo, L. Torresani, and A. Fitzgibbon. Picodes: Learning a compact code for novel-category recognition. Citeseer, 2011.

[42] W. Liu, J. Wang, S. Kumar, and S.F. Chang. Hashing with graphs. In *Proceedings of the 28th International Conference on Machine Learning*, pages 1–8, 2011.

[43] W. Liu, J. Wang, R. Ji, Y.G. Jiang, and S.F. Chang. Supervised hashing with kernels. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*, pages 2074–2081. IEEE, 2012.

[44] W. Kong, W.J. Li, and M. Guo. Manhattan hashing for large-scale image retrieval. In *Proceedings of the 35th international ACM SIGIR conference on Research and development in information retrieval*, pages 45–54. ACM, 2012.

[45] W. Kong and W.J. Li. Double-bit quantization for hashing. In *Twenty-Sixth AAAI Conference on Artificial Intelligence*, 2012.

[46] P. Jain, B. Kulis, and K. Grauman. Fast image search for learned metrics. In *In Proceedings of CVPR*, 2008.

[47] Y. Matsushita and T. Wada. Principal component hashing: An accelerated approximate nearest neighbor search. *Advances in Image and Video Technology*, pages 374–385, 2009.

[48] T. Shibata and O. Yamaguchi. Local fisher discriminant component hashing for fast nearest neighbor classification. *Structural, Syntactic, and Statistical Pattern Recognition*, pages 339–349, 2008.

[49] J. Sivic and A. Zisserman. Video google: A text retrieval approach to object matching in videos. In *ICCV*, 2003.

[50] H. Jégou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 2011.

[51] S. Lazebnik and M. Raginsky. Supervised learning of quantizer codebooks by information loss minimization. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 2009.

[52] T. Tuytelaars and C. Schmid. Vector quantizing feature space with a regular lattice. In *ICCV*, 2007.

[53] R. Motwani, A. Naor, and R. Panigrahi. Lower bounds on locality sensitive hashing. In *Proceedings of the twenty-second annual symposium on Computational geometry*, 2006.

[54] Aapo Hyvarinen and Erkki Oja. Independent component analysis: Algorithms and applications. 2000.

[55] Aapo Hyvarinen. Fast and robust fixed-point algorithms for independent component analysis. 1999.

[56] Karsten M. Borgwardt Nino Shervashidze. On the convergence of ica algorithms with symmetric orthogonalization. 2009.

[57] Karsten M. Borgwardt Nino Shervashidze. Fast subtree kernels on graphs. In *NIPS*, 2009.

[58] C. Schmid S. Lazebnik and J. Ponce. Beyond bags of features, spatial pyramid matching for recognizing natural scene categories. In *CVPR*, 2006.

[59] Matthias Seeger Christopher Williams. Using the nystrom method to speed up kernel machines. In *NIPS*, 2001.

[60] Ali Rahimi and Benjamin Recht. Random features for large-scale kernel machines. In *NIPS*, 2007.

[61] Wei Liu Junfeng He and Shih-Fu Chang. Scalable similarity search with optimized kernel hashing. In *KDD*, 2010.

[62] A. Oliva and A. Torralba. Modeling the shape of the scene: A holistic representation of the spatial envelope. *International Journal of Computer Vision*, 42(3):145–175, 2001.

[63] S. Winder and M. Brown. Learning local image descriptors. *CVPR*, 2007.

[64] R.O. Duda, P.E. Hart, and D.G. Stork. Pattern classification. *John Willey & Sons*, 2001.

[65] Y. Avrithis, Y. Kalantidis, G. Tolias, and E. Spyrou. Retrieving landmark and non-landmark images from community photo collections. In *ACM MM*, 2010.

[66] H. Bay, T. Tuytelaars, and L. Van Gool. Surf: Speeded up robust features. *ECCV*, 2006.

[67] Roelof van Zwol and Lluis Garcia Pueyo. Spatially-aware indexing for image object retrieval. In *WSDM 2012*, 2012.

[68] Ricardo Baeza-Yates and Berthier Ribeiro-Neto. *Modern Information Retrieval*. Boston, MA, USA, 1999.

[69] Andrei Z. Broder, David Carmel, Michael Herscovici, Aya Soffer, and Jason Zien. Efficient query evaluation using a two-level retrieval process. In *CIKM*, 2003.

[70] J. Yang, Y.G. Jiang, A.G. Hauptmann, and C.W. Ngo. Evaluating bag-of-visual-words representations in scene classification. In *MIR*, 2007.

[71] E. Voorhees and D. Harman. Trec experiment and evaluation in information retrieval. MIT Press, 2005.

[72] B. Girod, V. Chandrasekhar, D.M. Chen, N.M. Cheung, R. Grzeszczuk, Y. Reznik, G. Takacs, S.S. Tsai, and R. Vedantham. Mobile visual search. *Signal Processing Magazine, IEEE*, 2011.

[73] F.X. Yu, R. Ji, and S.F. Chang. Active query sensing for mobile location search. In *ACM MM*, 2011.

[74] J. He, T.H. Lin, J. Feng, and S.F. Chang. Mobile product search with bag of hash bits. In *Demo session of ACM MM*, 2011.

[75] G. Takacs, V. Chandrasekhar, N. Gelfand, Y. Xiong, W.C. Chen, T. Bismpigiannis, R. Grzeszczuk, K. Pulli, and B. Girod. Outdoors augmented reality on mobile phone using loxel-based visual feature organization. In *MIR*, 2008.

[76] D.G. Lowe. Distinctive image features from scale-invariant keypoints. *IJCV*, 2004.

[77] S.S. Tsai, D. Chen, V. Chandrasekhar, G. Takacs, N.M. Cheung, R. Vedantham, R. Grzeszczuk, and B. Girod. Mobile product recognition. In *ACM MM*, 2010.

[78] V. Chandrasekhar, M. Makar, G. Takacs, D. Chen, S.S. Tsai, N.M. Cheung, R. Grzeszczuk, Y. Reznik, and B. Girod. Survey of sift compression schemes. In *Proceedings of International Mobile Multimedia Workshop (IMMW)*, 2010.

[79] V. Chandrasekhar, Y. Reznik, G. Takacs, D. Chen, S. Tsai, R. Grzeszczuk, and B. Girod. Quantization schemes for low bitrate compressed histogram of gradients descriptors. In *CVPR Workshops*, 2010.

[80] D. Nister and H. Stewenius. Scalable recognition with a vocabulary tree. In *CVPR*, 2006.

[81] H. Jégou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *PAMI, IEEE Transactions on*, 2011.

[82] V. Chandrasekhar, G. Takacs, D. Chen, S. Tsai, R. Grzeszczuk, and B. Girod. Chog: Compressed histogram of gradients a low bit-rate feature descriptor. In *CVPR*, 2009.

[83] J. He, S.F. Chang, R. Radhakrishnan, and C. Bauer. Compact hashing with joint optimization of search accuracy and time. In *CVPR*, 2011.

[84] M. Yang, K. Kpalma, J. Ronsin, et al. A survey of shape feature extraction techniques. *Pattern Recognition*, 2008.

[85] F. Berrada, D. Aboutajdine, SE Ouatik, and A. Lachkar. Review of 2d shape descriptors based on the curvature scale space approach. In *ICMCS*, 2011.

[86] M.M. Cheng, G.X. Zhang, N.J. Mitra, X. Huang, and S.M. Hu. Global contrast based salient region detection. In *CVPR*, 2011.

[87] C. Rother, V. Kolmogorov, and A. Blake. Grabcut: Interactive foreground extraction using iterated graph cuts. In *ACM Transactions on Graphics*, 2004.

[88] R.B. Yadav, N.K. Nishchal, A.K. Gupta, and V.K. Rastogi. Retrieval and classification of shape-based objects using fourier, generic fourier, and wavelet-fourier descriptors technique: A comparative study. *Optics and Lasers in engineering*, 2007.

[89] V.R. Chandrasekhar, D.M. Chen, S.S. Tsai, N.M. Cheung, H. Chen, G. Takacs, Y. Reznik, R. Vedantham, R. Grzeszczuk, J. Bach, et al. The stanford mobile visual search data set. In *ACM conference on Multimedia systems*, 2011.

[90] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is nearest neighbor meaningful? *Database Theory ICDT99*, pages 217–235, 1999.

[91] C. Aggarwal, A. Hinneburg, and D. Keim. On the surprising behavior of distance metrics in high dimensional space. *Database Theory ICDT 2001*, pages 420–434, 2001.

[92] D. Francois, V. Wertz, and M. Verleysen. The concentration of fractional distances. *IEEE Transactions on Knowledge and Data Engineering*, pages 873–886, 2007.

Part VIII

Appendix

Chapter 9

Proofs

9.1 Proofs for Chapter 2

9.1.1 Sketch of the Proofs for Theorem 2.3.1

In this section, we will provide the sketch of proof for Theorem 2.3.1. More details, i.e., the proofs for theorems and lemmas in this section, can be found in next Section.

9.1.1.1 There is one global minimal k_{min}

From $P_{nn-miss}(k, L) = (1 - (p_{nn})^k)^L = \delta$, we have

$$L = \frac{\log \delta}{\log(1 - (p_{nn})^k)}.$$

Note that when z is a small positive number, $\log(1-z) = -\sum_{i=1}^{n} \frac{z^i}{i} \approx -z$. Since p_{nn}^k is a very small positive number, therefore $L = \frac{\log \delta}{\log(1-(p_{nn})^k)}$ can be well approximated as $L = \frac{\log \frac{1}{\delta}}{(p_{nn})^k}$. Since L now is a function of k, both $T(k, L)$ and $S(k, L)$ depend on k only. So we denote $T(k, L)$ and $S(k, L)$ as $T(k)$ and $S(k)$ in the following discussion, where

$$T(k) = U_{hash} kL + U_{check} Ln(p_{any})^k = [U_{hash} k + U_{check} n(p_{any})^k] \frac{\log \frac{1}{\delta}}{(p_{nn})^k} \tag{9.1}$$

and

$$S(k) = nL + nd = nd + n\frac{\log \delta}{-(p_{nn})^k} \tag{9.2}$$

First note that $\frac{dT(k)}{dk} = \frac{d((ak+bp_{any}^k)p_{nn}^{-k})}{dk}$

$\frac{dT(k)}{dk} = p_{nn}^{-k}a(k\log p_{nn}^{-1} + 1) - b(\frac{p_{any}}{p_{nn}})^k \log \frac{p_{nn}}{p_{any}}$.

The following two lemmas shows $\frac{dT(k)}{dk}$ will have only one local/global minimum.

Lemma 9.1.1. *$\frac{dT(k)}{dk}$ is monotonically increasing with k.*

Lemma 9.1.2. *When the database size n is large enough, there will be one and only one k_{min}, such that $T^{'}(k_{min}) = 0$, and hence $T(k_{min})$ will be minimal.*

9.1.1.2 The tight bound of k_{min}

From $T^{'}(k_{min}) = 0$, we can obtain Theorem 9.1.3, which gives some estimation about k_{min}.

Theorem 9.1.3. $p_{any}^{k_{min}} = \frac{U_{hash}}{U_{check}n}\frac{1}{\log \frac{p_{nn}}{p_{any}}}(k_{min}\log p_{nn}^{-1} + 1)$

We can find the tight bound k_{min}, as shown in the following two lemmas.

Lemma 9.1.4. $k_{min} = O(\log_{p_{any}^{-1}}(\alpha_1 n))$ *where* $\alpha_1 = \frac{U_{check}\log(p_{nn}/p_{any})}{U_{hash}}$.

Lemma 9.1.5. $k_{min} = \Omega(\log_{p_{any}^{-1}}(\alpha_1 n))$ *where* $\alpha_1 = \frac{U_{check}\log(p_{nn}/p_{any})}{U_{hash}}$.

From the above two lemmas, we know

$$k_{min} = \Theta(\log_{p_{any}^{-1}}(\alpha_1 n)).$$

9.1.1.3 The proof of Theorem 2.3.1

Putting Theorem 9.1.3 into (9.1), we can obtain Theorem 9.1.6 to provide the value of $T(k_{min})$ and the corresponding $S(k_{min})$.

Theorem 9.1.6. $T(k_{min}) = \log \frac{1}{\delta} U_{hash} n^{\rho} (k_{min} \log p_{any}^{-1} + 1)(k_{min} \log p_{nn}^{-1} + 1)^{-\rho} \alpha_1^{\rho} \frac{1}{\log(p_{nn}/p_{any})}$ $S(k_{min}) = dn + (\log \frac{1}{\delta}) n^{1+\rho} \alpha_1^{\rho} (k_{min} \log p_{nn}^{-1} + 1)^{\rho}$ *where* $\rho = \frac{\log p_{nn}}{\log p_{any}}$ *and* $\alpha_1 = \frac{U_{check} \log(p_{nn}/p_{any})}{U_{hash}}$. *Moreover, the number of hash tables* $L = (\log \frac{1}{\delta})[\frac{\alpha_1 n}{(k_{min} \log p_{nn}^{-1} + 1)}]^{\rho}$. *The number of returned points is* $(\log \frac{1}{\delta}) n^{\rho} {\alpha_1}^{1-\rho} (k_{min} \log p_{nn}^{-1} + 1)^{1-\rho}$.

However, $T(k_{min})$ and $S(k_{min})$ in Theorem 9.1.6 still depends on k_{min}.

Put $k_{min} = \Theta(\log_{p_{any}^{-1}}(\alpha_1 n))$ into Theorem 9.1.6, we can get

$$T(k_{min}) = \Theta(\log \frac{1}{\delta} n^{\rho} U_{hash} [\frac{U_{check}}{U_{hash}}]^{\rho} \alpha_0 \alpha^{-\rho})$$

where $\alpha_0 = \frac{\log(\tau \frac{U_{check}}{U_{hash}} n) + 1}{\tau}$, $\alpha = \frac{\rho \log(\tau \frac{U_{check}}{U_{hash}} n) + 1}{\tau}$, $\rho = \frac{\log p_{nn}}{\log p_{any}}$ and $\tau = \log(p_{nn}/p_{any})$.

Moreover, putting $k_{min} = \Theta(\frac{\log(\alpha_1 n)}{\log p_{any}^{-1}})$ into $S(k_{min}) = dn + (\log \frac{1}{\delta}) n^{1+\rho} \alpha_1^{\rho} (k_{min} \log p_{nn}^{-1} + 1)^{-\rho}$, we get

$$S(k_{min}) = \Theta(dn + (\log \frac{1}{\delta}) n^{1+\rho} [\frac{U_{check}}{U_{hash}}]^{\rho} \alpha^{-\rho})$$

Moreover, the number of hash tables $L = (\log \frac{1}{\delta})[\frac{\alpha_1 n}{(k_{min} \log p_{nn}^{-1} + 1)}]^{\rho} = \Theta(\log \frac{1}{\delta} n^{\rho} [\frac{U_{check}}{U_{hash}}]^{\rho} \alpha^{-\rho})$.

The number of returned points is $\log \frac{1}{\delta} n^{\rho} {\alpha_1}^{\rho-1} (k_{min} \log p_{nn}^{-1} + 1)^{1-\rho} = \Theta(\log \frac{1}{\delta} n^{\rho} [\frac{U_{check}}{U_{hash}}]^{\rho-1} \alpha^{1-\rho})$.

This completes the proof.

9.1.2 Details of Proofs for Theorem 2.3.1

Proof of Lemma 9.1.1:

First, it is easy to see $p_{nn}^{-k} a(k \log p_{nn}^{-1} + 1)$ is monotonically increasing with k. Moreover, note that $\frac{p_{any}}{p_{nn}} < 1$, so $-b(\frac{p_{any}}{p_{nn}})^k \log \frac{p_{nn}}{p_{any}}$ will be monotonically increasing with k too. So $\frac{dT(k)}{dk} = p_{nn}^{-k} a(k \log p_{nn}^{-1} + 1) - b(\frac{p_{any}}{p_{nn}})^k \log \frac{p_{nn}}{p_{any}}$ is monotonically increasing with k. This completes the proof.

Proof of Lemma 9.1.2:

When $k = 0$, $\frac{dT(k)}{dk} = U_{hash}(-\log \delta) - \log \frac{p_{nn}}{p_{any}} U_{check}(-\log \delta) n$. When the database size is large enough, i.e., $n > \frac{U_{hash}}{\log \frac{p_{nn}}{p_{any}} U_{check}}$, $\frac{dT(k)}{dk} < 0$, for $k = 0$.

Moreover, when $k \to +\infty$, $p_{nn}^{-k} a(k \log p_{nn}^{-1} + 1) \to +\infty$, and $-b(\frac{p_{any}}{p_{nn}})^k \log \frac{p_{nn}}{p_{any}} = -U_{check}(-\log \delta) n (\frac{p_{any}}{p_{nn}})^k \log \frac{p_{nn}}{p_{any}} \to 0$.

In conclusion, $\frac{dT(k)}{dk} < 0$, for $k = 0$. $\frac{dT(k)}{dk} > 0$, when k is large. There must be some k such that $\frac{dT(k)}{dk} = 0$.

Moreover, $\frac{dT(k)}{dk}$ is monotonically increasing with k, so there must be only one k_{min} such that $T^{'}(k_{min}) = 0$, at which $T(k_{min})$ will be minimal.

Proof of Lemma 9.1.4:

Since $p_{any}^{-k_{min}} = \frac{U_{check} n \log(p_{nn}/p_{any})}{U_{hash}} \frac{1}{(k_{min} \log p_{nn}^{-1}+1)}$,

$$p_{any}^{-k_{min}} \leq \frac{nU_{check} \log(p_{nn}/p_{any})}{U_{hash}}$$

. Or in other words,

$$k_{min} \leq \frac{\log(\alpha_1 n)}{(-\log p_{any})}$$

where

$$\alpha_1 = \frac{U_{check} \log(p_{nn}/p_{any})}{U_{hash}}.$$

Proof of Lemma 9.1.5:

First recall that $p_{any}^{-k_{min}} = \frac{\alpha_1 n}{(k_{min} \log p_{nn}^{-1}+1)}$ where $\alpha_1 = \frac{U_{check} \log(p_{nn}/p_{any})}{U_{hash}}$.

so $P_{any}^{-k_{opt}} \geq \frac{\alpha_1}{(\log P_{nn}^{-1}+1)} n k_{opt}^{-1}$,

$$k_{opt} \log P_{any}^{-1} \geq \log(\frac{\alpha_1 n}{(\log P_{nn}^{-1} + 1)}) - \log k_{opt}$$

From the lemma above, we know that $k_{opt} \leq \frac{\log(\alpha_1 n)}{\log P_{any}^{-1}}$, so when n is large enough

$$\log(k_{opt}) \leq \log(\frac{\log(\alpha_1 n)}{\log P_{any}^{-1}}) \ll \log(\frac{\alpha_1 n}{(\log P_{nn}^{-1} + 1)})$$

And we can ignore $\log k_{opt}$, and get

$$k_{opt} \log P_{any}^{-1} \geq \log(\frac{\alpha_1 n}{(\log P_{nn}^{-1} + 1)})$$

so

$$k_{opt} \geq \frac{\log(\frac{\alpha_1 n}{(\log P_{nn}^{-1}+1)})}{\log P_{any}^{-1}} = \Omega(\frac{\log(\alpha_1 n)}{\log P_{any}^{-1}})$$

Proof of Theorem 9.1.6:

From $T^{'}(k_{min}) = 0$, we have

$$T^{'}(k_{min}) = p_{nn}^{-k_{min}} a(k_{min} \log p_{nn}^{-1} + 1) - b(\frac{p_{any}}{p_{nn}})^{k_{min}} \log \frac{p_{nn}}{p_{any}} = 0.$$

$$p_{any}^{k_{min}} = \frac{1}{\log \frac{p_{nn}}{p_{any}}} \frac{a}{b}(k_{min} \log p_{nn}^{-1} + 1)$$
$$\Leftrightarrow p_{any}^{k_{min}} = \frac{U_{hash}}{U_{check} n} \frac{1}{\log \frac{p_{nn}}{p_{any}}}(k_{min} \log p_{nn}^{-1} + 1) \tag{9.3}$$

Putting (9.3) into

$$T(k_{min}) = (ak_{min} + bp_{any}^{k_{min}})p_{nn}^{-k_{min}}$$

we get

$$T(k_{min}) = (ak_{min} + bp_{any}^{k_{min}})p_{nn}^{-k_{min}}$$
$$= p_{nn}^{-k_{min}}[U_{hash} \log \frac{1}{\delta} k_{min} + U_{check} \log \frac{1}{\delta} n\{\frac{U_{hash}}{U_{check} n} \frac{1}{\log \frac{p_{nn}}{p_{any}}}(k_{min} \log p_{any}^{-1} + 1)\}]$$
$$= p_{nn}^{-k_{min}} U_{hash} \log \frac{1}{\delta}(\frac{k_{min} \log p_{any}^{-1} + 1}{\log(p_{nn}/p_{any})}) \tag{9.4}$$

Moreover, note that $p_{any}^{-k_{min}} = \frac{\alpha_1 n}{(k_{min} \log p_{nn}^{-1}+1)}$ where $\alpha_1 = \frac{U_{check} \log(p_{nn}/p_{any})}{U_{hash}}$ and $x = y^{\frac{\log x}{\log y}}$, so

$$p_{nn}^{-k_{min}} = (p_{any}^{-k_{min}})^{\frac{\log p_{nn}}{\log p_{any}}} = [\frac{\alpha_1 n}{(k_{min} \log p_{nn}^{-1} + 1)}]^{\rho}$$

$$T(k_{min}) = [\frac{\alpha_1 n}{(k_{min} \log p_{nn}^{-1} + 1)}]^{\rho}[U_{hash} \log \frac{1}{\delta}(\frac{k_{min} \log p_{nn}^{-1} + 1}{\log(p_{nn}/p_{any})})]$$

which can be rewritten as

$$T(k_{min}) = \log \frac{1}{\delta} U_{hash} n^{\rho}(k_{min} \log p_{nn}^{-1} + 1)(k_{min} \log p_{nn}^{-1} + 1)^{-\rho} \alpha_1^{\rho} \frac{1}{\log(p_{nn}/p_{any})} \tag{9.5}$$

Note that $nL = n\frac{-\log \delta}{p_{nn}^{k_{min}}} = n(-\log \delta)(p_{any}^{-k_{min}})^{\frac{\log p_{nn}}{\log p_{any}}}$ so the space complexity is

$$S(k_{min}) = dn + (\log \frac{1}{\delta})n[\frac{\alpha_1 n}{(k_{min} \log p_{nn}^{-1} + 1)}]^{\rho}$$

or rewritten as

$$S(k_{min}) = dn + (\log \frac{1}{\delta})n^{1+\rho}\alpha_1^{\rho}(k_{min} \log p_{nn}^{-1} + 1)^{-\rho}$$

Moreover, the number of hash tables $L = (\log \frac{1}{\delta})[\frac{\alpha_1 n}{(k_{min} \log p_{nn}^{-1}+1)}]^{\rho}$. The number of returned points is $Lnp_{any}^{k_{min}} = \frac{(k_{min} \log p_{nn}^{-1}+1)}{\alpha_1}(\log \frac{1}{\delta})[\frac{\alpha_1 n}{(k_{min} \log p_{nn}^{-1}+1)}]^{\rho} = (\log \frac{1}{\delta})n^{\rho}{\alpha_1}^{\rho-1}(k_{min} \log p_{nn}^{-1}+ 1)^{1-\rho}$

9.2 Proofs for Chapter 3

Proof of Theorem 3.1.1

Denote p_i as the probability for one random point to fall into cluster i. Then when n is large enough, the probability for a random query and a random database point both falls into the cluster i is p_i^2. Then

$$\widehat{P}_{any}(\Psi) = \sum_{i=1}^{K} p_i^2 = \frac{1}{n^2}\sum_{i=1}^{K} n_i^2$$

Note that $\sum\limits_{i=1,...,K} n_i = n$, $\sum\limits_{i=1}^{K} (n_i)^2$ will minimized if n_i are equal, i.e., $n_i = n/K$, $i = 1, ..., K$.

Proof of Theorem 3.1.2

Note that $Entropy(y) = \{-\sum\limits_{i=1}^{2^k} P(y = a_i) \log P(y = a_i)\} = \{-\sum\limits_{i=1}^{2^k} \frac{n_i}{n} \log(\frac{n_i}{n})\}$. It is easy to see that $n_i = \frac{n}{2^k}$ for $i = 1, ..., 2^k$, if and only if $Entropy(y)$ gets its maximum value.

As shown in [54, 55],

$$Entropy(y) = \sum_{m=1}^{k} Entropy(y_m) - I(y_1, ..., y_m, ..., y_k) \tag{9.6}$$

where $I()$ is the mutual information.

So $Entropy(y)$ would be maximized, if $\sum\limits_{m=1}^{k} Entropy(y_m)$ is maximized and $I(y_1, ..., y_m, ..., y_k)$ is minimized. Moreover, note that y_m is a binary random variable. If the mathematical expectation $E(y_m) = 0$, half samples would have bit $+1$ and the other half would have bit -1 for y_m, which means $Entropy(y_m) = 1$, and is maximized.

In conclusion, if $E(y_m) = 0, m = 1, ..., k$ and $I(y_1, ..., y_m, ..., y_k)$ is minimized, $Entropy(y)$ would be maximized, and the search time would be minimized.

Derivation for Algorithm 1:

The optimization for SPICA and GSPICA is non-convex. It is not trivial to get the solution especially when the data size is large. In this section, we provide a simple but fast and practical algorithm to solve the optimization problem. Moreover, by replacing $X_i$with K_{X_i} in the following discussion, we can easily extend the solution to GSPICA.

Suppose $E(xx^T) = \Sigma = \Omega\Lambda\Omega^T$, where $\Omega\Lambda\Omega^T$ is the SVD decomposition of Σ . Λ is a diagonal matrix, and Ω is the orthogonal matrix. Denote

$$Q = \Omega_k \Lambda_k^{-\frac{1}{2}} \tag{9.7}$$

where Λ_k is a diagonal matrix consisting of k largest eigen values of Λ, and Ω_k are the corresponding columns of Ω.

Denote $T_m = Q\widetilde{T}_m$, $\widetilde{x} = Q^T x$, $\widetilde{X}_i = Q^T X_i$, and $\widetilde{C} = Q^T C Q$. The formulation of SPICA equals to

$$\begin{aligned} &\max_{T_m, m=1...k} \sum_{m=1}^{k} ||g_0 - E(G(\widetilde{T}_m^T \widetilde{x}))||^2 \\ &s.t., \widetilde{T}_m^T \widetilde{T}_j = \delta_{mj}, 1 \le m, j \le k \\ &\sum_{m=1}^{k} \widetilde{T}_m^T \widetilde{C} \widetilde{T}_m \le \eta \end{aligned} \tag{9.8}$$

After $\widetilde{T}_m$ is obtained, T_m can be computed as: $T_m = Q\widetilde{T}_m$.

Since $||g_0 - f||^2$ can get maximal value only when f is maximized or minimized, the optimal solution in Equation (9.8) can only come from two following solutions:

$$\begin{aligned} &\min_{T_m, m=1...k} \sum_{m=1}^{k} E(G(\widetilde{T}_m^T \widetilde{x}))_m = \sum_{m=1}^{k} \tfrac{1}{N} \sum_{i=1}^{N} (G(\widetilde{T}_m^T \widetilde{X}_i)) \\ &s.t., \widetilde{T}_m^T \widetilde{T}_j = \delta_{mj}, 1 \le m, j \le k \\ &\sum_{m=1}^{k} \widetilde{T}_m^T \widetilde{C} \widetilde{T}_m \le \eta \end{aligned} \tag{9.9}$$

or

$$\begin{aligned} &\max_{T_m, m=1...k} \sum_{m=1}^{k} E(G(\widetilde{T}_m^T \widetilde{x})) = \sum_{m=1}^{k} \tfrac{1}{N} \sum_{i=1}^{N} (G(\widetilde{T}_m^T \widetilde{X}_i)) \\ &s.t., \widetilde{T}_m^T \widetilde{T}_j = \delta_{mj}, 1 \le m, j \le k \\ &\sum_{m=1}^{k} \widetilde{T}_m^T \widetilde{C} \widetilde{T}_m \le \eta \end{aligned} \tag{9.10}$$

Following a process similar to that used in [19] which is proved to be one of the fastest and most practical algorithms for ICA, we provide a similar iteration method to solve our optimization problem. For problems in equation (9.9) and (9.10), we would obtain $\widetilde{T}_m$ for $m = 1, .., k$ iteratively, i.e., one by one.

Suppose we already got $\widetilde{T}_1, ...\widetilde{T}_{m-1}$, now we want to obtain $\widetilde{T}_m$. First we apply QR decomposition to matrix $[\widetilde{T}_1, ...\widetilde{T}_{m-1}]$ to get matrix B, such that $[\widetilde{T}_1, ...\widetilde{T}_{m-1}, B]$ is an orthogonal matrix. Denote $\widetilde{T}_m = Bw$. The constraint

$$\widetilde{T}_m^T \widetilde{T}_j = \delta_{mj}, \text{for} 1 \le j \le m$$

now becomes

$$w^T B^T B w = w^T w = 1$$

Substituting $\widetilde{T}_m = Bw$ to equation (9.9), we can get the KKT condition in terms of w in equation (9.9) as

$$L(w) = E(G(w^T \widehat{x})) - \gamma w^T A w - \beta w^T w$$

with $\gamma \le 0$, where $\widehat{x} = B^T \widetilde{x} = B^T Q^T x, A = B^T \widetilde{C} B$.

The stationary point for the KKT condition can be found by

$$F(w) = \tfrac{\partial L}{\partial w} = E(\widehat{x} G'(w^T \widehat{x})) - \gamma A w - \beta w = 0, \gamma \le 0 \tag{9.11}$$

where G' is the derivative of function G.

Multiplicating w^T to equation (9.11), we can get

$$\begin{aligned} &E(w^T \widehat{x} G'(w^T \widehat{x})) - \gamma w^T A w - \beta w^T w = 0, \gamma \le 0 \\ &\Rightarrow \beta = E(w^T \widehat{x} G'(w^T \widehat{x})) - \gamma w^T A w, \gamma \le 0 \end{aligned} \tag{9.12}$$

Similarly, for equation (9.10) , we will have

$$F(w) = E(\widehat{x}G'(w^T\widehat{x})) - \gamma Aw - \beta w = 0, \gamma \geq 0 \tag{9.13}$$

where $\beta = E(w^T\widehat{x}G'(w^T\widehat{x})) - \gamma w^T Aw, \gamma \geq 0$.

By combining the above two cases (9.11) and (9.13), we have:

$$F(w) = E(\widehat{x}G'(w^T\widehat{x})) - \gamma Aw - \beta w = 0 \tag{9.14}$$

where $\beta = E(w^T\widehat{x}G'(w^T\widehat{x})) - \gamma w^T Aw$, and γ is a parameter to be tuned through empirical validation.

Similarly as discussed in [19] the Jacobin function for $F(w)$ is:

$$JF(w) \approx E(G''(w^T\widehat{x}))I - \gamma A - \beta I$$

where G'' is the second derivative of function G. So we will update w as $w = w - JF^{-1}(w)F(w)$.

A complete workflow to solve SPICA [1] is shown in algorithm 1 in Chapter 3.

Proof of Theorem 3.4.1:

With the same relaxation as in spectral hashing by ignoring the constraint of $Y_i \in \{-1, 1\}^k$, we will have

$$Y_i = A^T K_i - b$$

Hence, from the constraint of $\sum_{i=1}^{n} Y_i = 0$, we can get

$$\sum_{i=1}^{n} (A^T K_i - b) = 0 \Rightarrow b = A^T \bar{a} \tag{9.15}$$

[1] By replacing X_i with K_{X_i}, we can easily obtain the implementation for GSPICA.

Moreover, since

$$\begin{aligned}\sum_{i=1}^{n} Y_i Y_i^T &= \sum_{i=1}^{n} (A^T K_i - b)(A^T K_i - b)^T \\ &= A^T (K_{p\times n} K_{p\times n}^T - \sum_{i=1}^{n} K_i \bar{a}^T \\ &- \sum_{i=1}^{n} \bar{a} K_i^T + \sum_{i=1}^{n} \bar{a}\bar{a}^T) A \\ &= A^T (K_{p\times n} K_{p\times n}^T - n\bar{a}\bar{a}^T) A\end{aligned} \tag{9.16}$$

from the constraint of $\frac{1}{n}\sum_{i=1}^{n} Y_i Y_i^T = I$, we can get

$$A^T \frac{1}{n} (K_{p\times n} K_{p\times n}^T - n\bar{a}\bar{a}^T) A = I$$

And because

$$\frac{1}{2}\sum_{i,j=1}^{n} W_{ij} ||Y_i - Y_j||^2 \tag{9.17}$$

$$= tr\left(A^T (K_{p\times n}(D - W) K_{p\times n}^T) A\right) \tag{9.18}$$

So the optimization problem in (3.16) becomes:

$$\begin{aligned}\min_{A} \quad & tr(A^T C A) \\ s.t. \quad & A^T G A = I\end{aligned} \tag{9.19}$$

where

$$C = K_{p\times n}(D - W) K_{p\times n}^T$$

$$G = \frac{1}{n} K_{p\times n} K_{p\times n}^T - \bar{a}\bar{a}^T$$

Moreover, note that

$$tr(A^T C A) = tr((A^T C A)^T) = tr(A^T C^T A)$$

so

$$tr(A^T C A) = tr(A^T \frac{(C + C^T)}{2} A)$$

which completes the proof of Proposition 1.

9.3 Analysis for Chapter 4

The relationship between $\widehat{P}_{nn}(\Psi)$ and K-means cost function

First we show that larger $\widehat{P}_{nn}(\Psi)$ will somewhat lead to a smaller quantization error. Actually, when $\widehat{P}_{nn}(\Psi)$ is larger, a point and its nearest or near neighbors will have high probability to be in the cluster. For two points x and y, supposed C_x and C_y are their cluster center. Note that a point y and its nearest neighbor x should have small distance $D(x, y)$. Moreover, the distance between two cluster centers is usually much larger than the distance of two nearest neighbors. In other words, if two points are nearest neighbors but fall into different clusters, then usually we have $D(C_x, C_y) \gg D(x, y)$, and increase the quantization error, where the quantization error is defined as

$$\int (D(x, y) - D(C_x, C_y))^2 p(x)p(y)dxdy.$$

So larger $\widehat{P}_{nn}(\Psi)$ will often help to reduce the quantization error.

Moreover, following a similar discussion as in [50], we will see the quantization error is bounded by within-cluster distances, i.e., mean squared error (mse),

$$D(x, y) - D(y, C_y) - D(x, C_x) \leq D(x, C_y) - D(x, C_x) \leq D(C_x, C_y) \tag{9.20}$$

$$\leq D(x, C_y) + D(x, C_x) \leq D(x, y) + D(y, C_y) + D(x, C_x) \tag{9.21}$$

which equals to $(D(x, y) - D(C_x, C_y))^2 \leq (D(y, C_y) + D(x, C_x))^2 \leq 2[D(y, C_y)^2 + D(x, C_x)^2]$. And hence

$$\int (D(x, y) - D(C_x, C_y))^2 p(x)p(y)dxdy \tag{9.22}$$

$$\leq 2 \int [D(y, C_y)^2 + D(x, C_x)^2] p(x)p(y)dxdy \tag{9.23}$$

$$= 4 \int D(x, C_x)^2 p(x) dx \tag{9.24}$$

For a K-means clustering with data points X_i, $i = 1, ..., n$ and cluster center C_j, $\int D(x, c_x)^2 p(x) dx$ can be empirically computed as $\sum_{j=1}^{K} \sum_{i \in S_j} (X_i - C_j)^2$.

9.4 Proofs for Chapter 7

9.4.1 Proofs

Proof of Theorem 7.2.1:

Since R_j are independent and satisfy Lindeberg's condition, from central limit theorem, R will be distributed as Gaussian for large enough d with mean $\mu = \sum_j \mu_j$ and variance $\sigma^2 = \sum_j \sigma_j^2$. Normalizing the data by dividing by μ, the new mean is $\mu' = 1$, and new variance is σ'^2 as defined in (7.4). Now, the probability that $R \leq \alpha$ for any $0 \leq \alpha \leq 1$ is given as

$$P(R \leq \alpha) \approx \phi(\frac{\alpha - 1}{\sigma'}) - \phi(\frac{0 - 1}{\sigma'}), \tag{9.25}$$

where ϕ is the c.d.f of standard Gaussian, and the second term in RHS is the correction factor since R is always nonnegative.

Let's denote the number of samples for which $R \leq \alpha$ as $n(\alpha)$. Clearly, $n(\alpha)$ follows Binomial distribution with probability of success given in (9.25):

$$P(n(\alpha) = k) = \binom{n}{k} (P(R \leq \alpha))^k (1 - P(R \leq \alpha))^{n-k}.$$

Hence the expected number of database points, $\bar{n}(\alpha)$ that satisfy $R \leq \alpha$ can be computed as

$$\bar{n}(\alpha) = E[n(\alpha)] = nP(R \leq \alpha) = n(\phi(\frac{\alpha - 1}{\sigma'}) - \phi(\frac{-1}{\sigma'})).$$

Recall D_{min} is the expected distance to the nearest neighbor and $R_{min} \approx D_{min}^p$.[2]

[2]The approximation becomes exact when metric L_1 is considered. For other norms (e.g., $p = 2$), bounds on D_{min} can be further derived.

Thus, $\bar{n}(D^p_{min}) \approx \bar{n}(R_{min}) = 1$. Hence,

$$D_{min} \approx (\bar{n}^{-1}(1))^{\frac{1}{p}} \approx [1 + \phi^{-1}(\frac{1}{n} + \phi(\frac{-1}{\sigma'}))\sigma']^{\frac{1}{p}} \tag{9.26}$$

Moreover, after normalization, R follows a Gaussian distribution with mean 1. So, $R_{mean} = 1$, and $D_{mean} \approx R^{\frac{1}{p}}_{mean} = 1$. Thus, the relative contrast can be approximated as:

$$C_r = \frac{D_{mean}}{D_{min}} \approx \frac{1}{[1 + \phi^{-1}(\frac{1}{n} + \phi(\frac{-1}{\sigma'}))\sigma']^{\frac{1}{p}}}$$

which completes the proof.

Proof of Theorem 7.2.2:

The probability for both x^j and q^j to be non-zero is s_j^2, and the probability for one of them to be non-zero is $2(1 - s_j)s_j$. Hence, the mean

$$\mu_j = E[R_j] = E[|x^j - q^j|^p]$$

for sparse vectors can be computed as,

$$\mu_j = s_j^2 m'_{j,p} + 2(1 - s_j)s_j m_{j,p}$$

Similarly, the variance

$$\sigma_j^2 = Var[R_j] = E[R_j^2] - E[R_j]^2 = E[|x^j - q^j|^{2p}) - \mu_j^2$$

for sparse vectors can be given as,

$$\sigma_j^2 = s_j^2 m'_{j,2p} + 2(1 - s_j)s_j m_{j,2p} - \mu_j^2,$$

Thus, the normalized variance for sparse vectors is:

$$\sigma'^2 = \frac{\sum_{j=1}^d \sigma_j^2}{(\sum_{j=1}^d \mu_j)^2}. \tag{9.27}$$

If we assume each dimension to be i.i.d, i.e., all V_j have the same distribution with $E[V_j] = \mu_d$, $var[V_j] = \sigma_d^2$, and also assume $s_j = s$, $m_{j,p} = m_p$ and $m'_{j,p} = m'_p$, then

$$\sigma' = \frac{1}{d^{1/2}}\frac{\sigma_d}{\mu_d} = \frac{1}{d^{1/2}}\sqrt{\frac{s[(m'_{2p} - 2m_{2p})s + 2m_{2p}]}{s^2[(m'_p - 2m_p)s + 2m_p]^2} - 1} \tag{9.28}$$

Proof of Theorem 7.3.1:

With the hash functions of

$$h(x) = \lfloor \frac{v^T x + b}{w} \rfloor$$

it can be shown that([22]),

$$P(h(X_i) = h(q)) = f_h(||X_i - q||_p) \tag{9.29}$$

where function $f_h(a) = \int_0^w \frac{1}{a} f_p(\frac{z}{a})(1 - \frac{z}{w})dz$ is monotonically decreasing with a. Here f_p is the p.d.f. of the absolute value of a p-stable variable.

Suppose the data are normalized by a scale factor such that $D_{mean} = 1$. Note that such a normalization will not change the nearest neighbor search results at all. In this case, $D_{min} = 1/C_r$. Denote p_{nn} (p_{any}) as the probability for one random query q and its nearest neighbor (q and a random database point) to have the same code with one hash function. According to equation (9.29),

$$p_{nn} = f_h(1/C_r)$$

and

$$p_{any} = f_h(1),$$

since the expected distance between q and its nearest neighbor is $D_{min} = 1/C_r$, and the expected distance between q and a random database point is $D_{mean} = 1$.

So $\rho = \frac{\log p_{nn}}{\log p_{any}}$ is actually a function of C_r, denoted as $g(C_r)$.

$$\rho = g(C_r) = \frac{\log p_{nn}}{\log p_{any}} = \frac{\log f_h(1/C_r)}{\log f_h(1)}.$$

Since $f_h(\cdot)$ is a monotonically decreasing function, when C_r is larger, $g(C_r)$ will be smaller[3]

Since, the returning candidate points, the time complexity, space complexity and the number of hash tables needed are monotonically increasing with ρ, so they are monotonically decreasing with C_r.

[3]Note that both $\log f_h(1/C_r)$ and $\log f_h(1)$ are negative, since $f_h(\cdot)$ is always ≤ 1.

9.4.2 Previous Works on the Difficulty of Nearest Neighbor Search

One of the influential works that analyzed nearest neighbor search in high dimensional spaces is from Beyer et.al. ([90]), whose main result is shown in Theorem 9.4.1.

Theorem 9.4.1. *([90]) Denote* $D^q_{max} = \max_{i=1,...n} D(X_i, q)$ *and* $D^q_{min} = \min_{i=1,...n} D(X_i, q)$. *If* $\lim_{d\to\infty} var(\frac{D(X_i,q)^p}{E[D(X_i,q)^p]}) \to 0$, *then for every* $\epsilon \geq 0$,
$\lim_{d\to\infty} P[D^q_{max} \leq (1+\epsilon)D^q_{min}] = 1$.

Intuitively speaking, Theorem 9.4.1 points out if the condition $\lim_{d\to\infty} var(\frac{D(X_i,q)^p}{E[D(X_i,q)^p]}) \to 0$ is true (for example when the dimensions are i.i.d), D^q_{max} will be approximately the same as D^q_{min} with probability 1, and hence searching nearest neighbors will not be meaningful in the case of $d \to \infty$.

Moreover, under the assumption that every dimension is not only i.i.d., but also uniformly distributed, Aggarwal et.,al. ([91]) have extended Beyer's theory and proved that $D^q_{max} - D^q_{min}$ will asymptotically grow as $d^{\frac{1}{p}-\frac{1}{2}}$ (here p represents distance metric L_p), and hence smaller p results in better contrast. Its main result is shown in Theorem 9.4.2.

Theorem 9.4.2. *Aggarwal et.,al. ([91]:)*
Suppose every dimension of the data is i.i.d., and l_p *distance metric is considered. Denote* $D^q_{max} = \max_{i=1,...n} ||X_i - q||_p$, $D^q_{min} = \min_{i=1,...n} ||X_i - q||_p$, *then*

$$A_p \leq \lim_{d\to\infty} [\frac{D^q_{max} - D^q_{min}}{d^{\frac{1}{p}-\frac{1}{2}}}] \leq (n-1)A_p,$$

where A_p *is a constant related to p.*

In ([92]), a measurement called "relative variance" defined as $\frac{\sqrt{Var(||X_i-q||_p)}}{E(||X_i-q||_p)}$, which is a modification of the condition $var(\frac{D(X_i,q)^p}{E[D(X_i,q)^p]})$ in Beyer's work, is discussed. If every dimension is i.i.d, the result is shown in Theorem 9.4.3.

Theorem 9.4.3. *([92]) If every dimension of the data is i.i.d., when* $d \to \infty$, $\frac{\sqrt{Var(||X_i-q||_p)}}{E(||X_i-q||_p)} \approx \frac{1}{\sqrt{d}}\frac{1}{p}\frac{\sigma_j}{\mu_j}$, *where* $\sigma_j = Var(||X_i^j - q^j||_p^p)$ *and* $\mu_j = E(||X_i^j - q^j||_p^p)$ *are the variance and mean on each dimension.*

It shows the "relative variance" will be worse if d is larger or p is larger. However, it is not clear how "relative variance" affects D^q_{mean} and D^q_{min}, or the complexity of approximate NN search.

9.4.3 Relations Between Our Analysis and Previous Works

Relation to Beyer's Work

Note that if the distance function $D(X_i, q)$ in Beyer's work is L_p distance, then $var(\frac{D(X_i,q)^p}{E[D(X_i,q)^p]}) = \frac{\sigma^2}{\mu^2} = (\sigma')^2$. When $\sigma' \to 0 (d \to \infty)$, Beyer's work shows that $D^q_{max} \approx D^q_{min}$, and our theory shows $C_r \to 1$, or equivalently $D_{mean} \to D_{min}$. So we will get the same conclusion: when $d \to \infty$, NN search is not very "meaningful" , because we can not differentiate the nearest neighbor from other points. However, Beyer's theory works for the worst case (i.e., compare NN point to the worst point with maximum distance), while ours works for the average case.

Relation to Francois's Work

In Theorem 9.4.3, a measurement called "relative variance", defined as $\frac{\sqrt{Var(||X_i-q||_p)}}{E(||X_i-q||_p)}$, is discussed, which is a modification of the condition $var(\frac{D(X_i,q)^p}{E[D(X_i,q)^p]})$ in Beyer's work. If $\frac{\sqrt{Var(||X_i-q||_p)}}{E(||X_i-q||_p)} \to 0$, NN search will become meaningless. The following theory reveals the relationship between relative variance and relative contrast.

Theorem 9.4.4. *In (7.5), if* $\sigma' \to 0$ *(e.g.,* $d \to \infty$*),*

$C_r \approx \frac{1}{1+\phi^{-1}(\frac{1}{n})\frac{1}{p}\frac{1}{d^{1/2}}\frac{\sigma_j}{\mu_j}}$.

Proof: *If* σ' *is very small, for example,*

$$\phi(\frac{-1}{\sigma'}) \ll \frac{1}{n},$$

then in Theorem 2.1, we can omit $\phi(\frac{-1}{\sigma'})$ *and then we can get*

$$C_r = \frac{D_{mean}}{D_{min}} \approx \frac{1}{(1+\phi^{-1}(\frac{1}{n})\sigma')^{\frac{1}{p}}}.$$

Moreover, note that

$$\phi^{-1}(\frac{1}{n})\sigma' \gg \phi^{-1}(\phi(\frac{-1}{\sigma'}))\sigma' = -1$$

In other words, $\phi^{-1}(\frac{1}{n})\sigma'$ *is a negative number with very small absolute value, so we can further apporximate the result as*

$$(1 + \phi^{-1}(\frac{1}{n})\sigma')^{\frac{1}{p}} \approx 1 + \frac{1}{p}\phi^{-1}(\frac{1}{n})\sigma'.$$

If we have i.i.d assumption for each dimension, then

$$\sigma' = \frac{1}{d^{1/2}}\frac{\sigma_j}{\mu_j}.$$

And hence

$$\frac{D_{mean}}{D_{min}} = \frac{1}{1 + \phi^{-1}(\frac{1}{n})\frac{1}{p}\frac{1}{d^{1/2}}\frac{\sigma_j}{\mu_j}}.$$

From Theorem 9.4.4, we see when $\sigma' \to 0$ (e.g., $d \to \infty$), the relative contrast monotonically depends on $\frac{1}{p}\frac{1}{d^{1/2}}\frac{\sigma_j}{\mu_j}$, which equals to "relative variance" as in Theorem 9.4.3.

Though relative variance have been used as a measurement of contrast before, our work is the first one that explicitly discovers the relationship between relative variance and relative contrast, and hence connects it to the complexity of approximate NN search like LSH.

To summarize, most of the known analysis can be derived as special asymptotic cases (when $\sigma' \to 0$, e.g., $d \to \infty$) of the proposed measure with the focus on only one or two data properties. In contrast to the existing works, the proposed relative contrast can be utilized to analyze how NN search is affected by various data properties in not only asymptotic but also non-asymptotic cases.