Complementary Hashing for Approximate Nearest Neighbor Search

Hao Xu[†] Jingdong Wang[‡] Zhu Li[§] Gang Zeng[¶] Shipeng Li[‡] Nenghai Yu[†]
[†]MOE-MS KeyLab of MCC, University of Science and Technology of China, P. R. China
[‡]Microsoft Research Asia, P. R. China
[§]Core Networks Research, Huawei Technology, USA
[¶]Key Laboratory of Machine Perception, Peking University, P. R. China
xuhao657@ustc.edu, {jingdw, spli}@microsoft.com, zhu.li@ieee.org, gang.zeng@pku.edu.cn, ynh@ustc.edu.cn

Abstract

Recently, hashing based Approximate Nearest Neighbor (ANN) techniques have been attracting lots of attention in computer vision. The data-dependent hashing methods, e.g., Spectral Hashing, expects better performance than the data-blind counterparts, e.g., Locality Sensitive Hashing (LSH). However, most data-dependent hashing methods only employ a single hash table. When higher recall is desired, they have to retrieve exponentially growing number of hash buckets around the bucket containing the query, which may drag down the precision rapidly. In this paper, we propose a so-called complementary hashing approach, which is able to balance the precision and recall in a more effective way. The key idea is to employ multiple complementary hash tables, which are learned sequentially in a boosting manner, so that, given a query, its true nearest neighbors missed from the active bucket of one hash table are more likely to be found in the active bucket of the next hash table. Compared with LSH that also can exploit multiple hash tables, our approach is more effective to find true NNs, thanks to the complementarity property of the hash tables from our approach. Experimental results on large scale ANN search show that the proposed method significantly improves the performance and outperforms the state-of-the-art.

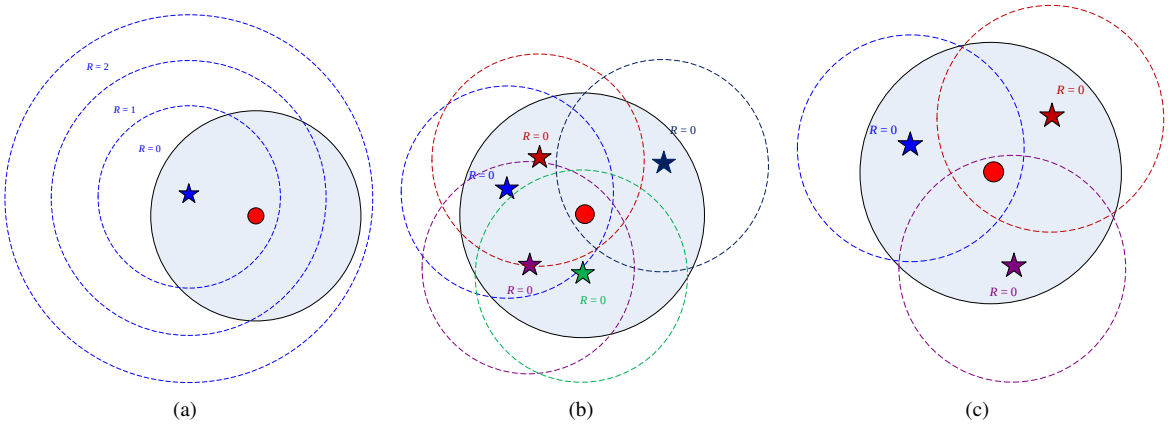
1. Introduction

Similarity search, also known as nearest neighbor search, addresses the problem of, given a query point, finding its most similar points from the database. It is a fundamental problem in many practical applications, such as k-nearest neighbor classification, Content Based Image Retrieval (CBIR) and so on. With the growth of the size of the database, the naive approach adopting linear scan becomes impractical. Therefore, recently a lot of research efforts have been devoted to investigate the alternative solution - Approximate Nearest Neighbor (ANN) search, which trades off a little search accuracy to greatly speed up the search process.

Among existing ANN methods, hashing based methods have demonstrated promising performances [1, 8, 19, 20, 21]. The basic idea is to construct hash functions to map the data points to finite number of hash codes, so that similar data points have larger probability of collision, i.e., having the same hash code. Without loss of generality, a hash code is considered to be made of a group of hash bits in this paper. Hashing based methods can be roughly divided into two categories, data-blind hashing and data-dependent hashing, according to whether they make use of the database to construct the hash functions.

Locality Sensitive Hashing (LSH) [1, 3, 7] is one of the best known methods in the first category. It produces each hash bit typically by projecting the data point to a random hyperplane and then conducting random thresholding. Multiple hash tables are independently constructed, aiming to enlarge the probability that similar data points are mapped to similar hash codes. In practice, due to the data-blindness and independence, LSH suffers from severe redundancy of the hash bits as well as redundancy of the hash tables. Consequently, on the one hand, LSH may need very long hash codes to encourage only similar points to be projected to similar hash codes, and on the other hand, lots of hash tables are needed to access enough points for the satisfactory recall. This leads to many practical problems, such as the increase of the query time and the big storage overhead for the large number of hash tables.

A representative method in the second category is Spectral Hashing (SH) [21]. It can produce very compact hash codes by thresholding with nonlinear functions along the principal directions of the data. Given a query, it retrieves all the points whose hash codes fall within a Hamming ball centered at the query's hash code, i.e., the Hamming distances between the retrieved points and the query are not larger than the radius of the ball. When higher recall is desired, they usually have to increase the radius of the Ham-



(a) (b) (c)

Figure 1. Illustration of the differences using (a) single hash table (e.g., spectral hashing), (b) multiple random hash tables (LSH), and (c) multiple learned hash tables (our approach). Suppose the true neighbors of the query (the red bullet ●) distribute in the shaded area. To cover the shaded region, (a) requires a big Hamming ball, (b) needs many small balls, while fewer small balls are enough shown in (c). We illustrate the inverse projection of the Hamming ball as the ball in the data space centered at the star, and the balls with $R = 0$ corresponds to the active hash buckets hit by the query in different hash tables.

ming ball to retrieve more points. We visually illustrate the arising problem in Fig. 1(a). A big Hamming ball with radius 2 is needed to cover the shaded region, which contains the true near neighbors. Since the ball grows homogenously and explosively, a large piece of unshaded area, which contains many irrelevant points, is also covered by the ball. This may drag down the precision rapidly.

In this paper, we present the complementary hashing, which indexes data points with multiple *complementary* hash tables and is able to balance the precision and recall in a more effective way. The hash tables are sequentially learned from the data in a boosting manner, so that, given a query, its true nearest neighbors missed from the active bucket of one hash table are very likely to be found in the active bucket of the subsequent hash table. Different from LSH, which employs multiple *independent* hash tables, the proposed method constructs the hash functions in a *complementary* manner. Compared with the methods adopting a single hash table (e.g., spectral hashing), employing multiple hash tables is more helpful to balance the precision and recall. Taking Fig. 1(c) as an example, where the query falls in three hash buckets belong to three different hash tables, indicated by three small balls. If we properly learn the hash tables from the data, the shaded area, which contains the true near neighbors, will be covered by a smaller number of small Hamming balls. Thus many irrelevant points are avoided and accordingly the search performance is improved. In contrast, LSH adopts the scheme, illustrated in Fig. 1(b), which is not as efficient as our method and results in requiring more hash tables to guarantee the recall.

We summarize the contributions in this paper as follows. We propose the complementary hashing approach to effectively balance the precision and the recall, and then present a boosting algorithm to effectively learn the multiple hash tables in a sequential order. Moreover, an incremental indexing scheme is proposed, so that only a fraction of data points are needed to be indexed by the subsequent hash tables. This from another point of view means the points can be assigned with hash codes of various length, so that we are able to trade off the search efficiency and storage cost. We experimentally illustrate the advantages of adopting multiple hash tables. Experimental results justify the superiority of our approach over existing representative methods.

2. Related Work

We first introduce some annotations for later convenience. Suppose the database $\mathbf{X}$ consists of N data points $\{\mathbf{x}_i\}_{i=1}^N$, $\mathbf{x}_i \in \mathbb{R}^d$. $\mathbf{A}$ is the similarity matrix and a_{ij} denotes the similarity of $\mathbf{x}_i$ and $\mathbf{x}_j$. A hashing method adopts K hash functions to map a data point $\mathbf{x}$ to a K-bit hash code $H(\mathbf{x}) = [h_1(\mathbf{x}), \ldots, h_K(\mathbf{x})]$, where each hash function maps the data point to a single bit $h_k(\mathbf{x}) \in \{-1, 1\}$. A hash code is also referred as a hash bucket, which contains the points being mapped to it. We define the combination of the K hash functions as a *hash projection*, denoted as H. Recently, many ANN search methods, e.g., kd-trees [2, 9] are proposed. In the following, we mainly review the closely-related hashing based methods.

2.1. Locality Sensitive Hashing and Extensions

Locality Sensitive Hashing (LSH) constructs L hash tables using the hash projections $\{H_l\}_{l=1}^L$. The k-th hash function of the l-th hash projection is in the form of:

$$h_k^l(\mathbf{x}) = \mathrm{sgn}((\mathbf{w}_k^l)^T\mathbf{x} + b_k^l), \tag{1}$$

where $\mathbf{w}_k^l$ is a random hyperplane and b_k^l is a random threshold. With these hash projections, LSH projects each

data point in the database to L K-bit hash codes. To perform search, a query is mapped to L hash codes in the same way and data points having the same hash codes are retrieved from each hash table. In practice, because LSH suffers from severe redundancy of the hash bits, K has to be sufficiently large to achieve satisfactory precision, which requires large L to maintain reasonable probability of collision (projected to the same hash code) for the similar points. This leads to the big storage burden of holding the the hash tables and high computational cost of projecting the query to the hash codes.

There are some works extending LSH from different perspectives. [4, 13, 16] try to reduce the number of desired hash tables of LSH without lost of the search accuracy. [8] presents a supervised version of LSH. In [10], Kulis et al. propose a kernelized version of LSH. However, they still suffer from the redundancy in the hash codes.

2.2. Spectral Hashing and Extensions

Spectral Hashing (SH) is proposed by Weiss et al., to learn hash functions from the data by minimizing the following objective function:

$$J(H) = \sum_{i,j=1}^{N} a_{ij} \|H(\mathbf{x}_i) - H(\mathbf{x}_j)\|^2 \quad (2)$$

$$s.t. \sum_{i=1}^{N} H(\mathbf{x}_i) = 0 \quad (3)$$

$$\frac{1}{N} \sum_{i=1}^{N} H(\mathbf{x}_i) H(\mathbf{x}_i)^T = \mathbf{I} \quad (4)$$

$$H(\mathbf{x}_i) \in \{-1, 1\}^K$$

The first two constraints are added to provide the following two good properties. 1) The hash codes are efficient: each hash bit partitions the data points into two balanced parts and 2) The hash codes are compact: the bits of a hash code are uncorrelated. As shown in [21], this minimization problem can be converted to an eigenvalue decomposition problem and efficiently solved. Thanks to the data-dependent hash functions, SH is able to produce better hash codes than LSH in practice. Many extensions of SH have been proposed in recent years, including the kernelized versions [6, 15], a semi-supervised scheme [19], a speed-up scheme [11] and a self-taught scheme which trains a classifier per bit [22].

More recently, Unsupervised Sequential Projection Learning for Hashing (USPLH) is proposed by Wang et al. [20]. Its basic idea is to learn the hash functions sequentially, in a way that the subsequent hash functions are in charge of correcting the "errors" made by the previous hash functions. Since USPLH, as well as SH and its extensions, only employs a single hash table, they suffer from the problem that we described in the Introduction Section. Compared with USPLH, instead of constructing a single hash table by learning multiple hash bits one by one, our algorithm learns multiple hash tables one by one, so that can balance the precision and the recall more effectively.

Table 1. Comparison of four representative hashing methods with our method.

Method	Data dependent?	Multiple hash tables?	Complementary hash tables?
LSH	no	yes	no
SH	yes	no	-
USPLH	yes	no	-
PCH	yes	yes	no
Our method	yes	yes	yes

There are two previous works on the data dependent hashing approaches with multiple hash tables. [17] presents a k-means based LSH, constructing multiple hash tables by using different randomly generated seeds for k-means. Principal Component Hashing (PCH) [14] constructs hash tables independently along the principle axes of the data points. Unlike our proposed method which learns complementary hash buckets, the hash buckets of above two approaches are independent to each other.

Finally, we compare the proposed method with four representative hashing methods in Tab. 1.

3. Complementary Hashing

The goal of the complementary hashing is to minimize the following objective function:

$$J(\{H_l\}_{l=1}^{L}) = \sum_{i,j=1}^{N} \left(a_{ij} \min_{l=1..L} \|H_l(\mathbf{x}_i) - H_l(\mathbf{x}_j)\|^2 \right). \quad (5)$$

Like LSH, CH employs multiple hash tables and we also consider the hash functions in the form of Eqn. (1) in this paper. But instead of using random hyperplanes $\mathbf{w}$ and random thresholds b to construct the hash function, we learn such parameters from the data. The idea behind this formulation is straightforward. For two data points $\mathbf{x}_i$ and $\mathbf{x}_j$, whose similarity a_{ij} is relatively large, we encourage that they have similar hash codes in at least one hash table. In the case that $L = 1$, say, only a single hash table is employed, the objective function becomes exactly the same with that of SH, i.e., Eqn. (2). Ideally, all the true neighbors of any query can be found by only retrieving the points in the active hash bucket of each hash table. This is infeasible using only a single hash table, since it corresponds to a single partition of the data space and the query near the boundary of the partition is likely to get untrue neighbors. For each hash table we impose the same constraints as SH does, i.e., Eqn. (3) and Eqn. (4), so as to produce efficient and compact hash codes in each hash table as well.

3.1. Algorithm

We adopt a boosting-based approach to solve this minimization problem. Taking each pair $(\mathbf{x}_i, \mathbf{x}_j)$ as an element

and a hash projection as a classifier to predict the label for each element, the boosting scheme learns the new classifier by paying more attention to the misclassified elements from the previous classifiers [5]. The label of an element is 1 or -1 according to whether the two component points of the element are sufficiently similar, i.e., $b_{ij} = \mathrm{sgn}(a_{ij} - \alpha)$. A hash projection H conducts the prediction by measuring the similarity of the hash codes of the element's two component points:

$$P_H(\mathbf{x}_i, \mathbf{x}_j) = \begin{cases} 1, & \frac{1}{4}\|H(\mathbf{x}_i) - H(\mathbf{x}_j)\|^2 < \beta \\ -1. & \text{otherwise} \end{cases} \quad (6)$$

In a boosting paradigm, each element is associated with a weight, and the classifier is learned in a way that the inaccurate predictions for the elements with larger weights incurs greater penalty. To make the subsequent classifier (hash projection) lay more emphasis on the misclassified elements by the previous classifiers, we need to assign larger weights to those misclassified elements than the elements being correctly classified. Different from the conventional AdaBoost [5], which makes prediction using the weighted sum of the outputs of the weak classifiers, our overall classifier uses the optimal decision of the member classifiers. Because in the scenario of ANN search, instead of voting for which point to retrieve, independent search is performed in each hash table and the retrieved points are then merged to generate the final search result. Hence the optimal decision made by a member hash table directly contributes to the final search performance. Taking account of such a difference, we update the weight matrix $\mathbf{S}$ according to current classifier H as:

$$s_{ij} = \begin{cases} 0, & b_{ij} = P_H(\mathbf{x}_i, \mathbf{x}_j) \\ \min(s_{ij}, f_{ij}), & b_{ij} = 1, P_H(\mathbf{x}_i, \mathbf{x}_j) = -1 \\ -\min(-s_{ij}, f_{ij}). & b_{ij} = -1, P_H(\mathbf{x}_i, \mathbf{x}_j) = 1 \end{cases} \quad (7)$$

$$f_{ij} = (a_{ij} - \alpha)(\frac{1}{4}\|H(\mathbf{x}_i) - H(\mathbf{x}_j)\|^2 - \beta).$$

Here, if an element is predicted correctly by current classifier, its weight is set to zero and will not change any more in the future updates. Otherwise, the prediction errors can be categorized into two types: 1) a pair of similar points are projected to the dissimilar hash codes and 2) a pair of dissimilar points are projected to the similar hash codes. In either case, we adjust the corresponding weight to reflect the degree of the contradiction of the original similarity and the similarity in the Hamming space. Following Eqn. (7), the weight of the pair of similar points will always be greater or equal than zero, while that of the pair of dissimilar points less or equal than zero. The magnitude of the weight is constantly decreasing, since we keep the old weight unchanged if some previous classifier works better than the current classifier.

Given the weighted elements, we learn a hash projection by maximizing the following objective function:

$$\hat{J}(H) = \sum\nolimits_{i,j=1}^{N} s_{ij} H(\mathbf{x}_i)^T H(\mathbf{x}_j), \quad (8)$$

where $H(\mathbf{x}_i)^T H(\mathbf{x}_j) = \sum_{k=1}^{K} h_k(\mathbf{x}_i) h_k(\mathbf{x}_j)$ reflects the similarity of the two hash codes. Here without loss of generality, we assume the data is normalized to have zero mean, so that $b_k = 0$ for mean thresholding. Straightforwardly, this objective function encourages a pair of points to be projected to the similar hash codes if the corresponding weight is large, and be projected to the dissimilar hash codes otherwise.

In order to make the learned hash projection subjects to the constraint Eqn. (3), we propose to maximize the variance of the projected data [19]:

$$\max_{\mathbf{W}} \mathbf{tr}\left[\mathbf{W}^T \mathbf{X}\mathbf{X}^T \mathbf{W}\right]. \quad (9)$$

where $\mathbf{W}$ is a $d \times K$ matrix, the rows of which is formed of $\{\mathbf{w}_k\}_{k=1}^{K}$. By relaxing $\mathrm{sgn}(\mathbf{w}^T\mathbf{x})$ to the signed magnitude $\mathbf{w}^T\mathbf{x}$ in Eqn. (8), and combining the regularization term Eqn. (9), we transform Eqn. (8) to the following objective function:

$$\begin{aligned} \hat{J}(H) &= \mathbf{tr}\left[\mathbf{W}^T \mathbf{X}\mathbf{S}\mathbf{X}^T \mathbf{W} + \eta \mathbf{W}^T \mathbf{X}\mathbf{X}^T \mathbf{W}\right] \\ &= \mathbf{tr}\left[\mathbf{W}^T \mathbf{M}\mathbf{W}\right]. \end{aligned}$$

where $\mathbf{M} = \mathbf{X}\mathbf{S}\mathbf{X}^T + \eta\mathbf{X}\mathbf{X}^T$ is a $d \times d$ matrix. Viewing the weight matrix $\mathbf{S}$ as the supervised data, this can be understood as a semi-supervised formulation. Parameter η trades off the effects of the supervised data and the regularizer. This objective function is similar to that in [19], expect that we use a real valued weight matrix $\mathbf{S}$ instead of the label matrix whose elements are $\{-1, 0, 1\}$. This problem has a closed-form solution, that $\mathbf{w}_k$ is the eigenvector corresponding to the k-th largest eigenvalue of $\mathbf{M}$, and b_k is the median value of $\mathbf{w}_k\mathbf{x}$ for $\mathbf{x} \in \mathbf{X}$. The orthogonality property for $\mathbf{w}$ approximately guarantees Eqn. (4).

The overall procedure of complementary hashing is summarized in Alg. 1. We initialize the weight matrix as $s_{ij} = K(a_{ij} - \alpha)$, and it is straightforward to incorporate the prior supervised data by modifying this initialization.

3.2. Scalability Extension

There are two practical issues in Alg. 1. First, two $N \times N$ matrices, the similarity matrix $\mathbf{A}$ and the weight matrix $\mathbf{S}$, are involved in the algorithm. For the large scale dataset, computing with such huge matrices is infeasible. Second, it produces multiple hash codes per point. The storage required to hold the hash tables grows linearly along with the growth of the number of hash tables. Actually, the big storage burden is one of the major problems that hinders LSH

Algorithm 1 Complementary Hashing (CH)

Input: data $\mathbf{X}$, length of hash codes K, number of hash tables L.

Output: hash projections $\{H_l\}_{l=1}^{L}$.

Initialize the similarity matrix and the weight matrix: $a_{ij} = \mathrm{sim}(\mathbf{x}_i, \mathbf{x}_j)$, $s_{ij} = K(a_{ij} - \alpha)$.
for $l = 1$ **to** L **do**
  Compute covariance matrix:
    $\mathbf{M} = \mathbf{X}\mathbf{S}\mathbf{X}^T + \eta\mathbf{X}\mathbf{X}^T$.
  Learn the hash projection H_l (i.e., $\{\mathbf{w}_k, b_k\}_{k=1}^{K}$):
    $\mathbf{w}_k$ is the eigenvector corresponding to the k-th largest eigenvalue of $\mathbf{M}$.
    b_k is the median value of $\mathbf{w}_k\mathbf{x}$ for $\mathbf{x} \in \mathbf{X}$.
  Update the weight matrix $\mathbf{S}$ by Eqn. (7).
end for

being used in the practical applications. In this subsection, we address these issues and present a scalable version of CH.

To handle the issue of the huge matrices, we make use of sparse matrices instead. To make $\mathbf{S}$ sparse, we only update the weights for a small proportion of the elements, which are more likely to be misclassified by the previous classifiers. In fact, the misclassified elements are more likely to consist of the points near the hash hyperplanes. Recall that there are two types of errors to consider when updating the weight matrix: the paired similar points being projected to dissimilar hash codes and the paired dissimilar points being projected to similar hash codes. For the first type of error, it is easy to see that the misclassified pair of points must satisfy the following two properties: 1) they are close to each other in the data space and 2) they are separated by many hash hyperplanes.From these observations, we can deduce that the misclassified pair of points must be distributed near the hash hyperplanes. For the second type of error, the misclassified pair consists of two dissimilar points. Consider the hyperplanes as walls, the points with the same hash code are confined to the same room. If two dissimilar points exist in a room, they are supposed to be close to the opposite walls, since they are far away from each other. Therefore, we first select a group of candidate points from the vicinity of the hash hyperplanes:

$$\mathcal{X} = \{\mathbf{x} | d_l(\mathbf{x}) < \epsilon\}, \tag{10}$$
$$d_l(\mathbf{x}) = \max\left(d_{l-1}(\mathbf{x}), \min_{k=1..K} |\mathbf{w}_k^l \mathbf{x} + b_k^l|\right),$$

where $d_0(\mathbf{x})$ is set to zero. $\mathcal{X}$ is also expressed in the matrix form $\hat{\mathbf{X}}$, the columns of which are the points contained in $\mathcal{X}$. Note that if a point is far away from all hash hyperplanes in one of the previous hash tables, it will be never selected. Next, we only update the weights for the data pairs formed by these candidate points, constructing a sparse weight matrix $\mathbf{S}$. To avoid computing the dense similarity matrix $\mathbf{A}$, we initialize the weight matrix with equal weight for each data pair, i.e., $s_{ij} = K$.

To handle the storage issue, we treat the points unequally so that a majority of points are only indexed by a part of hash tables, i.e., a majority of points have less than L hash codes. Since the hash projections, except the first one, play a complementary role of handling the misclassified elements from the previous hash projections, it is reasonable to only keep those misclassified points in the subsequent hash tables. Due to the boosting paradigm, the number of misclassified elements drops rapidly when more and more hash tables are employed. This means a majority of points are only indexed by the first few hash tables, dramatically reducing the overall storage to hold the hash tables. Instead of finding the points forming the misclassified pairs, which is quite computationally intensive, the points collected from the vicinity of the hash hyperplanes by Eqn. (10) are considered. Because, as aforementioned, the misclassified data pairs are more likely constituted by these points. In this way, for a single point, if we consider the concatenation of all its associated hash codes in the corresponding buckets as a single hash code of this point, the length of hash codes for different data points may be different. A point $\mathbf{x}_i$ being indexed in L_i buckets has a hash code of length KL_i. When a majority of points are only indexed by the first few hash tables, the total storage cost $\sum_{i=1}^{N} KL_i$ will be close to $O(NK)$, saved lots of space compared to the fully indexing scheme which requires $O(NKL)$ space.

The CH's scalability extension is described in Alg. 2. Different from Alg. 1, both hash projections and hash tables are constructed during the learning process.

Algorithm 2 CH's scalability extension

Input: data $\mathbf{X}$, length of hash codes K, number of hash tables L.

Output: hash projections $\{H_l\}_{l=1}^{L}$, hash tables $\{T_l\}_{l=1}^{L}$.

Initialize $\hat{\mathbf{X}} = \mathbf{X}$, $s_{ij} = K$, $T_l = \emptyset$.
for $l = 1$ **to** L **do**
  Compute covariance matrix:
    $\mathbf{M} = \hat{\mathbf{X}}\mathbf{S}\hat{\mathbf{X}}^T + \eta\hat{\mathbf{X}}\hat{\mathbf{X}}^T$.
  Learn the hash projection H_l (i.e., $\{\mathbf{w}_k, b_k\}_{k=1}^{K}$):
    $\mathbf{w}_k$ is the eigenvector corresponding to the k-th largest eigenvalue of $\mathbf{M}$.
    b_k is the median value of $\mathbf{w}_k\mathbf{x}$ for $\mathbf{x} \in \hat{\mathbf{X}}$.
  Construct the hash table:
    $T_l(H_l(\mathbf{x})) = T_l(H_l(\mathbf{x})) \cup \mathbf{x}$, for all $\mathbf{x} \in \hat{\mathbf{X}}$.
  Select a set of candidate points $\hat{\mathbf{X}}$ by Eqn. (10).
  Update $\mathbf{S}$ only for the data pairs formed by the points in $\hat{\mathbf{X}}$ according to Eqn. (7).
end for

4. Experiments

4.1. Setting

In this section, we compare the proposed method against some representative hashing methods, i.e., LSH, SH and USPLH, to justify the effectiveness of the proposed method. Two datasets are used in the experiments, 20K 512-dimensional Gist features extracted from the images of LabelMe dataset [18] and 1 million 128-dimensional SIFT descriptors extracted from random images [12]. We randomly select 2K points from the LabelMe dataset and 10K points from the SIFT dataset respectively as the testing queries, and the other points are taken as the database. The ground truth neighbors of a query are obtained by the brute force search, and a data point is considered to be a true neighbor if it lies in the top 2 percent points closest to the query, in terms of Euclidean distance. For the data-dependent hashing methods, i.e., SH, USPLH and CH, the whole database is used for learning the hash functions.

The following two schemes are usually adopted by the hashing based methods to conduct ANN search. 1) Hamming ranking: The Hamming distance between the hash codes of the query and each point in the database is calculated. The points are then ranked according to the corresponding Hamming distances, and a certain number of top ranked points are retrieved. Though the complexity of Hamming ranking is linear to the size of the database, it can be implemented very fast, taking advantage of the capability of the hardware to efficiently compute the Hamming distance. 2) Hash lookup: All the points whose hash codes fall within a Hamming radius around the query's hash code are retrieved. Hash lookup usually enjoys lower complexity than Hamming ranking. We will compare different hashing methods using both of these schemes in the experiments.

To perform Hamming ranking for the hashing methods with multiple hash tables, i.e., LSH and CH, we compute the Hamming distance of a point $\mathbf{x}_i$ and the query $\mathbf{x}_q$ by

$$d(\mathbf{x}_i, \mathbf{x}_q) = \min_{l=1..L} \frac{1}{4} \|H_l(\mathbf{x}_q) - H_l(\mathbf{x}_i)\|^2 / 1_{(\mathbf{x}_i \in T_l(H_l(\mathbf{x}_i)))},$$

where T_l is the l-th hash table, and $1_{(\cdot)}$ is the indicator function, which is used here to exclude the points that are not indexed by a hash table (such case is possible for CH). These Hamming distances are then used to perform ranking. Performing hamming ranking in this way can leverage the partial indexing structure of our hashing scheme, which is equivalent to adopting hash codes of various length.

4.2. Implementation

LSH is implemented according to [3] and the code of SH is obtained from the author's Web site. The parameters of USPLH are well adjusted with a validation dataset, which is randomly sampled from the database. We implement two versions of complementary hashing, CH_p and CH. CH_p is implemented as described in Alg. 2. Its hash tables, except the first one, only index a part of the points of the database, in order to save storage. CH is almost the same with CH_p and the only differences is that all the points are indexed in every hash table. For all the experiments, LSH, CH and CH_p employ the same number of hash tables.

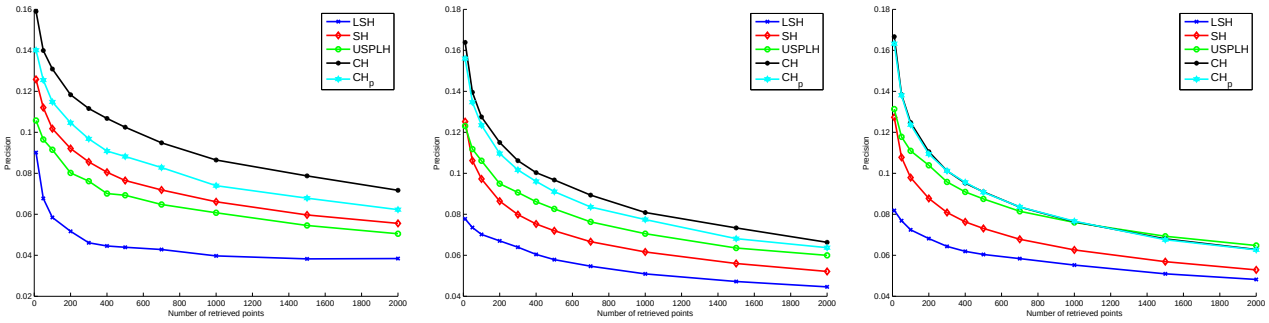
4.3. Comparison with Other Methods

We first do the experiment using Hamming ranking. The performances of the five methods, in terms of the precision versus the number of retrieved points, are illustrated in Fig. 2 and Fig. 3. Both LSH and CH use three hash tables in all the experiments in this subsection.

As can be seen in the figures, CH outperforms the other methods significantly, which justifies the statement in the Introduction Section. That is, to retrieve more points, the methods with a single hash table, say, SH and USPLH, need to check the nearby hash buckets by increasing the Hamming radius to search, which makes the number of active buckets as well as the number of irrelevant points involved grow exponentially. The ratio of true neighbors drops quickly along with the increase of Hamming radius to search, undermines the precision of these methods.

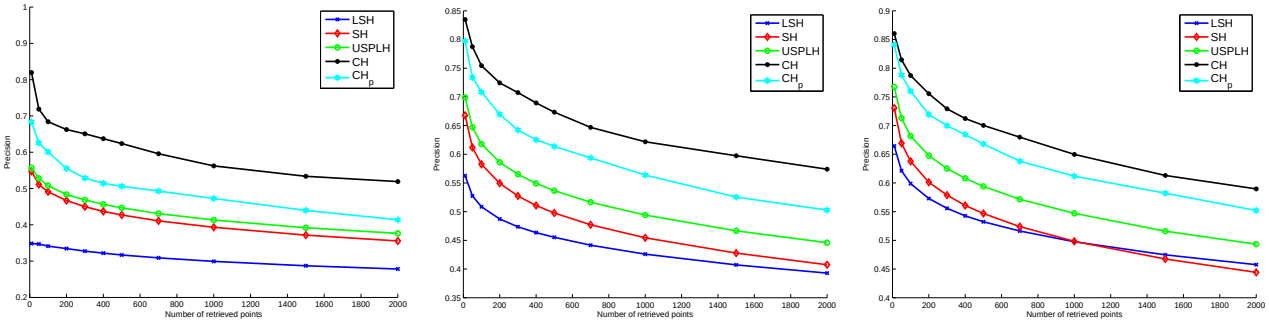
In contrast, the methods with multiple hash tables simultaneously check the nearby hash buckets in each hash table, which hopefully can find enough true neighbors without the necessity of growing the Hamming radius too large. LSH constructs the hash tables in a random way, resulting in the different hash tables may suffer from severe redundancy, i.e., find the same proportion of true neighbors. CH learns the hash tables in a complementary mechanism so that such a redundancy across the hash tables is dramatically reduced. The problem is more obvious for the short hash codes, since in that case the hash bucket usually contains more points. That's why CH outperforms the other methods more significantly for the short hash codes, e.g., $K = 16$. The performance of CH_p is worse than CH but better than the other methods. This is understandable since it trades off the search accuracy for the storage efficiency, i.e., CH_p requires around 60% of the storage of CH in this experiment.

Fig. 4 illustrates the performance of the five methods using Hash lookup. The points within Hamming radius 2 are retrieved. Both CH and CH_p outperform the other methods using the hash codes of short and moderate length. USPLH works better when long hash codes, say, $K = 48, 64$, are used. This is probably because it relaxes the orthogonal constraint (Eqn. (4)) of the hash hyperplanes, which forces the hash projection to progressively select the directions that the variance of the data is small. This problem is more obvious when the number of hash bit grows. That's why, in the case that long hash codes is used, the methods



(a) (b) (c)

Figure 2. Comparison of the performance using Hamming ranking on the 20K LabelMe dataset. (a), (b) and (c) are the performances for the hash codes of 16, 24 and 32 bits respectively.



(a) (b) (c)

Figure 3. Comparison of the performance using Hamming ranking on the 1M SIFT dataset. (a), (b) and (c) are the performances for the hash codes of 16, 24 and 32 bits respectively.

with the orthogonality constraint, say SH, CH and CH$_p$, even work worse than LSH, which randomly selects the hash hyperplanes, for the LabelMe dataset. Our approach can also have such an extension by relaxing the orthogonal constraints.

4.4. Accuracy-Storage Tradeoff

The proposed method enjoys the flexibility of trading off the search accuracy with the storage cost. Such a trade-off can be controlled by the two parameters, L and ϵ. L is the number of hash tables, and fewer hash tables leads to less storage overhead. In Fig. 5, we show the performance of LSH and CH according to different L for the SIFT dataset. Generally, both LSH and CH perform better when more hash tables are used, and their performances trend to be converged when $L > 10$. Due to the data-dependent hash functions, CH performs better even with only a single hash table. The second and third hash tables bring significant performance gain, and the contributions of the later hash tables become less significant. This observation is in line with the intuition, since the hash projections are learned in a boosting way. In summary, CH constantly outperforms LSH in the case that the same storage is required.

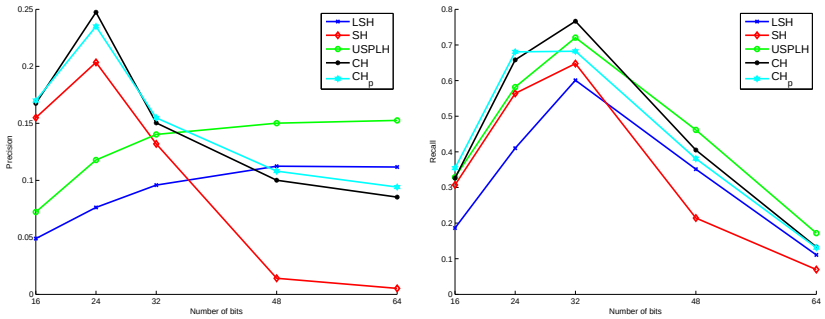
In the proposed method, we can also trade off the search accuracy and the storage cost by adjusting the parameter ϵ, which controls the number of points to be indexed by the hash tables. In Fig. 6, we show the performance of CH$_p$ for different ϵ for the SIFT dataset, together with the performance of USPLH and SH for comparison. All the methods use 24-bit hash codes. We can see that larger ϵ leads to better performance, since larger ϵ indicates more data points are indexed by the hash tables. The corresponding number of data points indexed by the hash tables is shown in Tab. 2. In the case of small ϵ, e.g., $\epsilon = 0.01$, the proposed method can still perform better than USPLH and SH, with only a little storage overhead.

Table 2. The percentage of points indexed in the three hash tables of CH$_p$ in the cases of different ϵ, corresponding to the performance curves in Fig. 6.

Parameter	Table#1	Table#2	Table#3
$\epsilon = 0.01$	100%	13.3%	1.9%
$\epsilon = 0.03$	100%	34.9%	12.4%
$\epsilon = 0.05$	100%	50.9%	26.1%
$\epsilon = 0.2$	100%	100%	100%

5. Conclusion

In this paper, we present the complementary hashing, which is a kind of data-dependent hashing with multiple



(a) (b)

Figure 4. Comparison of the performance with Hash lookup on (a) 20K LabelMe dataset and (b) 1M SIFT dataset. The points within Hamming radius 2 are retrieved.

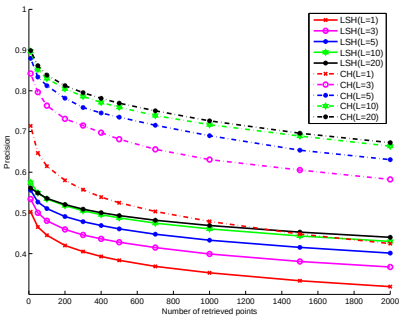


Figure 5. Comparison of LSH and CH in the cases that different numbers of hash tables are employed.

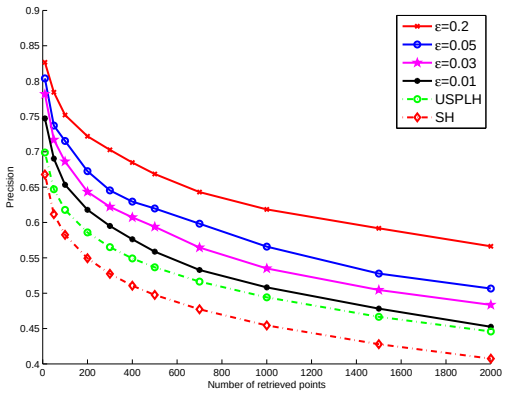


Figure 6. The performance of CH_p in the cases of different ϵ, together with the performance of USPLH and SH for comparison.

hash tables and is able to balance the precision and recall more effectively. The hash tables are sequentially learned from the data in a boosting manner, so that different hash tables are likely to contribute complementary parts of the true neighbors of the query. We experimentally illustrate the advantages of adopting multiple complementary hash tables, compared with LSH that constructs multiple hash tables in the data-blind way, and the methods adopting only a single hash table.

Acknowledgement

We would like to thank anonymous reviewers for their helpful suggestions and thank Jun Wang for sharing the dataset. This work is supported in part by the Fundamental Research Funds for the Central Universities WK2100230002.

References

[1] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *FOCS*, pages 459–468, 2006.

[2] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[3] M. Charikar. Similarity estimation techniques from rounding algorithms. In *STOC*, pages 380–388. ACM, 2002.

[4] W. Dong, Z. Wang, W. Josephson, M. Charikar, and K. Li. Modeling LSH for performance tuning. In *CIKM*, pages 669–678. ACM, 2008.

[5] J. Friedman, T. Hastie, and R. Tibshirani. Additive logistic regression: a statistical view of boosting. *The annals of statistics*, 28(2):337–407, 2000.

[6] J. He, W. Liu, and S. Chang. Scalable similarity search with optimized kernel hashing. In *SIGKDD*, pages 1129–1138. ACM, 2010.

[7] P. Indyk and R. Motwani. Approximate nearest neighbors: towards removing the curse of dimensionality. In *STOC*, pages 604–613, 1998.

[8] P. Jain, B. Kulis, and K. Grauman. Fast image search for learned metrics. In *CVPR*, pages 1–8. IEEE, 2008.

[9] Y. Jia, J. Wang, G. Zeng, H. Zha, and X.-S. Hua. Optimizing kd-trees for scalable visual descriptor indexing. In *CVPR*, pages 3392–3399, 2010.

[10] B. Kulis and K. Grauman. Kernelized locality-sensitive hashing for scalable image search. In *ICCV*, volume 1, page 3, 2009.

[11] W. Liu, J. Wang, S. Kumar, and S.-F. Chang. Hashing with graphs. In *ICML*, pages 1–8, June 2011.

[12] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *IJCV*, 60(2):91–110, 2004.

[13] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-probe lsh: Efficient indexing for high-dimensional similarity search. In *VLDB*, pages 950–961, 2007.

[14] Y. Matsushita and T. Wada. Principal component hashing: An accelerated approximate nearest neighbor search. In *PSIVT*, pages 374–385, 2009.

[15] Y. Mu, J. Shen, and S. Yan. Weakly-supervised hashing in kernel space. In *CVPR*, pages 3344–3351, 2010.

[16] R. Panigrahy. Entropy based nearest neighbor search in high dimensions. In *SODA*, pages 1186–1195. ACM, 2006.

[17] L. Paulevé, H. Jégou, and L. Amsaleg. Locality sensitive hashing: a comparison of hash function types and querying mechanisms. *Pattern Recognition Letters*, 2010.

[18] A. Torralba, R. Fergus, and Y. Weiss. Small codes and large image databases for recognition. In *CVPR*, pages 1–8, 2008.

[19] J. Wang, S. Kumar, and S. Chang. Semi-supervised hashing for scalable image retrieval. In *CVPR*, pages 3424–3431, 2010.

[20] J. Wang, S. Kumar, and S. Chang. Sequential Projection Learning for Hashing with Compact Codes. In *ICML*, pages 1127–1134, 2010.

[21] Y. Weiss, A. Torralba, and R. Fergus. Spectral hashing. In *NIPS*, volume 21, pages 1753–1760, 2009.

[22] D. Zhang, J. Wang, D. Cai, and J. Lu. Self-taught hashing for fast similarity search. In *SIGIR*, pages 18–25, 2010.