k-Nearest Neighbor Search for Large Scale High Dimensional data

k-Nearest Neighbor Search for Large Scale High Dimensional data

Mai Ngoc Anh Vu

February 2016

Acknowledgements

First of all, I would like to express the deepest appreciation to my supervisor, Prof. Takeshi Tokuyama, who responsibly guided me during the master course and also cared of my life in Japan. His patience and support helped me overcome many crisis situations and finish this thesis. I could not have imagined having a better supervisor and mentor for my study.

I am also deeply grateful to Assoc. Prof. Jinhee Chun for her generous supports for both academic and daily's life. Special thanks to all members of Tokuyama Laboratory and my dear friends for the unconditional supports and friendships in the last three years.

Last but not the least, I would like to thank my family for supporting me spiritually throughout all my studies at Tohoku University and my life in general.

Contents

List of Figures

Chapter 1

Introduction

Nearest neighbor search problem is simply defined as follows: find the closest point to a given query point from a collection of points. When the amount of data is large, it is needed to construct a data structure that help to retrieve the nearest neighbor fast and effectually. The nearest neighbor search problem is important in many applications such as pattern recognition, machine learning, data mining, data compression, data analysis and signal processing. For instance, in pattern recognition, the k-nearest neighbor algorithm and it variation are methods used for classification and regression that use nearest neighbor search as the core technique.

k-nearest neighbor search also can be served as a content-based search engine, which is widely used in many different domains such as multimedia, biology, finance, sensor, etc. For example, for image retrieval problem, the Content-based Image Retrieval system which find similar images based on visual content of the input images, such as colors, shapes or texture has showed better result compared to Text-based Image Indexing which is based on associated metadata such as keywords, tags and descriptions. These metadata are often given by users, sometimes could be faked or not much concerned. In turn, visual contents such as colors, shapes or texture information are more reliable to identify each image. In other domains, the database and query object can be DNA sequences in biology, trend curse from stock history data in finance, sensor network data log, and so on. These kinds of data are difficult to visualize or add keywords and captions, but suitable for using content-based search engine.

A straightforward way for this problem is *exhaustive search* that compares a query to each point from the database. However, it can not scale up to huge data sets which have more than billions points with the ranges of dimension from ten to millions.

There are many efficient algorithms has been proposed for the case when the dimension is low (up to about 10) such as *kd*-trees [6], R-trees [13], SR-tree or cover-trees [7]. These early proposed tree-based methods can return accurate results, but they are not time-efficient for data with high dimensional. [29] shown that when the dimensionality exceeds 10, The running time of these methods grown up in exponential of d (dimension) and be slower than the exhaustive search.

In many applications, instead of finding the accurate nearest neighbor that take a lot of time, approximate algorithms are usually used to speed up the performing in an acceptable accuracy. Typical efficient approximate nearest neighbor search algorithms for high dimensional data are Locality Sensitive Hashing (LSH) and the randomized *kd*-forest. Hashing based methods such as locality sensitive hashing and it variations have been shown scalability for high-dimensional data, and hence more suitable for data mining. This study try to improve LSH algorithm on Euclidean distance space by applying Principal Component Analysis as an data-learning technique in order to adapting the distributions of given data sets. Our experiment results shows our approach clearly outperforms the original LSH.

1.1 Thesis Structures

The rest of thesis is organized as follow:

- In chapter 2, we introduce the nearest neighbor search problem and give an overview of recent works from the viewpoint of how data/space partition is given.
- In chapter 3, we will give a brief introduction to the basic Locality Sensitive Hashing algorithms and it variations.
- Then, in chapter 4, we describe our improving of LSH by hashing on principal components.
- In chapter 5, we will explain the result of experiments.
- Finally, in chapter 6, we summarize the main idea of the thesis and give conclusion.

Chapter 2

Nearest neighbor search

2.1 Problem Definition

In this thesis we deal with the problem of efficient nearest neighbor search in metric spaces. The nearest neighbor search problem can be defined as follows:

Definition 1 *(Nearest neighbor search)*
Given a set X with n points $X = \{x_1, x_2, ..., x_n\}$ in a metric space M, construct a data structure such that for any query point $q \in M$, it reports the point X_{nn} in X that is closest to q with respect to a metric distance $D : M \times M \to \mathbb{R}$. That is,

$$D(X_{nn}, q) \leq D(x_i, q), i = 1, ..., n.$$

In the rest of this thesis, we focus on the approximate nearest neighbor problem. The formal definition of the approximate version is as follows:

Definition 2 *(c-approximate nearest neighbor)*
Given a set X with n points $X = \{x_1, x_2, ..., x_n\}$ in a metric space M, construct a data structure such that for any query point $q \in M$, it reports all points p in X that is c-approximate nearest neighbor of q with a probability guarantee at least $1 - \delta$, where

$$D(p, q) \leq cD(X_{nn}, q)$$

In practice, we are often interested in finding not only the first closest neighbor, but top k closest neighbors. This problem is called *approximate k-nearest neighbors search* problem.

2.2 Distance and Similarity Measurements

For any two points **a** and **b**, the l_p distance between them is defined as

$$\|\mathbf{a}-\mathbf{b}\|_s = \left(\sum_{i=1}^{d} |\mathbf{a}-\mathbf{b}|^p\right)^{1/p}$$

for $p > 0$; this distance function is often called the l_p norm. The typical cases include $p = 2$ (the Euclidean distance) or $p = 1$ (the Manhattan distance). Hamming distance, which is Manhattan distance in the binary code $\{0,1\}^d$.

In this thesis, we refer to use the Euclidean distance, since it is one of the most appropriate distance metric to many high dimensional feature-rich data. To simplify the notation, we often omit the subscript 2 that is, $\|\mathbf{a}-\mathbf{b}\| = \|\mathbf{a}-\mathbf{b}\|_2$.

2.3 Related Works

Most of nearest neighbor search methods consist of the following two phases:

1. **Pre-processing:** Partition the data space into regions with some combinatorial structures (e.g.: tree, hashing, cluster), then construct an indexing of the dataset to record which points are contained in each region.

2. **Searching:** Compute the index of the query point to locate the query region. Then obtain all data points in that region or expand to query's nearby regions as the candidates. Finally re-rank the candidates to report selected data.

In this section, we review previous nearest neighbor search methods from the viewpoint of how data partition is given. Most of previous works can be categorized into two groups of data structures: tree based methods and hashing based methods.

2.3.1 Tree Based Partition Methods

Tree based nearest neighbor methods use a hierarchical tree (usually binary tree) as the indexing structure in which the whole space will be divided into several areas, then each area will be further divided into smaller areas until

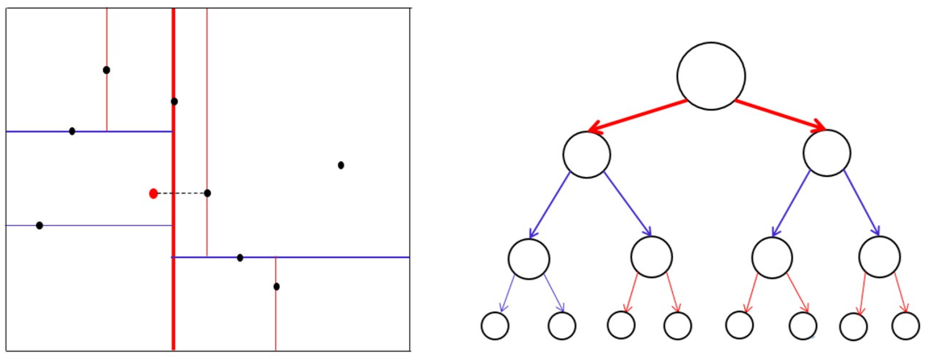


Figure 2.1: *kd*-tree

some stopping criteria is satisfied (e.g. the number of data points remain in each area less than a threshold τ). The whole space is present at the root node, smaller areas is so-called internal nodes and the smallest one where contains IDs of the data points is so-called leaf node.

The early proposed *kd*-tree [6] is the simplest one, which constructs a binary tree by using axis aligned hyperplanes to split the search space along the axis with the highest variance into two regions containing half of the points of the parent region. The iterative decomposition are continued until one point remain in each leaf node. At the searching phase, traversing the tree from the root to the closest leaf node where the query point belongs to obtain a first nearest neighbor candidate, and then continue to traverse next node by backtracking until all candidate nodes that may contain nearest neighbor have been checked.

The *kd*-tree data structure is very effective for low dimensional space, when during the tree exploration many subspaces can be ignored due to being further away than the best nearest neighbor candidate. The binary tree structure is close to signal of computer, so it is easy to construct and can return exact nearest neighbors of the query. However, when the dimensionality increases, "axis aligned hyperplanes" property required number of intermediate nodes are exponential in d. Therefore when searching more and more nodes of the tree needs to be explored by backtracking, so it showed less efficient in running time even comparing to linear search.

A modify at the searching stage of the original *kd*-tree to use it for approximate approaching that instead of traversing all necessary nodes [3], the search stop when it satisfied a “error bound” threshold on the solution accuracy or a “time bound” threshold on the running time for search [5]. [3] also proposes the use of a priority queue to speed up the search in a tree by traversing nodes in order of their distance from the query point. This make the search more efficient by increasing the likelihood of visiting closer neighbors early and pruning away more further branches.

Other researches try to improve on the way *kd*-tree partition the space by some non axis aligned splitting tools, such as PCA-tree [27], random projection trees [10], vantage point tree (vp-tree) [33], etc. In PCA-tree, the partitioning hyperplanes is chosen to be perpendicular on the top eigenvectors generated by Principal Component Analysis (PCA) in order to obtain a better decomposition of the data space. In random projection trees, the splitting hyperplanes are picked to be random directions on the unit hypersphere. And in vp-tree method, in stead of hyperplanes, the hyper-spheres are used to partition the data space. More specifically, in each internal node, selecting a “vantage point” and draw a hyper-spheres with the center as the vantage point and the radius as a threshold, then the data points are split into two set, the “near” and “far” points whether they are inside or outside the hyper-sphere.

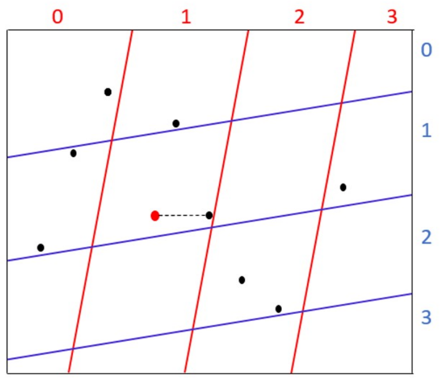
In a single *kd*-tree, if the query point is close to one of the splitting hyperplane, it is high probability that the nearest neighbor lies on the other side of the hyperplane, so that more and more nodes in the tree are need to be explored. In that case of the approximate search when the tree exploration is stopped early, the cell with the closest neighbor might not get visited at all. Multiple randomized *kd*-trees (or randomized *kd*-forest), proposed in [26] creates several different *kd*-trees with different splitting hyperplanes to improve the accuracy of the results for using high dimensional data. When the closest neighbor happens to lie on the other side of hyperplane from the query point, it’s possible that in other tree with different decomposition these two points will be partitioned into the same cell. The construction of randomized *kd*-trees is almost similar in manner to the classic *kd*-tree described above, but instead of splits data on the dimension with the highest variance, the split dimension is chosen randomly from the top N dimensions with the highest variance. In the query stage, the search is performed simultaneously in the multiple trees through a shared priority queue, that is ordered by increasing distance to the decision boundary of each leaf node in the queue, so the search will visit first the closest leaves from all the trees. Once a data point is visited inside one tree, it is marked so that it will not be re-visited from other trees. The search is stopped when it reaches a determined maximum number of leaf nodes.

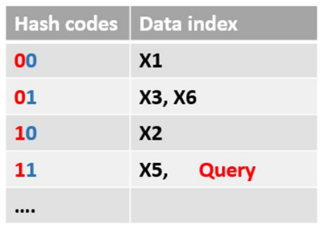
Hierarchical k-means tree [25] is another approach of tree based partition methods, combining with k-means clustering in order to exploit the natural structure existing in the data. It is constructed by splitting the data points at each level into k distinct regions (so-called k clusters) using k-means clustering, and then recursively applying the same splitting method for the points in each region, until the number of points in each region is smaller than a threshold. The searching stage is similar to *kd*-trees, we might also use a priority queue to speed up the search.

2.3.2 Hashing Based Partition Methods

Hashing is an indexing method that using a mathematical function to convert data point from the original large-scale size to shorter fixed-length value or key which easier to search or interact with. Hashing based nearest neighbor algorithms build data structure as a hash table, in which the space will be partitioned to many regions, each entry in the hash table represents one region and contains all IDs of the data points belong to that region. Those regions are so-called "buckets", described by its hash codes computed by the hash functions. Note that on a hash table, each point in the data space is mapping to one bucket determined by its own hash codes, while one bucket could contain one or more items. Items that fall into the same bucket (points that assigned the same hash codes as the region) would have more or less sharing properties, so they have high probability to be near-by or similar. Afterward, one can find near neighbors by hashing query point and retrieving all elements stored in the bucket containing that point as the candidates.

Locality Sensitive Hashing (LSH) is one of the most popular hashing based algorithms for performing approximate nearest neighbor search in high dimensional space. LSH partitions the data space into smaller buckets by using a family of hash functions (usually geometric functions such as hyperplanes or lattice) to ensure that nearby objects have higher probability to fall into the same bucket than far apart objects. It can also be regarded as a form of probabilistic dimensionality reduction. Variations of LSH algorithms have been proposed in recent literatures to expand its usage to the cases of Euclidean space with l_s norms ($s = 1, 2$) [11], for Cosine similarity on $\mathbb{R}^d$ [9], Jaccard for similarity of sets [8], and learned metrics [17], Chi2 distance[12],





Hash codes	Data index
00	X1
01	X3, X6
10	X2
11	X5, Query
....	

Figure 2.2: Space partition by hash functions

etc.

For instance, Euclidean LSH uses random projection to partition the space. Data points are projected into k random directions choosing independently from Gaussian distribution, and then split into several buckets with a preset window size. (Figure 2.2)

We can see that both Euclidean LSH and kd-trees indexing methods are using hyperplanes, but LSH only needs $O(\log M)$ hyperplanes to partition the space to M regions while tree based methods need $O(M)$ hyperplanes. As a result, hashing based methods are requiring less memory, and making multiple partitions convenient and practical. Another big advantages of LSH is data independent (random) partitions. In LSH, the partition structures are randomly generated, independent of the database points, helps they overcomes "the curse of dimensionality" in some sense and can usually deal with high dimensional data quite well. No training of data is required in LSH, and hence it is very easy to scale up to large data set.

But the data-independent properties also become the lack of LSH. Since the hash functions in LSH are randomly generated and independent of the data, it is often not very efficient. And hence many hash tables are often needed to get a good recall to keep a high precision. This would heavily increase the requirement of storage, causing problems for very large scale applications.

Consequently, many resent research has been targeted at improving hashing methods by using various learning techniques to generate data-dependent hashing functions: spectral hashing [30] , kernelized LSH [22], learnt binary embeddings [23], semi-supervised hashing [28], optimized kernel hashing [15] and complementary hashing [31], etc. But they usually need massive computation to learn the partitions.

This study try to improve locality sensitive hashing algorithm on Euclidean distance space by applying some data-learning technique when design the hash functions but not seriously increase the computation. Next chapter we will represent more detail about LSH algorithm on Euclidean distance space and explain our improving points in chapter 4.

Chapter 3

Locality Sensitive Hashing

Locality Sensitive Hashing (LSH) is one of the most popular hashing based algorithms for performing approximate nearest neighbor search in high dimensional space. LSH partitions the data space into smaller buckets by using a family of hash functions (usually geometric functions such as hyper-planes or lattice) to ensure that nearby objects have higher probability to fall into the same bucket than far apart objects. Then, online searching consists of two steps: (1) hash the query point and retrieves all items belong to its bucket as potential candidates, and (2) determine the nearest neighbor by ranking the candidates according to their distances to the query point.

3.1 LSH Definition

The LSH algorithm relies on the existence of locality sensitive hash functions. Let S be the domain of objects with D be the distance measure between objects. Let $\mathcal{H}$ be a family of hash functions mapping S to some universe U. The family $\mathcal{H}$ is called locality sensitive if it satisfies the following condition.

Definition 3 *(Locality sensitive hashing function family)*
A function family $\mathcal{H} = h : S \to U$ is called (r, cr, P_1, P_2)-sensitive for D if for any $q, p \in S$

- *If $D(q,p) \leq r$ then $P_{rH}[h(q) = h(p)] \geq P_1$,*
- *If $D(q,p) > cr$ then $P_{rH}[h(q) = h(p)] \leq P_2$.*

To use LSH for approximate nearest neighbor search, we pick $c > 1$ and $P_1 > P_2$. With these choices, nearby objects (those within distance r) have a greater chance (P_1 vs. P_2) of being hashed to the same value than objects that are far apart (those at a distance greater than cr away)

Several LSH families have been discovered for the use of different distance functions D. For instance, when the data are binary vectors from $\{0,1\}^d$ with Hamming distance, we can use a particularly simple family of functions $\mathcal{H}$ which contains all functions h_i from $\{0,1\}^d$ to $\{0,1\}$ such that $h_i(p) = p_i$ [16]. Choosing one hash function h uniformly at random from $\mathcal{H}$ means that $h(p)$ returns a random coordinate of p. To confirm the family $\mathcal{H}$ is locality-sensitive, observe that the probability $Pr_{\mathcal{H}}[h(q) = h(p)]$ is equal to the fraction of coordinates on which p and q agree. Therefore, $P_1 = 1 - R/d$, while $P_2 = 1 - cR/d$. Since the approximation factor c is always picked greater than 1, we have $P_1 > P_2$.

Other LSH families have been also proposed for Euclidean space with l_s norms ($s = 1, 2$), for Cosine similarity on $\mathbb{R}^d$ [9], or Jaccard for similarity of sets [8], etc.

3.2 LSH Algorithm

3.2.1 LSH Indexing Method

Using a family of LSH functions $\mathcal{H}$, we can construct indexing data structures for similarity search. The basic LSH indexing method works as follows:

- For an integer k, define a function family $G = g : S \to U^k$, and for $g \in G, g(v) = (h_1(v), ..., h_k(v))$, where $h_j \in H for 1 \leq j \leq k$ (i.e., g is the concatenation of k LSH functions).

- For an integer L, choose $g_1, ..., g_L$ from G, independently and uniformly at random. Each of the L functions $g_i (1 \leq i \leq L)$ is used to construct one hash table, resulting in L hash tables

This LSH indexing method can further amplify the difference between P_1 and P_2, by concatenating several functions. This increases the ratio of the probabilities (given above) that points at different separations will fall into the same quantization bin, since $(P_1/P_2)^k > (P_1/P_2)$.

By concatenating k functions, the probability of the query and the nearest neighbor are in the same bin will be P_1^k, which decreases as we include more functions (increase k). Therefore, performing L independent projections and pool the neighbors from all of these will ensure that a true near neighbor could fall into the same buckets at least in one table. Then, by increasing L we can find the true nearest neighbor with arbitrarily high probability.

3.2.2 Basic Querying Process

To search the nearest neighbor for a query q, we hash the query point and scan through the buckets $g_1(q), ..., g_L(q)$ to retrieve the points stored in them. After retrieving the points, we compute their distances to the query point, and report any point that is a valid answer to the query.

Some optional optimizations can be used in query process.

- In practice, we can stop processing a query as soon as the number of reported points is more than a threshold L' to keep time efficiency.
- A near neighbor might be encountered and be computed the distance to the query point more than once because it appears in more than one of the buckets $g_1(q), ..., g_L(q)$. We can skip these repeated points by making a tracking vector e_i of the points for which we already computed the distance $\|q - x_i\|$, and not compute the distance a second time ($e_i = 1$ if we already encountered the point $x_i \in X$ in an earlier bucket and computed the distance $\|q - x_i\|$), and $e_i = 0$ otherwise). Another trick is retrieving candidates set is the union of all query's buckets before computing the distance.

3.3 Euclidean LSH and Its Variations

In this section, we will mainly introduce LSH for Euclidean space with l_2 norm distance, which is more popular in practice, its variations and other ideas that promising to apply into Euclidean space.

3.3.1 Euclidean LSH with l_2 norm distance

For Euclidean space with, Datar et al. [11] have proposed LSH families for l_s norms ($s = 1, 2$), based on s-stable distributions. Here, each hash function is defined as:

$$h_{a,b}(v) = \left\lfloor \frac{\mathbf{a} \cdot \mathbf{v} + b}{W} \right\rfloor$$

where $\mathbf{a}$ is a d-dimensional random vector with entries chosen independently from a s-stable distribution and b is a real number chosen uniformly from the range $[0, W]$. Each hash function $h_{a,b} : \mathbb{R}^d \to \mathbb{Z}$ maps a d-dimensional vector $\mathbf{v}$ onto the set of integers. Due to the linearity of the dot product, the difference between two projections $|\mathbf{a} \cdot \mathbf{p} - \mathbf{a} \cdot \mathbf{q}|$ has a magnitude whose distribution is proportional to $\|\mathbf{p} - \mathbf{q}\|$ therefore, $P_1 > P_2$.

Increasing the quantization bucket width W will increase the number of points that fall into each bucket. To obtain our final nearest neighbor result we will have to perform a linear search through all the points that fall into the same bucket as the query, so varying W effects a trade-off between a larger table with a smaller final linear search, or a more compact table with more points to consider in the final search. On the other hand, if k is large then P_1^k is small, which means that L must be sufficiently large to ensure that a near neighbor collides with the query point at least once.

The basic LSH scheme has several drawbacks. Firstly, it is difficult to choose parameters such as k, W that suitable for a given dataset and different queries, which may result in large deviation in runtime and quality. Secondly, in practice it needs a large number of hash tables L to achieve high recall ratio and low error ratio and this results in high selectivity with more memory-demanding. These drawbacks can affect the efficiency and quality of basic LSH-based algorithms on large datasets. Many techniques have been proposed to overcome some of the drawbacks of the basic LSH algorithm.

3.3.2 Variations of Euclidean LSH

Multi-probe LSH methods [24] [21] are proposed to decrease the need of using large number of hash tables. The idea of multi-probe LSH is if the nearest neighbor does not fall into the query's bucket, it would fall into the near-by buckets which hash code is different in $\{-1, 0, +1\}$. Multi-probe LSH [24] systematically probes the buckets near the query points in a query-dependent manner, instead of only probing the bucket that contains the query point. It can obtain higher recall ratio with fewer hash tables, but may result in larger selectivity from additional probes. The different approaches mainly differ in terms of how they select the multiple buckets per hash table. Joly et al. [21] improve the multi-probe LSH by using prior information collected from sampled dataset.

Bawa et al. [4] proposed the LSH Forest for Hamming space, which is a data structure representing the hash table as a prefix tree. The prefix trees can hold hash keys of variable length, allowing the LSH forest to adapt better to the data, compared to the classic LSH for which the key length parameter needs to be hand tuned. Jegou et al. [18] applied similar idea into Euclidean space, which trades memory against accuracy by defining a larger pool of M hash functions but use only for the most relevant ones likely to return the nearest neighbors on a per-query basis.

In [1] Andoni and Indyk proposed a near-optimal LSH that uses a Leech lattice for the geometric hashing instead of one-dimensional random projections. The idea is that the Leech lattices (24 dimensions) offer better quantization properties for the mean square error dissimilarity measure used in Euclidean spaces. But it requires significantly calculating operations to decoding each lattice point. Jegou et al.[18] proposed using another lattice structure (E_8 lattice with 8 dimensions) that still excellent quantization properties but more efficient decoding compare to Leech lattice.

Pauleve et al. [24] present k-means Locality Sensitive Hashing scheme that combine k-means clustering with LSH. The idea is instead of building L hash tables using some locality sensitive hash family, they generate L different k-means clustering for each table with different set of centroids. At search time, the nearest centroid for each of the L k-means tables is found for the query point. The data points assigned to these same centroids are then retrieve as the candidates. The indexing scheme and query processing is similar to the basis scheme of LSH. Using k-means clustering as partition method makes the hashing process adaptive to datasets with different distributions.

Zixiang et al. [32] proposed the idea of using hierarchical and non-uniform bucket partitioning based on the distribution of points to handle the non-homogeneous datasets with a specific video similarity measurement. The hierarchical LSH in [32] has a prospect that be able to build a data-dependent structure that learn well the distribution of the input dataset that can be apply to other metric space or distance measurement.

Chapter 4

Our works: Locality Sensitive Hashing on Principal Components

4.1 Discussion

Locality sensitive hashing (LSH) algorithm has prominent merit is that they overcome "*the curse of dimensionality*" in some sense and can usually deal with high dimensional data quite well. Moreover, it makes multiple partitions convenient and practical but only needs $O(\log M)$ partition functions to create M regions, requiring less memory. However, since the partitions of LSH are randomly generated (data independent), it may not be optimal/high-quality for all kind of datasets. Therefore, our study try to improve LSH algorithm on Euclidean distance space by applying some data-learning technique when design the hash functions but not seriously increase the computation.

The initial ideas is that instead of using random projection, we could find another way to choose a better projecting direction that be able to adapt the data.

Intuitively, the dot product (projection) using in Euclidean LSH can be express geometrically as follows:

$$\mathbf{a} \cdot \mathbf{v} = \|\mathbf{a}\| \, \|\mathbf{v}\| \cos(\angle(\mathbf{a}, \mathbf{v}))$$

The magnitude of the dot product is proportional to the length of two vectors and the angle between them. For any two points $\mathbf{p}, \mathbf{q}$ the distance between

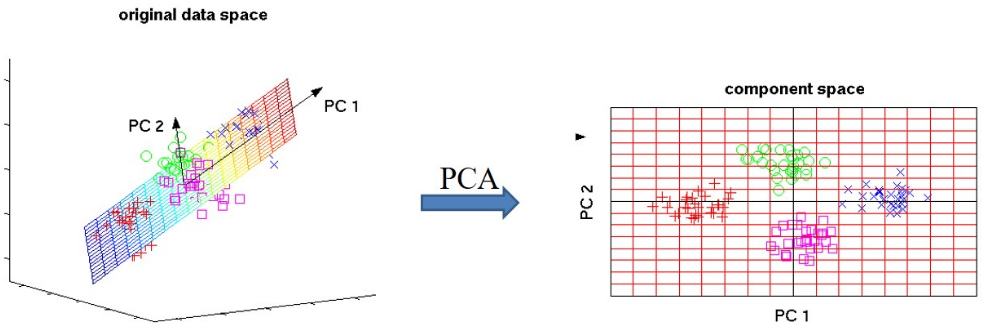


Figure 4.1: Principle Component Analysis (PCA)

their projections is

$$\mathbf{a} \cdot \mathbf{p} - \mathbf{a} \cdot \mathbf{q} = \mathbf{a} \cdot (\mathbf{p} - \mathbf{q}) = \|\mathbf{a}\| \, \|\mathbf{p} - \mathbf{q}\| \cos(\angle(\mathbf{a}, \mathbf{p} - \mathbf{q}))$$

Therefore, two points that are close together are also close together when projected onto any direction. This is true no matter how we rotate the projecting direction. Two other points that are far apart will be close together on the projection, only happen with some orientations when $\cos(\angle(\mathbf{a}, \mathbf{p}-\mathbf{q}))$ is approach to zero. This point of view orients us working on how to choosing a better direction for projection compare to the random vectors of Euclidean LSH. Projection on the top highest variance of PCA eigenvectors is this kind of thing.

4.2 Locality Sensitive Hashing on Principal Components

4.2.1 Principal Component Analysis

Principal Component Analysis (PCA) is known as a dimension reduction procedure that uses an orthogonal transformation to generate the principle components from a set of data. It rotates the original data space into a new coordinate system in which the axis are arranged in order of variance from largest to smallest. Then, we can omit some axis with small value of variance without losing the scaling information of data. (Figure 4.1)

This transformation is defined in such that the first principal component has the largest possible variance, then the next largest variance come to the second coordinate under the constraint that it is orthogonal to the preceding component, and so on. The number of principal components is less than or equal to the number of original variables. The eigenvectors of PCA are always an uncorrelated orthogonal unit set of vectors.

PCA can be performed via *Eigenvalue Decomposition* (EVD) of a data covariance matrix or *Singular Value Decomposition* (SVD) of the data matrix.

In the case of using eigenvalue decomposition, the covariance matrix computation costs $O(d^2n)$ floating-point operations (flops), its eigenvalue decomposition is $O(d^3)$. So, the complexity of PCA is $O(d^2n + d^3)$ for a $n \times d$ data matrix, where n is number of data points and d is the dimension size. Each row represents a data point, and each of the d columns represents a feature of data. When $n \geq d$, the total complexity of PCA is $O(d^2n)$.

In the case of using singular value decomposition, the total complexity of *Exact SVD* is almost same as using eigenvalue decomposition $O(d^2n)$, while the first step reducing the data matrix into bidiagonal form needs $O(d^2n)$ flops and the second step making the decomposition needs $O(d^3)$. There are some technique for computing an approximate SVD such as *Truncated SVD* that computes only the v largest singular values and associated singular vectors of a data matrix. This can be much quicker and more economical than the exact SVD. The cost of this approach is typically $O(vdn)$ flops. Halco et al. in [14] presented *Randomized SVD* that using only $O(nd\log(v)+(n+d)v^2)$ flops is fastest scheme as we know.

4.2.2 Hashing on Principle Components

Projection on the top PCA eigenvectors with highest variance can adapt the distribution of the whole points from the set of data. Using the top PCA eigenvectors as projecting direction does not change the linearity attitude of the dot product explained in section 4.1, then the difference between two image points $|h(\mathbf{p})-h(\mathbf{q})|$ has a magnitude whose distribution is proportional to $\|\mathbf{p}-\mathbf{q}\|$, therefore it satisfies the condition $P_1 > P_2$ in the Definition 3 about LSH family. More clearly, projection on highest variance components does not increase the probability of two nearby points would fall into the same bucket P_1, but it can decrease the probability P_2 for two far away points be hash to one bucket. Thus, apply PCA to LSH could bring better space partition than random projection.

The preprocessing algorithm of LSH on Principle Components (PCALSH) can be rewrite as follows:

1. Doing PCA for all points in the dataset and choosing top V eigenvectors.

2. Construction of L hash functions $g_1, ..., g_L$ by for each g_j concatenating k LSH functions $g_j(v) = (h_{1,j}(v), ..., h_{k,j}(v))$, with $h_{i,j}(i = 1, ...k, j = 1, ..., L)$ is defined as follows:

$$h_{a,b}(v) = \left\lfloor \frac{\mathbf{a} \cdot \mathbf{v} + b}{W} \right\rfloor$$

   where **a** is choosing randomly from top V eigenvectors and b is a real number chosen uniformly from the range $[0, W]$

3. Construct L hash tables, where each table contains all of data points hashed using the function $g_1, ..., g_L$

Morever, hashing on principle components has been shown other beautiful properties. Firstly, the data space can be partition into well-balanced buckets since PCA component is orthogonal unit vectors. Figure 4.2 is an example of the regions obtained using projection on orthogonal unit vectors and projection on random vectors. Secondly, the orthogonal properties makes it is easy to calculate real distance from one query point to bucket boundary that can be used to improve the multi-probe looking up on the query process.

By the way, one issue of PCALSH is how to choose the number of top principal components V. The number of PCA components limited in the number of dimension d, but not all of them are principal. For example, for dataset SIFT1M included 1,000,000 items with 128 dimensions of images SIFT description, using only the first 50 principal components is correlated with more than 90 percents of the real distances. Therefore, we can not generate independently and uniformly $k \times L$ random vectors like Euclidean LSH.

Note that, in order to construct L hash tables, each table needs k distinct hash keys, it's no need to prepare $L * k$ distinct hash functions. We can repeatedly use some hash functions on different hash table. In the original E2LSH Package [2], Indyk et al. used only $O(k\sqrt{L})$ distinct keys instead of $L * k$ in order to reduce the total computing time for hash functions. It makes the function g_j are not totally independent, but still guarantees for the success probability of the approximate problem.

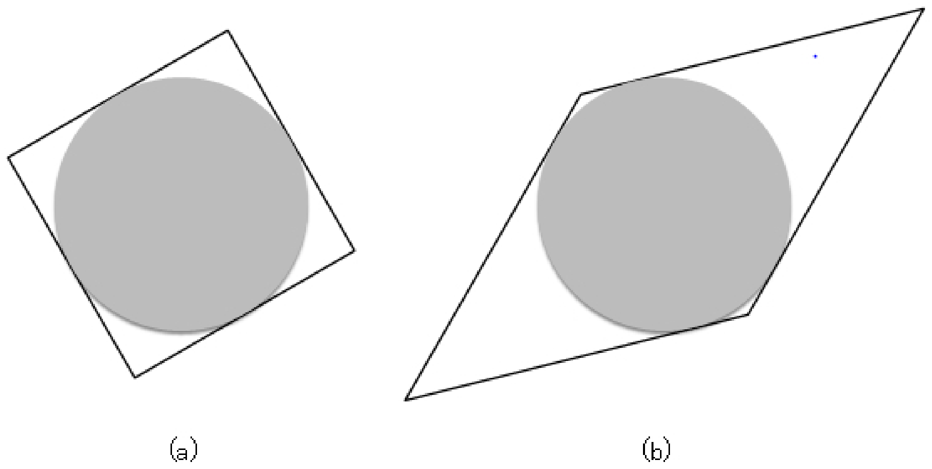


Figure 4.2: Fundamental regions obtained using projection on Orthogonal unit vector (a) and random vector (b)

For our PCALSH: Since the number of principal components is limited, we propose to use a smaller number of top components: $V = O(k\sqrt[k]{L})$. For instance, when $L = 20, k = 10$, the value of V is 14; when $L = 100, k = 20$ the value of V is only 25. This small number of V still guarantees $L < \frac{V!}{k!(V-k)!}$, that means there is always enough different combination of k distinct hash keys for construction L tables.

We will compare PCALSH with different number of principal components in experiment 5.3.1.

4.2.3 Time complexity of PCALSH indexing process

The time cost of a LSH indexing process consists of four parts as follows:

1. Generating hash functions parameters

   - For Euclidean LSH, generating $L * k$ independent d dimension-vectors randomly from Gaussian distribution cost $O(dkL)$

- For PCALSH, calculating exact PCA with all eigenvectors cost as most $O(nd^2)$ flops. It is difficult to scale up to very high-dimensional datasets.

  Since our PCALSH needs only small number of V principal components to construct L hash tables, we implemented an approximate PCA using the Randomized SVD as given in [14] requires only $O(nd \log V)$ flops.

2. Calculating hash keys

   - For Euclidean LSH, in order to construct L tables, each table includes k keys, the time consuming of calculating hash keys is $O(ndkL)$
   - For PCALSH, because some PCA's eigenvectors are reused on different tables, we can reduce the time consuming of calculating hash keys as follows:

     (a) Firstly, making the projection of all data points into V principal components needs $O(ndV)$,

     (b) Then calculating hash keys in $O(nLk)$ flops.

     The total time cost of calculating hash keys is $O(ndV + nLk)$

3. Classifying into distinct buckets

   This step is the same in both Euclidean LSH and PCALSH, it costs at most $O(n \log(n)k)$

The total time complexity of our PCALSH indexing process is

$$O(nd \log(V) + ndV + nLk + n \log(n)k)$$

and the total time complexity of Euclidean LSH indexing is

$$O(dkL + ndkL + n \log(n)k)$$

In most of case, we have $\log(n) \leq k \leq V \leq d \leq n$ and $L \leq d$. Therefore, the time complexity of PCALSH indexing can be rewritten as $O(ndV)$ and the time complexity of Euclidean LSH indexing is $O(ndkL)$.

If we choose $V = O(k\sqrt[k]{L})$, then the time complexity of PCALSH even smaller than the time complexity of the Euclidean LSH.

4.3 Orthogonal Locality Sensitive Hashing with QR decomposition

As explained before, PCALSH have two main beautiful properties. The first one is projecting on top eigenvectors can adapting the distribution of data. The second is orthogonal unitary properties of the eigenvectors. In order to confirm the efficiency of the second property, we design another algorithm so-called Orthogonal Locality Sensitive Hashing that projection onto sets of orthogonal random vectors. There are some methods to generate a set of orthogonal vectors such as singular value decomposition and QR decomposition. In this paper, we choose to use QR decomposition since it is simpler and faster in practice.

The algorithm of generating k orthogonal random vectors as follows:

- First, generate a set of k random vectors independently from Gaussian distribution, denote $A = [\mathbf{a}_1, ..., \mathbf{a}_k]$. Then A is a $d \times k$ matrix.
- Using QR decomposition to generate a uniform random rotation matrix $Q = [\mathbf{q}_1, ..., \mathbf{q}_d]$ from random matrix A, such that $A = QR$
- Taking the first k column of matrix Q as a set of k independent orthogonal random vectors $[\mathbf{q}_1, ..., \mathbf{q}_k]$

The set of k independently orthogonal random vectors $[\mathbf{q}_1, ..., \mathbf{q}_k]$ then can be used to construct a hash function $g(v) = (h_1(v), ..., h_k(v))$ In order to construct L tables with independent function $g_j(v)$, if $L \times k$ is smaller than dimension size d, we can produce $L \times k$ independent orthogonal random vectors using QR decomposition, if not, repeating QR decomposition until perceive enough L set of k orthogonal random vectors.

Then, the indexing algorithm of Orthogonal Locality Sensitive Hashing is completely the same as Euclidean LSH.

4.3.1 QR factorization

If A is $m \times n$ and left-invertible then it can be factored as

$$A = QR$$

where:

- R is $m \times n$ and upper triangular with $r_{ii} > 0$
- Q is $m \times m$ and orthogonal ($Q^T Q = I$)

There are several ways of computing the QR decomposition, such as the Gram-Schmidt process, Householder transformations and Givens rotations. It can be computed in $O(mn^2)$.

Chapter 5

Experiments

5.1 Experimental Setup

5.1.1 Evaluation Datasets

We perform our experiments on SIFT1M dataset which is particularly generated for evaluating the quality of approximate nearest neighbors search [20] [1]. The SIFT1M dataset included 1,000,000 feature vectors of 128 dimensions local SIFT descriptors of images. The elements of SIFT feature vectors are integers in the range $[0, 255]$. The query subset is generated from distinct source of image with dataset. These datasets are extracted from the publicly available INRIA Holidays dataset and Flickr images [19].

5.1.2 Evaluation Benchmarks

For each descriptor data type, we only picked 100 objects from query subsets as query object. For each query object, the ground truth is defined to be the query object K nearest neighbors based on the Euclidean distance of their feature vectors calculated by exhaustive search. Unless otherwise specified, K is 50 in our experiments. We mainly compare the quality and efficiency of our PCALSH and Orthogonal LSH with the original E2LSH.

There are two parameters for evaluating the search quality.

Firstly, the search quality is measured by *recall*: Given a query object $\mathbf{q}$, let I $(\mathbf{q})$ be the set of ideal answers (i.e., the K nearest neighbors of $\mathbf{q}$), let

[1] Downloaded from http://corpus-texmex.irisa.fr/

$A(\mathbf{q})$ be the set of actual answers, then

$$recall = \frac{|A(\mathbf{q}) \cap I(\mathbf{q})|}{|I(\mathbf{q})|}$$

Note that we do not need to consider precision here, since all the candidate objects will be ranked based on their Euclidean distances to the query object and only the top K candidates will be returned.

Secondly, the quality of approximate nearest neighbor search will be presented in terms of *effective error ratio* calculated as follow:

$$error = \frac{1}{|Q|K} \sum_{q \in Q} \sum_{k=1}^{K} \frac{d_{LSH_k}}{d_k*}$$

where d_{LSH_k} is the k-th nearest neighbor retrieved by a LSH method, and d_k* is the true k-th nearest neighbor. In other words, it measures how close the distances of the K nearest neighbors found by LSH are compared to the exact K nearest neighbors distances. High recall and very small error ratio are expected for a good nearest neighbor search algorithm.

In order to confirm the time complexity of indexing process of our PCALSH and E2LSH that explained in section 4.2.3, we measure running time on the indexing process that construct table with the same number of buckets.

Both E2LSH and our PCALSH use the same scheme on query searching process, in which most of runtime is used for doing linear search on the candidates set. Therefore, instead of measure the real running time depends on the system status, we will use *selectivity* to measure the runtime cost of the short-list search:

$$selectivity = \frac{|A(q)|}{|S|}$$

where $|S|$ is the size of the dataset and $A(q)$ is the candidates set. The searching methods with smaller selectivity is faster in real running time and more efficient in practice.

All performance measures are averaged over the 100 queries. Also, since the hash functions are randomly picked, each experiment is repeated 10 times and the average is reported.

5.1.3 Parameters setting

The main parameters of our proposed technique PCALSH and QRLSH are similar to the common E2LSH parameters L, k and w. The effect of the common parameters L, k and W on whole three algorithms are generalized in Table 5.1.

Table 5.1: The effect of the common parameters L, k and W

	$L\uparrow$	$k\uparrow$	$w\uparrow$
Selectivity	$\uparrow$	$\downarrow$	$\uparrow$
Recall	$\uparrow$	-	$\uparrow$
Memory usage	$\uparrow$	$\uparrow$	-

In practice, using large number of k and L to guarantee the search quality is not recommended since it is the trade-off to the space requirement and running time. Therefore, we limit the value of $L \leq 20, k \leq= 15$ for all experiments.

In PCALSH, there is a new parameter V, number of top PCA components that choosing for projection. We use different value of $V(\left\lceil k\sqrt{L}\right\rceil = 45$, $\left\lceil k\sqrt[3]{L}\right\rceil = 27$ and $\left\lceil k\sqrt[k]{L}\right\rceil = 14$) and compare the effect to our PCALSH.

5.2 Experimental results

There are four experiments that designed as follows:

1. Experiment 1: Quality comparison of PCALSH and E2LSH

   With the fixed value $L = 20, k = 10$, we increase the window size W gradually which will result in *selectivity* in an ascending order and observe the behaviors of *recall* and *error* ratio.

2. Experiment 2: Independent property of hash tables in PCALSH

   We fixed the value of $k = 10$ and increase the number of hash tables L gradually from 1 to 20 and trace the changes of the *recall* and *selectivity* of PCALSH compare with E2LSH. The window size W is choosing different for each algorithm in order to observe the variation clearly ($W = 800$ for E2LSH and $W = 150$ for PCALSH).

3. Experiment 3: Orthogonal property

   With fixed the value of $L = 20, k = 10$, we increase the window size W gradually which will result in selectivity in an ascending order and observe the behaviors of *recall* and *error* ratio of Orthogonal QRLSH compare to E2LSH and PCALSH.

4. Experiment 4: Running time of PCALSH indexing process

   We measure running time on the indexing process that by fixing the value of L, k and choosing appropriate value of W for PCALSH and E2LSH in order to construct tables with almost same number of buckets (eg.:$W = 700$ for E2LSH and $W = 123$ for PCALSH).

5.2.1 Experiment 1: Quality comparison of PCALSH and E2LSH

Figure 5.1 and 5.2 show the varying of *recall* and *error* ratio belong to the same selectivity for Euclidean LSH and our PCALSH with different setting of V.

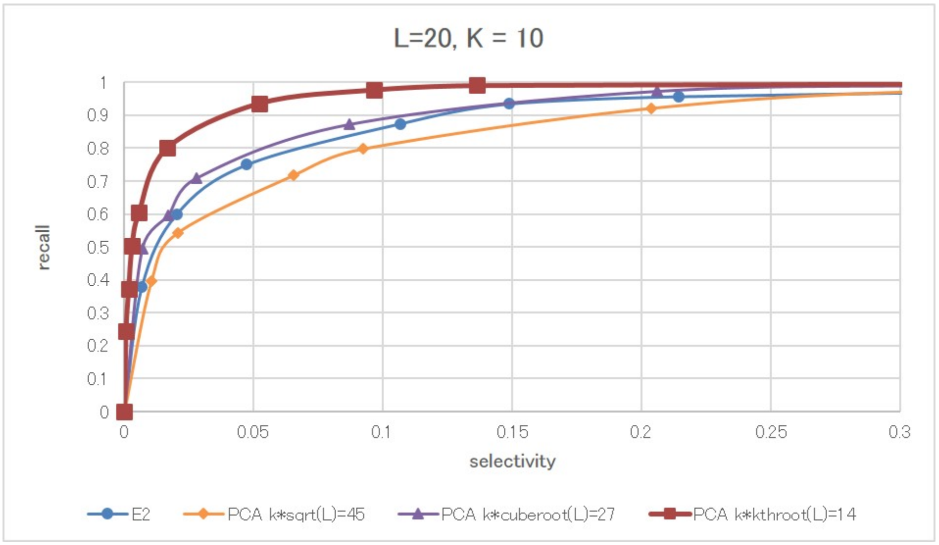


Figure 5.1: Recall comparison of PCALSH and E2LSH by varying bucket's window size W

We come up the following facts from the results:

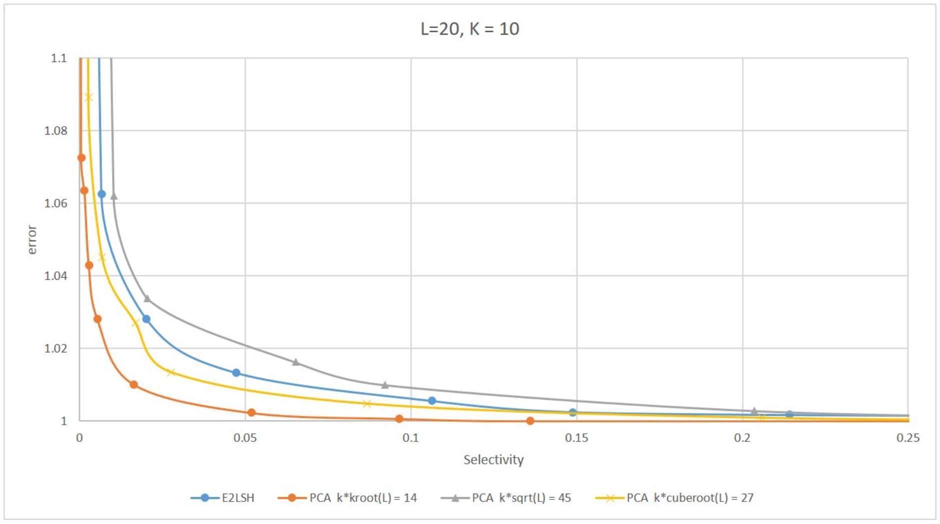


Figure 5.2: Error ratio comparison of PCALSH and E2LSH by varying bucket's window size W

Firstly, with the same selectivity, using top 14, 27 of PCA returns higher *recall* and smaller *error* ratio compare to E2LSH. Especially, hashing with top 14 PCA components (setting $V = k\sqrt[k]{L}$) showed clearly outperforms the original E2LSH. From another point of view, in order to retrieve a demand recall, PCALSH can use smaller selectivity than E2LSH (for instance, over 90% of the grouthtruth, PCALSH using top 14 components retrieve $selectivity = 0.05$ equal 5% of the dataset as the candidates set for checking by the linear search, while E2LSH needs to check more than 20% of the whole dataset, that means PCALSH is 4 times faster than E2LSH for the query search time).

Secondly, consider about the effect of different number of top PCA components V into performance of PCALSH, using smaller value of V showed better result compare to the large one. The reason is that the importance of the principal components is significantly change with the variation of variance in decending order. Therefore, hashing on the rearward due to less effective than hashing on the frontal components.

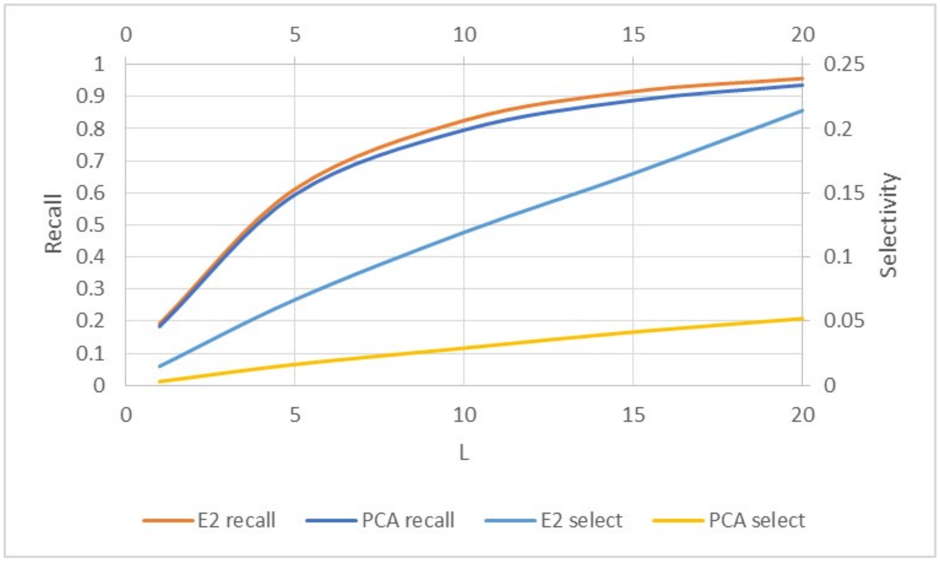


Figure 5.3: Independent property comparison of PCALSH and E2LSH

5.2.2 Experiment 2: Independent property of hash tables in PCALSH

One significant change of our PCALSH algorithm that different to the original E2LSH is using small number of hashing directions to concatenate L hash functions $g_1, ..., g_L$ instead of $L * k$ independent random hash functions of E2LSH. In this case, the functions g_j are not totally independent as in E2LSH. Therefore, it is needed to make an observation on the behavior of PCALSH when changing the number of hash tables.

In this experiment, we fixed the value of $k = 10$ and increase the number of hash tables L gradually from 1 to 20 and trace the changes of the *recall* and *selectivity* of PCALSH compare with E2LSH. For PCALSH, we use top 14 PCA components (setting $V = k\sqrt[k]{L}$) to construction 20 hash tables. And for E2LSH, we generate 200 distinct random vectors from Gaussian distribution to construction 20 independent hash tables. The window size W is choosing different for each algorithm in order to observe the variation clearly ($W = 100$ for E2LSH and $W = 150$ for PCALSH).

The results are showed in Figure 5.3. We can see that the selectivity of PCALSH are extremely smaller than E2LSH, but still linear increasing with the number of hash tables L. It proves that our indexing method with small number of hash directions (setting $V = k\sqrt[k]{L}$) is able to guarantee the independent property of L hash functions $g_1, ..., g_L$. It is very important property to help our algorithm be scalable to other dataset with higher dimensions.

5.2.3 Experiment 3: Orthogonal LSH

With fixed the value of $L = 20, k = 10$, we increase the window size W gradually which will result in selectivity in an ascending order.

Figure 5.4 showed that the Orthogonal properties of QRLSH cannot improve the search quality compare to the original E2LSH. While in PCALSH, hashing on top highest variance components which adapting the distribution of data is the most important properties cause its out performance.

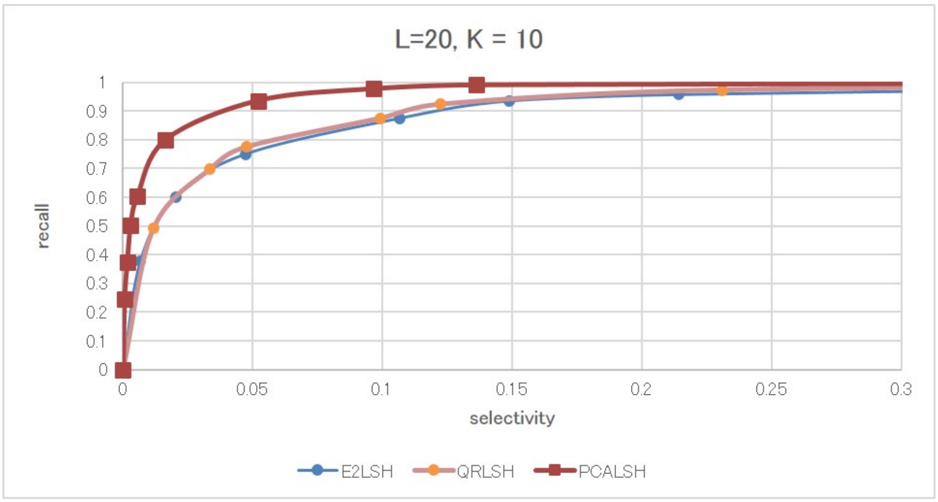


Figure 5.4: Quality comparison of QRLSH and E2LSH, PCALSH

5.2.4 Experiment 4: Indexing time comparison

In section 4.2.3, we have analyzed the time complexity of PCALSH indexing process is $O(ndV)$, while the time complexity of E2LSH indexing is $O(ndkL)$. It mean that when we choose $V = O(k\sqrt[k]{L})$, then the running time of PCALSH indexing even smaller than E2LSH.

In this experiment, by fixing $k = 10$ and choosing appropriate value of W for PCALSH and E2LSH in order to construct tables with almost same number of buckets ($W = 700$ for E2LSH and $W = 123$ for PCALSH) , we confirm the running time of indexing process PCALSH and E2LSH to construct $L = 1$, 5, 10 and 20 tables.

The result are given in Table 5.2. It showed that the time complexity of PCALSH is really reduced and more efficient than E2LSH.

Table 5.2: Indexing time comparison of PCALSH and E2LSH

Number of tables	E2LSH	PCALSH
1	2.15s	1.93s
5	7.93s	4.38s
10	16.23s	7.66s
20	32.28	13.46s

Chapter 6

Conclusion

We prerented a new data-dependent hashing method that projection on the top PCA eigenvectors with highest variance instead of using random vectors. Our PCALSH is able to adapt the distributions of given data sets, thus lead to better space partition than random projection. By using small number $V = O(k\sqrt[k]{L})$ top principal components which are generated by Randomized SVD, we reduce time complexity of PCALSH to $O(ndV)$, and hence more suitable for large scale data. Our experiment results has showed our approach with small enough value of V clearly outperforms the original LSH. We also confirmed the effect of hashing on the orthogonal unit vectors by Orthogonal LSH, and conclude that the orthogonal properties is not important in Locality Sensitive Hashing methods.

For the future works, we need to study more about the behaviors of PCALSH by implement on other dataset with higher dimension size with several value in parameter setting. It is possible to combine PCALSH with multi-probe LSH, E8-lattice LSH, hierarchical LSH. Multi-probe querying on PCALSH does not need to construct a lot of tables, hence reduce memory usage, PCALSH with E8-lattice might offer better quantization than using projection, and PCALSH with hierarchical structure could adapt the distribution in each dataset. Therefore, we want to developing a full Nearest Neighbor Search package based on PCALSH and its variation with parameters auto-tuning that is comparable with state-of-the-art ANNs library FLANN for both search quality and time-efficiency.

Bibliography

[1] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *Foundations of Computer Science, 2006. FOCS '06. 47th Annual IEEE Symposium on*, pages 459–468, Oct 2006.

[2] Alexandr Andoni and Piotr Indyk. E2LSH 0.1 User Manual. `http://www.mit.edu/~andoni/LSH/manual.pdf`, 2005. [Online; accessed 10-July-2015].

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

[4] Mayank Bawa, Tyson Condie, and Prasanna Ganesan. Lsh forest: Self-tuning indexes for similarity search. In *Proceedings of the 14th International Conference on World Wide Web*, WWW '05, pages 651–660. ACM, 2005.

[5] Jeffrey S. Beis and David G. Lowe. Shape indexing using approximate nearest-neighbour search in high-dimensional spaces. In *Proceedings of the 1997 Conference on Computer Vision and Pattern Recognition (CVPR '97)*, pages 1000–, 1997.

[6] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975.

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

[8] A. Broder. On the resemblance and containment of documents. In *Proceedings of the Compression and Complexity of Sequences 1997*, SEQUENCES '97, pages 21–, Washington, DC, USA, 1997. IEEE Computer Society.

[9] Moses S. Charikar. Similarity estimation techniques from rounding algorithms. In *Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing*, STOC '02, pages 380–388, New York, NY, USA, 2002. ACM.

[10] Sanjoy Dasgupta and Yoav Freund. Random projection trees and low dimensional manifolds. In *Proceedings of the Fortieth Annual ACM Symposium on Theory of Computing*, STOC '08, pages 537–546, New York, NY, USA, 2008. ACM.

[11] Mayur Datar and Piotr Indyk. Locality-sensitive hashing scheme based on p-stable distributions. In *In SCG 04: Proceedings of the twentieth annual symposium on Computational geometry*, pages 253–262. ACM Press, 2004.

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

[13] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. In *INTERNATIONAL CONFERENCE ON MANAGEMENT OF DATA*, pages 47–57. ACM, 1984.

[14] Nathan Halko, Per-Gunnar Martinsson, Yoel Shkolnisky, and Mark Tygert. An algorithm for the principal component analysis of large data sets. *SIAM J. Sci. Comput.*, 33(5):2580–2594, October 2011.

[15] Junfeng He, Wei Liu, and Shih-Fu Chang. Scalable similarity search with optimized kernel hashing. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '10, pages 1129–1138, New York, NY, USA, 2010. ACM.

[16] Piotr Indyk and Rajeev Motwani. Approximate nearest neighbors: Towards removing the curse of dimensionality. In *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, pages 604–613. ACM, 1998.

[17] P. Jain, B. Kulis, and K. Grauman. Fast image search for learned metrics. In *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, pages 1–8, June 2008.

[18] H. Jegou, L. Amsaleg, C. Schmid, and P. Gros. Query adaptative locality sensitive hashing. In *Acoustics, Speech and Signal Processing, 2008. ICASSP 2008. IEEE International Conference on*, pages 825–828, March 2008.

[19] Herve Jegou, Matthijs Douze, and Cordelia Schmid. Hamming embedding and weak geometric consistency for large scale image search. In *Proceedings of the 10th European Conference on Computer Vision: Part I*, ECCV '08, pages 304–317, Berlin, Heidelberg, 2008. Springer-Verlag.

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

[21] Alexis Joly and Olivier Buisson. A posteriori multi-probe locality sensitive hashing. In *Proceedings of the 16th ACM International Conference on Multimedia*, MM '08, pages 209–218, New York, NY, USA, 2008. ACM.

[22] B. Kulis and K. Grauman. Kernelized locality-sensitive hashing. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 34(6):1092–1104, June 2012.

[23] Brian Kulis and Trevor Darrell. Learning to hash with binary reconstructive embeddings. In Y. Bengio, D. Schuurmans, J.D. Lafferty, C.K.I. Williams, and A. Culotta, editors, *Advances in Neural Information Processing Systems 22*, pages 1042–1050. Curran Associates, Inc., 2009.

[24] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. Multi-probe lsh: Efficient indexing for high-dimensional similarity search. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07, pages 950–961. VLDB Endowment, 2007.

[25] K. Mikolajczyk and J. Matas. Improving descriptors for fast tree matching by optimal linear projection. In *Computer Vision, 2007. ICCV 2007. IEEE 11th International Conference on*, pages 1–8, Oct 2007.

[26] C. Silpa-Anan and R. Hartley. Optimised kd-trees for fast image descriptor matching. In *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, pages 1–8, June 2008.

[27] Nakul Verma, Samory Kpotufe, and Sanjoy Dasgupta. Which spatial partition trees are adaptive to intrinsic dimension? In *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence*, UAI '09, pages 565–574, Arlington, Virginia, United States, 2009. AUAI Press.

[28] Jun Wang, S. Kumar, and Shih-Fu Chang. Semi-supervised hashing for scalable image retrieval. In *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on*, pages 3424–3431, June 2010.

[29] Roger Weber, Hans-Jörg Schek, and Stephen Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, pages 194–205, 1998.

[30] Yair Weiss, Antonio Torralba, and Rob Fergus. Spectral hashing. In D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, editors, *Advances in Neural Information Processing Systems 21*, pages 1753–1760. Curran Associates, Inc., 2009.

[31] Hao Xu, Jingdong Wang, Zhu Li, Gang Zeng, Shipeng Li, and Nenghai Yu. Complementary hashing for approximate nearest neighbor search. In *Computer Vision (ICCV), 2011 IEEE International Conference on*, pages 1631–1638, Nov 2011.

[32] Zixiang Yang, Wei Tsang Ooi, and Qibin Sun. Hierarchical, non-uniform locality sensitive hashing and its application to video identification. In *Multimedia and Expo, 2004. ICME '04. 2004 IEEE International Conference on*, volume 1, pages 743–746 Vol.1, June 2004.

[33] Peter N. Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '93, pages 311–321, 1993.