Dissertation

Unsupervised Binary Code Learning for Approximate Nearest Neighbor Search in Large-scale Datasets

Submitted by

Hao Zhang

Department of Computer Science

In partial fulfillment of the requirements

For the Degree of Doctor of Philosophy

Colorado State University

Fort Collins, Colorado

Spring 2016

Doctoral Committee:

Advisor: Ross Beveridge
Co-Advisor: Bruce Draper

Charles Anderson
Yongcheng Zhou

Abstract

Unsupervised Binary Code Learning for Approximate Nearest Neighbor Search in Large-scale Datasets

Nearest neighbor search is an important operation whose goal is to find items in the dataset that are similar to a given query. It has a number of applications such as content based image retrieval (CBIR), near duplicate image detection and recommender systems. With the rapid development of the Internet and digital devices, it becomes easy to share and collect data. Taking a modern social network as an example, Facebook was reported in 2012 to be collecting more than 500 terabytes of text, images and videos each day. Conventional nearest neighbor search using linear scan becomes prohibitive when dealing with large-scale datasets like this. This thesis proposed a new quantization-based binary code learning algorithm, called Unit Query and Location Sensitive Hashing (UnitQLSH), to solve the problem of approximate nearest neighbor search for large-scale, unsupervised and unit-length data. UnitQLSH maps each high dimensional data sample to a binary code constrained to be residing on the unit-sphere. This constraint is very helpful in improving the retrieval performance. Also, UnitQLSH takes advantage of the approximate linearity of local neighborhoods of data to further improve performance. Moreover, given a query, a weight vector is computed based on it, indicating the significance of different bits. The Hamming distances are weighed by this vector to provide much more accurate retrievals than traditional approaches without any weighting schemes. Compared to existing state-of-the-art approaches, the proposed algorithm outperforms them significantly.

Acknowledgements

The completion of this thesis would not have been achieved without the support from my advisors, my family and my friends. I would like to first give my gratitude to my advisor, Dr. Ross Beveridge for his selfless support and great mentoring. He is a very nice and smart person with a devotion to research and teaching. I benefit a lot from him in our countless research meetings. He especially showed his super strong personality when facing a family emergency. It is beyond my imagination how he managed nicely to teach and take care of his family member at the same time. He is not only my research tutor, but also my life tutor. I would also like to thank Dr. Bruce Draper for his guidance in my research. He is highly knowledgeable about different research fields so he is able to give me helpful advice when I am lost or stuck in my research. Like Dr. Beveridge, Dr. Draper is also very considerate of his students, which I very much appreciate.

I want to thank my parents for raising me. I know it must be very difficult for them to let their child go abroad, but they let me go anyway. They have sacrificed a lot for me. I want to express my special appreciation to my mother, who teaches me to be an honest and friendly person. She cooks for me, cuts fruit for me, washes clothes for me and does everything she can to make my life warm and comfortable. She is the greatest mother one can ask for.

Also, I would like to thank my girlfriend, Zichun Xu, for her caring and unconditional love. Pursuing a PhD degree is very hard. The hardest part is not figuring out a solution to a problem, but loneliness, especially when you are studying abroad. Her love is a precious gift that I will treasure till the end of time.

Finally, I want to thank my colleague and friend, Quanyi Mo. He is a smart and fun person to work with. Discussions with him about research are actually the reason why I chose this topic for my PhD thesis. Besides, we also play games and have dinner together all the time. Those are the happy days that I will remember forever.

This dissertation is typset in LaTeX using a document class designed by Leif Anderson.

Table of Contents

List of Figures

CHAPTER 1

Introduction

With the rapid development of the Internet and digital devices, data are collected, manipulated and shared on a scale unimaginable 20 years ago. Taking a modern social network as an example, Facebook was reported in 2012 to be collecting more than 500 terabytes of text, images and videos each day [1]. The term "big data" became popular to describe this explosion of data, characterizing datasets that are too large to be processed using traditional techniques [2]. Nearest neighbor search is one of the operations affected by "big data". Its goal is to find items in the dataset that are similar to a given query. This is important since there are many applications that relate to nearest neighbor search, such as content based image retrieval (CBIR), duplicate detection and recommender systems. A conventional way to do nearest neighbor search of a given query is to perform a linear scan through all items in the dataset and retrieve the most similar ones. However, the computational complexity of a linear scan has the same order as the number of data samples in the dataset. When it comes to large-scale datasets which contain millions, billions or even more data samples, this procedure is prohibitive because scanning all the items can be extremely time-consuming. Therefore, more efficient methods are needed to handle this problem in practice. This chapter contains a detailed description of the problem tackled in this thesis and a brief summary of important solutions. The goals of this thesis are also presented in this chapter.

1.1. An Overview of Approximate Nearest Neighbor Search Problem

This section introduces the approximate nearest neighbor search problem. More specifically, it first describes how to define the nearest neighbors of a data sample. Then it introduces what approximate nearest neighbor search is and why it is important.

1.1.1. The Definition of Nearest Neighbors. The data we use in this thesis are called unsupervised data, i.e., data samples not associated with any additional metadata (e.g., a label indicating its category). Given a query, its nearest neighbors are conventionally defined as the closest data samples in terms of certain distance metrics. Euclidean distances are the most widely used metric for high dimensional data. Other options are also available, such as inner product and city-block distance. On the other hand, data are called supervised if they are associated with metadata. However, collecting metadata is a time-consuming task.

It is very convenient and practical to collect unsupervised data. For example, people can use their cell phones or digital cameras to take pictures very conveniently and then upload them online. Most of the time, the pictures are not tagged or labeled. Because collection does not require people to generate labels, unsupervised data now accounts for most of all the existing data. As a result, searching unsupervised data is a much more practical and common scenario. Due to the popularity and importance of unsupervised data , this thesis will solely focus on unsupervised data and the nearest neighbors are also defined in an unsupervised manner accordingly.

1.1.2. Definition of Approximate Nearest Neighbor Search and Its Importance. The goal of nearest neighbor search is to search a dataset for nearest neighbors of a given query or multiple queries. The data samples and the queries are often represented as a vector of numbers. This indicates that data often live in a high dimensional space. The desired nearest neighbors of a query are considered to be similar to it. A conventional way to find nearest neighbors of a query is to go through all data samples in the dataset and pick the samples with minimal distances to the query.

Nearest neighbor search is very important because it is so widely used. One example is Content Based Image Retrieval (CBIR) [3]. CBIR is needed when there is no metadata

associated with a query image and one wants to find other images similar to it (e.g., product search and art collection search). Another example is near duplicate image detection [4]. A large number of images online are duplicates or near duplicates of others. Detecting this redundancy will greatly help to reduce storage and processing overheads. Other examples of nearest neighbor applications are recommender systems, data compression and spell checking.

Despite the simplicity of the conventional nearest neighbor search method, it has encountered an obstacle. The datasets that are searched in response to the query are usually very large. Therefore, it is not practical to scan all items in the datasets in order to find similar ones. A workaround for very large datasets is that, most of the time, it suffices to obtain only approximately accurate solutions of nearest neighbors rather than the exact solution. As a result, much faster and memory-efficient methods can be developed, e.g., [5–7]. This is called approximate nearest neighbor (ANN) search. This problem and its solutions will be the focus of the study in this thesis.

1.2. An Overview of Existing Approaches for ANN Search

To enable efficient ANN for large-scale datasets, two main issues need to be addressed. One is excessive memory consumption. It is highly desirable that all the data samples can be loaded into memory so that search can be performed without involving hard disks in data exchange, which is much slower than accessing data from memory. However, it is often impossible to directly load all of data samples into memory. Therefore, a compact representation is needed for each data sample to resolve this issue. The other issue is that search should not go through all data samples. In other words, a linear scan must be avoided and the search complexity should be sublinear (e.g., $log_2(n)$ or a constant complexity, where

n is the number of all data samples). Therefore, finding a way to directly locate a small subset of the data samples close to a given query is very important. This section presents an overview of existing approaches for ANN search.

Due to the importance of ANN search, a number of methods have been proposed to enable efficient search in large-scale datasets. Tree-based methods [8] construct a tree that tries to partition data efficiently so that data samples in the same partition are similar to each other. In theory, finding nearest neighbors of a new query is in logarithmic expected time if the tree is approximately balanced. However, learning such trees can be very time-consuming. Also, the nearest neighbor search for high dimensional data can degrade to a linear scan when many data samples are close to the partitioning boundary. Clustering based methods, related to product quantization [7, 9], partition data samples into multiple orthogonal subspaces and then run clustering in each subspace. The resulting cluster centers are treated as quantizers to represent all data samples. Learning these quantizers can take a long time since clustering needs to be run in each subspace. Moreover, there can be large memory overhead due to the storage of lookup tables that contain the distances between the query and quantizers. Also, there can be computation overhead due to the generation of nearest neighbor candidates from these lookup tables.

Besides the two aforementioned categories of methods, binary code learning approaches [10–19] are the most popular and dominant. The general idea is to learn a binary representation of each data sample and then compute the similarity of pairs of data samples using the Hamming distance between their binary codes. The Hamming distances between pairs of data samples are expected to approximate their true relative distances. In other words, similar data samples should have similar binary codes and dissimilar samples dissimilar binary codes.

Binary code learning techniques have two main advantages, offering a solution to both the memory and retrieval efficiency issue mentioned above. One advantage is efficient data storage, since a binary code consumes only a few bits of memory compared to a real-valued vector representation of a data sample (Each real value takes 32 or 64 bits.). This makes it feasible to load a whole dataset into memory to perform search. The other is efficient search of the Hamming ball centered at the query. Given a query, usually its binary code is first computed. Then items in the dataset are retrieved based on the ranking of their Hamming distance to the query until enough items are retrieved. In practice, only data samples with very similar binary codes to the query will be explored. For instance, one will only explore binary codes whose Hamming distance to the query is within 3 and retrieve data samples in those codes. This is important since most samples are never compared to the query, making the search time independent of the size of datasets.

The upper half of Figure 1.1 shows an overview of the ANN search problem: Given a query and a large-scale dataset, find the items in the dataset that are most similar to it. In this figure, we set the range of similarity scores to be between 0 and 1, where larger values denote a higher similarity. The values in red are similarity scores between the query and the retrieved items. The lower half of Figure 1.1 shows how general binary code learning techniques work. Both the query and the items in the dataset are mapped to binary codes as shown below each item. Items with small Hamming distances to the query are retrieved using hash tables whose keys are the binary codes. Ideally, the Hamming distances between pairs of data samples should reflect their similarity. As a result, the ranking based on Hamming distances should be the same as or close to the ranking based on the similarity score to enable fast and accurate retrieval. Since ANN algorithms only provide approximate

answers, Hamming distances cannot reflect the exact real similarities. So data samples with the same Hamming distance to the query can have different real similarities.

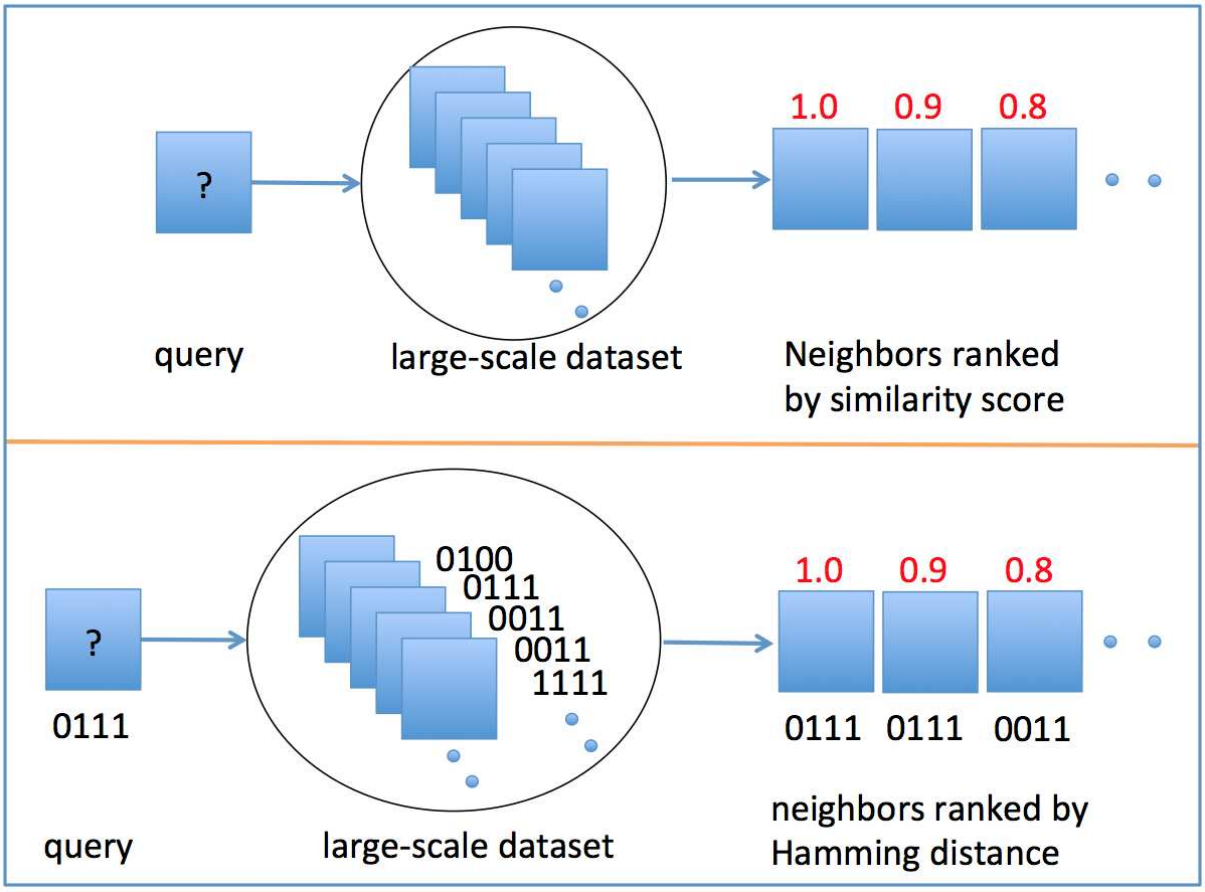


FIGURE 1.1. An overview of the nearest neighbor search problem and how general binary code learning techniques work

1.3. AN OVERVIEW OF THE PROPOSED APPROACH

There exist some practical constraints of the binary code learning approaches. To make the representation compact and the retrieval fast in practice, the number of bits in the binary code representation cannot be large because the number of binary codes increase exponentially as the number of bits increases. This will lead to a very small number of items to retrieve when only a small set of binary codes close to query are explored since most of the binary codes will associate with no data samples. However, current binary learning

algorithms tend to have low retrieval performance when the number of bits is small. This is one of the common weaknesses with algorithms in this category. The other weakness is a coarse ranking of the binary codes. For example, given a specific 32-bit binary code of a query, there exist $\binom{32}{3} = 4960$ binary codes with a Hamming distance of exactly 3 to the query. There is no further way to distinguish these binary codes. Therefore, all these 4960 codes need to be treated equally. Otherwise, one needs to perform some post processing to give them a finer ranking, which can lead to poor performance.

In this paper, we propose a method named Unit Query and Location Sensitive Hashing (UnitQLSH) to perform fast retrieval of nearest neighbors in large-scale datasets. This approach falls in the category of binary code learning methods. The proposed method is designed to work on unit-length data. It learns a binary code for each data sample. The binary codes are a concatenation of two parts. One part indicates the index of a cluster and the other indicates a quantizer in that cluster. Details of this design will be presented later in this paper. Given a new query, UnitQLSH is able to quickly and accurately compute a finely-ranked set of binary codes to be explored. Data samples retrieved from these explored binary codes are its approximate nearest neighbors.

UnitQLSH overcomes the two aforementioned issues. It takes full advantage of the benefits brought by neighborhood localization (location sensitive) which decompose the whole dataset into a modest number of local neighborhoods, also known as clusters. UnitQLSH also intelligently computes a weight vector to compute a weighted Hamming distance in order to more finely rank binary codes. Moreover, instead of making the binary code unit-length after they are learned (as in [16]), we found that adding a unit-length constraint on the binary codes during the learning process helps improve retrieval performance significantly. Compared to other state-of-the-art methods on 3 large-scale datasets, each of which contains

at least 1 million data samples, UnitQLSH outperforms them by a significant amount at a various range of code lengths.

To give a general yet informative idea why UnitQLSH works much better than others, we show in Figure 1.2 the 2-bit binary codes learned on 2-D unit-length data using three algorithms: ITQ, AQBC and UnitQLSH. ITQ and AQBC are two well known hashing algorithms for nearest neighbor search. UnitQLSH is the method this paper will propose. The data points are marked as blue and the learned binary codes are marked as red. Each red dot corresponds to a binary code. These red dots are also called quantizers since the binary code of each data sample is computed as the binary code of the closest red dot.

We can see from the left plot that since ITQ is not designed specifically for unit-length data, it still tries to find a cube whose vertices yield the least representation error. As a result, two of the learned quantizers are far away from data samples and only the other two quantizers are actually effective, leading to large quantization error. In the middle plot, we see only three quantizers for AQBC since it uses the origin as a fixed quantizer and the origin is never used to quantize unit-length data. That's why the origin is not shown in the plot. We observe that all of the three quantizers are not close the data, which means that the quantization error is large. The right plot shows the binary codes learned by the proposed UnitQLSH. The data are first partitioned to two local pieces, each of which can be much more closely represented by a linear subspace. Then we learn a hyper-rectangle (in this case, a 1-D rectangle or a line segment) in each of the local linear subspace. The binary quantizers are guaranteed to be unit-length (i.e., residing on the unit hypersphere). We can see that the learned quantizers using UnitQLSH approximate the data samples very well. It should be noted that another major contribution of UnitQLSH, using a weight vector to rank binary codes, is not presented in the plot since that is done during retrieval.

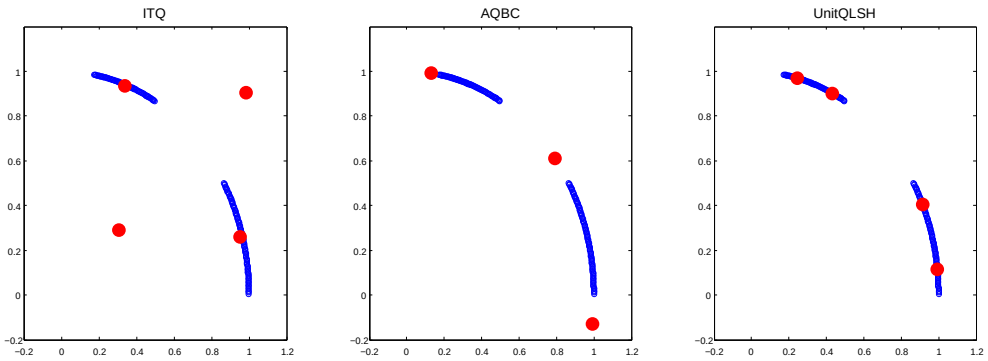


FIGURE 1.2. Binary codes (red dots) learned using ITQ, AQBC (The origin is the fourth binary code. It is omitted since it is never used to encode data.) and UnitQLSH

CHAPTER 2

Related Work

There has been a rapid development of large datasets nowadays thanks to cheap and convenient data acquisition equipments. How to find approximate nearest neighbors efficiently in large-scale datasets is very useful yet challenging. As a result, a large collection of approximate nearest neighbor search algorithms for large-scale datasets have been proposed. The work can be categorized based upon different criteria:

- Whether data are associated with category labels:
  - unsupervised nearest neighbor search
  - semi-supervised nearest neighbor search
  - supervised nearest neighbor search
- How the learning is performed (optimization objectives):
  - similarity based learning
  - quantization based learning
- Types of hash/binary functions:
  - linear hash functions
  - nonlinear hash function (e.g., kernel-based learning)

To measure the performance of an ANN algorithm, there are two common protocols. One is the recall@R curve. It reports the recall scores when varying the number of retrieved items. Basically, it measures how often the retrieved items are true nearest neighbors. Recall@R curves do not consider the number of binary codes that are retrieved in order to obtain the desired number of R (retrieved items). This is an important factor since it is related to the effort spent in retrieval. As a result, a second protocol is also common: recall/precision at a

fixed Hamming radius. This protocol fixes the number of binary codes to retrieve and report the corresponding retrieval performance. However, this protocol still fails to fully measure the retrieval efforts. An important discussion will be presented later in this paper that shows how to fully measure retrieval efforts.

Since this thesis focuses on unsupervised nearest neighbor search, several representative algorithms in this category will be reviewed and summarized in this section. The algorithms are divided based on whether the they are similarity based or quantization based learning. Section 2.1 presents algorithms that learn binary codes based on similarity of pairs of data samples. Section 2.2 presents algorithms seeking binary codes that can quantize/approximate individual data samples. Concluding remarks of both the advantages and disadvantages of these methods will be presented at the end of this section.

2.1. Similarity Based Binary Code Learning Algorithms

Multiple representative similarity based binary code learning methods are reviewed in this section. Most binary code learning approaches reviewed here seek binary representations of data such that their Hamming distance matrix well approximates a target distance matrix containing distances between pairs of data samples. Note that a distance matrix can be easily converted to a similarity matrix by negating each entry. For unsupervised scenarios, the target distances between pairs of data samples are often defined as Euclidean distances, inner product or Radial Basis Functions (RBF).

2.1.1. Locality Sensitive Hashing.

Gionis et al. proposed Locality Sensitive Hashing (LSH) [10] to perform approximate ANN search. Their goal is to ensure that close data samples will have a higher collision (mapped to the same binary code) probability than data samples that are far apart. The corresponding hash functions are called locality sensitive

functions. These locality sensitive functions have a formal mathematical form. Denote $\mathcal{H}$ a family of functions that map data samples in a set S to some universe U. $\mathcal{H}$ is called (p_1, p_2, r_1, r_2) sensitive for a distance metric $D(*,*)$ if any $x_1, x_2 \in S$ satisfies the following:

- if $x_1 \in B(x_2, r_1)$, then $Pr_{\mathcal{H}}[h(x_1) = h(x_2)] \geq p_1$
- if $x_1 \notin B(x_2, r_2)$, then $Pr_{\mathcal{H}}[h(x_1) = h(x_2)] \leq p_2$

Here, $B(x_2, r)$ denotes all points in S that are within a distance r to x_2 where the distance function is defined by $D(*,*)$. $Pr_{\mathcal{H}}[h(x_1) = h(x_2)]$ denotes the probability of choosing a hash function $h()$ from $\mathcal{H}$ such that $h(x_1) = h(x_2)$. The cases where x_1 and x_2 are mapped to the same hash code $(h(x_1) = h(x_2))$ is called collision. To make these locality sensitive functions useful, it is always required that $p_1 > p_2$ and $r_1 < r_2$.

Although LSH is classified as a similarity based learning method in this thesis, unlike other similarity based methods introduced later, it does not aim to learn hash functions that approximate *predefined* similarity scores between pairs of data samples. What LSH does is to randomly select hash functions from a family of functions that yield high collision probability for data samples that are close to each other.

In the case where $D(*,*)$ is defined as Euclidean distance, the hash functions are linear functions and the parameters of these linear functions are drawn independently from a normal distribution [20]. Therefore, these linear hash functions are independent of the size of the dataset, so training is very fast. Unfortunately, long binary codes are necessary to achieve reasonable search performance of LSH. Hundreds of bits are needed for LSH to achieve good performance while other methods such as SH [11] only need 32 bits.

An obvious drawback of LSH is that the linear hash functions do not take into consideration the distribution of data. In other words, the hash functions are data-independent or there is no learning involved in it. If hash functions can be learned in a way that adapts to

the distribution or structure of the data, more compact binary codes yielding better retrieval performance may be learned.

2.1.2. Spectral Hashing. Weiss et al. proposed Spectral Hashing (SH) [11] to learn binary codes such that Hamming distance strongly correlates to the similarity of pairs of data samples. Specifically, they use a Radial Basis Function (RBF) to define the similarity W_{ij} of the i-th and j-th data sample:

$$W_{ij} = exp(-||x_i - x_j||^2/\sigma^2) \tag{1}$$

where x_i and x_j denote the i-th and j-th data sample and σ controls the threshold that defines similar data samples. Denote y_i as the binary code (a row vector) of i-th data sample, the objective function they are trying to optimize may be written as follows:

$$\begin{aligned} minimize: \quad & \sum_{i,j} W_{ij}||y_i - y_j||^2 \\ \text{subject to} \quad & y_i \in \{-1, 1\}^k, \\ & \sum_i y_i = 0, \\ & \frac{1}{n}\sum_i y_i^T y_i = I_{k\times k} \end{aligned} \tag{2}$$

where k is the number of bits in a binary code, n is the number of data samples and $I_{k\times k}$ is an identity matrix of size $k \times k$. This is a standard optimization function in spectral analysis except for the constraints. To understand this equation, we can look at W_{ij} as a weighted penalty imposed on the Hamming distance $||y_i - y_j||^2$. If two data samples are very similar to each other (large W_{ij}), we would like their Hamming distance to be small. Therefore, a

large penalty should be used to weight their Hamming distance. The constraint $\sum_i y_i = 0$ means that the each bit will partition all data samples evenly (Each bit will "fire" 50% of the time.). The intent is to maximize the information carried by each bit. The other constraint $\frac{1}{n}\sum_i y_i^T y_i = I_{k\times k}$ means that pairs of bits are uncorrelated to each other. This constraint is to reduce redundancy among bits.

Equation 2 can be written more elegantly by using linear algebraic notations. $\sum_{i,j} W_{ij}||y_i - y_j||^2$ is equal to $trace(Y^T(D-W)Y)$, where Y is an $n \times k$ matrix whose i-th row is y_i, D is an $n \times n$ diagonal matrix whose i-th diagonal entry $D(i,i) = \sum_j W(i,j)$. D is called a degree matrix in graph theory. W can be treated as the adjacency matrix of a weighted graph. After they relax the problem by removing the binary constraint of Y, the solutions can be easily obtained via computing eigenvectors of the Laplacian matrix $L = D - W$ corresponding to minimal nonzero eigenvalues. These eigenvectors are also called the spectral embedding of this weighted graph represented by W. The binary form of Y is simply obtained by taking the signs of the eigenvectors.

The method described above can only be used to compute the binary codes of training data. To compute the binary code of new data samples (also known as out-of-sample extension), the Nystrom method [21] is usually applied. However, it can be very computationally expensive if the number of data samples is large, which is often the case for ANN problems. To obtain efficient out-of-sample extensions, the authors assume that data are sampled from separable multidimensional uniform distributions after data are rotated by PCA. Then they choose from all analytical eigenfunctions learned in each single dimension to obtain ones with the smallest eigenvalues. The chosen eigenfunctions are thresholded at zero to provide binary codes of data.

A major drawback of SH is its strong assumption that data come from a separable multidimensional uniform distribution. Although this assumption makes it efficient to compute binary data of new data samples, it is often not true of real data. Therefore, retrieval performance on real data can be poor. Another drawback is a highly dynamic choice of σ which controls the radius in the RBF kernel. The authors mentioned in their released code that σ has to change every time the number of bits changes. This requires a lot of additional efforts to tune parameters and such changes are very expensive when datasets are large. A third drawback is the relaxation of the problem by dropping the binary constraints. No work has been done to show that this relaxation results in solutions approximate to the real solutions. Therefore, whether it will provide good approximate solutions to Equation 2 is unknown.

2.1.3. Anchor Graph Hashing. In order to relax the strong assumption made in SH that data have a separable multidimensional uniform distribution, Liu et al. [12] proposed Anchor Graph Hashing (AGH) which can efficiently compute eigenvectors of a Laplacian matrix and easily generate binary codes for new data samples.

AGH works by first running K-means [22] clustering on a small subset of training data with a few iterations to quickly obtain a small number of cluster centers called anchors. Each data sample x_i is than represented as a new feature z_i using these anchors. Denote $Z \in \mathcal{R}^{n \times m}$ as the new data matrix where each row is this new feature of a data sample. n is the number of data samples and m is the number of anchors. The i-th row and j-th column of Z is computed by the following equation.

$$Z_{ij} = \begin{cases} \frac{exp(-D^2(x_i,u_j)/t)}{N_i} & \forall u_j \text{ close to } x_i \\ 0 & \text{otherwise} \end{cases}$$

where u_j is the j-th anchor. u_j is considered to be close to x_i if it is among the top s anchor points that are closest to x_i. $D^2(x_i, u_j)$ denotes the Euclidean distance between x_i and u_j. N_i is a normalizer for the i-th row of Z such that the entries in the i-th row sum up to 1. Basically, this equation computes a new representation of each data sample using its similarity to each of the anchor points. If an anchor u_j is far from the data sample x_i, the similarity is 0. Otherwise, the similarity is computed as $\frac{exp(-D^2(x_i,u_j)/t)}{N_i}$.

The purpose of Z is to approximate the adjacency matrix. Let $\Lambda = diag(Z^T \mathbf{1}_{n\times 1}) \in \mathbb{R}^{m\times m}$ where $\mathbf{1}_{n\times 1}$ denotes an $n \times 1$ matrix (vector) of all 1's. $\hat{A} = Z\Lambda^{-1}Z^T$ is an approximation to the true adjacency matrix A [23]. More importantly, $\hat{A}$ is a low rank matrix (maximal rank is m) and each of its columns or rows sum up to 1. As a result, an $n \times n$ identity matrix $I_{n\times n}$ is its degree matrix. Therefore, the graph Laplacian can be approximated as $L = I_{n\times n} - \hat{A}$.

It is easy to prove that if v is an eigenvector of L and λ is the associated eigenvalue, then v is also an eigenvector of $\hat{A}$ with an eigenvalue of $1 - \lambda$. Therefore, to get r graph Laplacian eigenvectors, one only needs to solve for the eigenvectors of $\hat{A}$ corresponding to r largest eigenvalues (ignoring eigenvalue 1 since it corresponds to zero eigenvalues of L). Furthermore, the low rank property of $\hat{A}$ can be used to speed up the eigenvector computation of L. Instead of solving eigenvectors of $\hat{A}$ which is of size $n \times n$, the authors solve for the eigenvectors of an $m \times m$ matrix $M = \Lambda^{-1/2}Z^T Z\Lambda^{-1/2}$. Let the result be an $m \times r$ matrix V, each column of which is a chosen eigenvector. Also, let $\Sigma \in \mathbb{R}^{r\times r}$ be a diagonal matrix where the diagonal entries contain the corresponding eigenvalues. The final spectral embedding matrix E of L can be computed as

$$E = \sqrt{n}Z\Lambda^{-1/2}V\Sigma^{-1/2} = ZW \tag{3}$$

where $W = \sqrt{n}\Lambda^{-1/2}V\Sigma^{-1/2}$. The final binary code associated with each data sample is obtained by thresholding E at zero: $sign(ZW)$. As shown in Equation 3, the process of computing the graph Laplacian eigenvectors E of L can be seen as first computing a new representation Z of data samples using anchor points and then projecting the resulting representation on W.

A very important contribution of this work is that the authors show that for a new data sample, its binary code can also be computed by first computing its anchor representation and then projecting it on W followed by thresholding at zero. In other words, the binary code for new data samples can be computed as $sign(Z_tW)$, where Z_t is the anchor representation of the new data samples. Therefore, the out-of-sample generalization is computed very efficiently.

Another contribution of this work is that they propose a hierarchical hashing by assigning 2 bits for each eigenvector instead of using one eigenvector for each bit. It is because that not all the chosen eigenvectors (bits) are equally important: The eigenvectors of L corresponding to low eigenvalues are more important than others [24]. The authors show how 2 bits can be associated with each eigenvector to alleviate this bias. The procedure is as follows. The first bit is simply obtained by thresholding the eigenvector. However, some points close to each other in the original space can be assigned with different bit values when the they are close to the thresholding boundary. The second bit is used to mitigate this problem by assigning these points the same bit values. Specifically, there are two thresholds corresponding to the second bit. One threshold is for further thresholding positive values computed by the first bit (before thresholding) and the other is for the negative values. These two thresholds are still obtained by finding the spectral embedding of the approximate adjacency matrix $\hat{A}$.

Their experiments on NUS-WIDE and MNIST show that AGH is fast in terms of computing binary codes of test data. It also yields higher Mean Average Precision (MAP, which will be described later in this thesis) compared to SH, LSH and so on. Moreover, the authors show that AGH is capable of retrieving semantically similar items, which is possibly an outcome of using their anchor representation.

AGH has three disadvantages. One is that the training time of AGH is long, which can be a problem when applying this method to large-scale datasets. The second disadvantage is that how well the anchor graph $\hat{A}$ can approximate the true adjacency matrix is unknown. The authors did not provide any experiments to measure the approximation error. The third disadvantage is the relaxation that binary constraints are replaced by signed magnitude. Similar to the issue of SH, although this relaxation makes the problem tractable, there is no guarantee this relaxation will provide sufficiently good solutions.

2.1.4. Binary Reconstructive Embeddings. Kulis et al. [13] proposed Binary Reconstructive Embeddings (BRE) to learn binary hash codes for ANN problems. The main idea of BRE is to find hash functions such that Hamming distances between pairs of samples are as close to their Euclidean distances as possible. Denote the data samples as $x_1, x_2, \ldots, x_n$, where n is the total number of data samples. Also denote b hash (binary) functions as $h_1, h_2, \ldots, h_b$, where b is the number of bits. Note that each hash function h_i produces only one bit. The authors design the hash functions similar to the representation of the classification boundary in a kernel Support Vector Machine (SVM) [25]. Specifically, the p-th bit of a data sample x is designed as follows:

$$h_p(x) = sign(\sum_{q=1}^{s} W_{pq}\kappa(x_{pq}, x)) \tag{4}$$

where W is a $b \times n$ matrix, $\kappa(*,*)$ denotes a specific kernel function and $x_{pq}(q = 1, \ldots, s)$ are s data samples corresponding to h_p. These s data samples can be related directly to the support vectors of an SVM. Although the authors do not mention how the samples x_{pq} are chosen in their paper. In their released MATLAB code, x_{pq} are chosen randomly and they set $s = 50$. The binary representation of x is denoted as $\tilde{x} = [h_1(x), h_2(x), \ldots, h_b(x)]$. To achieve the goal that the Euclidean distances between pairs of data samples are well approximated by their Hamming distances, the authors propose the following objective function:

$$\underset{W}{\operatorname{argmin}} \sum_{(i,j)\in\mathcal{N}} \Big(d(x_i, x_j) - \tilde{d}(x_i, x_j)\Big)^2 \tag{5}$$

where $\mathcal{N}$ is a selection of pairs of points, $d(x_i, x_j) = \frac{1}{2}||x_i - x_j||^2$ and $\tilde{d}(x_i, x_j) = \frac{1}{b}||\tilde{x_i} - \tilde{x_j}||^2$. A detail about $\mathcal{N}$ is that, for each of the n training data samples, they choose k other data samples to pair with it. Therefore the total size of $\mathcal{N}$ is nk. All data samples are normalized to have length 1. It is clear that the goal of Equation 5 is to minimize the mean approximation error.

The authors then propose a coordinate-descent method to find a locally optimal solution to Equation 5. In each iteration, their method updates only one entry of W while fixing all others. In the paper, it is proved at length that updating all hash functions can be computed in $\mathcal{O}(nb(logn + k))$, which is considered reasonable from the authors' perspective.

The authors compare their method to LSH, SH and so forth on 6 datasets. They compute the precision of the retrieval results when the Hamming distance is less then or equal to 3. On four datasets, their approach yields the highest precision scores when varying the number of bits. On the other two datasets, SH achieves the highest performance and BRE performs very similarly to SH.

In my opinion, the most desirable property of BRE is that it can use any kernels to account for a nonlinear structure of real data. Although in the paper, a linear kernel is adopted for computational efficiency. A major drawback of BRE is the computation time. The authors have explicitly provided the time complexity of their training. Although BRE may be feasible for small datasets (still a long time), it is too expensive to be applied to large-scale datasets. Besides the need for long training times, the time used for computing query data can also be inefficient since it involves kernel computation (an issue similar to kernel SVM). Another disadvantage is that the data samples used in kernel computation (i.e., x_{pq}) are chosen randomly. There should be a more principled way to select these points, e.g., K-means.

2.2. Quantization Based Binary Code Learning Algorithms

In this section, multiple representative quantization based methods for binary code learning are reviewed in detail. The goal of quantization based methods is to learn binary codes such that the learned binary codes can approximate data samples well. A key difference between these methods and similarity based methods is that they focus on individual data samples rather than pairs of data samples. If the number of data samples is denoted as n, they only need to optimize over $\mathcal{O}(n)$ data samples rather than $\mathcal{O}(n^2)$ pairs. Therefore, these methods are generally more efficient. However, since they are focused on approximating each data sample, they are not as flexible when considering alternative distance metrics.

2.2.1. K-d Tree. K-d tree [8] is a classical and very well known method for nearest neighbor search. A k-d tree can be seen as a quantization based method. Its basic idea is to partition the space in a hierarchy way and the location of a query in the space can be

found quickly by comparing it to the dividing point at each level. The partition of the space is stored in a tree-based structure.

For k-dimensional data, the simplest construction of a k-d tree works as follows.

(1) Initialization: Set $i = 1$ and the current partition to the whole dataset.

(2) Select a data sample whose (i *mod* k)-th axis is the median of all data samples' (i *mod* k)-th axis in the current partition. This naturally further partitions the data samples in the current partition into 2 balanced parts.

(3) Set $i = i + 1$, either go back to the Step 2 or return when certain criterion is met.

The term (i *mod* k) means that there is a cyclic order of all axis's used for comparison and one axis can also be used multiple times. For example, if the data are 3D with axis X, Y and Z, then the order can be $X, Y, Z, X, Y, Z, X, \ldots$. It should be noted that it is not required to choose the median as a dividing point. Random dividing points can be used if computation complexity is a major concern. However, this can lead to unbalanced trees and can sometimes make search very inefficient. Picking the median in each partition yields balanced trees, but the time complexity of computing the median is linear in terms of the number of data samples. To circumvent this heavy computation, especially when the number of data samples is large, usually a random subsample of data samples are selected and their median is computed. This yields good balanced trees in practice. The leaves of a k-d tree are called buckets.

For nearest neighbor search, it is possible that if a query is close to the partitioning boundary, its nearest neighbors may lie on the other side of the boundary. These boundaries can be found by looking for boundaries that intersect a hypersphere centered at the query with a small radius. For each intersection (corresponding to a node in a k-d tree), a new

search has to be performed to check if nearest neighbors exist. If a lot of such intersecting boundaries exist, the search will be expensive.

Although the learning process is simple, the application of k-d tree is quite limited. The time complexity of constructing a tree is $\mathcal{O}(nlogn)$, meaning that it is not feasible for building k-d trees for large-scale datasets. More importantly, searching a k-d tree will degrade to a linear exhaustive search when the dimension of data is high because there exist a large number of those aforementioned intersecting boundaries that require a new search.

2.2.2. PCA Hashing. Wang et al. [14] proposed PCA Hashing (PCAH) for ANN search on semi-supervised data. Although their method assumes data are semi-supervised (partly labeled), it is more widely applied to unsupervised data. I will first present a detailed description of how PCAH is conducted on semi-supervised data and then show how it can be easily fit to unsupervised data.

The hash functions used in their work are linear functions. Specifically, $h_k(x) = sign(w_k^T x - b_k)$ where $h_k(x)$ denotes the hash function of k-th bit for a data sample x and $b_k = \frac{1}{n}\sum_{i=1}^{n} w_k^T x_i$, where n is the number of data samples. For zero-centered data, $b_k = 0$ and therefore $h_k(x) = sign(w_k^T x)$. The labeled data are presented, using pair information, in a matrix S of size $l \times l$

$$S_{ij} = \begin{cases} 1 & (x_i, x_j) \in \mathcal{M} \\ -1 & (x_i, x_j) \in \mathcal{C} \\ 0 & \text{otherwise} \end{cases}$$

where $\mathcal{M}$ is the set of pairs of similar data samples, $\mathcal{C}$ is the set of pairs of dissimilar data samples and l is the number of data samples associated with at least one of the similar or dissimilar pairs. Denote these data samples as X_l and the set of hash functions as W. Each column of X_l is a data sample and each row of W is the coefficients of a hash function (i.e.,

w). The objective function $J(W)$ aims at assigning similar data samples similar binary codes and vice versa:

$$\begin{aligned} \max J(W) &= \sum_k [\sum_{(x_i,x_j)\in\mathcal{M}} h_k(x_i)h_k(x_j) - \sum_{(x_i,x_j)\in\mathcal{C}} h_k(x_i)h_k(x_j)] \\ &= \frac{1}{2} tr[sign(W^T X_l) S sign(W^T X_l)^T] \end{aligned} \tag{6}$$

To obtain an easier optimization function, the authors further relax $J(W)$ by dropping the $sign()$ function:

$$J(W) = \frac{1}{2} tr(W^T X_l S X_l^T W) \tag{7}$$

Solving $J(W)$ only addresses the labeled data. The vast majority of samples are neither in $\mathcal{M}$ nor in $\mathcal{C}$. The motivation of PCAH to handle these unlabeled data samples is to maximize the information (entropy) carried by each bit. In other words, each hash function should not only assign the same bit to similar data samples and different bits to dissimilar data samples, but also partition the whole dataset evenly to maximize the information of each bit:

$$\max entropy(h_k(x)) \tag{8}$$

It is easy to show that Equation 8 is the same as maximizing the variance of each bit:

$$\max entropy(h_k(x)) \Leftrightarrow \max var(h_k(x)) \tag{9}$$

The maximal value is reached when the partition is balanced. Again, the $sign()$ function is dropped directly so that Equation 9 becomes

$$\max var(w_k^T x) \tag{10}$$

This term, written in a matrix form, is added to $J(W)$ as a regularizer:

$$J(W) = \frac{1}{2} tr(W^T X_l S X_l^T W) + \frac{\lambda}{2} tr(W^T X X^T W) \tag{11}$$

where X denotes the whole dataset (Each column is a data sample.).

Denoting $M = X_l S X_l^T + \lambda X X^T$, Equation 11 can be rewritten in a cleaner form:

$$\underset{W}{\operatorname{argmax}}\, tr(W^T M W) \tag{12}$$

Also, to obtain approximately independent hash bits, each hash function is expected to be orthogonal to each other, yielding $W^T W = I$. Equation 12 now becomes a standard PCA problem whose goal is to find linear projections that maximize data variance. The covariance matrix in this PCA problem is M. The bits are obtained by taking the sign of the PCA projected data.

It is easy to notice that projections found by PCA are not equally important in terms of maximizing Equation 12. The projections with larger eigenvalues are more important than the ones with smaller eigenvalues. This will lead to bits with different qualities. In other words, some bits are more important in deciding whether two data samples are close than other bits. To balance the importance of each bit, the authors propose to solve a

nonorthogonal relaxation of Equation 12:

$$\underset{W}{\operatorname{argmax}}\, tr(W^T M W) - \frac{\rho}{2}||W^T W - I||_F^2 \tag{13}$$

Equation 12 only requires the projections to be *approximately* orthogonal to each other. Therefore, bits with balanced importance can be obtained. When ρ is chosen properly, Equation 13 can be solved by setting its derivative to zero.

The solution described so far is for semi-supervised data. If data are unsupervised, the same procedure can still be used to obtain hash functions. The only difference is that $M = XX^T$ instead of $X_l S X_l^T + \lambda X X^T$. In other words, simply setting S to zero will yield the solution for unsupervised data.

Experiments are conducted on MNIST and GIST1M [7] datasets. The authors show that the orthogonal version of the solution yields inferior retrieval performance while the nonorthogonal version performs the best compared to other algorithms. In more detail, high precision is observed when looking at the top K (e.g., $K = 500$) retrieved neighbors or looking at binary codes within a small, fixed Hamming distance.

Although the proposed method is relatively efficient and yields better performance than some other approaches. It still has a few drawbacks. The first one is that there is a trade-off between orthogonality and balance of bits due to the nonorthogonality constraint. That is to say, although the bits are relatively balanced, the correlation between them may be high. In other words, there exists redundancy among the bits. The second drawback is the format of the linear hash functions. The authors fix the bias term to zero when the data is zero centered, which means the hash functions will always go through the origin. This is not an optimal choice for linear hash functions. Moreover, if data have a low-dimensional nonlinear structure, these linear functions do not have the ability to take advantage of it.

2.2.3. Iterative Quantization and a Kernelized Version. Iterative Quantization (ITQ) is an unsupervised hashing approach proposed recently by Gong et al. [15]. It is motivated by the orthogonal version of PCA Hashing (PCAH) proposed by Wang et al [14]. Before I talk about ITQ, let me first describe how orthogonal PCAH works.

The orthogonal version of PCAH projects data to a b-dimensional linear subspace (b: the number of bits in a binary code) using PCA and computes their binary codes using the sign of the projected data. In more detail, represent training data as $X \in \mathbb{R}^{n \times d}$, where n is the number of training samples and d is the number of dimensions of a data sample. PCA is applied to training data to obtain b principal components corresponding to the b largest eigen-values. They form a PCA projection matrix $P \in \mathbb{R}^{d \times b}$, where each column of P is a principal component. The training data are projected to this b-dimensional linear subspace: $V = XP$, where V is the projected data. The binary codes B are simply obtained by $B = sign(V)$.

A novel way to look at PCAH is through the perspective of quantization. Quantization is very help when one wants to replace/approximate many data samples with only a small number of data points. A good example of quantization is to use an integer to replace/approximate a floating number by rounding it. In the case of PCAH, by computing the binary codes using $B = sign(V)$, each data sample is replaced/approximated by its sign. For example, a 2D data sample $[0.8, -0.9]$ is approximated by $[1, -1]$. The code $[1, -1]$ is also called the quantizer of $[0.8, -0.9]$. It is easy to see that all possible b-bit binary codes form the vertices of a b-dimensional hyper-cube. For example, in 2D, there are all four possible 2-bit binary codes: $[-1, -1]$, $[1, -1]$, $[-1, 1]$ and $[1, 1]$, which are four vertices of a 2D square. If B is close to V, then the Hamming distance of data samples represented in B is generally close to the Euclidean distance of data samples represented in V (as well as

X since V preserves most of the data variances in X), which is the goal of unsupervised hashing. Therefore, now the problem can be cast as finding a b-dimensional hyper-cube such that the approximation error is minimized when each data samples is replaced/quantized by the closest vertex on that hyper-cube.

Figure 2.1 further illustrates quantization in a 2D space. The left plot of Figure 2.1 shows 2D synthetic data and their binary codes (red dots). The data are already aligned/rotated by PCA such that the first dimension is also the principal component corresponding to the largest eigen value, etc. data samples in each quadrant is quantized by the red dot in the same quadrant. Note that the 2D hyper-cube (square) is properly scaled to minimize the quantization/approximation error, though the scale is actually not relevant in retrieval since it does not change the rank of the distances between data samples. It can be seen that data samples very close to the boundary (x axis) get assigned to different bits. Therefore, the mean quantization error will be large for this example.

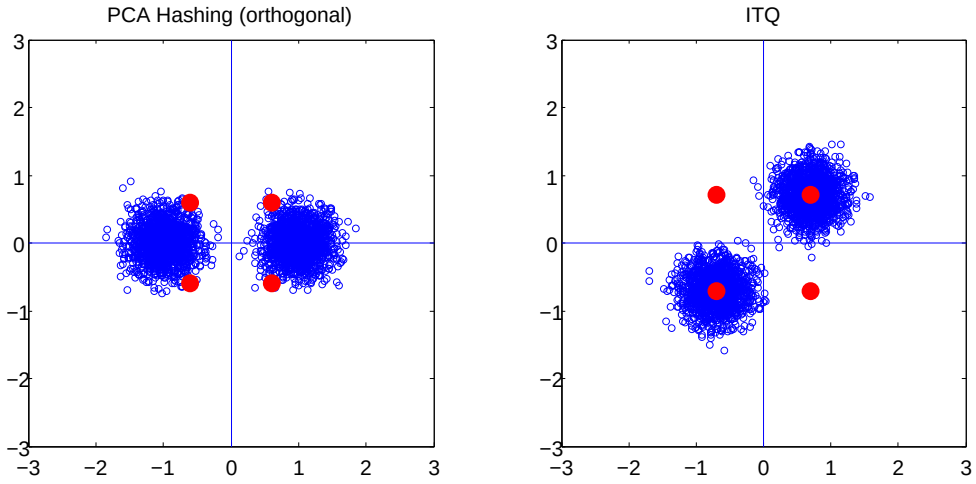


FIGURE 2.1. A comparison of PCA Hashing and ITQ

The key idea behind ITQ is to *rotate* the reduced data V such that the approximation error is minimized when each data sample is replaced by the closest vertex on that hyper-cube. As shown in the right plot of Figure 2.1, the same data are rotated and then converted to binary codes by taking the sign at each axis. The binary codes (red dots, also scaled properly) now better approximate data samples.

The procedure of ITQ for finding such binary codes is as follows. The first few steps are the same as PCAH, where reduced data V are obtained by PCA projection: $V = XP$. To find a rotation that yields binary codes with better quantization, the objective function is written as follows:

$$\underset{B,R}{\operatorname{argmin}} Q(B, R) = ||B - VR||_F^2, \tag{14}$$

where $R \in \mathbb{R}^{b \times b}$ is a rotation matrix and $||M||_F$ denotes the Frobenius norm of matrix M.

A greedy method is proposed to find B and V by updating one with the other one fixed. When R is fixed, B is computed as $B = sign(VR)$ so that each data sample is quantized as its closest vertex on the hyper-cube. When B is fixed, finding R is equal to an orthogonal Procrustes problem [26]. R is computed by first taking the SVD of $B^T V$ such that $B^T V = S_1 \Sigma S_2^T$ and setting R as $R = S_1 S_2^T$.

A great advantage of ITQ is computational efficiency. Since PCA is computed by taking the singular value decomposition (SVD) of the $d \times d$ data covariance matrix, it can be very quickly computed if the dimensionality of data is not too large, even for a very large number of data samples. Moreover, ITQ yields impressive retrieval performance compared to multiple state-of-the-art methods such as Spectral Hashing [11] and PCA Hashing [14] on CIFAR dataset [27] and a dataset containing 580,000 tiny images [28].

Although this approach yields good performance, it assumes that the data live in a single linear subspace. To deal with real-world scenarios where data exhibit a nonlinear structure, Gong et al. [5] subsequently proposed a kernel version of ITQ (KITQ) where data samples are transformed by Random Fourier Features (RFF) so that their dot product approximates an RBF kernel. Unfortunately, RFF requires many dimensions for a reasonable approximation, which increases the computational burden. More importantly, the performance gain of KITQ over ITQ is very limited. In many cases, KITQ performs even worse than standard ITQ.

2.2.4. Angular Quantization-based Binary Codes. In many applications, data samples are often represented as vectors of unit length whose l_2 norm is 1. In other words, they live on a unit hypersphere. For example, each entry of a vector representing a document can be the frequency of a particular word and this vector is often normalized to have length one. The similarity between pairs of such vectors is defined as their inner product, or equivalently, the cosine of the angle between them. Note that Euclidean distances and inner products are interchangeable for vectors of unit length. Traditional hashing methods can readily be applied to such data. However, they generally cannot take advantage of the fact that data reside on a unit hypersphere. Gong et al. [16] propose Angular Quantization-based Binary Codes (AQBC) to conduct fast search in unit-length data.

The basic idea of AQBC is to approximate each point on the unit sphere using a vertex of a unit hypercube such that the angle between the point and its approximation is minimal. Figure 2.2 shows this basic idea in a 2D case. AQBC deals with only data with all positive entries. The yellow curve is the positive orthant of the 2D unit (hyper-)sphere where data reside. The 2D unit (hyper-)cube is also shown where the coordinate of each of its vertices corresponds to a binary code naturally. The binary codes are denoted using red dots. Note

that since "00" has zero length, it is not used for approximation. A data point x is approximated using the binary code "11" since their angle is the smallest (the angle between the two dash lines). This is also equivalent to projecting each binary code to the 2D unit (hyper-)sphere and finding the projected code with the smallest Euclidean distance. In this case, the projections of "01" and "10" stay the same since they are already on the yellow curve and the projection of "11" is denoted as the green dot x'. In other words, every point on this yellow curve is quantized by three points on the same curve: "01", x', and "10".

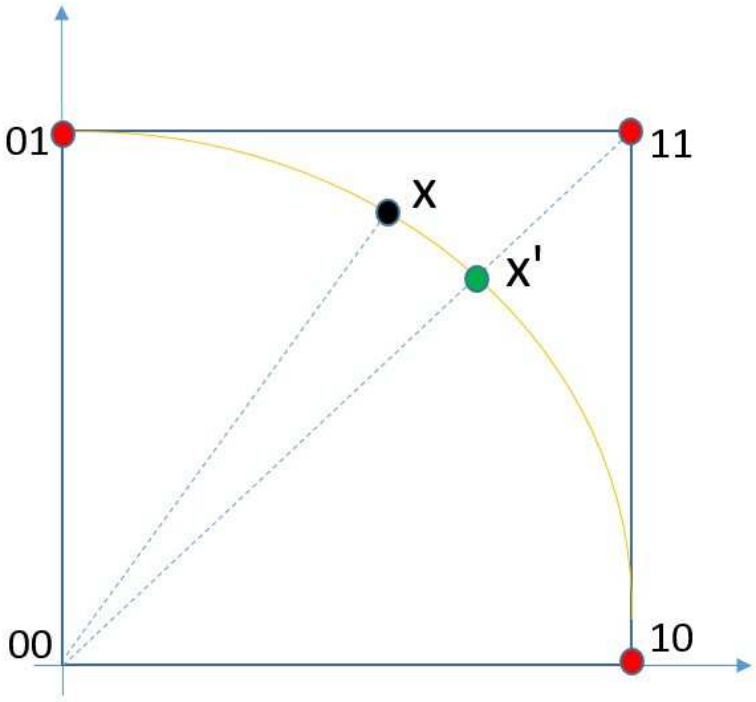


FIGURE 2.2. Basic idea of AQBC

Assuming that b is an arbitrary binary vector whose entries are either 0 or 1, then $||b||_1 = m$ and b' is another arbitrary binary vector with a Hamming distance of r from b. The authors show that the cosine of the angles between b and b' are bounded by $\left[\sqrt{\frac{m-r}{m}}, \sqrt{\frac{m}{m+r}}\right]$. Note that when r is fixed and m becomes large, the range of this bound, $\sqrt{\frac{m}{m+r}} - \sqrt{\frac{m-r}{m}}$, is smaller. This means that if b has a lot of 1's, b and its Hamming neighbors will be more densely distributed on the unit hypersphere. The reason is as follows. The number of binary

codes with a Hamming distance of r from b is fixed. When m is large, the range of the cosine of the angles between these binary codes and b becomes smaller. Due to a smaller range but a fixed number of binary codes in this range, the distribution of these codes is denser. Given this observation, the authors aim to assign dense regions on the hypersphere using binary codes with large l_1 value (large m). The idea is to rotate data such that dense regions will be close to binary codes with large m.

The objective function is:

$$Q(B,R) = \underset{B,R}{\operatorname{argmax}} \sum_{i=1}^{n} \frac{b_i^T}{||b_i||_2} \frac{R^T x_i}{||R^T x_i||_2} \qquad s.t. \quad R^T R = I_{c\times c} \tag{15}$$

where $b_i \in \{0,1\}^c$ is the binary code of i-th data sample x_i with d dimensions, c is the number of bits and $R \in d \times c$ is a matrix that performs dimension reduction and rotation at the same time. $I_{c\times c}$ is a $c \times c$ identity matrix. Equation 15 is quite straightforward. It aims to maximize the sum of the cosine of the angle between every rotated data point and its binary code.

To make Equation 15 easier to solve, the author remove the normalization of $R^T x_i$ and solve the following equation instead:

$$Q(B,R) = \underset{B,R}{\operatorname{argmax}} \sum_{i=1}^{n} \frac{b_i^T}{||b_i||_2} R^T x_i \qquad s.t. \quad R^T R = I_{c\times c} \tag{16}$$

Similar to ITQ, a greedy method alternating between two steps is proposed to solve Equation 16. In one step, R is fixed and b_i's are found one by one independently. In the other step, b_i's are fixed and R is solved via polar decomposition [29].

Compared to traditional hashing methods which do not assume unit-length data, AQBC performs better on multiple datasets. Note that the similarity between a query and a training

sample is measured as the cosine of the angle between their binary codes, which is less computationally efficient. It is one of a few methods that explicitly consider the case where data live on a unit hypersphere. However, there exist significant drawbacks of this method.

One obvious drawback is that AQBC can only handle data samples represented as non-negative vectors since the binary codes all reside in one quadrant. There exist other feature extraction methods (such as PCA with normalization) where the features are unit-length but they contain negative values. The second is the relaxation from Equation 15 to Equation 16. This relaxation is sort of abrupt and there are many cases where the approximate solution can be far from the real one. No evaluation is conducted to show how this relaxation will affect the search performance.

2.2.5. K-means Hashing. The effectiveness of quantization based hashing has been demonstrated by ITQ. Since it is well-known that K-means can do data quantization very well, it is promising that a binary code learning method can integrate K-means into the learning process. He et al. propose K-means Hashing (KMH) [17] to learn binary codes of data via a K-means like approach. A very straightforward method is to first cluster training samples using K-means and then assign binary codes to each cluster center such that the Euclidean distance between any two centers is approximated by the Hamming distance between their binary codes. However, finding a good assignment of the binary codes to the cluster centers is very expensive.

Moreover, the straightforward method mentioned above has two types of approximation errors. One is from the K-means clustering and the other is from binary code assignment to cluster centers. He et al. propose to combine the binary code assignment with the K-means clustering as a single method with a single approximation error.

Denote by x a data sample, c_i the i-th cluster center and $id(x)$ the index of the cluster x belongs to. Note that the index i can be represented in a binary form. The total number of cluster centers is K. Like K-means, the learning process also involves two alternating steps. These two steps are:

- Fix c_i and update $id(x)$.
- Fix $id(x)$ and update c_i.

The first step is the same as K-means: Each data sample is assigned to the closest cluster center. The second step considers two aspects. One is the quantization error which is the same in K-means. The other aspect is the approximation error of the Euclidean distance between any two centers and the Hamming distance between their indices. The optimization function is as follows:

$$c_i = \underset{c_i}{\operatorname{argmin}}\Big(\sum_{x;id(x)=i} ||x - c_i||^2 + \lambda \sum_{j;j\neq i} \frac{n_i n_j}{n^2} (d(c_i, c_j) - d_h(i,j))^2 \Big) \tag{17}$$

where n_i, n_j and n are the number of data samples in the i-th cluster, j-th cluster and the whole dataset. $d(c_i, c_j)$ denotes the Euclidean distance between c_i and c_j and $d_h(i,j)$ denotes the square root of the Hamming distance (properly scaled) between their indices. This step is repeated K times in order to update all centers. It should be noted that when updating one center c_i, other centers are fixed.

There are actually two greedy methods in the proposed approach. One is apparent, since the two alternating steps are greedy. The other is a little subtle: In the second step, one center c_i is updated while other centers are fixed while it would be better to update all the centers together. These two greedy methods may make the quality of the learned bits low.

There is a very important practical issue with the proposed method. Since the number of clusters K is equal to 2^b, where b is the number of bits, the learning can be prohibitive if b is large. As a result, the authors propose a method similar to Product Quantization [7] which represents each data sample x as a concatenation of M subvectors: $x = [\hat{x}^1, \hat{x}^2, \ldots, \hat{x}^M]$. The i-th subvector of all data samples form an independent subspace. KMH is conducted in each subspace and the binary code of a data sample x is obtained by concatenating the binary code of each of its subvectors. Let $c = b/M$. If 2^c cluster centers are found in each subspace, then the total number of different binary codes of a whole data sample is $2^{Mc} = 2^b$. This equation means that a data sample can still have a b bit code. However, since there are 2^c quantizers in each of the M subspaces, the total number of quantizers is $M2^c$. Clearly, $M2^c$ is much smaller than 2^b and subdividing the problem in this way extends the size range of the solvable problems.

Since the quantizers are found in each subspace independently, it would be desirable to require that each subspace is independent. Also, since binary codes in each subspace are concatenated to form a whole binary code, each subspace should have approximately the same variance so that the learned binary codes are directly comparable and can be directly concatenated. In other words, when the binary codes are scaled to approximate the Euclidean distance in each subspace, the scale should be similar for each subspace. So the authors conduct PCA rotation (without dimension reduction) to obtain uncorrelated dimensions. Note that any rotation, translation or scale does not affect the learning of binary codes. Moreover, they reorder the principal components such that the eigenvalues associated with each subspace have a similar sum, which means that each subspace has a similar variance.

He et al. compare KMH with multiple state-of-the-art methods such as ITQ and SH on SIFT1M dataset and also a GIST dataset they built on their own. KMH performs the best on both datasets. However, they did not conduct any experiment on the standard GIST 1M dataset [7]. This thesis will show later that KMH is inferior compared to ITQ on the standard GIST 1M dataset.

Learning binary codes via a K-means like method has the ability to fully explore how quantization can be beneficial to binary code learning. However, as mentioned before, the learning process may be too "greedy" to obtain good local optima. Also, another significant disadvantage of KMH is that it is very computationally expensive when the number of data samples is large. The authors advocate subsampling the data before running the algorithm, but the performance can be worse if the subsampling cannot preserve the characteristics of the data.

2.2.6. Ranking Binary Codes using Weighted Hamming Distance. When we retrieve nearest neighbors of a given query, we will compute the binary code of the query and then look up the hash table using the binary code as our key. The items stored in that key are the retrieved nearest neighbors. However, this is an ideal case because the binary code may not exist in the hash table. Often, we need to explore similar keys, i.e., keys with a small Hamming distance to the key of the query. In other words, we need to do a Hamming ball exploration centered at the key of the query. ANN search of a query is still fast using this Hamming ball search with an increasing radius. However, there can be a number of binary codes with the same Hamming radius to the query. Assuming the number of bits is b, there exist $\binom{b}{d}$ binary codes with a Hamming distance of d to the query. This number can be very large even when d is small. For example, for a query with a 32-bit code, there are $\binom{32}{3} = 4960$ binary codes with a Hamming distance of 3 to the query. Zhang et al. [18]

propose to associate each bit with a weight when computing Hamming distances to further distinguish the binary codes with the same Hamming distance to a query. Note that they are not designing new hashing algorithms but computing weights given an existing hashing algorithm.

Denoting a binary hash function of the k-th bit of a data sample x as $h_k(x)$, $h_k(x) = sign(f_k(x) - T_k)$ where T_k is a threshold giving the binary value. As we have seen so far, this representation is very common for many hashing methods: computing a real value and then thresholding it to get a bit value. Also, if a data sample p is one of the nearest neighbors of another data sample q, they denote it as $p \in N(q)$. There are two criteria considered in the designing of the weight of each bit, listed in the following.

- **Data sensitivity**: Given a query q, the hash functions are not equally discriminative. If the k-th bit is discriminative, then $s_k(p, q) = f_k(p) - f_k(q)$ would be close to zero for $\forall p \in N(q)$, and vice versa. (The authors use $f_k(p) - f_k(q)$ instead of $h_k(p) - h_k(q)$ since useful information can be lost in the binarized form.) Data sensitivity means the weights should have a positive correlation with the discriminativity of hash functions. As a result, the weight of the k-th bit, denoted as w_k, is a function of the distribution of s_k.
- **Query sensitivity**: For a query q, let us assume that the k-th bit is highly discriminative (small s_k). By data sensitivity, a large weight w_k should be assigned to it. However, if q lies close to the decision boundary corresponding to this bit, it is highly possible that a lot of its neighbors will have a different binary value in terms of this bit. Therefore, assigning a large w_k is not appropriate. This means that the value of w_k should also depend upon the query q.

Given a query q and a data sample whose binary code is g, denote their desired weighted Hamming distance as $D_H^w(H(q), g)$ where $H(q)$ is the binary code of q. $D_H^w(H(q), g)$ encodes similarity between q and this data sample g. In their paper, Zhang et al. relate $D_H^w(H(q), g)$ to the probability of $H(q + n) = g$, where n is a random noise. However, since in their experiment, $q + n$ are chosen as all neighbors of q, n is not completely random. Therefore, I relate $D_H^w(H(q), g)$ to the probability that given $H(q)$, how likely will its neighbor's binary code be g. I find this interpretation more helpful.

Since $D_H^w(H(q), g)$ should be a non-increasing function of $Pr(g|H(q))$, the authors select the information entropy function and compute $D_H^w(H(q), g)$ as:

$$D_H^w(H(q), g) = -logPr(g|H(q)) \tag{18}$$

To compute $Pr(g|H(q))$, they assume that each hash bit is independent so that $Pr(g|H(q))$ can be computed as a multiplication of the probabilities of each bit.

$$Pr(g|H(q)) = \Pi_{k=1}^{b} Pr(g_k|h_k(q)) \tag{19}$$

$Pr(g_k|h_k(q))$ can be further divided into two cases: $g_k = h_k(q)$ and $g_k \neq h_k(q)$. Therefore, $Pr(g_k|h_k(q))$ can be written as $Pr(\Delta h_k(q) = 0)$ when $g_k = h_k(q)$ and $Pr(\Delta h_k(q) \neq 0)$ when $g_k \neq h_k(q)$, where $\Delta h_k(q)$ denotes the difference between g_k and $h_k(q)$.

Simply dividing $Pr(g|H(q))$ by a constant term results in the following:

$$D_H^w(H(q), g) = \sum_{k \in S} \lambda_k(q) \tag{20}$$

where S is the set of hash bits in g that are different compared to $H(q)$ and

$$\lambda_k(q) = log\frac{1 - Pr(\Delta h_k(q) \neq 0)}{Pr(\Delta h_k(q) \neq 0)} \tag{21}$$

Now the only thing left is to compute $Pr(\Delta h_k(q) \neq 0)$, which is the probability of q's neighbor having a binary value different from q for the k-th bit. Figure 2.3 shows a clear illustration of how to compute it. The real values of q and its neighbors are shown on the horizontal line. The vertical line is the threshold used to binarize the data. The red dot is the query q and all the blue dots are its neighbors. Other data samples which are not the neighbors of q are not shown in the figure. The authors assume that the distribution of neighbors follows a Gaussian model. $Pr(\Delta h_k(q) \neq 0)$ is computed as the number of data samples on the left side of the threshold divided by the number of all the neighbors of q. These two numbers are computed as the area under the probability density function (PDF) curve of the distribution of q's neighbors. $Pr(\Delta h_k(q) \neq 0)$ is closely related to the two types of sensitively mentioned above since it is based upon the values of the blue and red dots (data sensitivity) and the threshold (query sensitivity).

The final weight for the k-th bit given a query q is $w_k(q) = \lambda_k(q)$. For comparison, the authors also provide a simple but straightforward weighting scheme: $w_k(q) = |T_k - f(q)/\sigma_k|$, where σ_k is the standard deviation of the aforementioned Gaussian model. This scheme means that if q is far from the threshold and if its neighbors spread tightly, the weight would be large.

Zhang et al.'s approach recomputes a set of weights each time a different query is given. As a result, the Hamming ball search can be adapted to the given query. The weights reorder the Hamming ball search. This reordering is not restricted only at the level of binary codes with the same Hamming distance to the query but among all binary codes. For example, a

binary code with (unweighted) a Hamming diatance of 2 may be searched earlier than some binary codes with a Hamming distance of 1 to the query.

Their experimental results on multiple datasets show that the search performance of PCAH, LSH, ITQ can be improved by this weighting scheme. However, the performance of the proposed method is not significantly better than the straightforward weighting scheme mentioned above (i.e., $w_k(q) = |T_k - f(q)/\sigma_k|$). It seems that it may not be necessary to go through such complicated processes to find a good weight for a bit.

There are two other drawbacks of this method. One is that the weights are only computed based on the distribution of the neighbors of a query. This means that the weights are useful when comparing the distance between a query and a gallery sample close to it. If the gallery sample is far away from the query, the weights are not informative. The second drawback is the computation cost. Since a set of weights have to be computed each time a new query is given, the procedure can be slow when the number of queries is large.

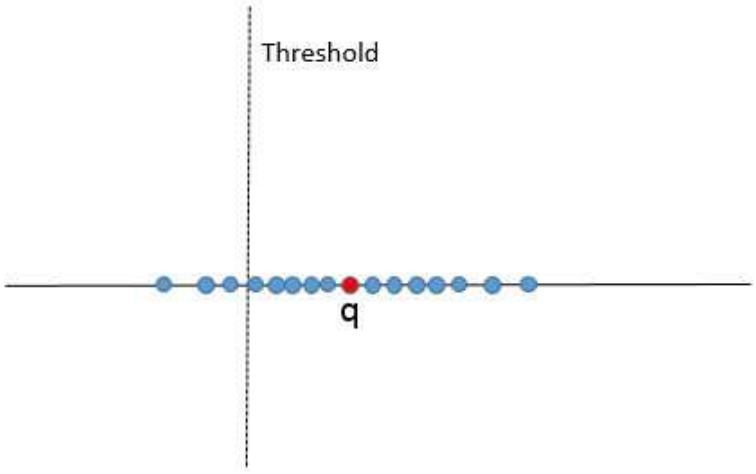


FIGURE 2.3. An illustration of computing $Pr(\Delta h_k(q) \neq 0)$ given q

2.2.7. SPARSE EMBEDDING AND LEAST VARIANCE ENCODING. Zhu and Zhang proposed Sparse Embedding and Least Variance Encoding (SELVE) [19] to learn binary codes

for large-scale datasets. The idea is to encode each transformed data sample using a dictionary matrix, where each data sample can be approximated using a linear combination of the columns in the dictionary. Each data sample is then represented as the coefficient vector. The coefficient vectors are constrained to have low variance such that similar data samples will have similar coefficient vectors. The weight vector is then thresholded using the mean of all weight vectors to get a binary code.

SELVE first conducts clustering on the whole dataset and then represents each data sample based on its distance to the few closest centers. This is the same as how Anchor Graph Hashing (AGH) generates its Z matrix, which is described in Section 2.1.3. One minor difference is that SELVE adopts a clustering method called linear spectral clustering [30] while AGH uses K-means. Although whether this difference will influence the performance is not discussed in the paper, I suspect that these two methods will perform equally well.

Denote the represented data samples as $P \in \mathcal{R}^{k \times n}$ ($P = Z^T$ if the notation from AGH is used), where k is the number of cluster centers and n is the number of data samples. Each column in P is a data sample. The authors then propose to represent data samples using a dictionary matrix $\Phi \in \mathcal{R}^{k \times c}$ where c is the number of dictionary atoms (Each column is called an atom). Each atom/column in Φ has unit length. Every data sample is approximated as a linear combination of the atoms in Φ. To write the dictionary representation in a matrix form, we have

$$\underset{\Phi,\Lambda}{\operatorname{argmin}} ||P - \Phi\Lambda||_F^2 \tag{22}$$

where $\Lambda \in \mathcal{R}^{c \times n}$ contains the linear weights (also called coefficients) for each data sample in P. The i-th column in Λ, denoted as α_i, contains the coefficients corresponding to the i-th data sample.

Moreover, the authors aim to learn coefficients with small variance such that similar data samples will have similar coefficients. Therefore, they add this constraint in the dictionary learning process and propose the following optimization function:

$$\underset{\Phi,\Lambda}{\operatorname{argmin}} ||P - \Phi\Lambda||_F^2 + \lambda \sum_{i=1}^{n} ||\alpha_i - \mu||^2 \tag{23}$$

where $\mu = \frac{1}{n}\sum_{i=1}^{n} \alpha_i$. It is easy to see that $\sum_{i=1}^{n} ||\alpha_i - \mu||^2$ represents the variance of the coefficients.

A greedy method that alternates between two steps is presented in the paper to solve Equation 23. In one step, Φ is updated while Λ is fixed. In the other step, Λ is updated while Φ is fixed. Multiple iterations are needed so that the solutions converge. μ is also used as a threshold of α_i to obtain the binary code of the i-th data sample: $sign(\alpha_i - \mu)$.

The authors also propose an enhanced version of their method: Between the step of clustering and obtaining P, they add one more step which finds a linear orthogonal projection W that"tightens" each cluster. In other words, after projection on W, data samples lying in the same cluster are closer to the center. As a result, P will be more informative, since each data sample is related to fewer cluster centers. Let X be the original data where each column is a data sample. Denote $\Gamma_j = X_j - M_j$, where each column of X_j is an original data sample in the j-th cluster, each column of M_j is the j-th cluster center (i.e., M_j simply contains multiple copies of a single cluster center). Each cluster is "tightened" by minimizing $\sum_{j=1}^{k} ||W\Gamma_j||_F^2$. Also, it should be noted that the orthogonal projection W should minimally distort the locality structure of the data. Otherwise, true nearest neighbors may become far apart. Therefore, W should be able to approximately reconstruct X, which leads to minimizing $||X - W^TWX||_F^2$. Here, WX is the projected values of X and W^TWX is the back projection of the projected values which is a reconstruction of X. The formal objective

function is:

$$\underset{W}{\operatorname{argmin}} ||X - W^T W X||_F^2 + \lambda \sum_{j=1}^{k} ||W\Gamma_j||_F^2, \ s.t. \ WW^T = I \tag{24}$$

This equation is solved using SVD.

Compared to other hashing methods such as SH, ITQ and AGH on four datasets, enhanced SELVE yields higher precision at the same recall and also higher precision if only binary codes within a Hamming distance of 2 are retrieved. Moreover, the enhanced version of SELVE performs consistently better than the regular version. It indicates that finding an orthogonal linear projection that "tightens" data helps improve retrieval performance.

Although their experiments show good performance, there is a major disadvantage of this approach. It is true that minimizing variance of the coefficients results in similar data samples having a higher probability of being assigned the same binary code. However, since the variance is minimized over the whole dataset, dissimilar data samples may also have similar binary codes. Therefore, it is uncertain that the learned binary codes of dissimilar data samples will have a larger Hamming distance than similar data samples. One possible answer is that: In their enhanced version, they propose to find a linear projection to "tighten" each cluster, which may somehow alleviate the aforementioned disadvantage.

2.2.8. Hashing with Binary Autoencoders. Carreira-Perpinan and Ramin Raziperchikolaei [31] propose to learn binary codes using binary auto-encoders. The basic concept of auto-encoders is to learn both a encoder and a decoder given data samples, where the encoder maps original data samples to a lower dimensional space and the decoder tries to learn a mapping that can reconstruct the original data samples from that space. Auto-encoders can be useful in applications such as dimensional reduction, feature extraction, etc..

In the case of learning binary codes, Carreira-Perpinan and Ramin Raziperchikolaei propose to learn a binary encoder that maps each original data sample to a vector of binary bits and a decoder that reconstructs data using the binary vectors. More specifically, the optimization function adopted in their paper is as follows:

$$E_Q(h, f, Z; \mu) = \sum_{n=1}^{N} (||x_n - f(z_n)||^2 + \mu ||z_n - h(x_n)||^2) \tag{25}$$

where N is the number of data samples, x_n denotes the n-th data sample, $z_n \in \{0, 1\}^b$ (b being the number of bits) is the desired output of $h(x_n)$, $h()$ is the encoder function and $f()$ is the decoder function.

To solve Equation 25, the authors iteratively alternate between two steps. One steps is solving Z when fixing h and f. The other is solving h and f when fixing Z. Moreover, they gradually increase the value of μ so that z_n will be equal to $h(x_n)$ in the end. It is worth mentioning that solving Z when f and h are fixed is NP-complete. For very small code lengths, they use enumeration to find the global optima. For large code lengths, they use alternating optimization and warm-start tricks to achieve a good solution. They apply multiple tricks to accelerate their computation such as space pruning, parallel programming and so on.

A merit of this paper is that, different from some other methods that transform data and then simply binarize/threshold the output, their method well respects the binary constraints of the problem when trying to learn auto-encoders. In other words, they take a lot of efforts in searching the binary space for good binary codes.

There exist some drawbacks of this method. First of all, a lot of compromises and relaxation have been made to make the optimization function solvable and computationally feasible. There is little evidence in the paper to demonstrate that the found auto-encoders

are close to optimal. Second, although all kinds of acceleration are conducted, the method still can only handle a very small number of bits in the binary codes (up to 32 bits in the paper). Third, the improvement of the performance compared to other state-of-the-art algorithms is not very significant.

2.3. Conclusion

This section reviewed a number of important and representative binary code learning approaches for approximate nearest neighbor search in large-scale datasets. They are categorized in two major groups. One is similarity based methods and the other is quantization based methods. Similarity based methods are generally computationally expensive. If the number of data samples is large, the size of the similarity matrix, which is the square of the number of data samples, can be extremely large. Approximation is often needed to obtain a similarity matrix (e.g., subsampling, low-rank approximation). Despite these disadvantages, similarity based methods can be more versatile than quantization based methods since the definition of similarity can be specialized to better suit specific problems. For example, in stead of using (negative) Euclidean distance to indicate similarity, an RBF function can be used. Also, if the similarity matrix is defined by supervised information, many of these methods can be easily adapted to supervised data.

Quantization based methods are more efficient. They aim at minimizing the approximation/quantization error of data samples using binary codes. They are suitable for unsupervised data where the groundtruth similarity is defined by an unsupervised metric such as Euclidean distance or inner product since if each data sample is well approximated, distance between pairs of them is also well approximated in general. Experimental results have shown that they can yield better retrieval performance compared to many similarity based

methods. A disadvantage of quantization based methods is that they are harder to employ in situations where standard measures of distance do not reflect well the goals of a specific task.

CHAPTER 3

The Proposed Approach: Unit Query and Location Sensitive Hashing (UnitQLSH)

This section describes in detail my proposed approach. I will first describe a method called Local Quantization Hashing (LQH) which extends ITQ to nonlinear data by local neighborhood approximation. LQH is also invented by me and my colleagues. It performs much better than ITQ on two large-scale, real datasets. Disadvantages of LQH will also be presented. Solving these disadvantages leads to the invention of the primary algorithm presented in this thesis, namely Unit Query and Location Sensitive Hashing (UnitQLSH).

Figure 3.1 shows the relationship between ITQ, LQH and UnitQLSH. The orange rectangle is the new element added to them. LQH is proposed based on integrating ITQ with nonlinear structure approximation via local neighborhoods. UnitQLSH is proposed based on adding query sensitivity to LQH and also constraining the binary quantizers to be unit-length. Each element solves significant disadvantages of the preceding algorithms.

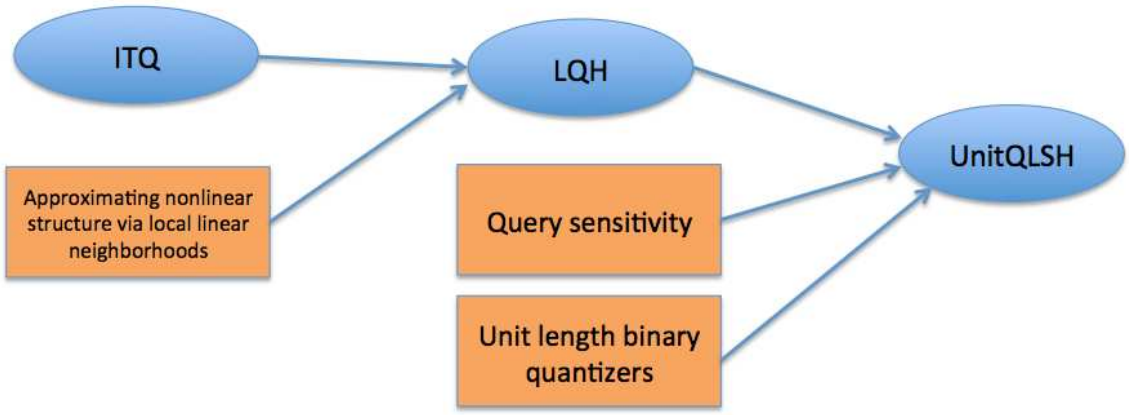


Figure 3.1. Relations between ITQ, LQH and UnitQLSH

3.1. Local Quantization Hashing

Despite the simplicity and impressive performance of ITQ, its hashing functions are linear functions. Therefore, if data live in a nonlinear subspace, ITQ will not be able to take advantage of this property. In fact, ITQ implicitly assumes that all the data samples live in a single linear space because it performs PCA on them as a first step. This is a weak assumption since it is common for real-world data to live in nonlinear subspaces that occupy only a fraction of the full feature space. Exploring these low-dimensional nonlinear subspaces can greatly benefit the learning of compact binary codes. As a result, multiple approaches including Kernelized Locality Sensitive Hashing (KLSH) [6], Kernelized Iterative Quantization (KITQ) [5] and Spectral Hashing (SH) [11] have been proposed to handle the nonlinear structure of data. Most of these approaches apply kernels such as Radial Basis Functions (RBFs) to model the similarity of pairs of data in nonlinear subspaces.

Unfortunately, choosing a good kernel function and the right kernel function parameters for a specific dataset is difficult. Some methods even require a different parameter set for a different number of bits, making learning more complicated. Also, there can be cases where the structure of the data is too complicated to be approximated by any known kernel function. Another disadvantage of using kernel to handle nonlinearity is that a kernel matrix is often involved in the computation, which is an $n \times n$ matrix, where n is the number of data samples. The resulting additional computation can be very expensive for large-scale datasets.

I describe in detail here a method proposed by ourselves, called Local Quantization Hashing (LQH). Instead of using kernels, it nicely tackles the nonlinear structure of data by approximating it via multiple local linear neighborhoods. Section 3.1.1 presents an introduction of LQH. Section 3.1.2 and Section 3.1.3 presents the training process and the nearest

neighbor search process of LQH, respectively. Section 3.1.5 presents concluding remarks of LQH.

3.1.1. Introduction of LQH. LQH is built on the assumption that nonlinear data have a smooth structure or can be decomposed to multiple smooth structures. LQH decomposes a smooth structure into multiple local neighborhoods, each of which can be approximated using a linear subspace. These neighborhoods well approximate the nonlinear structure of data. In each local neighborhood, the potential of ITQ can be truly realized since the space is very close to a linear one. Therefore, we apply ITQ independently in each local neighborhood to obtain a binary code for each data sample.

Figure 3.2 illustrates the differences between our method (LQH) and ITQ on nonlinear data. In this example, the synthetic data live in a nonlinear subspace, and the goal is to produce a 2-bit binary code. In the left subplot, the vertices of the hyper-cube found by ITQ are shown as red dots. These vertices are the quantizers of the data samples. In the right subplot, LQH divides the data into two clusters (shown in green and yellow), each of which forms a local neighborhood that can be approximated as a 1D subspace. LQH then applies ITQ to each cluster independently, creating a local 1-bit binary representation for each data sample. As a result, each data sample is still represented by 2 bits, where one bit indicates the index of the local neighborhood (i.e. cluster) and the other bit indicates its binary representation. Each cluster is therefore quantized by 2 points as shown by the red dots in the right subplot. The 2 points in each cluster form a 1D rectangle within each cluster. The quantizers found by LQH therefore yield a closer approximation of the original data due to the linear and low-dimensional structure of the local neighborhoods found by clustering.

There are two advantages of LQH.

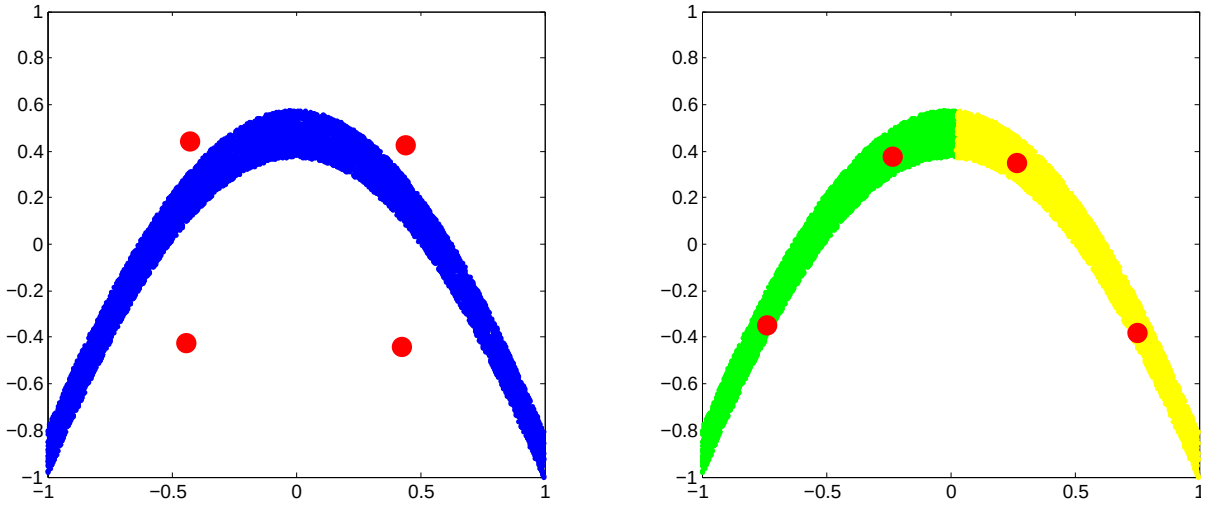


FIGURE 3.2. An illustration of LQH on 2D synthetic data. Left: ITQ; Right: LQH

- These local neighborhoods satisfy the assumption of ITQ that data live in a linear subspace. Therefore, the performance of ITQ over these neighborhoods is greatly enhanced.
- The nearest neighbors of a query have a high probability of coming from the few neighborhoods closest to the query. Therefore, the search space of its nearest neighbors is much smaller and more accurate. This can also be seen as location sensitivity since only a particular subset of all gallery data samples will be compared given a query.

This method also enables us to circumvent the difficulty of finding a good kernel and a right set of parameters of the kernel for a specific dataset.

3.1.2. LQH: TRAINING. LQH is introduced to improve approximate nearest neighbor search for nonlinear data. Unlike kernel-based methods, LQH assumes that nonlinear data have a smooth structure or can be decomposed to multiple smooth structures. LQH therefore exploits local linearity. More specifically, local neighborhoods of nonlinear data can be

approximated well by linear subspaces. In this way, LQH avoids explicit kernels yet while approximating nonlinear data better than kernel methods.

LQH builds on ITQ. ITQ assumes that data live in a single linear space, and fits a rotated hyper-cube to quantize them. This overlooks the fact that real data generally reside in lower dimensional nonlinear subspaces. LQH approximates nonlinear data using sets of local linear subspaces. A local neighborhood is defined as a set of data points close to each other in terms of Euclidean distance. K-means [32] clustering is a good algorithm for finding local neighborhoods. Each K-means cluster forms a local neighborhood, and local neighborhoods are approximated as linear PCA subspaces.

LQH then applies ITQ in each local neighborhood independently to learn binary codes for each data sample. More specifically, assume that there are n data points $\{x_1, x_2, \ldots, x_n\}$ in one cluster, each of which is a d dimensional row vector. LQH centers the data within the cluster by subtracting the mean (cluster center) $m = \Sigma_{i=1}^{n} x_i$. The centered data are represented as $y_i = x_i - m, (i = 1, \ldots, n)$. They form the rows of a data matrix $Y \in \mathbb{R}^{n \times d}$. The mean vector m is stored for processing new query data.

For b bits, ITQ learns a set of b linear encoders $\{w_1, w_2, \ldots, w_b\}$, each of which is a column vector. They form the columns of the encoding matrix $W \in \mathbb{R}^{d \times b}$. The binary encoding for the jth bit for a data point y_i is $h_j(y_i) = sgn(y_i w_j)$, where $sgn(v) = 1$ if $v \geq 0$ and -1 otherwise. The encoding of all data points can be written in a matrix form: $B = sgn(YW)$, where sgn performs a pointwise sign operation for the matrix. Each row of B is a binarized data sample. To learn the b linear encoders, ITQ first reduces each data point to b dimensions using Principal Component Analysis (PCA). Denoting the PCA projection matrix as $P \in \mathbb{R}^{d \times b}$, the reduced data can be written as $V = YP$. Then ITQ seeks to rotate

the reduced data V to minimize the quantization loss when data are binarized:

$$\underset{B,R}{\operatorname{argmin}} Q(B,R) = ||B - VR||_F^2, \tag{26}$$

where B is the binarized data, $R \in \mathbb{R}^{b \times b}$ is a rotation matrix and $||M||_F$ denotes the Frobenius norm of matrix M. After solving Equation 26 using the optimization process proposed in [15], the b-bit binary representations of Y are computed as $B = sgn(YPR)$.

Note that the above learning process is independently applied to each cluster. As a result, each cluster will be associated with a cluster center and a set of binary encoders $W = PR$ accustomed specifically to the cluster.

3.1.3. LQH: Approximate Nearest Neighbor Search. Given a query q, LQH finds the closest cluster by measuring the distance to each cluster center. After locating the closest cluster (denoted as k_1), we compute q's binary representation with respect to that cluster using the associated binary encoders and cluster center. The distance between the query and each data point within Cluster k_1 is simply the Hamming distance. In most cases, many nearest neighbors of the query should be in this cluster. If all the nearest neighbors are in k_1, then the search is done. However, the nearest neighbors may be distributed across multiple clusters. Therefore, LQH locates the next closest cluster (denoted as k_2) to q and computes its binary representation with respect to that cluster. The distance between q and data points in Cluster k_2 is also approximated by their Hamming distance. However, the Hamming distances within Cluster k_1 and Cluster k_2 are computed independently and they are not directly comparable. Therefore, in order to compare the distance between q and a data point in Cluster k_1 and the distance between q and a data point in Cluster k_2, we need to take advantage of the geometric interpretation of ITQ.

ITQ rotates the data in order to minimize the quantization error after data are binarized. This can be interpreted as finding a hyper-cube with minimal quantization error when data are mapped to the nearest vertex on the hyper-cube. For standard ITQ, since there is only one hyper-cube, the scale of the hyper-cube does not affect the ranking of Hamming distances or the choice of quantizers. In fact, the hyper-cube yielding the minimal quantization error has a unique real-value scale, denoted as s, that can be easily computed by Equation 27 after Equation 26 is solved.

$$\underset{s}{\operatorname{argmin}} \, ||V - sBR^T||_F^2, \tag{27}$$

The scaled hyper-cube is the quantizer of the original data. As a result, the Hamming distance between two data points should be scaled by s to approximate their Euclidean distance. In LQH, multiple hyper-cubes are learned, one for each cluster. For direct comparisons across clusters, we scale each hyper-cube to become the actual quantizers of the original data in the cluster. The resulting scaled Hamming distances from different clusters can then be compared directly.

Figure 3.3 shows how to compute the scaled Hamming distance in each cluster given a query on 2D synthetic data. Two local neighborhoods denoted by blue and green colors are first located by K-means clustering. Then ITQ is applied to each neighborhood. The red dots show the vertices of the hyper-cubes (squares in this case) found by ITQ, each of which corresponds to a binary string. Note that, unlike standard ITQ, the cubes are scaled such that they are the actual quantizers of the data in the same cluster. Denote the centers, the binary encoders and the scales of the cubes/hyper-cubes in these two clusters as $c_{b,g}, W_{b,g}$ and $s_{b,g}$ respectively, where b and g denote blue and green respectively. Given a query in the form of a row vector q (shown as a black star in Figure 3.3), its closest

cluster is the blue one. So LQH computes a binary representation with respect to the blue cluster: $bin_b = sgn((q - c_b)W_b)$, which will assign q to the lower-right vertex of the cube (as shown by the left black arrow). Denoting the Hamming distance between q and the ith data sample in the blue cluster (denoted as $x_{b,i}$) as $d(q, x_{b,i})$, their scaled Hamming distance is $d_s(q, x_{b,i}) = s_b \times d(q, x_{b,i})$. $d_s(q, x_{b,i})$ is an approximation of their Euclidean distance.

The second closest cluster to the query q is the green cluster. The same procedure is applied to compute a binary representation of q with respect to the green cluster (shown as the right black arrow) and a scaled Hamming distance $d_s(q, x_{g,j})$ between q and the jth data point in the green cluster (denoted as $x_{g,j}$). $d_s(q, x_{b,i})$ and $d_s(q, x_{g,j})$ can then be compared directly to indicate whether $x_{b,i}$ or $x_{g,j}$ is closer to q. It should be noted that since these two clusters are quantized by cubes with different scales, a single bit difference between two codes in the same clusters indicates different Euclidean distance for different clusters. Using this method, we can determine the relative distance of the query between two data samples from different neighborhoods.

Given a query q, it is important to point out that there is a trade-off between the number of nearby clusters explored and the accuracy of the Euclidean distance approximation using scaled Hamming distance. When LQH computes q's binary representation, it quantizes q as the closest vertex of the hyper-cube. However, it does not consider how far q is from its closest vertex of the hyper-cube. In other words, q may be very far from the cluster but it will still be assigned to one vertex of the hyper-cube before computing the scaled Hamming distance. Therefore, the distance between q and its quantizer/vertex on the hyper-cube is not included in the scaled Hamming distance to obtain q's binary representation for fast hashing. However, the above discussion indicates that it is not suitable for estimating distances between q and data points in far away clusters.

Since the goal is to find q's nearest neighbors, and the nearest neighbors probably reside in clusters close to q, we are comfortable restricting the search to nearby clusters. We can further reduce the set of clusters searched based on the criterion that the next cluster is explored only if its cluster center is as close as the current center to the query. This significantly improves retrieval efficiency.

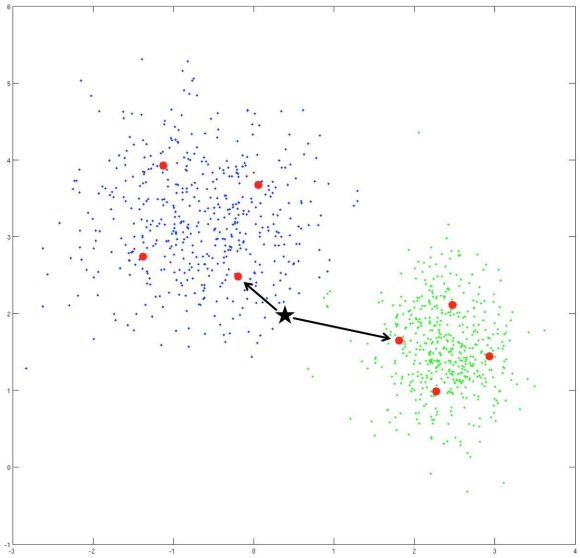


FIGURE 3.3. An illustration of computing scaled Hamming distances to enable a direct comparison of distances from different clusters. The quantizers in each cluster are scaled properly to minimize the approximation error in the cluster. The magnitude of the sacle is indicated as the side length of the cube.

3.1.4. PARAMETERS OF LQH. One important parameter is the number of clusters used in clustering procedure denoted as N_c. If N_c is too small, the local neighborhoods of the data will be too large to be approximated by a linear subspace. If N_c is too large, the nearest neighbors of a query may spread across a large number of clusters, which will increase search error due to the trade-off discussed in Section 3.1.3.

Another parameter is the number of clusters explored per query. This number should not be too small or too large. This is because, in general, a query's nearest neighbors exist in multiple clusters close to it. Therefore, multiple clusters should be searched for nearest

neighbors, not just the single closest cluster. However, due to the trade-off, we should only explore a few clusters in order to maintain a high search accuracy. In LQH, the number of clusters explored is adaptive. A maximal number of clusters M to explore is set and fixed. For a query q, the $(i+1)$th closest cluster is only explored when $\frac{||q, c(i+1)||_2}{||q, c(i)||_2} - 1 \leq \sigma$. Here $||q,\, c(i+1)||_2$ is the Euclidean distance between the query and the center of the $(i+1)$th closest cluster and $||q,\, c(i)||_2$ is the Euclidean distance between the query and the center of the ith closest cluster. σ is a small positive number. If this constraint is met, it means that the $(i+1)$th cluster is potentially as close to the query as the ith cluster and should be explored. The total number of explored clusters will not exceed M.

3.1.5. Conclusion of LQH. A hashing method called Local Quantization Hashing (LQH) is proposed for unsupervised approximate nearest neighbor search. LQH handles nonlinear data by finding local, smooth neighborhoods that can be explicitly approximated with linear subspaces. ITQ is then applied independently in each local neighborhood to achieve a more accurate binary representation of data points. LQH gracefully circumvents the problem of having to define a kernel function to approximate the nonlinear structure of the data.

The experimental results are presented later in Section 4. Compared to other state-of-the-art hashing methods, LQH performs better in most cases when varying the number of bits. More importantly, it yields much better retrieval performance when the number of retrieved items is small, which is highly desirable for a number of applications. We also conducted experiments where we apply LQH on four synthetic datasets to shed light upon why this simple method works so well.

Although providing very promising retrieval results, LQH also has some disadvantages. For a given query q, its true nearest neighbors will typically reside in multiple local neighborhoods, which means that multiple neighborhoods close to q should be explored. It seems that the when more neighborhoods are explored, the less chance a true nearest neighbor will be missed. However, the trade-off we just described means that when more neighborhoods (also farther from q) are considered, the distance estimated by Hamming distance becomes less accurate. In the experiments of LQH, the number of explored neighborhoods is limited to at most 2. There can be some true nearest neighbors of q that exist in other neighborhoods that are never retrieved.

3.2. Unit Query and Location Sensitive Hashing

Although LQH performs very well on multiple large-scale datasets, its potential is limited by the aforementioned drawback that, given a query, only a very few clusters closest to it can be explored. The query's true nearest neighbors can exist in other distant clusters. To address this disadvantage, we propose unit Query and Location Sensitive Hashing (UnitQLSH).

In UnitQLSH, it is assumed that all data are expressed as unit length vectors so that the similarity between pairs of data samples is defined as their inner product. Our method can also be classified as a quantization based hashing technique. In other words, we seek binary representations that can approximate original data samples. We formulate the approximation problem as a matrix factorization problem with binary components. Given a query q, we show that the order of Hamming ball exploration can be easily computed without knowing how to obtain the binary code of q. In more detail, we use q's original data representation directly to compute a real-valued vector whose length is the number of bits.

It guides the order of Hamming ball exploration. Since this real-valued vector is different for different queries, it is called query sensitivity. Further, we find that we can first locate local neighborhoods as we have done in LQH and apply the aforementioned process independently in each neighborhood. This is called location sensitivity.

A great benefit of UnitQLSH over LQH is that there is no need to compute the binary representation of query data. It not only avoids this computational overhead, but also addresses the issue of LQH nicely: Since original query data are used directly, there does not exist any quantization error that makes the distance estimation worse for distant neighborhoods. As a result, we are now able to explore as many neighborhoods as desired. Moreover, since there is no quantization error of the query, the retrieval accuracy can be greatly improved.

I first describe how binary codes are learned on the training data, i.e., the key optimization function and its solution in the following two sections. Then I present query sensitivity of UnitQLSH in Section 3.2.3. After that, I will present how location sensitivity (as in LQH) can be well integrated together with query sensitivity in Section 3.2.5, i.e., the whole UnitQLSH method.

3.2.1. Binary Code Learning on Training Data. In this work, we aim to perform nearest neighbor search in data with unit length. That is to say, denoting the i-th data sample as x_i, we always have $||x_i||_2 = 1$. We also describe data with this property as data living on a unit hypersphere. The similarity between two data points on a unit hypersphere can be defined as their inner product, which is also the cosine of the angle between them. Equivalently, we can define their distance as their Euclidean distance since for two data samples x_i and x_j, $||x_i - x_j||_2^2 = 2 - 2x_i x_j^T$. In other words, their Euclidean distance has a negative linear correlation with their inner product.

Inspired by quantization based hashing techniques, in this work, we propose to learn a set of quantizers **on the unit hypersphere** that minimize the approximation error between data points and their corresponding quantizers where the approximation error is defined as the angle between them. Each different quantizer corresponds to a different binary code. For hashing methods, it is often desirable to have independent bits so that information is not redundant. This independence is often replaced by orthogonality. As a result, similar to ITQ, we want these quantizers to be vertices of a hyper-rectangle. Note that ITQ finds a hyper-cube to quantize data while this new approach seeks a more general hyper-rectangle. Furthermore, ITQ finds a hyper-cube in the PCA space. It takes a part of the subspace out of consideration due to PCA. In our method, we would like to find a hyper-rectangle directly in the original data space, which may yield more accurate quantization. The initial optimization function is as follows:

$$\underset{B,C}{\operatorname{argmin}} ||X - BC||_F^2, \quad s.t. \quad CC^T = D^2, \ ||B_\forall C||_2 = 1 \tag{28}$$

where X is the data matrix. $B_\forall$ stands for any arbitrary binary code which has the same length as a row of B. C is the basis matrix satisfying $CC^T = D^2$, where D is a diagonal matrix. D controls the length of each side of the hyper-rectangle. Each row of C is a basis vector of the hyper-rectangle. $||B_\forall C||$ represents any quantizer (also a vertex of the hyper-rectangle). $||B_\forall C||_2 = 1$ means that all the quantizers should reside on the unit hypersphere. Note that since the quantizers and data samples all have unit-length, the statement of the optimization problem in Equation 28 is the same as minimizing the angle between data samples and their quantizers. We will later compare this method to a simpler version where

quantizers are not constrained to have unit length and show the practical value of Equation 28 and the unit length constraint on real problems.

It should be noted that the mean of this hyper-rectangle is the origin since each direction C_i can only take a coefficient of either -1 or 1. Since data samples in X are not necessarily zero centered, the hyper-rectangle does not need to be centered at origin either. Therefore, we add a translation vector which is the center of the hyper-rectangle. Denoting this translation vector as μ, we have

$$B' = [B, 1_{n\times 1}], \quad and \quad C' = [C; \mu] \tag{29}$$

where n is the number of data samples. B' is obtained by appending a column of all $1'$s on the right of B and C' is obtained by appending a row vector, μ, at the bottom of C. The vertices of the translated hyper-rectangle can then be expressed as $B'C'$. In more detail, the i-th row of $B'C'$, which is also equal to B'_iC', can be expressed as $B_iC + \mu$. Our objective function from Equation 28 now becomes:

$$\underset{B',C'}{\operatorname{argmin}} ||X - B'C'||_F^2, \quad s.t. \quad CC^T = D^2, \ ||B'_\forall C'||_2 = 1 \tag{30}$$

where $B' = [B, 1_{n\times 1}]$ and $C' = [C; \mu]$. The geometric interpretation is as follows. We seek a translated hyper-rectangle, whose vertices lie on a unit hypersphere, that minimizes the mean angle between data points and its corresponding quantizers (closest vertices of the hyper-rectangle).

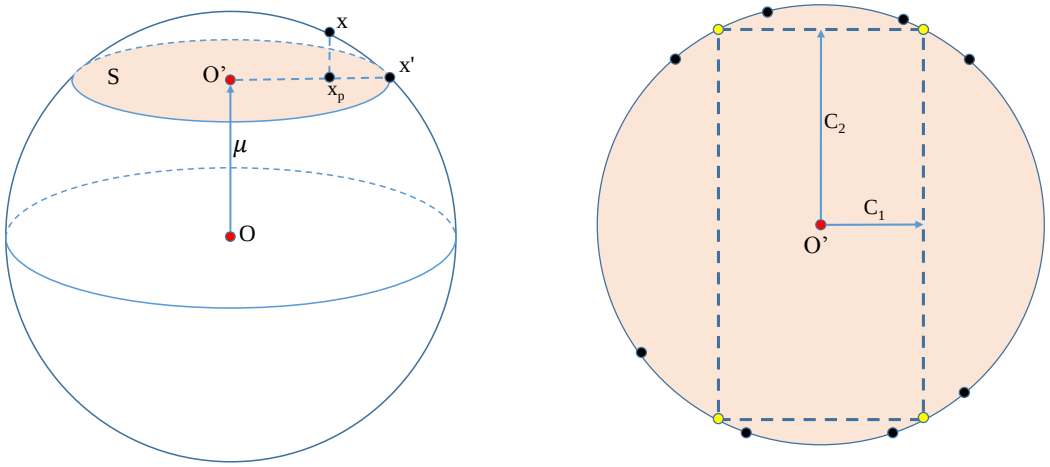
Theorem 1: If $||B'_\forall C'||_2 = 1$, then $C\mu^T = 0$.

Proof: $||B'_\forall C'||_2 = ||B_\forall C + \mu||_2 = B_\forall C C^T B_\forall^T + 2B_\forall C\mu^T + \mu\mu^T = tr(D^2) + 2B_\forall C\mu^T + \mu\mu^T = 1$ So, $B_\forall C\mu^T = \frac{1-\mu\mu^T - tr(D^2))}{2}$. For any hyper-rectangle that satisfies $||B'_\forall C'||_2 = 1$, we know $\mu\mu^T$ and $tr(D^2)$ are two constants whose value does not change, no matter what $B_\forall$ is. So, for any such hyper-rectangle, $B_\forall C\mu^T = const$. Denoting $v = C\mu^T$, v is a column vector. If $v \neq 0$, without loss of generality, let us assume the first entry of v is not zero, i.e., $v_1 \neq 0$. Then if B_1 and B_2 are two binary codes that only differ in the first entry: $B_1 = [1, \ldots]$ and $B_2 = [-1, \ldots]$, $B_1 v = v_1 + \ldots$ and $B_2 v = -v_1 + \ldots$ are different. It contradicts the fact that $B_\forall v = const$. So $v = C\mu^T = 0$. $\square$

The geometric meaning of $C\mu^T = 0$ is that the space spanned by these orthogonal bases of this hyper-cube (disregarding the translation vector μ) is perpendicular to μ. Note that the center of this hyper-cube is μ. This observation provides us with a nice way to solve Equation 30. In the first step, μ is found by simply taking the mean of all data samples. Next, locate a linear subspace S perpendicular/orthogonal to μ. The origin of S is μ. Each data sample is then projected to S. Note that these projected data samples are on the unit hyper-sphere as well. B, C and D (i.e., a hyper-rectangle without translation) are found in S given the projected data samples.

Figure 3.4 presents a 3D example that illustrates our optimization process. Figure 3.4(a) shows a 3D unit sphere. O is the origin. O' is the mean of data samples (The data samples are not shown in the figure for better display, except for x.). In other words, the vector from O to O' is μ. The linear subspace S perpendicular to μ is located, as shown in orange. The origin of this subspace is O'. The intersection of S and the sphere is a circle. It follows that O' is the center of the circle since S and μ are orthogonal. x is one of the data samples on the sphere. Since we want to find a hyper-rectangle (rectangle in this case) in S, we would

like to project every data sample to S. The projected data sample should be as close to the original data sample as possible. Moreover, to maintain the property that Euclidean distances reflect inner products (or angle) of any two points, we constrain their projections in S to have unit length as well. In this 3D example, we denote the desired projection of x as x'. x' can be found by first finding the orthogonal projection of x in S, denoted as x_p and then draw a line from O' to x_p until it intersects with the sphere (or equivalently, the circle denoted by orange). The intersecting point is the desired projection x'.



(a) Project a sample on the sphere to subspace S (b) Fit a rectangle in subspace S

FIGURE 3.4. A 3D example illustrating the optimization process.

It can be easily proven that x' and x have the smallest angle between them compared to all other projections on the orange circle (i.e., the subspace S): We know that $||x - x'||_2^2 = ||x - x_p||_2^2 + ||x_p - x'||_2^2$. $||x - x_p||_2^2$ is a constant and $||x_p - x'||_2^2$ is smallest when O', x_p and x' are on the same direction/line. Therefore, the found x' minimizes $||x - x'||_2^2$ as well as the angle between x and x'. $\square$

After we map each data sample in S, the remaining task is to find a rectangle in S, centered at O', that minimizes the quantization error. Figure 3.4(b) shows S when looking at it from the direction of μ. The black dots on the circle are the projections of all data samples. Note that since the vertices (yellow dots) of the rectangle are required to be on the sphere, they will also on the circle. More will be said about how to find this (hyper-)rectangle. For now, let us say that the dashed rectangle (centered at O') is the one yielding the smallest quantization error, i.e., the smallest mean Euclidean distance or angle between each data sample and its quantizer. Then the C_1 and C_2 in this example constitute the bases C of the rectangle: $C = [C_1; C_2]$ and $B = sign(X'C^T)$ where X' are the projections of the data samples.

3.2.2. Optimization. We have briefly described the optimization procedure in the example given in Figure 3.4. In this section, we show how to solve B' and C' in Equation 30. To start, assume a set of n data samples $\{x_1, x_2, \ldots, x_n\}$, each of which has d dimensions. These samples form the rows of a data matrix $X \in \mathbb{R}^{n \times d}$. Each data sample has unit length: $||x_i||_2 = 1, i = 1, 2, \ldots, n$. Further assume the number of bits we want to learn is b. Denote the sample mean as $\mu = \sum_{i=1}^{n} x_i$. Now translate all data samples such that their mean becomes the new origin: $y_i = x_i - \mu, i = 1, 2, \ldots, n$. Denote this new origin as O'. S is a linear subspace that goes through O' and is perpendicular to μ. The intersection of S and the unit hypersphere is a truncated hypersphere with one less degree of freedom. An orthogonal projection of a data sample y_i onto S is found using:

$$y_{p_i} = y_i - \frac{\mu}{||\mu||_2} y_i^T \tag{31}$$

Observe the notation here is slightly different from the 3D example in Figure 3.4. In that example, the origin does not change while here we make O' the new origin so the resulting equations are clearer. y_p is simply equal to $x_p - \mu$ in the 3D example. The projection of y_i in S is then obtained using:

$$y'_i = \alpha \frac{y_{p_i}}{||y_{p_i}||_2} \tag{32}$$

where $\alpha = \sqrt{1 - ||\mu||_2^2}$, the radius of the truncated hypersphere (the orange circle in the 3D example). Equation 32 defines y'_i to be on the same line as O' and y_p, and also makes sure that y'_i is on both the truncated hypersphere and the original hypersphere.

Next, the task is to find a hyper-rectangle in S that yields the minimal quantization error. The vertices of the hyper-rectangle are constrained to be on the truncated hypersphere so that they are also on the original hypersphere. That is to say, we need to find B and C such that:

$$min||Y' - BC||_F^2, \quad s.t. \quad ||B_\forall C||_2 = \alpha, \quad CC^T = D^2 \tag{33}$$

where the i-th row of Y' is y'_i, B is an $n \times b$ binary matrix, $C \in \mathbb{R}^{b \times d}$ and D is a diagonal matrix. Without loss of generality, constrain D to be positive. Since $CC^T = D^2$, we can equivalently represent C as $C = DR$, where $RR^T = I_{b \times b}$ and I is a $b \times b$ identity matrix.

The optimization function now may be written as:

$$\begin{aligned} &\underset{B,D,R}{\operatorname{argmin}} Q(B, D, R) = ||Y' - BDR||_F^2 \\ &s.t. \quad RR^T = I, \quad R\mu = 0, \quad ||B_\forall DR||_2 = \alpha \end{aligned} \tag{34}$$

Similar to the solution method used with ITQ, we propose an iterative method which updates one parameter at a time while keeping others fixed until a local optimum is reached. The alternating steps are listed as follows:

(1) Fix D and R, update B: Compute B as $B = sgn(Y'(R)^T)$. D is ignored since it is composed of all non-negative values.

(2) Fix R and B, update D: Geometrically speaking, D controls the scale and the aspect ratio of the hyper-rectangle. D should be chosen such that all the vertices of the hyper-rectangle land on the truncated hypersphere to ensure that they are also on the original hypersphere. Since the center of the truncated hypersphere, O', is also the center of this hyper-rectangle (due to Theorem 1), we only need to make sure that one of its vertices is on the hypersphere. When this condition is met, then the other vertices are guaranteed to be on it as well. That is to say, we can randomly choose one vertex, e.g., $B_0 = 1_{1\times b}$ such that $||B_0DR||_2 = \alpha$. $1_{1\times b}$ denotes a $1 \times b$ matrix of all $1'$s. This sub-problem then becomes

$$\underset{D}{\operatorname{argmin}} ||Y' - BDR||_F^2 \quad s.t. \quad ||B_0DR||_2 = \alpha \tag{35}$$

Using linear algebraic manipulations, Equation 35 becomes:

$$\underset{D}{\operatorname{argmin}}\, n\sum_{i=1}^{b} D_{ii}^2 - 2\sum_{i=1}^{b}(B^TY'R^T)_{ii}D_{ii}, \quad s.t. \quad \sum_{i=1}^{b} D_{ii}^2 = \alpha^2 \tag{36}$$

where $(B^TY'R^T)_{ii}$ denotes the i-th entry on the diagonal of $B^TY'R^T$ and D_{ii} denotes the i-th entry on the diagonal of D. The solution of Equation 36 is $D = \alpha\frac{diag(B^TY'R^T)}{||diag(B^TY'R^T)||_2}$, where $diag(B^TY'R^T)$ is a vector composed of the diagonal entries of $B^TY'R^T$.

(3) Fix D and B, update R: Since B and D are known, this optimization problem is reduced to the classical Procrustes problem [26]. Take the SVD of $Y'^T BD$ such that $Y'^T BD = U\Sigma V^T$, and then R is computed as $R = VU^T$.

R is initialized to be a random matrix R_0 with orthonormal rows and one more constraint: Each row of R_0 should be orthogonal to μ so that $C\mu = 0$ is satisfied. The criterion for convergence is that $||Y' - BDR||_F^2$ stops decreasing. In practice the proposed iterative technique converges within 50 iterations. Upon convergence, the value of $||Y' - BDR||_F^2$ does not change much given different random initializations. This means that the convergence is stable under different initializations.

3.2.3. Query Sensitivity. We have shown that a data matrix X can be approximated as multiplication of a binary matrix B' and a real-value matrix C', shown as follows (constraints omitted).

$$\underset{B',C'}{\operatorname{argmin}} ||X - B'C'||_F^2 \tag{37}$$

Let us skip for now how B' and C' are found and simply assume that we have found B', C'. For data with unit-length, the similarity between pairs of samples is defined as their inner product. Then the approximated similarity matrix M between a new query sample q and all training data X can be calculated as:

$$M = X_{approx} q^T = B'C'q^T \tag{38}$$

Denoting $w = C'q^T$, we have $M = B'w$. Each row of B' is the binary code of a training data sample and w is a column vector that contains real valued weights for each bit of the binary codes. We observe that the approximate similarity between a training data sample

x and the query q is simply the inner product of B'_x and w, where B'_x is the binary code of x. Therefore, this weight vector defines the order of the Hamming ball search. For example, if $w = [0.5; -0.8; 0.1]$, the 2-bit binary codes will be explored in the following order: $[1, -1, 1], [-1, -1, 1], [1, 1, 1], [-1, 1, 1]$ so that the inner products are ordered from largest to smallest: $[1.4, 0.4, -0.2, -1.2]$. Note that the last entry in w, 0.1, is the weight of the bit corresponding to the mean vector μ. That bit is always set to 1 for all data samples. Therefore, as long as w is known, efficient Hamming ball search can be performed for large-scale datasets. A major difference between our method and others is that other methods often need to compute the binary code of the query before they can perform retrieval, while our method uses the query directly to obtain a weight vector that guides the retrieval. As a result, our method does not approximate the query using a binary code and the similarity estimate is more accurate and more discriminating.

Two major advantages arise out of the weight vector formulation. One is that there is no need to compute the binary code of a query, and this in turn reduces computation. The other is that the approximation of similarity is more accurate since the query is not approximated. Since a new weight vector for the bits has to be computed every time a different query is given, we call this query sensitivity.

Query sensitivity can be related to a distance computation method in Product Quantization (PQ) [7], another popular approach in ANN. In PQ, two options exist for computing the distance of a query and any sample in a dataset. One is called symmetric distance. It maps the query to a binary code and uses stored look-up tables to compute the distance. The other is called asymmetric distance. It uses the original query data directly and generates a look-up table online to compute the distance. The latter option is somewhat expensive in terms

of computation, but it always yields better retrieval performance. UnitQLSH introduces a way to compute asymmetric distances for hashing approaches.

Also different from traditional methods where the (coarse) order of binary codes to explore is easily known by their Hamming distance to the binary code of the query, our method uses the weight vector w to determine the (fine) order of binary codes to explore. In the next section, we present an efficient tree-based algorithm to compute the binary codes that should be explored when a query is given.

3.2.4. An Efficient Tree-based Algorithm to Compute Ranked Binary Codes. We propose to use a weight vector to compute a weighted Hamming distance in order to rank binary codes in a finer level. In this section, we give the algorithm to compute the ranked binary codes given a weight vector computed from a query. Denoting a query as q, the corresponding weight vector as w and the number of bits in the binary code as n_b, w is of length n_b. To make the idea clear, we omit the last bit of the binary code since it is always 1 (corresponding to the mean of data). Given any binary code B of length n_b, its similarity to the query can be computed as their dot product: $< B, w >$. Without loss of generality, we assume that all elements in w are positive.

The first step is to sort the elements in the weight vector in ascending order and record the original indices of the sorted elements. It is apparent that the binary code with the largest similarity is composed of all 1's, yielding the largest dot product. So it is the first binary code that will be retrieved. The next binary code to retrieve is also easy to know. We can obtain it by assigning the first bit -1 and keeping the rest as 1's since the first bit corresponds to the smallest weight. This procedure can be seen as a tree expansion process and this second binary code is a child of the first binary code with one more negative bit. Note that the first binary code has n_b children with one more negative bit but this second

child is the one with the largest similarity among all its siblings. Therefore, we do not need to expand all its children to get the next possible binary code as long as we have its "largest" child. In general, when we retrieve one binary code, we will expand its "next sibling" and its largest children since they are both possible to be picked next. By "next sibling", we mean that the unexpanded sibling with the largest similarity. These two new binary codes (or nodes in the tree) are added to a candidate set containing all expanded nodes and the next retrieved node is chosen from this set by selecting the one with the largest similarity.

Figure 3.5 presents the first few steps of how the tree expansion works. The presorted weight vector w is [1,3,6,8]. Given w, we will retrieve 4-bit binary codes in order. The root of the tree is [1,1,1,1], which has the largest similarity, 18. The root is added to a candidate set C initialized to be empty. In the first iteration, the root is picked and then removed from C since it is the only node in C. Its largest child (child with largest similarity, 16), [-1,1,1,1] is expanded and added to C. The root has no siblings since there is only one binary code with 0 negative bits. In the second iteration, [-1,1,1,1] is picked and removed from C and its next sibling [1,-1,1,1] with similarity 12 and its largest child [-1,-1,1,1] with similarity 10 are expanded and added to C. In the third iteration, [1,-1,1,1] is picked and removed from C since it has the largest similarity. Its next sibling [1,1,-1,1] with similarity 6 and its largest child [1,-1,-1,1] with similarity 0 are expanded and added to C. C now contains [-1,-1,1,1], [1,1,-1,1] and [1,-1,-1,1]. In the fourth step, [-1,-1,1,1] is picked and removed from C. Its next sibling and largest child, [-1,1,-1,1] and [-1,-1,-1,1] are expanded and added to C. The red number on the left of each node shows which iteration that node is expanded and added to C. The first 4 retrieved nodes are [1,1,1,1], [-1,1,1,1], [1,-1,1,1] and [-1,-1,1,1]. The same procedure can go on until a desired number of binary codes are retrieved. The correctness of

this algorithm is guaranteed by the fact that the candidate set C always contains the nodes with largest possible similarity at each level of the tree.

The computational complexity of this tree-based retrieval algorithm is composed of the following. The first operation is to sort the weight vector in ascending order. This cost is negligible since the number of bits is small. Second, in each iteration, one binary code with the largest similarity is selected and removed, and generally two binary codes (the next sibling and the largest child) will be inserted into C. A good data structure for C is a max binary heap. The similarity the query associated with each node is that node's value in the heap. Note that the similarity of a node's (denoted as A) sibling and child does not need to be computed using a full dot product. Instead, it can be obtained by locating which bits are flipped and making a particular adjustment to A's similarity using the corresponding weight values. So the computation of similarity is $\mathcal{O}(1)$. For max binary heap, insertion and deletion are both $log(n)$ where n is the number of nodes in the tree. As a result, to retrieve the first n binary codes, the time complexity is $\mathcal{O}(nlog(n))$; which is very fast.

This proposed tree-based algorithm bears some interesting resemblance to A* search [33], a popular method in artificial intelligence that finds the best path from a starting node to a goal node. These two methods both use a priority queue to guide the next move: In each iteration, the node with the highest priority is picked, then its neighboring nodes are added to the priority queue. For our tree-based method, the priority is the similarity between the binary code and the weight vector, while in A* search, the priority is the inverse path cost (called $f(n)$) if the path goes through the node. It is composed of two parts: $f(n) = g(n) + h(n)$. $g(n)$ is the path cost from start node to the current node n, and $h(n)$ is the estimated path cost from current node n to goal node.

There are some key differences between this tree-based method and A* search. First of all, A* search usually requires users to define a function to estimate $h(n)$, which is called heuristics. Any heuristic would be valid as long as $h(n)$ never overestimates the cost. Therefore, the priority of each node in the priority queue of A* search is an estimate. However, in our method, the similarity between the binary code and the weight vector is simply their dot product, which is an accurate indication of the priority. The second difference is a result of the first difference. Since the priority estimate in our method is accurate, we do not need to update the priority value of existing nodes in the queue, leading to a simple implementation using heaps. However, A* search usually updates the priority values and the implementation of its priority queue is more complicated. The third difference is that, even if we assume A* search uses an accurate heuristic, the structures of the problems suitable for each method are different. A* search is to find the best path between a start node and a goal node. If its heuristic is accurate, then all the nodes on the best path should have the same path cost $f(n)$, i.e., the same priority. On the other hand, in our method, we aim at ranking these nodes by picking the next one with the highest similarity/priority. The nodes we pick often have different priorities.

3.2.5. Location sensitivity + Query Sensitivity. As described in Section 3.1, for LQH, there are two advantages of locating neighborhoods. One is that data in the same neighborhood/cluster can be well approximated by a linear subspace where linear hash functions perform well. The other is that, given a query, we can quickly locate its candidate nearest neighbors by looking at the few neighborhoods that are closest to it, resulting in better hashing performance as well as a significant efficiency improvement.

A disadvantage of LQH is that the number of explored clusters is limited due to the need to assign a binary code for each query. Therefore, some of its groundtruth nearest neighbors

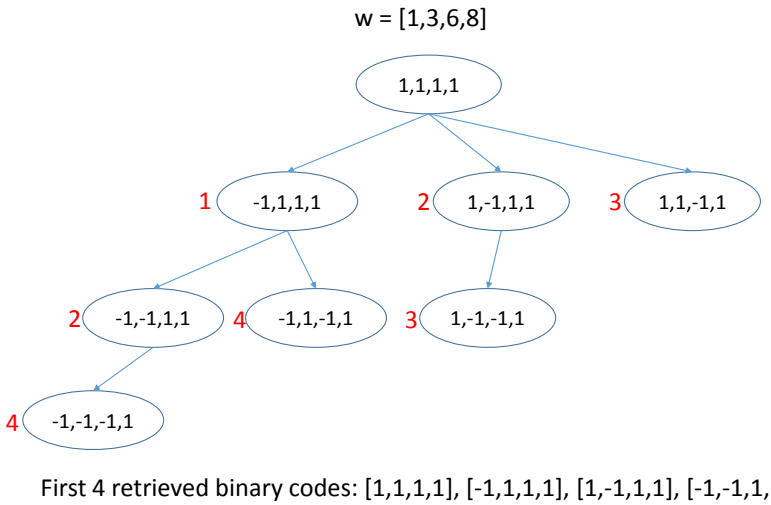


FIGURE 3.5. An example of the first few steps of tree expansion

will never be found. If we can integrate query sensitivity with LQH, there is no need to compute binary codes for queries since queries are used directly. Therefore, any number of clusters can be explored. All the groundtruth nearest neighbors of a given query will always be found as long as the number of explored clusters is large enough (no tradeoff).

In UnitQLSH, query sensitivity can be nicely integrated with neighborhood localization. The procedure is as follows. First, local neighborhoods are found using a clustering method such as K-means. In this paper, we are using a fully vectorized version of K-means [34] which runs very fast. Assume that the number of clusters is K. Denote the k-th cluster as G_k and the data samples in it as X_k where each row of X_k is a data sample in it. Then, for each cluster G_k, we construct a binary matrix B'_k and a real valued matrix C'_k such that $B'_k C'_k$ approximates X_k.

Given a query q, we need to compute the rank of binary codes to retrieve. The algorithm described in Section 3.2.4 can be slightly updated to fit this need. When multiple clusters are explored, a weight vector $w_k = C'_k q^T$ and a max binary heap is computed for

each cluster. The dot product of a weight vector with binary codes in the same cluster computes the similarity of the query and the quantized training data. It is possible that binary codes learned in different clusters will have different quantization errors, yielding different approximations of the similarity. However, we found in our experiments that a direct comparison of the similarity between query and data samples from different clusters achieves great performance.

It is important to note that given a query, it is not necessary to compute a weight vector in each cluster. This is because that most of the query's true nearest neighbors only exist in a few clusters closest to it. Therefore, we only compute a weight for only a few closest clusters and build a max binary heap in each of them.

During retrieval, the next binary code to retrieve is the one with the largest similarity among all roots of the max heaps. A little extra computation is to find the maximal value after obtaining all the roots. Since the number of max heaps is often very small (usually less than 5), the computation complexity can be ignored. After a binary code is retrieved, it is deleted from the corresponding heap, then its next sibling and first child are added to the same heap. This procedure is repeated multiple times until a desired number of data samples or binary codes are retrieved.

This proposed algorithm is both query sensitive and location sensitive. It is query sensitive because weight vectors w_i are computed to weight the Hamming distance (or to guide the search) and these weight vectors are different given different queries. It is location sensitive because given a query, depending on its location, a few clusters closest to it will be searched for nearest neighbors. These chosen clusters can be different given different queries. Also, since the quantizers are constrained to reside on the **unit** hyper-sphere, this algorithm is called Unit Query & Location Sensitive Hashing (UnitQLSH). It should be noted that the

training of UnitQLSH is very fast: It takes about 15 minutes to train on 1 million samples (each with 1000 dimensions).

CHAPTER 4

Experimental Results

This section presents experimental results of the two proposed methods, namely LQH and UnitQLSH. In more detail, common protocols used to evaluate the performance of hashing techniques are introduced first in Section 4.1. Detailed experimental results of LQH are then presented in Section 4.2, where LQH is compared with multiple state-of-the-art methods on multiple large-scale datasets and synthetic datasets. Then a comparison between LQH and UnitQLSH is conducted in Section 4.3. Finally, extensive experimental results of the ultimate method on 3 large-scale datasets, UnitQLSH, are shown in Section 4.4.

4.1. Common Evaluation Protocols

To evaluate the performance of a binary code based hashing method, multiple evaluation protocols can be applied. Common protocols include recall@R curves, precision-recall curves, mean average precision (MAP), precision/recall within a fixed Hamming distance. This section will describe these common protocols in detail.

4.1.1. Precision-recall curves. Precision-recall curves are widely used in the field of document retrieval where the goal is to retrieve relevant documents/items given a query document. This goal is very similar to nearest neighbor search. Therefore, hashing methods can also be evaluated by recall and precision.

Given a query document, let us define the total number of retrieved documents as R and the total number of documents in the gallery that are relevant to the query as M. Recall and precision are defined as follows:

- Precision = number of relevant documents in the retrieved documents / R

- Recall = number of relevant documents in the retrieved documents / M

Note that the numerator in precision and recall is the same. Precision measures the accuracy of the retrieved documents being relevant, i.e., the fraction of relevant documents in the R retrieved documents. Recall measures the fraction of retrieved, relevant documents in all M relevant documents in the gallery. M is a fixed number given a gallery and a query. Ideally, all these M documents should be retrieved since they are all relevant to the query. However, usually only a fraction are retrieved given a limited budget, which is measured by recall. Note that recall=100% can be achieved if all documents in the gallery are retrieved, but the precision is very low. Therefore, recall should be combined with precision to indicate the performance of algorithms.

The number of retrieved documents R can vary, each of which corresponds to a recall-precision pair. As a result, a recall-precision curve can be plotted as a 2D curve by varying R. Usually, the X axis represents recall and the Y axis represents precision. A perfect algorithm should yield a horizontal line whose X axis range is [0, 1] and Y value is always 1. Note that when there are multiple queries, a single precision-recall curve can be generated by averaging the precision at the same recall value for each query.

4.1.2. Recall@R Curves. In recall-precision curves, varying R will lead to a change of precision and recall. Instead of plotting precision at X axis, we can directly plot R at X axis. The Y axis represents recall scores. Each point on the recall@R curve measures, when R documents are retrieved, how many relevant documents they contain. Since we can calculated number of relevant documents in the retrieved documents by $M \times recall$, we can calculate precision via $\frac{M \times recall}{R}$. Therefore, theoretically speaking, recall@R curves can be converted to precision-recall curves, and vice versa.

4.1.3. Mean Average Precision. It is sometimes helpful to quickly evaluate hashing algorithms using a single score rather than generating curves. The most widely used score is mean average precision (MAP). MAP is computed by taking the mean of average precision (AP) of each query. So we need to know how to compute AP of a query. AP measures the area under the precision-recall curve.

$$AP = \int_0^1 precision(r)dr = \sum_{k=1}^{n} precision(k) \triangle r(k) \tag{39}$$

where n is the number of retrieved documents and r is the recall value. In general, n should be equal to the total number of documents in the gallery so that the range of integral is $[0, 1]$. However, to speed up computation, a much smaller n can sometimes be used. In these cases, the range of the integral in Equation 39 is cutoff accordingly.

4.1.4. Precision Values and Recall Values within a Fixed Hamming Distance. When retrieval is done by Hamming ball exploration centered on a query, it is desirable that its nearest neighbors exist in the top binary codes closest to it. This is because that the Hamming ball exploration can be expensive if the Hamming distance is large. For example, for a 32-bit code, the number of binary codes having a Hamming distance of 3 to a given query is $\binom{32}{3} = 4960$. As a result, if a true nearest neighbor is not very close to the query, a large number of binary codes have to be explored before this neighbor is retrieved. Precision values and recall values within a small fixed hamming distance takes this issue into consideration. It measures the average precision and recall score within a very small hamming distance such as 2. High recall scores indicate that many true nearest neighbors are actually retrieved in the first few explorations. High precision scores mean that only a few mistakes are made in these retrieved neighbors.

4.2. Experimental Results of LQH

This section presents two sets of experiments. The first compares Local Quantization Hashing (LQH) to Iterative Quantization (ITQ) [15] and K-Means Hashing (KMH) [17] on synthetic datasets with known distributions. Although the synthetic data are not realistic, these experiments provide insights into the nature of the three algorithms. The second set of experiments compare LQH to ITQ, KMH, Spectral Hashing (SH) [11] and Locality Sensitive Hashing (LSH) [10] on two large-scale datasets, SIFT1M and GIST1M.

4.2.1. Experiments on Synthetic Datasets.

We compare LQH to two state-of-the-art hashing methods, Iterative Quantization (ITQ) and K-means Hashing (KMH) on four synthetic datasets. The goal is to show for which kinds of data LQH works well, and for what types of data other algorithms perform better. This may provide insights as to why LQH works well for real datasets.

Each synthetic dataset contains 1 million training samples and 2000 query samples, and every data sample has 128 dimensions. The datasets are generated as follows:

- **Uniform**: Each dimension of a data point is independently sampled from a uniform distribution between 0 and 1.
- **Gaussian**: Each dimension of a data point is independently sampled from a Gaussian distribution with a mean of 0 and a standard deviation randomly chosen from [0, 1] for each dimension. The covariance matrix is therefore diagonal.
- **Mixture of Gaussians**: 40 cluster centers are randomly generated, and points are generated for each cluster from a multivariate Gaussian distribution whose mean is the cluster center. The covariance matrix of each Gaussian distribution is also a random diagonal matrix. The average magnitude of the covariance matrix is

designed to be less than the distances between any two cluster centers so that these clusters can be easily separated.

- **Nonlinear**: For each data point, the first 64 dimensions are randomly sampled from a normal Gaussian distribution. The $64 + i$th dimension is a cosine function of the $i$the dimension, where $i = 1, 2, \ldots, 64$. The format of the cosine function is $X_{64+i} = cos(wX_i + b)$. The parameter w of the cosine function is randomly chosen from a Gaussian distribution with a mean of 0 and a standard deviation of 8. The other parameter b is randomly chosen from a Gaussian distribution with a mean of 0 and a standard deviation of 3. The resulting 128 dimensional data points reside in a nonlinear subspace of 64 dimensions.

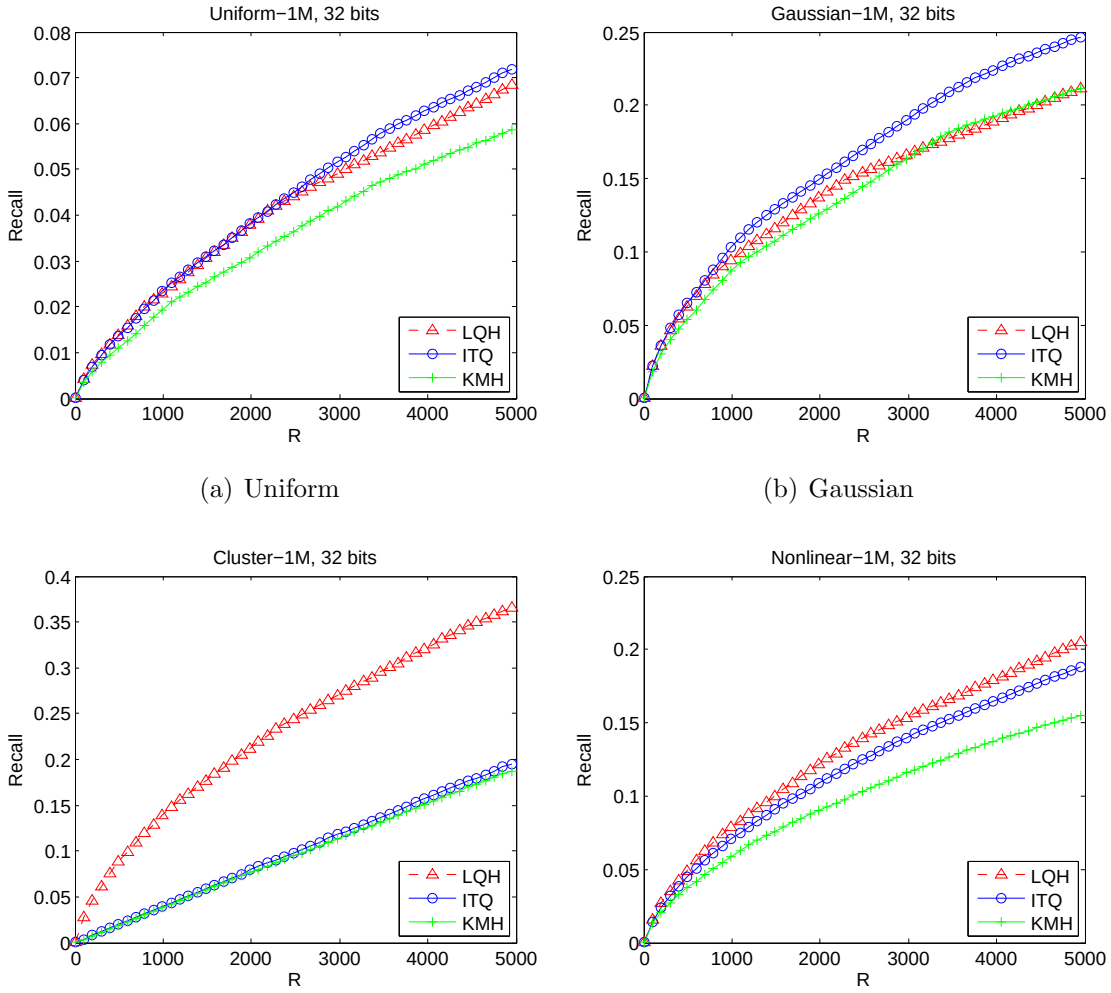
The ground truth nearest neighbors for every query are defined as the closest 100 samples in the training set, in terms of Euclidean distance. Figure 4.1 shows the recall@R curves for each dataset using LQH, ITQ and KMH to generate 32 bit binary codes. The horizontal axis shows the number of top ranked nearest neighbors and the vertical axis shows the recall score. For LQH, we use K-means with 16 clusters. This means that $log_2(16) = 4$ bits are used to encode the index of the cluster and the remaining 28 bits are used to store the learned binary representation. For each query, we explore at most 2 nearest clusters in LQH. The phrase "at most" means that we will only explore the next closest cluster if the distance between its center to the query is not much larger than the distance between the center of the previous cluster to the query. For KMH, we set the number of bits per subspace as 4 in all experiments.

For the comparison on UNIFORM DATA shown in Figure 4.1(a), LQH and ITQ have very similar performance at the lower range of R. LQH is worse than ITQ for large values of R because it explores at most 2 nearest clusters for each query, and some true nearest neighbors

can be missed. The result in Figure 4.1(a) tells us that for uniform data, clustering does not improve the performance of ITQ. For the comparison on the Gaussian data shown in Figure 4.1(b), ITQ performs consistently better than LQH. This could be because that a Gaussian distribution is ideal for the PCA model in ITQ. Again, Figure 4.1(b) indicates that clustering does not help improve ITQ when data come from a single Gaussian distribution. A major reason why clustering fails to help for uniform and Gaussian data may be that the whole data set already forms a single linear space and there is no need to subdivide it.

For the mixture of gaussians data in Figure 4.1(c), LQH yields much better performance than either ITQ or KMH. LQH performs well for two reasons. One is that a mixture of Gaussians can be divided into a set of smooth linear subspaces where ITQ performs well. The other is that, for each query, almost all of its true nearest neighbors reside in the top few closest clusters explored by LQH. The chance of missing true nearest neighbors is very small. It is interesting to note that although KMH is in some sense similar to K-means clustering, it does not perform well on this test. Figure 4.1(d) presents the comparison on a dataset composed of data samples living in a low-dimensional nonlinear subspace. LQH approximates local neighborhoods of its nonlinear structure using linear subspaces; ITQ and KMH do not. The recall@R curves show that LQH is a better fit than ITQ and KMH for this kind of data. We suggest that LQH outperforms other methods on real datasets because many reside in nonlinear low-dimensional subspace, and/or have multiple clusters.

4.2.2. Comparison between LQH and Others on Real Datasets: SIFT1M and GIST1M. This section presents a comparison of LQH between other multiple state-of-the-art hashing algorithms on two large-scale datasets with real data. One is SIFT1M dataset and the other is GIST1M dataset. A brief of each dataset is described and experimental results are shown. The performance of each algorithm is evaluated by recall@R curves.



(a) Uniform

(b) Gaussian

(c) Mixture of Gaussians

(d) Nonlinear

FIGURE 4.1. Comparison of hashing approaches on 4 synthetic datasets

In the SIFT1M dataset, both the training and query data samples are local SIFT features [35] extracted from the INRIA Holidays images [36]. In this dataset, there are 1 million training data samples and 10000 queries. Each sample is a 128-dimension vector. For each query, the ground truth nearest neighbors are defined as the 100 closest training samples in terms of Euclidean distance. For KMH, we use the recommended parameter settings. For SH, the radius in its RBF function is set to one half of the mean distance in the training set. We present the recall@R curves at different bit-lengths: 32, 64 and 128 in Figure 4.2. Note that the performance of the kernel version of ITQ (KITQ) is worse than ITQ when the

number of bits is small, therefore we do not include KITQ in the comparison. Considering the factors of efficiency and accuracy, we set the parameters of LQH, namely N_c, M and σ as 16, 2 and 0.1 respectively in all of the following experiments.

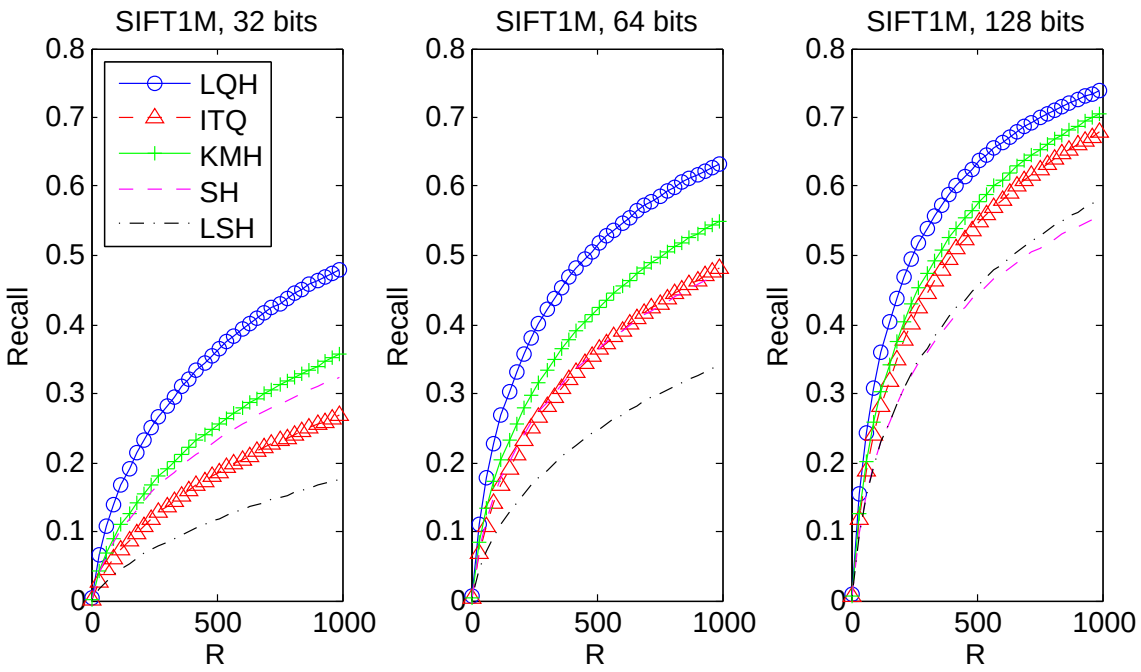


FIGURE 4.2. Recall@R curves on SIFT1M using 32, 64 and 128 bits

It can be observed in Figure 4.2 that the recall score of LQH is much higher than other methods when the number of bits is small, as shown in the left and middle subplots. When the number of bit is equal to 128, we see an interesting intersection of the LQH curve and other curves such as KMH or ITQ. LQH works best when the number of retrieved nearest neighbors is low, for example when $R < 2000$. In contrast, KMH and ITQ outperform LQH as R becomes large. This result is within our expectation since we only explore at most the top 2 closest clusters given a query. It is possible that there are some true nearest neighbors of a query that lie in other clusters that never get selected. Therefore, the maximal recall score will not achieve 100% even when R is very large. However, for many applications such as recommender systems, retrieving a large number of candidates for a query is neither

practical nor efficient. A much more important property of a hashing method is that it can return a lot of relevant items when the number of retrieved items is limited, i.e., high recall when R is small. Our proposed LQH is attractive from this perspective. LQH also works better than the other methods when the number of encoding bits is small.

For the other dataset GIST1M, there are 1 million training data samples and 1000 queries. These samples are color GIST global descriptors [37] extracted from a subsample of the tiny image dataset [38]. Each sample is a 960-dimension vector. For each query, the ground truth nearest neighbors are also defined as the 100 closest training samples in terms of Euclidean distance. We present the same set of results as shown in Figure 4.3.

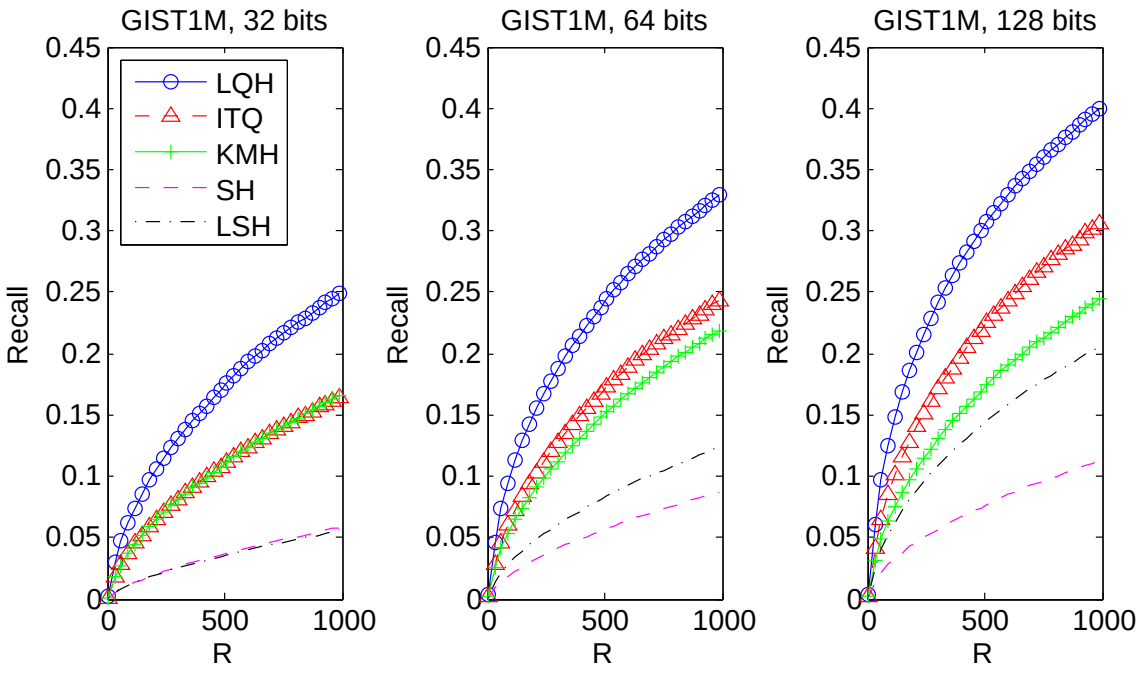


FIGURE 4.3. Recall@R curves on GIST1M using 32, 64 and 128 bits

In Figure 4.3, we can see that LQH achieves the best performance in all cases. Again, in all three cases, the recall@R curve is very steep at the lower range of R, which is highly desirable for approximate nearest neighbor search. For instance, for the 32-bit case, the

recall score given by LQH almost doubles the recall scores of ITQ and KMH when R is small.

4.3. Comparing UnitQLSH to LQH

This section shows experimental results of UnitQLSH compared to LQH and other state-of-the-art methods. Figure 4.4 shows the recall@R curves of different algorithms on SIFT1M data using binary codes of 32 and 64 bits. The data are normalized to be length 1 for all algorithms. AQBC is proposed by Gong et al. in [16]. QLSH (NonUnit) is created specifically to show the advantages of UnitQLSH. It is similar to the proposed UnitQLSH, except that the quantizers are not constrained to be on the unit hypersphere.

In Figure 4.4, it can be seen that all UnitQLSH, QLSH and LQH greatly outperform AQBC, a state-of-the-art hashing method designed for unit-length data. Comparing QLSH to LQH, we can see that when the number of bits is 32, QLSH performs slightly better than LQH. Since QLSH takes advantage of both query and location sensitivity while LQH only considers location sensitivity, it shows that query sensitive is helpful in this case. However, when the number of bits becomes larger, e.g., 64 bits, LQH shows superior performance. This reveals a potential drawback of QLSH: Since the quantization of training data is not constrained to be unit length, the dot product of a query q (unit length) and a quantized training sample $\hat{x}$ measures $\hat{x}$'s projection on q rather than the cosine of their angle. This projection may not be a good approximation of their true cosine similarity. As demonstrated by the superior performance of UnitQLSH, when the quantization is constrained to be unit length, the true similarity between query and training data is very well preserved. Moreover, the performance increase of UnitQLSH is very significant compared to others in both small and large code lengths.

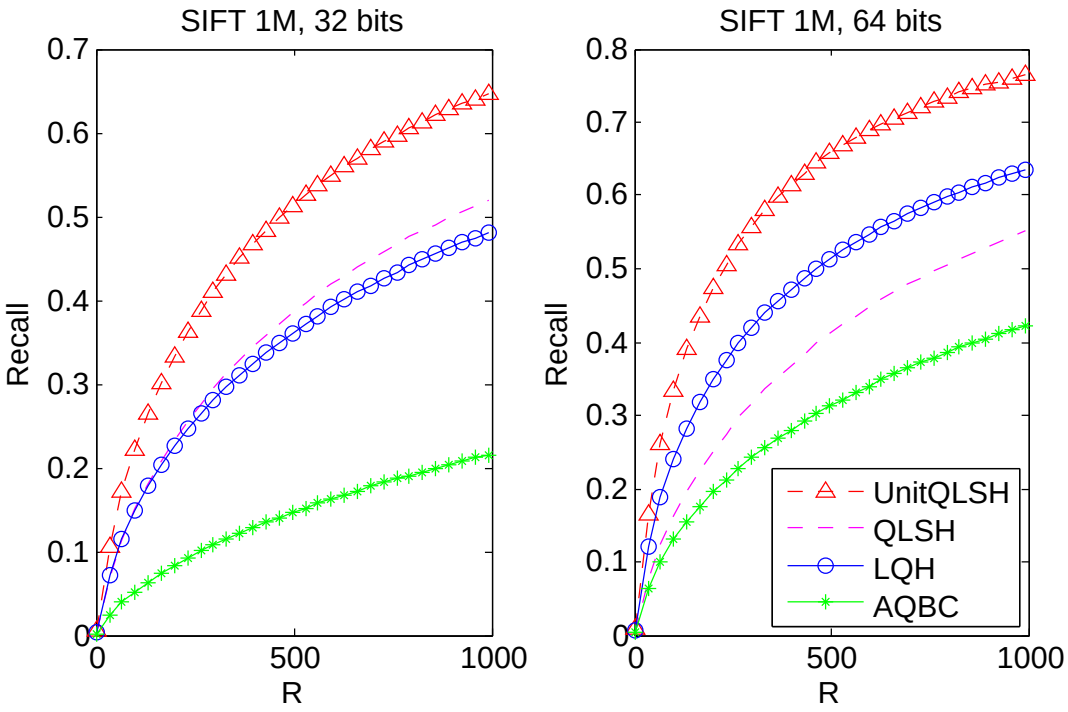


FIGURE 4.4. Recall@R curve: Comparing UnitQLSH to QLSH, LQH and AQBC on SIFT1M using 32 and 64 bits.

Although we only show the comparison on SIFT1M in Figure 4.4, we have found that UnitQLSH always outperforms LQH in all of the future experiments using different datasets. Therefore, we will not include LQH in our following experiments where we compare UnitQLSH to other state-of-the-art methods.

4.4. Experimental Results of UnitQLSH on 3 Datasets

In this section, a number of experiments are performed that compare the proposed UnitQLSH to multiple state-of-the-art algorithms on three large-scale datasets. Besides SIFT1M and GIST1M, we adopt another challenging large-scale dataset called ImageNet to further explore the potential of UnitQLSH.

In ImageNet, there are 14,197,122 images in total. Among these images, 1,261,406 of them are associated with SIFT features. Also, 1000-dimensional Bag of Words (BoW) features are computed using these SIFT features and are available on ImageNet website. More

importantly, these images are associated with category labels so we can easily know how good a retrieval is by looking at the label of the retrieved items. There are in total 1000 category labels. We use these 1.2 million images to conduct an image retrieval task. The features we use for these images are their 1000-dimensional BoW features. It should be noted that the label information is not used since all methods in this paper are unsupervised. Among these 1,261,406 images, we randomly pick 2 images from each category to form 2000 query images. The rest of the images are used as our training/gallery set. The BoW features are normalized. For each query, its ground-truth neighbors are defined as the closest 100 gallery samples in terms of Euclidean distance, or equivalently, cosine similarity.

Section 4.4.1 shows the recall@R curves of different algorithms. Section 4.4.2 shows their recall and precision within a fixed Hamming radius. Section 4.5 shows that the clustering algorithm can be vastly accelerated. It also presents experimental results where the number of clusters is varied to show how this is affecting the retrieval performance. Section 4.6 presents a practical perspective towards efficient retrieval in large-scale datasets. Section 4.7 presents the method of how to integrate UnitQLSH with a distributed system to handle an extremely large amount of data.

4.4.1. Recall@R curves on 3 datasets. Figure 4.5 shows the recall@R curves of different algorithms using 32, 64 and 128 bits on ImageNet. R denotes the number of retrieved items. Note that for the case of 128 bits, KMH is too time-consuming to be included in the comparison because the number of clusters has to be 2^16 in order to make KMH work.

In these three plots, we can easily observe that UnitQLSH performs the best. Moreover, the performance improvement is very significant compared to other state-of-the-art methods. Also, we note that the two variants of UnitQLSH, Q-ITQ and NonUnit-QLSH are performing

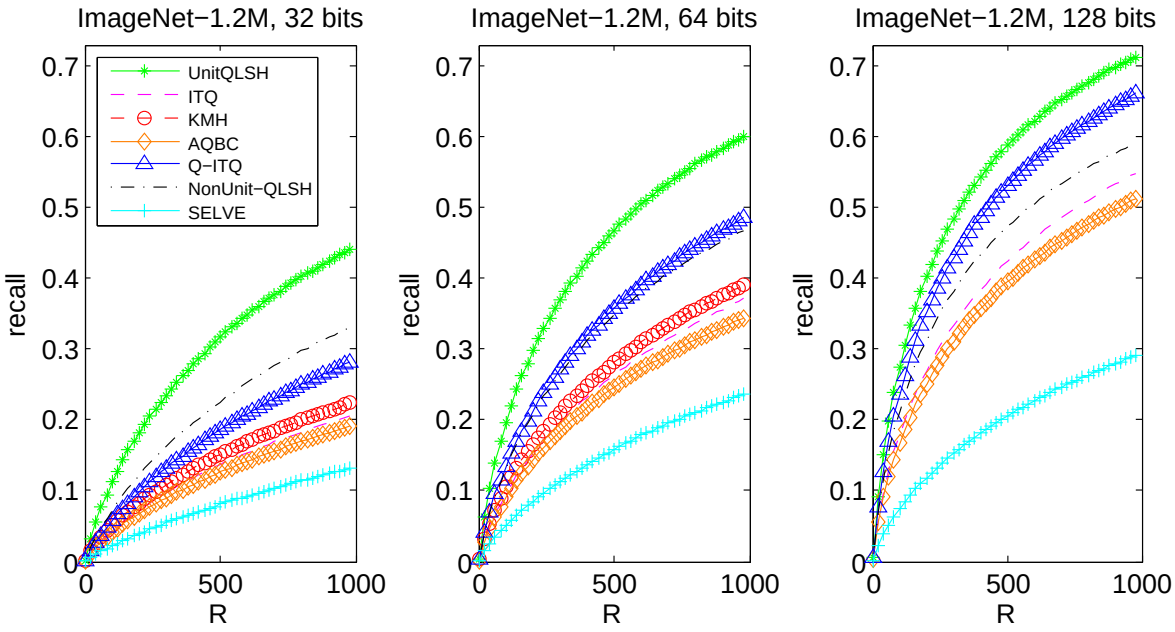


FIGURE 4.5. Recall@R curve: Comparing UnitQLSH to ITQ, KMH, AQBC, Q-ITQ and NonUnit-QLSH using 32, 64 and 128 bits on ImageNet.

better than other state-of-the-art methods as well. It demonstrates the effectiveness of each part of the proposed approach. In particular, the result of Q-ITQ demonstrates that the query sensitivity is very useful and the result of NonUnit-QLSH shows that the integrating location sensitivity with query sensitivity helps improve the retrieval performance. AQBC performs poorly on this dataset. This may be due to the over-relaxation of its optimization function. SELVE yields the lowest result compared to others.

Figure 4.6 shows the recall@R curves of different algorithms using 32, 64 and 128 bits on GIST1M. Note that the performance of AQBC cannot be measured on this dataset because AQBC only works on positive data while GIST1M contains negative data. Again, we observe that UnitQLSH outperforms all other methods with small, median and large code lengths. On this dataset, we can see that NonUnit-QLSH performs worse than Q-ITQ in all cases and it performs the worst of all when the number of bits is 64, while UnitQLSH stay the best. This result also shows that the constraint that the binary code should reside on the

unit hypersphere is very helpful for the retrieval of unit-length data. Also, we see again that Q-ITQ is working very well, which means that query sensitivity is playing an important role in improving retrieval performance.

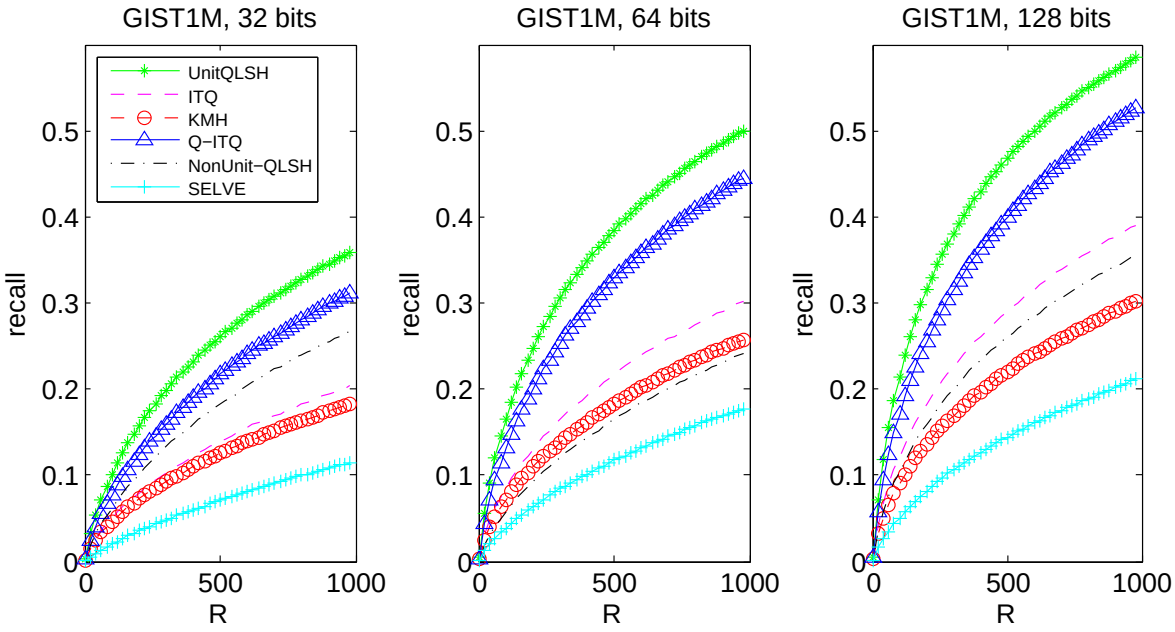


FIGURE 4.6. Recall@R curve: Comparing UnitQLSH to ITQ, KMH, Q-ITQ and NonUnit-QLSH using 32, 64 and 128 bits on GIST1M.

The same set of experiments are also conducted on SIFT1M. Figure 4.7 shows the recall@R curves of different algorithms using 32, 64 and 128 bits. We observe a result similar to the ImageNet dataset. One exception is that the relative performance of NonUnit-QLSH drops significantly when the number of bits becomes large.

Another interesting finding in these plots is that when the number of bits is small, we can see that NonUnit-QLSH outperforms Q-ITQ on SIFT1M and ImageNet. When the number of bits increases, Q-ITQ performs better. This reveals a potential drawback of NonUnit-QLSH: Since the quantization of training data is not constrained to be unit length, the dot product of a query q (unit length) and a quantized training sample $\hat{x}$ measures $\hat{x}$'s projection on q rather than the cosine of their angle. This projection may not be a good approximation

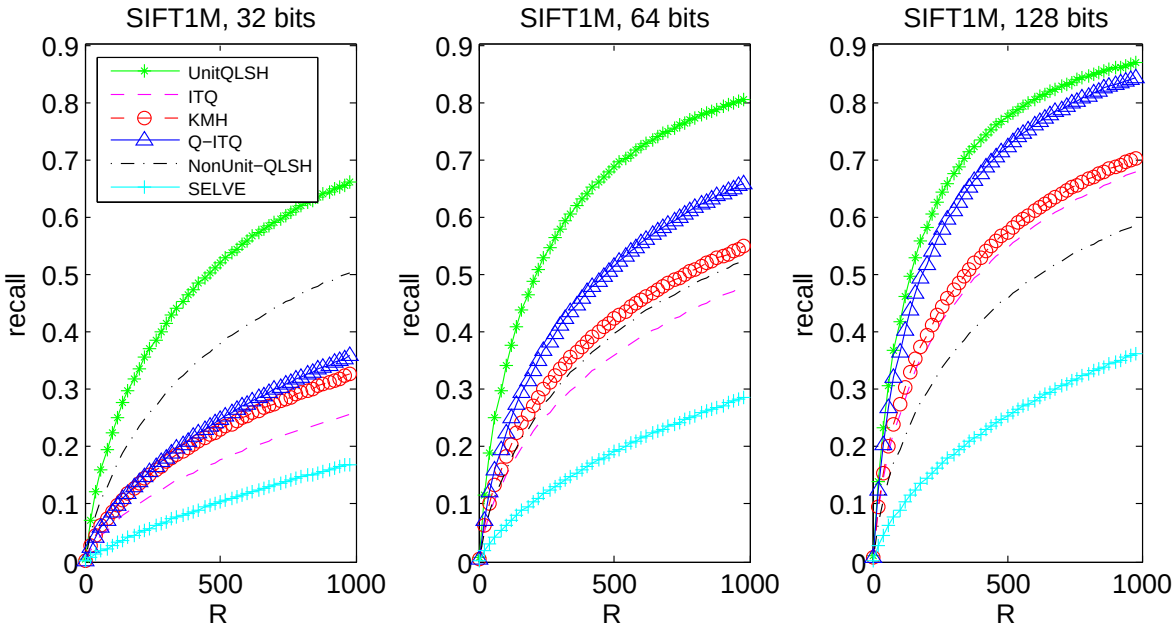


FIGURE 4.7. Recall@R curve: Comparing UnitQLSH to ITQ, KMH, AQBC, Q-ITQ and NonUnit-QLSH using 32, 64 and 128 bits on SIFT1M.

of their true cosine similarity. As demonstrated by the superior performance of UnitQLSH in all cases, when the quantization is constrained to be unit length, the true similarity between query and training data is very well preserved.

4.4.2. PRECISION AND RECALL WITHIN FIXED HAMMING RADIUS ON 3 DATASETS. Another important evaluation protocol of hashing methods is the recall and precision of different algorithms when the retrieval is done within a Hamming radius of 1 and 2. Note that since UnitQLSH computes a weight vector to weigh the Hamming distance, the weighted Hamming distance is in general a floating point number with a much finer range of values. To perform a fair comparison, we calculate the number of binary codes within a Hamming radius of 1 and 2 for unweighted cases and retrieve the same number of binary codes for UnitQLSH. For example, for a 32-bit case, we will retrieve 33 binary codes if the Hamming radius is 1 and 529 binary codes if the radius is 2. More importantly, due to the weight vector, we are able to rank the binary codes in a much finer level. As a result, we are able

to plot the precision and recall scores when varying the number of retrieved binary codes smoothly. In contrast, other conventional hashing methods are not able to distinguish binary codes with the same Hamming distance to the query. Therefore, the number of binary codes they retrieve will vary rapidly when increasing the Hamming radius. If one wants to retrieve the first 2000 binary codes with 32 bits, random sampling needs to be done among all the binary codes with a Hamming radius of 3, which can lead to a drop of accuracy. Using the proposed UnitQLSH, one can conveniently adjust the number of binary codes to retrieve nearest neighbors with high accuracy.

Figure 4.8 shows the average precision and recall scores of different algorithms at Hamming radius of 0, 1 and 2. SELVE and AQBC performed poorly so their result is not presented in the figure. The number of bits is 32 since the recall scores within a low Hamming radius will be extremely low when the number of bits increases, which is not feasible for real applications. We also show the number of retrieved binary codes within each different Hamming radius in the figure. For conventional algorithms, there are actually only 3 points, each corresponding to a Hamming radius, since that is the finest level to rank the binary codes. However, for UnitQLSH, the number of binary codes can be ranked smoothly. In this plot, we take a increasing step of 20 binary codes and plot the corresponding recall and precision. Each dot on the line is one step. For some dots (if too many, text will overlap), we show the number of binary codes so readers can have an idea of how the number of retrieved binary codes can affect the precision and recall scores. Readers can easily estimate the position of 529 binary codes (radius of 2) and 33 binary codes (radius of 1) on the line to compare directly to other algorithms.

Before we compare the results in Figure 4.8, there is one additional important factor in retrieval using binary codes that we need to bring up: the number of items that are retrieved.

This number is important because it relates to the actual time spent on data retrieval in practice. For retrieval in large-scale datasets, original data are often too large to be stored in memory. Therefore, data samples are usually stored on local disks. What is stored in the memory is a hash map. A hash map a data structure that can map keys to values. In this case, the keys are the binary codes and the values are generally some sort of identifiers of the corresponding data samples. A key property of hash maps is that, given any key, mapping it to the associated value is extremely fast. Usually, the complexity of this mapping is either $O(1)$ or $O(logn)$, depending on the implementation details. The hash map is very small comparing to all the original data samples.

Therefore, only the hash map is small enough to be loaded into memory. It is well known that disk I/O is much slower than accessing memory. Therefore, if a large number of items are to be retrieved, the actual retrieval time can be long due to the disk I/O. As a result, one would like the number of retrieved items to be small. Also, to retrieve a reasonable number of true nearest neighbors with a limited number of retrieved items, the precision should be high. The background color of Figure 4.8 shows the average number of retrieved items corresponding to each precision-recall pair. The color bar on the right side shows the mapping from a color to the actual number. The formula to calculate this number is $100 \cdot recall/precision$, where 100 is the number of true nearest neighbors for each query. To show detailed gradation of the color, we set an upperbound of the color bar. The actual range of the color bar is from 0 (close to vertical axis) to infinity (close to the horizontal axis). However, it is impossible to capture this range. This is why we set an upperbound of the color bar manually.

Comparing UnitQLSH to KMH and ITQ in Figure 4.8, we can see that at Hamming radius of 0, 1 and 2, UnitQLSH yields good recall scores and much higher precision scores

than others. At the same recall, UnitQLSH has much better precision scores. Both the precision and recall of UnitQLSH are higher than KMH at the same number of retrieved binary codes. Although ITQ has a higher recall score, its precision is very low and it retrieves a lot more items with the same number of binary codes and thus is not appealing in practice. UnitQLSH retrieves a more limited number of items with high precision, which is very desirable in practice. Also, since UnitQLSH can rank binary codes in a finer level, a user of UnitQLSH is able to gradually vary the number of binary codes to better meet the needs of specific applications.

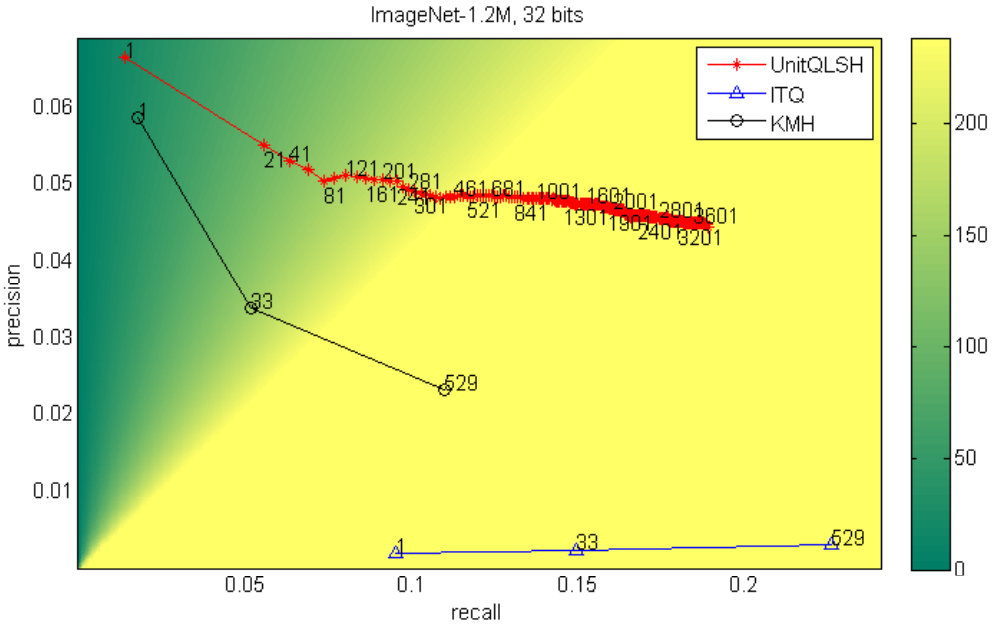


FIGURE 4.8. Precision vs. Recall curve: Precision and Recall with different Hamming radius / numbers of binary codes for 32 bits on ImageNet. The setting of this experiment is explained in detail in this section: Section 4.4.2

Figure 4.9 shows the precision and recall scores when varying the number of retrieved binary codes on GIST1M. We observe again that, compared to the other two algorithms, UnitQLSH yields very high precision and good recall scores. It yields much higher precision when comparing to others at the same recall. ITQ has very low precision and it still retrieves

a very large number of items even within a short Hamming radius, making it infeasible for real large-scale retrieval applications. Compared to ITQ, KMH also has higher precision and retrieves a small number of items. The precision and recall of UnitQLSH are higher than KMH when the Hamming radius is 1 and 2.

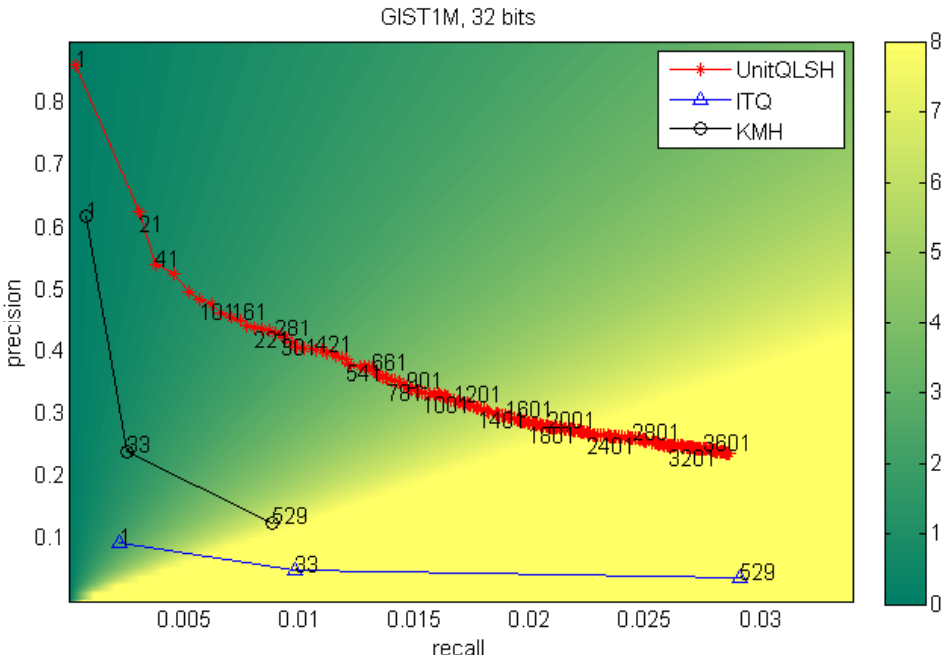


FIGURE 4.9. Precision vs. Recall curve: Precision and Recall with different Hamming radius / numbers of binary codes for 32 bits on GIST1M.

Figure 4.10 shows the precision and recall scores of UnitQLSH, KMH and ITQ when varying the number of retrieved binary codes on SIFT1M. UnitQLSH yields higher precision and recall than KMH at Hamming radius of 1 and 2. UnitQLSH also has higher precision at the same recall, except in the one case where KMH outperforms UnitQLSH when only a single binary code is retrieved. ITQ still has relatively high recall rates, but very low precision, indicating the algorithm retrieves more items.

The result that ITQ is retrieving more items than UnitQLSH and KMH within the same Hamming radius is interesting. Therefore, we plot the distribution of all the non-empty

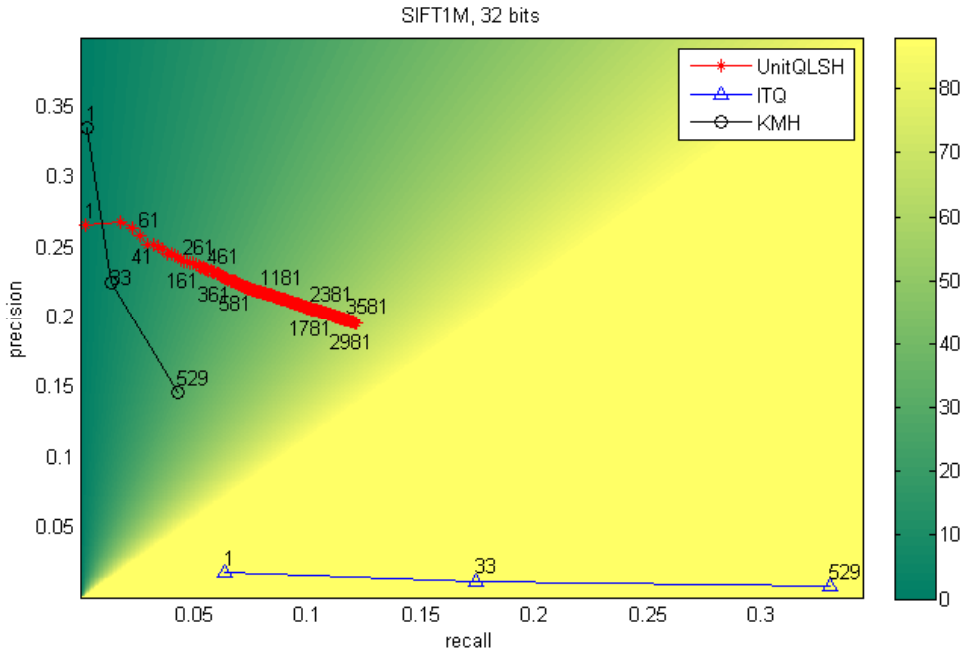


FIGURE 4.10. Precision vs. Recall curve: Precision and Recall with different Hamming radius / numbers of binary codes for 32 bits on SIFT1M.

32-bit binary codes (binary codes containing at least one item) learned on training data for each algorithm in Figure 4.11. The top, middle and bottom subplot corresponds to ITQ, KMH and UnitQLSH, respectively. The horizontal axis shows the indices of binary codes and the vertical axis shows the number of items mapped to the same binary code. We set an upper bound of the vertical axis to be 1000 so that readers can see the details of the lower range of the distribution. There are actually some binary codes containing more than 1000 items. In the title of each subplot, we also report the total number of non-empty binary codes, the number of binary codes with 1, 2 and more than 2 items. For instance, the title of the top subplot (ITQ) says: "Total-720355, 1-620392, 2-52116, ¿=3-47847". It means that among all the binary codes learned by ITQ, 720355 of them correspond at least one data sample. Among these 720355 samples, 620392 binary codes correspond to 1 data sample, 52116 binary codes correspond to 2 data samples and 47847 binary codes correspond to the rest (great than or equal to 3 data samples).

It can be seen that UnitQLSH and ITQ have the largest and smallest number of non-zero binary codes, respectively. A number of codes in ITQ contain a lot of items while UnitQLSH is more evenly distributed. The distribution of the code of KMH has certain properties of both ITQ and UnitQLSH: It is not as evenly distributed as UnitQLSH but is more even than ITQ. The reason why ITQ retrieves a large number of items within a small Hamming radius can be explained as follows. For each query, there probably exist a few number of binary codes within the Hamming radius that contain many items. This happens less often using UnitQLSH and KMH.

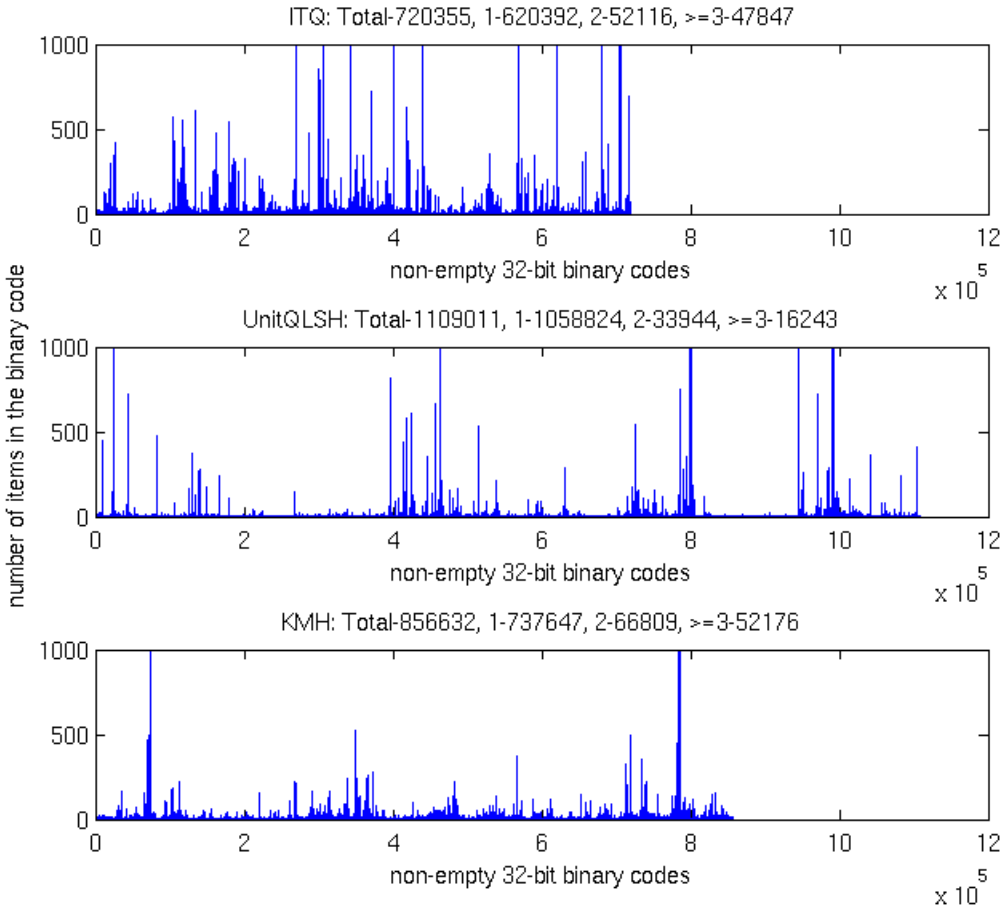


FIGURE 4.11. Distribution of non-empty 32-bit binary codes learned using ITQ, KMH and UnitQLSH on ImageNet.

4.5. The Influence of Clustering Step

The first step of learning binary codes using UnitQLSH is to cluster the data using K-means. Clustering is a very important step since data in the same local neighborhood are better quantized with binary codes (a hyper-rectangle) compared to conducting quantization in all the data. Moreover, during query time, only a few local neighborhoods are selected for exploration, yielding higher accuracy and efficiency. However, clustering can be computationally expensive when the number of data samples, the number of dimensions of data or the number of desired clusters is very large. Tricks can be applied to accelerate K-means. One can apply K-means only on a subset of the data or find a better initialization instead of a random initialization.

It may surprise some that the K-means implementation in MATLAB has a lot of room for improvement, even without applying any of the approximation tricks mentioned above. Deng [34] has made available a fast version of K-means in MATLAB. The acceleration is achieved via a full vectorization of data and an adoption of sparse matrices to store the index of each data sample. As a result, the computation can be highly parallel without using any explicit parallelism. The time saving is tremendous: it only took about 7 minutes to cluster one million data samples (each with 960 dimensions) into 16 clusters. Note that this acceleration does not sacrifice the retrieval performance of UnitQLSH.

The number of clusters, denoted as K, should not be too small or too large. The reasons are as follows. K roughly controls the size of each local neighborhood approximated by a linear subspace. It seems that the smaller these neighborhoods are, the better they will be approximated by linear subspaces. However, this is only true for continuously distributed data samples. For real world cases, it is often incorrect to assume that data are continuously distributed. Also, since a subset of the bits in binary codes are used to indicate the index

of the cluster, fewer bits will be assigned to each cluster if the number of clusters becomes large. Moreover, if K is too large, clustering will become very computationally expensive.

Figure 4.12 shows the recall@R curves when varying the number of clusters in training UnitQLSH. This experiment is conducted on both SIFT1M and GIST1M. The number of clusters K is set to 1, 8, 16, and 32 on both datasets. $K = 1$ is a special case where no clustering is performed in the training. In all other cases where K is greater than 1, we explore the 3 closest clusters when searching for nearest neighbors of a query.

It can be seen from Figure 4.12 that $K = 1$ performs poorly compared to others, indicating that clustering plays an important role in improving the retrieval performance of UnitQLSH. When K increases, the recall score increases in general. However, the increase is not very significant. Considering both the computational burden of K-means and the slight performance increase brought by large K, we have chosen K to be 16 in all of our experiments.

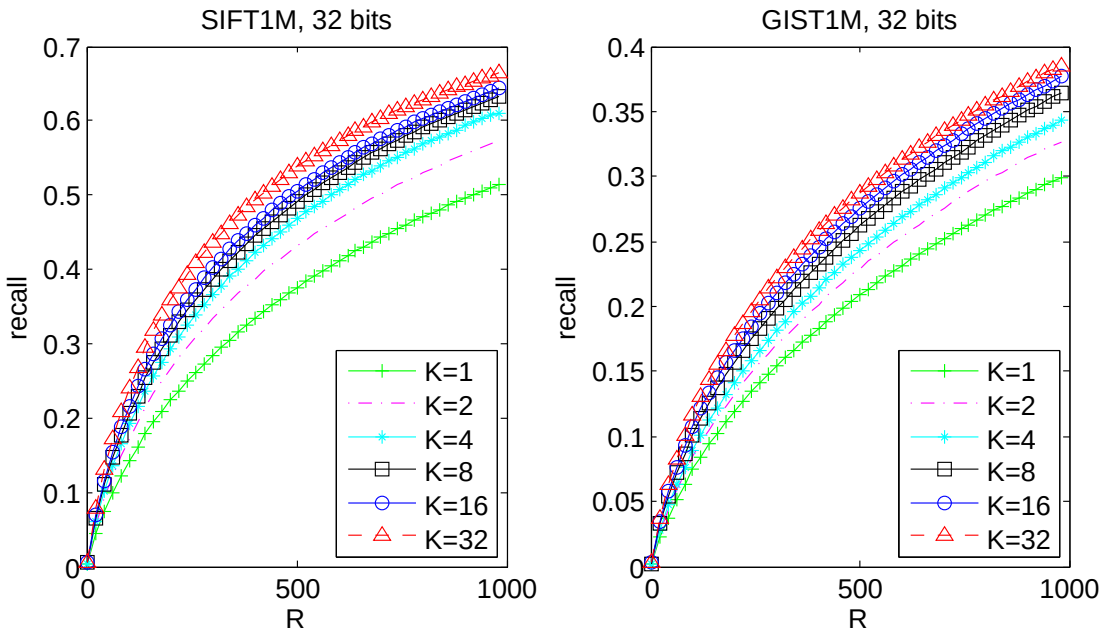


FIGURE 4.12. Recall@R curves of UnitQLSH when varying the number of clusters.

4.6. Looking at Data Retrieval from a Practical Perspective

The number of bits for a binary code cannot be large because the total number of binary codes will increase exponentially with respect to the number of bits. In our experiments, we found that when the number of bits is 64, there are so many binary codes in the hash map with no content that few items can be retrieved within a reasonable search range since the number of data samples is 1 million. As a result, we regard 32 bits as a reasonable choice. Beyond that, it can be difficult to conduct quick retrieval in practice.

Let us assume that we have already learned a binary code for each gallery/training data sample. Given a query, we will first compute its binary code and then search similar binary codes to retrieve its nearest neighbors. The time complexity required to retrieve its nearest neighbors is then composed of two parts. One is the hash map lookup using binary codes as indices: given a particular binary code, provide the data items associated with that binary code. The other part is to load data samples given the content (e.g., indices of images) in that binary code. As a result, the total complexity of finding nearest neighbors without any post-processing can be formulated as: $\mathcal{O}(C_1 N_{bc} + C_2 N_{img})$. In this equation, C_1 represents the time spent on performing a look-up using a binary code in the memory and C_2 the time spent on loading a data sample from disk into memory.

Imagine the following application. Given a large-scale image dataset, the goal is to provide the user, in real time, with a limited number of similar images when he/she provides a query image. Often the dataset is too large to be loaded to memory so each image is stored on a local hard disk. Once the hash map with binary codes is learned and built, each image is concisely represented by a binary code. Therefore, the whole hash map can be loaded to memory and look-ups can be done very quickly. Also assume that for each query, there are a large number of gallery images similar to it. Our experiments show that $C_2 : C_1 = 17826 : 1$

in Python language and $C_2 : C_1 = 1000 : 1$ in MATLAB. Although the ratio of C_1 and C_2 differs in different programming languages, we can see that as long as the number of retrieved binary codes is not extremely different among algorithms, the cost of hash map lookup can be ignored during comparison. As a result, to retrieve the same amount of actually similar images, algorithms with higher precision win. This is because they retrieve fewer total images, yielding higher efficiency. This is a big advantage of UnitQLSH. Note that post-processing is needed to filter truly similar images from all the retrieved images. Hashing algorithms with higher precisions also have low post-processing overhead.

Another great advantage of UnitQLSH is that it ranks each binary code. Other traditional algorithms do not rank binary codes, making it impossible to distinguish binary codes with the same Hamming distance to a query. In the case of traditional algorithms, the number of binary codes to retrieve increases exponentially with respect to Hamming distance. For UnitQLSH, we can know what is the next binary code to retrieve. As a result, we can conveniently adjust the number of binary codes to satisfy different needs.

4.7. Future Work: Extending UnitQLSH on a Distributed System

For the experiments presented in this thesis, training of UnitQLSH is performed by first loading the whole data matrix into the memory of a single workstation. However, when the number of data samples becomes extremely large, it is impossible to load the whole matrix into memory. This is a common question when handling large-scale data. The learning of hashing methods should be able to handle this scalability issue when they are adopted for large-scale applications. In this section, I present a method of learning UnitQLSH when the dataset is too large to be loaded into the memory of a single workstation.

There exist two key components of UnitQLSH, listed as follows.

- K-means clustering
  - Compute mean of each cluster
  - Compute index of each data sample
- Optimization procedure shown in Equation 34
  - Fix D and R, update B: $B = sgn(Y'(R)^T)$
  - Fix R and B, update D: $D = \alpha \frac{diag(B^T Y' R^T)}{||diag(B^T Y' R^T)||_2}$
  - Fix D and B, update R: compute SVD of $Y'^T BD = U\Sigma V^T$, then $R = VU^T$

Let us assume that the number of data samples is so large that one machine cannot store all the data samples in memory. In order for UnitQLSH to scale, the key components should be able to scale easily. Given a distribution system with multiple commodity computers, the data samples can be randomly partitioned and each partition is put on a different machine.

To cluster the data, the K-means algorithm needs to be able to alternate between the two aforementioned steps in the list. The map-reduce system [39] proposed by Google offers a good solution. To compute the mean of each cluster, mappers will emit an index-data pair. Then reducers will take all the data samples of the same index and compute the summation and the count of them. The average is then computed by dividing the summation by the count. To compute the index of each data sample, mappers can directly compute the index of each data sample. Reducers are not needed since there is no need to "reduce" the result from the mapper.

To scale the optimization procedure, the three alternating steps should be scalable. Since the optimization is applied to each of the clusters independently, we show how the optimization in one cluster can be scaled. Denote the number of data samples in the cluster as n, the dimension of each data sample as d and the number of bits as b. In general, n is very large, but d and b are reasonably small. The sizes of the related matrices are:

- $Y' \in \mathbb{R}^{n \times d}$: normalized data samples
- $B \in \mathbb{R}^{b \times b}$: binary codes of data
- $R \in \mathbb{R}^{b \times d}$: matrix with orthonormal rows
- $D \in \mathbb{R}^{d \times d}$: diagonal matrix

Let us also make the reasonable assumption that the data and the corresponding binary codes will always live on the same machine. In other words, we can partition data onto different machines, but we never partition a data sample and its binary code.

The computation of $B = sgn(Y'(R)^T)$ is relatively simple. Since R is small enough to store in the memory of each machine, each machine is responsible for computing the binary codes of its own data. Then B would be the union of all the binary codes in all machines.

To compute D, we need to compute $diag(B^T Y' R^T)$. Since R is small, we only need to show how to compute $B^T Y'$. Also, since $B^T Y' \in \mathbb{R}^{b \times d}$ is small as well, we will only show how to compute the i, j-th element in $B^T Y'$. Denoting the result of $B^T Y'$ as Z, we have $Z_{i,j} = \sum_{k=1}^{n} B_{i,k}^T Y'_{k,j} = \sum_{k=1}^{n} B_{k,i} Y'_{k,j}$. Resorting to map-reduce, we can have mappers that emit, for each data sample, the multiplication of the i-th value of its binary code and the j-th value of the data vector. This can be easily done since these two values corresponding to a data sample live on the same machine. Reducers can then sum up the emitted values of all data samples, which gives $Z_{i,j}$. After Z is obtained, D can be computed.

To obtain R, we need to obtain $Y'^T BD$, which is equal to $Z^T D$. Also, note that $Z^T D \in \mathbb{R}^{d \times d}$ and D are also a small matrices. Since Z is already obtained from the previous step, $Z^T D$ is convenient to compute. Then SVD can be conducted on a single machine to get R.

This section has presented the key components of UnitQLSH, namely the K-means algorithm and the optimization procedure shown in Equation 34. Then the nice scalability of these key components has been shown to handle extremely large datasets using a cluster

of machines. Therefore, UnitQLSH is very useful in practice to handle real-world data that can be very large.

CHAPTER 5

Conclusion

5.1. A Summary of This Thesis

This thesis first introduces an approximate nearest neighbor search problem, which is very important for various applications such as image retrieval, duplicate detection and recommender systems. Important related works are then divided into two major categories: similarity based learning and quantization based learning. These works are carefully reviewed and critiqued. Then a novel approach proposed by us, Local Quantization Hashing, is introduced. Its advantages and disadvantages are evaluated and discussed. Then a novel approach called Unit Query and Location Sensitive Hashing (UnitQLSH) is proposed to solve unsupervised approximate neighbor search problems in large-scale datasets. It is greatly improved from LQH. UnitQLSH solves the drawbacks of Local Quantization Hashing (LQH) by integrating query sensitivity with location sensitivity in the hashing scheme and introducing a unit-length constraint of the quantizers.

UnitQLSH assumes that all data samples reside on a unit hypersphere centered at the origin, i.e., data are unit-length. It is true that the original data coming from different application domains are not unit-length. However, these data samples can be processed to be unit-length during the feature extraction step because a lot of feature extraction techniques normalize data to be unit-length. The similarity between two unit-length data samples is defined as their inner product. It is equivalent to say that the distance between these two data samples is defined as their Euclidean distance. Multiple local neighborhoods are first located via certain clustering methods such as K-means. These neighborhoods can be well approximated as linear subspaces, which is highly beneficial for our binary code learning.

Another advantage of finding local neighborhoods is that, during retrieval, the search can be constrained in a small number of local neighborhoods closest to the query, which makes the search more efficient and accurate. After local neighborhoods are located, a hyper-rectangle is learned for each local neighborhood such that the approximation/quantization error is minimized when data samples in the same neighborhood are replaced by the closest vertex on the hyper-rectangle. A very important constraint of the hyper-rectangles is that all their vertices (binary quantizers) should be on the hypersphere. It is equally important that enforcing this constraint helps improve retrieval performance.

Given a query, a weight vector is computed based on the query in each neighborhood to rank the binary codes to be retrieved. This not only helps finely rank all the candidate binary codes, but also makes the computed similarity more accurate since the query is directly used in the computation without any quantization. This is called query sensitivity. Moreover, as aforementioned, a small number of neighborhoods closest to the query is located first to conduct the exploration. This is called location sensitivity. These two sensitivities and the unit-length constraint on the binary quantizers make the retrieval highly efficient and accurate.

Evaluations of UnitQLSH are conducted on SIFT1M, GIST1M and ImageNet, 3 large-scale and challenging datasets. Under different evaluation protocols such as recall@R curves and recall/precision curves, UnitQLSH stands out by a significant amount when compared to other state-of-the-art hashing algorithms. In these evaluations, the merits of location sensitivity, query sensitivity and the unit-length constraint are also demonstrated. More experiments are also performed on evaluating the influence of the parameters in UnitQLSH, such as K - the number of clusters. It is shown that K does not need to be large. A small value of K is already able to achieve good performance. To summarize these experiments,

UnitQLSH yields the best retrieval performance. More importantly, the performance improvement of UnitQLSH is very significant. For example, on the SIFT1M data the recall for UnitQLSH is about 0.5 when $R = 500$ with 32-bit binary code. The next best result is for the KMH algorithm yielding only about 0.25.

5.2. Some Thoughts about the Future for ANN Algorithms and Binary Codes

In this thesis, the future work is discussed, which is to extend UnitQLSH on a distributed system. This is needed when one single machine cannot handle a large-scale dataset in practice. It is described in the future work how UnitQLSH can be extended to multiple machines by exploring the scalability of each key components in UnitQLSH algorithm: the clustering process and the optimization procedure. A map-reduce system is the key to conveniently handle the scalability of each component. Although this is only a thought experiment, the application of UnitQLSH to real tasks is very promising given the great performance seen in the experiments.

The following discussion is beyond the scope of this thesis but important. The UnitQLSH algorithm proposed in this paper assumes that the data are provides as they are and there is no need to generate new representations of the data. In other words, the Euclidean distance between data samples represents their true similarity. This is reasonable if no additional information is known about the data. However, in many practical cases, there are some additional constraints people would like to meet. As a result, it is important to learn a new feature representation of the data that can make problems easier. Feature representation is regarded as one of the most fundamental and difficult problems in computer vision. For unsupervised data, finding nearest neighbors can be a very important step in generating

feature representations. UnitQLSH is able to solve this step for large-scale datasets. That is to say, UnitQLSH may be integrated into a more sophisticated system that alters data representation to better match the needs of a particular problem.

Another thought is about datasets. Currently, there are only a small number of large-scale datasets available for academic researchers. Many researchers are using datasets of less than 100K data samples in their experiments while still claiming them to be large-scale. It became clear to me during my research that there is a gap between industry and academia. Big Internet companies like Google and Facebook will never be short of data. They have so many users and these users are contributing data to them all the time without even realizing it. Frankly speaking, their data should also be more fun to play with. These companies also have their own laboratories to carry out new ideas and new products. As a result, the chances of connecting big companies with universities are sort of small. I personally would like to see a much closer cooperation between academic researchers and big IT companies in the future. It will provide academic researchers, especially PhD students, with a chance to gain insights in real data and really enjoy the challenge.

Bibliography

[1] "How big is facebooks data? 2.5 billion pieces of content and 500+ terabytes ingested every day." http://techcrunch.com/2012/08/22/. Posted: 2012-08-22.

[2] "Big data." `http://en.wikipedia.org/wiki/Big_data`.

[3] Q. Iqbal and J. K. Aggarwal, "Cires: A system for content-based retrieval in digital image libraries," in *Control, Automation, Robotics and Vision, 2002. ICARCV 2002. 7th International Conference on*, vol. 1, pp. 205–210, IEEE, 2002.

[4] D. Xu, T.-J. Cham, S. Yan, and S.-F. Chang, "Near duplicate image identification with patially aligned pyramid matching," in *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, pp. 1–7, IEEE, 2008.

[5] Y. Gong, S. Lazebnik, A. Gordo, and F. Perronnin, "Iterative quantization: A procrustean approach to learning binary codes for large-scale image retrieval," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 35, no. 12, pp. 2916–2929, 2013.

[6] B. Kulis and K. Grauman, "Kernelized locality-sensitive hashing for scalable image search," in *Computer Vision, 2009 IEEE 12th International Conference on*, pp. 2130–2137, IEEE, 2009.

[7] H. Jegou, M. Douze, and C. Schmid, "Product quantization for nearest neighbor search," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 33, no. 1, pp. 117–128, 2011.

[8] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[9] T. Ge, K. He, Q. Ke, and J. Sun, "Optimized product quantization for approximate nearest neighbor search," in *Computer Vision and Pattern Recognition (CVPR), 2013 IEEE Conference on*, pp. 2946–2953, IEEE, 2013.

[10] A. Gionis, P. Indyk, R. Motwani, *et al.*, "Similarity search in high dimensions via hashing," in *VLDB*, vol. 99, pp. 518–529, 1999.

[11] Y. Weiss, A. Torralba, and R. Fergus, "Spectral hashing," in *Advances in neural information processing systems*, pp. 1753–1760, 2009.

[12] W. Liu, J. Wang, S. Kumar, and S.-F. Chang, "Hashing with graphs," in *Proceedings of the 28th International Conference on Machine Learning (ICML-11)*, pp. 1–8, 2011.

[13] B. Kulis and T. Darrell, "Learning to hash with binary reconstructive embeddings," in *Advances in Neural Information Processing Systems 22* (Y. Bengio, D. Schuurmans, J. Lafferty, C. Williams, and A. Culotta, eds.), pp. 1042–1050, Curran Associates, Inc., 2009.

[14] J. Wang, S. Kumar, and S.-F. Chang, "Semi-supervised hashing for scalable image retrieval," in *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pp. 3424–3431, IEEE, 2010.

[15] 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*, pp. 817–824, IEEE, 2011.

[16] Y. Gong, S. Kumar, V. Verma, and S. Lazebnik, "Angular quantization-based binary codes for fast similarity search," in *Advances in Neural Information Processing Systems*, pp. 1196–1204, 2012.

[17] K. He, F. Wen, and J. Sun, "K-means hashing: an affinity-preserving quantization method for learning binary compact codes," in *Computer Vision and Pattern Recognition (CVPR), 2013 IEEE Conference on*, pp. 2938–2945, IEEE, 2013.

[18] L. Zhang, Y. Zhang, J. Tang, K. Lu, and Q. Tian, "Binary code ranking with weighted hamming distance," in *Computer Vision and Pattern Recognition (CVPR), 2013 IEEE Conference on*, pp. 1586–1593, IEEE, 2013.

[19] X. Zhu, L. Zhang, and Z. Huang, "A sparse embedding and least variance encoding approach to hashing," *Image Processing, IEEE Transactions on*, vol. 23, pp. 3737–3750, Sept 2014.

[20] 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*, pp. 253–262, ACM, 2004.

[21] C. Fowlkes, S. Belongie, F. Chung, and J. Malik, "Spectral grouping using the nystrom method," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 26, no. 2, pp. 214–225, 2004.

[22] J. A. Hartigan and M. A. Wong, "Algorithm as 136: A k-means clustering algorithm," *Applied statistics*, pp. 100–108, 1979.

[23] W. Liu, J. He, and S.-F. Chang, "Large graph construction for scalable semi-supervised learning," in *Proceedings of the 27th International Conference on Machine Learning (ICML-10)*, pp. 679–686, 2010.

[24] J. Shi and J. Malik, "Normalized cuts and image segmentation," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 22, no. 8, pp. 888–905, 2000.

[25] V. N. Vapnik and V. Vapnik, *Statistical learning theory*, vol. 2. Wiley New York, 1998.

[26] P. H. Schönemann, "A generalized solution of the orthogonal procrustes problem," *Psychometrika*, vol. 31, no. 1, pp. 1–10, 1966.

[27] A. Krizhevsky and G. Hinton, "Learning multiple layers of features from tiny images," *Computer Science Department, University of Toronto, Tech. Rep*, 2009.

[28] R. Fergus, Y. Weiss, and A. Torralba, "Semi-supervised learning in gigantic image collections," in *Advances in neural information processing systems*, pp. 522–530, 2009.

[29] X. Chen, Y. Qi, B. Bai, Q. Lin, and J. G. Carbonell, "Sparse latent semantic analysis.," in *SDM*, pp. 474–485, SIAM, 2011.

[30] X. Chen and D. Cai, "Large scale spectral clustering with landmark-based representation.," in *AAAI*, 2011.

[31] M. A. Carreira-Perpinan and R. Raziperchikolaei, "Hashing with binary autoencoders," in *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015.

[32] J. A. Hartigan and M. A. Wong, "Algorithm as 136: A k-means clustering algorithm," *Applied statistics*, pp. 100–108, 1979.

[33] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *Systems Science and Cybernetics, IEEE Transactions on*, vol. 4, no. 2, pp. 100–107, 1968.

[34] D. Cai, "Litekmeans: the fastest matlab implementation of kmeans," *available at: http://www.zjucadcg.cn/dengcai/Data/Clustering.html*, 2011.

[35] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International journal of computer vision*, vol. 60, no. 2, pp. 91–110, 2004.

[36] H. Jegou, M. Douze, and C. Schmid, "Hamming embedding and weak geometric consistency for large scale image search," in *Computer Vision–ECCV 2008*, pp. 304–317, Springer, 2008.

[37] A. Oliva and A. Torralba, "Modeling the shape of the scene: A holistic representation of the spatial envelope," *International journal of computer vision*, vol. 42, no. 3, pp. 145–175, 2001.

[38] A. Torralba, R. Fergus, and W. T. Freeman, "80 million tiny images: A large data set for nonparametric object and scene recognition," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 30, no. 11, pp. 1958–1970, 2008.

[39] J. Dean and S. Ghemawat, "Mapreduce: simplified data processing on large clusters," *Communications of the ACM*, vol. 51, no. 1, pp. 107–113, 2008.