Complementary Projection Hashing

Zhongming Jin[1], Yao Hu[1], Yue Lin[1], Debing Zhang[1], Shiding Lin[2], Deng Cai[1], Xuelong Li[3]
[1]State Key Lab of CAD&CG, College of Computer Science, Zhejiang University, Hangzhou, P. R. China
[2]Baidu, Inc., Beijing, P. R. China
[3]Center for OPTical IMagery Analysis and Learning (OPTIMAL), State Key Laboratory of Transient Optics and Photonics, Xi'an Institute of Optics and Precision Mechanics, Chinese Academy of Sciences, Xi'an 710119, Shaanxi, P. R. China
{jinzhongming888, huyao001, linyue29, debingzhangchina}@gmail.com, linshiding@baidu.com, dengcai@cad.zju.edu.cn, xuelong_li@opt.ac.cn

Abstract

Recently, hashing techniques have been widely applied to solve the approximate nearest neighbors search problem in many vision applications. Generally, these hashing approaches generate 2^c buckets, where c is the length of the hash code. A good hashing method should satisfy the following two requirements: 1) mapping the nearby data points into the same bucket or nearby (measured by the Hamming distance) buckets. 2) all the data points are evenly distributed among all the buckets. In this paper, we propose a novel algorithm named Complementary Projection Hashing (CPH) *to find the optimal hashing functions which explicitly considers the above two requirements. Specifically, CPH aims at sequentially finding a series of hyperplanes (hashing functions) which cross the sparse region of the data. At the same time, the data points are evenly distributed in the hypercubes generated by these hyperplanes. The experiments comparing with the state-of-the-art hashing methods demonstrate the effectiveness of the proposed method.*

1. Introduction

Nearest Neighbors (NN) search is a fundamental problem and has found applications in many computer vision tasks [23, 10, 29]. A number of efficient algorithms, based on pre-built index structures (e.g. KD-tree [4] and R-tree [2]), have been proposed for nearest neighbors search. Unfortunately, these approaches perform worse than a linear scan when the dimensionality of the space is high [5], which is often the case of computer vision applications.

Given the intrinsic difficulty of exact nearest neighbors search, many hashing algorithms are proposed for Approximate Nearest Neighbors (ANN) search [1, 25, 27, 16, 7, 9].

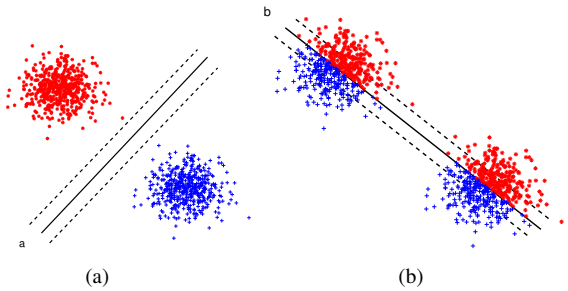


Figure 1. Illustration for the first motivation. (a) The hyperplane *a* crosses the sparse region and the neighbors are quantized into the same bucket; (b) The hyperplane *b* crosses the dense region and the neighbors are quantized into the different buckets. Apparently, the hyperplane *a* is more suitable as a hashing function.

The key idea of these approaches is to represent data points by binary codes which can preserve the pairwise similarities.

Given a data set $\mathbf{X} \in \mathbb{R}^{d\times n}$ containing n d-dimensional points, a hashing algorithm uses c hash functions to generate a c-bit Hamming embedding $\mathbf{Y} \in \mathbb{B}^{c\times n}$. The k-th hash function can be expressed as: $h_k(\mathbf{x}) = sgn(\mathbf{w}_k^T\mathbf{x} - b_k)$[1]. Each hash function can be seen as a hyperplane to split the feature space into two regions. Using c hash functions, a hash index can be built by assigning each point into a c-bit hash bucket corresponding to its c-bit binary code. Given a query point, the hashing approaches use three stages to perform the search: 1) *coding stage*: the query point is compressed into a c-bit binary code using the c hash functions; 2) *locating stage*: all the points in the buckets that fall within a hamming radius r of the hamming code of the query are returned. 3) *linear scan stage*: a linear scan over these points is performed to return the required neighbors.

[1]The corresponding binary hash bit can be simply computed as: $y_k(\mathbf{x}) = (1 + h_k(\mathbf{x}))/2$.

The above procedure shows that a good hashing method should satisfy two requirements: 1) mapping the nearby data points into the same bucket or nearby (measured by the hamming distance) buckets to ensure the accuracy. 2) all the data points are evenly distributed among all the buckets to reduce the linear scan time.

To satisfy the first requirement, the hyperplanes associated with the hash functions should cross the sparse region of the data distribution. In Fig. 1, the hyperplane *a* crosses the sparse region and the neighbors are quantized into the same bucket while the hyperplane *b* crosses the dense region and the neighbors are quantized into the different buckets. Apparently, the hyperplane *a* is more suitable as a hashing function. However, many popular hashing algorithms (e.g., Locality Sensitive Hashing (LSH) [1], Entropy based LSH [22], Multi-Probe LSH [11, 17], Kernelized Locality Sensitive Hashing (KLSH) [13]) are based on the random projection. These methods generate the hash functions randomly and fail to consider this requirement.

In order to satisfy the second requirement, many existing hashing algorithms (e.g., [7, 25, 24]) require that the data points are evenly separated by each hash function (hyperplane). However, this does not guarantee that the data points are evenly distributed among all the hypercubes generated by the hyperplanes (hash functions). Fig. 2 gives an example: Both the hyperplane *a* and the hyperplane *b* partition the data evenly and they are both good one bit hash functions. However, putting them together does not generate a good two bits hash function, as shown in Fig. 2(c). A better choice for two bits hash functions are hyperplanes *c* and *d* in Fig. 2(d).

In this paper, we propose a novel algorithm named *Complementary Projection Hashing (CPH)* to find the optimal hashing functions which explicitly considering the above two requirements. Specifically, CPH aims at sequentially finding a series of hyperplanes (hashing functions) which cross the sparse region of the data. At the same time, the data points are evenly distributed in the hypercubes generated by these hyperplanes. The experiments comparing with the state-of-the-art hashing methods demonstrate the effectiveness of the proposed method.

2. Background and Related Work

The generic hashing problem is as follows: Given n data points $\mathbf{X} = [\mathbf{x}_1, ..., \mathbf{x}_n] \in \mathbb{R}^{d \times n}$, find c hash functions to map a data point $\mathbf{x}$ to a c-bits hash code

$$H(x) = [(1 + h_1(\mathbf{x}))/2, \cdots, (1 + h_c(\mathbf{x}))/2],$$

where $h_k(\mathbf{x}) \in \{-1, 1\}$ is the k-th hash function. For the linear projection-based hashing, we have [24]

$$h_k(\mathbf{x}) = sgn(\mathbf{w}_k^T \mathbf{x} - b_k)$$

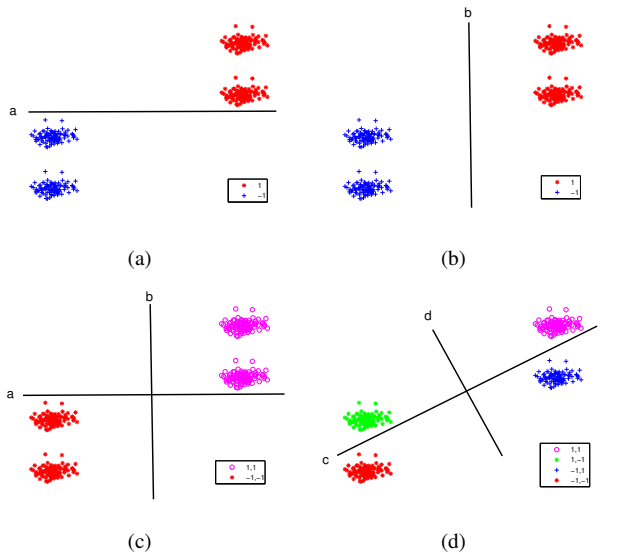


Figure 2. Illustration for the second motivation. (a) (b) Both the hyperplane *a* and the hyperplane *b* can evenly separated the data. (c) However, putting them together does not generate a good two bits hash function. (d) A better example for two bits hash function.

where $\mathbf{w}_k$ is the projection vector and b_k is the threshold. Different hashing algorithms aim at finding different $\mathbf{w}_k$ and b_k with respect to the different objective functions.

One of the most popular hashing algorithms is Locality Sensitive Hashing (LSH) [1]. LSH is fundamentally based on the random projection and uses randomly generated $\mathbf{w}_k$. Empirical studies [1] showed that the LSH is significantly more efficient than the methods based on hierarchical tree decomposition. It has been successfully used in various computer vision applications [26, 25]. There are many extensions for LSH [11, 22, 17, 15]. Entropy based LSH [22] and Multi-Probe LSH [11, 17] are proposed to reduce the space requirement in LSH but need much longer time to deal with the query. Kernelized Locality Sensitive Hashing (KLSH) [13] is introduced in the case of high-dimensional kernelized data when the underlying feature embedding for the kernel is unknown. All these methods are fundamentally based on the random projection and do not aware of the data distribution.

Recently, many learning-based hashing methods [27, 16, 7, 9, 28, 14] are proposed to make use of the data distribution. Many of them [27, 24, 16] exploit the spectral properties of the data affinity (i.e., item-item similarity) matrix for binary coding. The spectral analysis of the data affinity matrix is usually time consuming. To avoid the high computational cost, Weiss et al. [27] made a strong assumption that data is uniformly distributed and proposed a Spectral Hashing method (SH). The assumption in SH leads to a simple analytical eigenfunction solution of 1-D Laplacians, but the geometric structure of the original data is almost ignored, leading to a suboptimal performance. Anchor Graph

Hashing (AGH) [16] is a recently proposed method to overcome this shortcoming. AGH generates k anchor points from the data and represent all the data points by sparse linear combinations of the anchors. In this way, the spectral analysis of the data affinity can be efficiently performed. Some other learning based hashing methods include Iterative Quantization (ITQ) [7] which finds a rotation of zero-centered data so as to minimize the quantization error of mapping this data to the vertices of a zero-centered binary hypercube and Spherical Hashing (SPH) [9] which learns hypersphere-based hash functions. There are also many efforts on leveraging the label information into hash function learning, which leads to supervised hashing [20, 15] and semi-supervised hashing [25, 18].

There are some key points indicate the differences between our method and the previous methods. In [7, 25, 24], the orthogonality constraint of projections has been proposed. For obtaining more balanced buckets, we use a pair-wise hash buckets balance condition to formulate the constraint of hyperplanes. Mu et al. [18] proposed a maximum margin based method. But, we use a soft constraint of minimizing the number of data points which nearby the hyperplanes to find the hyperplanes which cross the sparse region of the data. Liu et al. [15] proposed a supervised hashing method, which used a label matrix involving three different kinds of labels (i.e., similar label, dissimilar label and unknown label). The optimization of the proposed method is motivated by [15], including spectral relaxation and sigmoid smoothing. But, our algorithm is an unsupervised hashing method and does not use this label matrix. Heo et al. [9] proposed a hypersphere based hashing method and mainly focused on the pair-wise hash buckets balance. However, [9] fails to consider the first requirement we described. Our method is a hyperplane based hashing method and explicitly considers the two requirements. Xu et al. [29] also proposed a complementary information based hashing method. But, [29] uses it between hash tables. The complementary information of the proposed method is dependent on the two requirements we described, which is different from [29] and used between projections in one hash table.

3. Complementary Projection Hashing

In this section, we give the detailed description of our proposed *Complementary Projection Hashing* (CPH).

3.1. Crossing The Sparse Region

Given a hyperplane $f(\mathbf{x}) = \mathbf{w}^T\mathbf{x} - b$ crossing the sparse region, the number of data points in the boundary region of this hyperplane should be small. It is easy to check that the distance of a point $\mathbf{x}_i$ to the hyperplane [21] is

$$d_i = \frac{|\mathbf{w}^T\mathbf{x}_i - b|}{\|\mathbf{w}\|}.$$

Without loss of generality, we assume $\|\mathbf{w}\| = 1$. Then $d_i = |\mathbf{w}^T\mathbf{x}_i - b|$.

Given the boundary parameter $\varepsilon > 0$, we can find the hyperplane which cross the sparse region by solving the optimization problem as follows:

$$\min_{\mathbf{w},b} \sum_{i=1}^{n} \mathcal{H}(\varepsilon - |\mathbf{w}^T\mathbf{x}_i - b|)$$

where $\mathcal{H}()$ is the unit step function.

We compute the first hash function (hyperplane) by solving the above optimization problem. If one point is inside the small region around previous learned hyperplanes, it is obvious that this data point should receive a large penalty when learning the new hyperplane. To compute the k-th hash function, the penalty u_i^k for data point $\mathbf{x}_i$ is defined as:

$$u_i^k = 1 + \sum_{j=1}^{k-1} \mathcal{H}(\varepsilon - |\mathbf{w}_j^T\mathbf{x}_i - b_j|). \tag{1}$$

It is easy to check that $u_i^1 = 1$ $(i = 1 \cdots n)$.

The final objective function to learn the k-th bit hashing function can be written as:

$$\min_{\mathbf{w}_k,b_k} \sum_{i=1}^{n} u_i^k \mathcal{H}(\varepsilon - |\mathbf{w}_k^T\mathbf{x}_i - b_k|). \tag{2}$$

By using the accumulative penalty u_i^k, the hashing function for a new bit is complementary to the hashing functions of previous bits.

3.2. Approximating Balanced Buckets

When we learn c-bits hashing functions, we have noticed that all the single bit hashing functions evenly separate the data set do not guarantee balanced buckets (all the data points are evenly distributed among all the 2^c buckets). Thus requiring one bit hashing function to evenly separate the data is not enough. However, learning c hyperplanes which distributes all the data points into 2^c hypercubes is generally NP-hard [8]. We use a *pair-wise hash buckets balance condition* [9] to get a reasonable approximation. The *pair-wise hash buckets balance* requires that every two hyperplanes split the whole space into four regions and each region has $n/4$ data points. This requirement can be nicely formulated as suggested by the following lemma:

Lemma 1. *(pair-wise hash buckets balance condition). Suppose we have two hash functions* $h_1(\mathbf{x}) = sgn(\mathbf{w}_1^T\mathbf{x} - b_1)$ *and* $h_2(\mathbf{x}) = sgn(\mathbf{w}_2^T\mathbf{x} - b_2)$*, if we have:*

$$\begin{cases} \sum_{i=1}^{n} h_1(\mathbf{x}_i) & = 0 \\ \sum_{i=1}^{n} h_2(\mathbf{x}_i) & = 0 \\ \sum_{i=1}^{n} h_1(\mathbf{x}_i)h_2(\mathbf{x}_i) & = 0 \end{cases}$$

Then, $n_{-1,-1} = n_{1,-1} = n_{-1,1} = n_{1,1} = n/4$*, where* $n_{a,b}$ *is the number of points which are satisfied* $h_1(\mathbf{x}) = a$ *and* $h_2(\mathbf{x}) = b$*.*

Proof. According to the conditions, we have:

$$\begin{cases} (n_{-1,-1} + n_{-1,1}) = (n_{1,-1} + n_{1,1}) \ \dots \ (a) \\ (n_{-1,-1} + n_{1,-1}) = (n_{-1,1} + n_{1,1}) \ \dots \ (b) \\ (n_{-1,1} + n_{1,-1}) = (n_{-1,-1} + n_{1,1}) \ \dots \ (c) \end{cases}$$

Now, we have

$$\begin{cases} (a) + (b) \ \Rightarrow \ n_{-1,-1} = n_{1,1} \\ (a) + (c) \ \Rightarrow \ n_{-1,1} = n_{1,1} \\ (b) + (c) \ \Rightarrow \ n_{1,-1} = n_{1,1} \end{cases}$$

and $n_{-1,-1} + n_{1,-1} + n_{-1,1} + n_{1,1} = n$, so we get: $n_{-1,-1} = n_{1,-1} = n_{-1,1} = n_{1,1} = n/4$. □

To learn the k-th bit hashing function, the *pair-wise hash buckets balance condition* can be formulated as:

$$\begin{cases} \sum_{i=1}^{n} sgn(\mathbf{w}_k^T \mathbf{x}_i - b_k) = 0 \\ \sum_{i=1}^{n} sgn(\mathbf{w}_j^T \mathbf{x}_i - b_j) sgn(\mathbf{w}_k^T \mathbf{x}_i - b_k) = 0, j = 1 \cdots k-1 \end{cases}$$

Define n-dimensional vector $\mathbf{v}_k$ as

$$\mathbf{v}_k = [sgn(\mathbf{w}_k^T \mathbf{x}_1 - b_k), \cdots, sgn(\mathbf{w}_k^T \mathbf{x}_n - b_k)]^T \quad (3)$$

and $n \times k$ matrix $\mathbf{V}_{k-1}$ as

$$\mathbf{V}_{k-1} = [\mathbf{1}, \mathbf{v}_1, \cdots, \mathbf{v}_{k-1}], \quad (4)$$

the *pair-wise hash buckets balance condition* has the matrix formulation:

$$\mathbf{V}_{k-1}^T \mathbf{v}_k = \mathbf{0},$$

where $\mathbf{1}$ is an n-dimensional vector of all ones and $\mathbf{0}$ is a k-dimensional vector of all zeros. This suggests that the *pair-wise hash buckets balance condition* has a close connections to the orthogonal constrains of the graph-based hashing methods [27, 16]. $\mathbf{v}_j^T \mathbf{v}_k = 0$ $(j = 1, \cdots, k-1)$ forces two bits to be mutually uncorrelated in order to minimize redundancy among bits [27, 16].

In reality, it is hard to ensure perfect balanced partitions for a real data set. Thus, we replace the above "hard" constraint by a "soft" constraint as follow:

$$\min_{\mathbf{w}_k, b_k} \|\mathbf{V}_{k-1}^T \mathbf{v}_k\|^2 \quad (5)$$

3.3. The Objective Function

Combining the above two requirements, the objective function to learn the k-th bit hashing function can be formulated as:

$$\min_{\mathbf{w}_k, b_k} \sum_{i=1}^{n} u_i^k \mathcal{H}(\varepsilon - |\mathbf{w}_k^T \mathbf{x}_i - b_k|) + \alpha \|\mathbf{V}_{k-1}^T \mathbf{v}_k\|^2 \quad (6)$$

where u_i^k is defined in Eq. (1), $\mathbf{v}_k$ is defined in Eq. (3), $\mathbf{V}_{k-1}$ is defined in Eq. (4), and α is a balancing parameter to find a good trade-off between the two requirements.

In real applications, it is hard to find a set of *linear* hashing functions which achieve a good minimizer of Eq. (6). Motivated by Kernelized Locality Sensitive Hashing (KLSH) [13], we instead try to find a set of *nonlinear* hashing functions using the kernel trick.

For some unknown embedding function $\psi(\cdot)$, we can use a kernel function K to present the dot product of two data points in this unknown embedding [13]:

$$\mathrm{K}(\mathbf{x}_i, \mathbf{x}_j) = \psi(\mathbf{x}_i)^T \psi(\mathbf{x}_j).$$

Suppose we uniformly randomly selected $m (m \ll n)$ samples Θ in $\mathbf{X}$ and denote the k-th projection in kernel space as $\mathbf{z}_k$. According to [13], we can compute the projection:

$$\begin{aligned} \mathbf{z}_k^T \psi(\mathbf{x}_i) &= \sum_{j=1}^{m} \mathbf{p}_k(j) \psi(\mathbf{x}_j)^T \psi(\mathbf{x}_i) \\ &= \sum_{j=1}^{m} \mathbf{p}_k(j) \, \mathrm{K}(\mathbf{x}_j, \mathbf{x}_i) = \mathbf{p}_k^T \, \mathbf{k}(\mathbf{x}_i) \end{aligned}$$

where $\mathbf{p}_k(j)$ denotes j-th element of $\mathbf{p}_k$ which is a coefficient vector we need to learn. $\mathbf{k}(\mathbf{x})$: $\mathbb{R}^d \mapsto \mathbb{R}^m$ is a vectorial map defined by: $\mathbf{k}(\mathbf{x}) = [\mathrm{K}(\mathbf{x}, \Theta_1), \ldots, \mathrm{K}(\mathbf{x}, \Theta_m)]^T$.

Thus, the k-th bit nonlinear function can be written as $f_k(\mathbf{x}) = \mathbf{p}_k^T \, \mathbf{k}(\mathbf{x}) - b_k$ and the objective function of CPH in the kernel space can be written as:

$$\min_{f_k} \sum_{i=1}^{n} \tilde{u}_i^k \mathcal{H}(\varepsilon - |f_k(\mathbf{x}_i)|) + \alpha \|\tilde{\mathbf{V}}_{k-1}^T \tilde{\mathbf{v}}_k\|^2$$

where

$$\begin{aligned} \tilde{u}_i^k &= 1 + \sum_{j=1}^{k-1} \mathcal{H}(\varepsilon - |f_j(\mathbf{x}_i)|), \\ \tilde{\mathbf{v}}_k &= [sgn(f_k(\mathbf{x}_1)), \cdots, sgn(f_k(\mathbf{x}_n))]^T, \\ \tilde{\mathbf{V}}_{k-1} &= [\mathbf{1}, \tilde{\mathbf{v}}_1, \cdots, \tilde{\mathbf{v}}_{k-1}]. \end{aligned} \quad (7)$$

Since $\mathcal{H}(x) = \frac{1}{2}(1 + sgn(x))$ and $\mathcal{H}(\varepsilon - |x|) = \frac{1}{2}(1 + sgn(\varepsilon - |x|)) = \frac{1}{2} + \frac{1}{2} sgn(\varepsilon - x \cdot sgn(x))$, the objective function of CPH in the kernel space can be rewritten as:

$$\min_{f_k} \sum_{i=1}^{n} \tilde{u}_i^k sgn(\varepsilon - f_k(\mathbf{x}_i) sgn(f_k(\mathbf{x}_i))) + \alpha \|\tilde{\mathbf{V}}_{k-1}^T \tilde{\mathbf{v}}_k\|^2 \quad (8)$$

It can be easily checked that directly solving the above optimization problem is NP-hard [16]. Inspired by [15], we use spectral relaxation to compute an initial result and use gradient descent in pursuit of a better result.

3.4. Spectral Relaxation

In this subsection, we discuss how to use spectral relaxation to compute an initial result of $f_k(\mathbf{x}) = \mathbf{p}_k^T \mathbf{k}(\mathbf{x}) - b_k$. To simplify the relaxation, we centralize the kernel matrix and use $b_k = 0$ as an initial threshold. Please refer to [15] for details. Now we have

$$f_k(\mathbf{x}) = \mathbf{p}_k^T \bar{\mathbf{k}}(\mathbf{x}),$$

where $\bar{\mathbf{k}}(\mathbf{x}) = \mathbf{k}(\mathbf{x}) - \frac{1}{n}\sum_{i=1}^{n} \mathbf{k}(\mathbf{x}_i)$ and our goal is computing the coefficient vector $\mathbf{p}_k$.

By dropping the sign function outside of $f_k(\mathbf{x})$, Eq.(8) can be relaxed as:

$$\min_{\mathbf{p}_k} \sum_{i=1}^{n} \tilde{u}_i^k(\varepsilon - \mathbf{p}_k^T \bar{\mathbf{k}}(\mathbf{x}_i)\mathbf{p}_k^T \bar{\mathbf{k}}(\mathbf{x}_i)) + \alpha \tilde{\mathbf{v}}_k^T \tilde{\mathbf{V}}_{k-1} \tilde{\mathbf{V}}_{k-1}^T \tilde{\mathbf{v}}_k$$

which is equivalent to

$$\max_{\mathbf{p}_k} \mathbf{p}_k^T \bar{\mathbf{K}} (diag(\tilde{\mathbf{u}}^k) - \alpha \tilde{\mathbf{V}}_{k-1} \tilde{\mathbf{V}}_{k-1}^T) \bar{\mathbf{K}}^T \mathbf{p}_k \qquad (9)$$

where

$$\bar{\mathbf{K}} = [\bar{\mathbf{k}}(\mathbf{x}_1), \cdots, \bar{\mathbf{k}}(\mathbf{x}_n)]$$
$$\tilde{\mathbf{u}}^k = [\tilde{u}_1^k, \cdots, \tilde{u}_n^k]^T$$

Recall that we have assumed $\|\mathbf{p}_k\|^2 = 1$, the optimization problem (9) can be solved by computing the eigenvector associated with the largest eigenvalue of eigen-problem as follows:

$$\bar{\mathbf{K}} (diag(\tilde{\mathbf{u}}^k) - \alpha \tilde{\mathbf{V}}_{k-1} \tilde{\mathbf{V}}_{k-1}^T) \bar{\mathbf{K}}^T \mathbf{p}_k = \lambda \mathbf{p}_k \qquad (10)$$

3.5. Gradient descent

The eigenvector associated with the largest eigenvalue of eigen-problem (10) provides us an initial solution of $\mathbf{p}_k$ (the initial value for b_k is 0), we then use the gradient descent in pursuit of a better result.

Following [15], we use

$$\varphi(x) = \frac{2}{1 + e^{-x}} - 1$$

to approximate the non-differentiable function $sgn(x)$[2]. Thus, Eq. (8) can be formulated as a smooth surrogate:

$$\min_{\mathbf{p}_k, b_k} J(\mathbf{p}_k, b_k) = \sum_{i=1}^{n} \tilde{u}_i^k \varphi(\varepsilon - (\mathbf{p}_k^T \bar{\mathbf{k}}(\mathbf{x}) - b_k)\varphi(\mathbf{p}_k^T \bar{\mathbf{k}}(\mathbf{x}) - b_k)) + \alpha \|\tilde{\mathbf{V}}_{k-1}^T \varphi(\bar{\mathbf{K}}^T \mathbf{p}_k - \mathbf{1} b_k)\|^2$$

[2]For convenient presentation, we generalize $\varphi()$ to take the element-wise operation for any vector input.

Algorithm 1 Complementary Projection Hashing(CPH)

Input:

n training samples $\mathbf{X} = \{\mathbf{x}_i \in \mathbb{R}^d\}_{i=1}^n$;
m uniformly randomly selected samples ($m \ll n$);
c the number of bits for hashing codes;
α, ε the parameters of CPH;
$\mathrm{K}(\cdot,\cdot)$ the kernel function.

1: Compute the kernel matrix $\mathbf{K}$, then centralize it to obtain $\bar{\mathbf{K}}$.
2: **for** $k = 1, 2, \cdots, c$ **do**
3: Use Eq.(7) to calculate the $\tilde{u}_i^k$ and $\tilde{\mathbf{V}}_{k-1}$.
4: Compute the eigenvector associated with the largest eigenvalue of eigen-problem in Eq.(10) as a initial solution of $\mathbf{p}_k$, $b_k \leftarrow 0$.
5: Use the gradient descent method Eq.(11) to obtain the k-th optimal coefficient vector $\mathbf{p}_k^*$ and optimal threshold b_k^*.
6: **end for**
7: Use c hash functions $\{h_k(\mathbf{x}) = sgn(\mathbf{p}_k^{*T} \mathbf{k}(\mathbf{x}) - b_k^*)\}_{k=1}^c$ to create binary codes of $\mathbf{X}$.

Output:

c hash functions $\{h_k(\mathbf{x}) = sgn(\mathbf{p}_k^{*T} \mathbf{k}(\mathbf{x}) - b_k^*)\}_{k=1}^c$;
Binary codes for the training samples: $\mathbf{Y} \in \{0,1\}^{n \times c}$.

Since $\frac{\partial \varphi(x)}{\partial x} = \frac{1}{2}(1 - \varphi(x)^2)$, by simple algebra, the gradients of J respect to $\mathbf{p}_k$ and b_k are:

$$\frac{\partial J(\mathbf{p}_k, b_k)}{\partial \mathbf{p}_k} = \bar{\mathbf{K}}\mathbf{Q}, \ \frac{\partial J(\mathbf{p}_k, b_k)}{\partial b_k} = (-1) \cdot \mathbf{1}^T \mathbf{Q} \qquad (11)$$

where

$$\mathbf{Q} = \tilde{\mathbf{u}}^k \odot (\mathbf{1} - \mathbf{q}_4 \odot \mathbf{q}_4) \odot (-\mathbf{q}_2 - \mathbf{q}_1 \odot \mathbf{q}_3) + \alpha \tilde{\mathbf{V}}_{k-1} \tilde{\mathbf{V}}_{k-1}^T \mathbf{q}_2 \odot \mathbf{q}_3$$

and

$$\mathbf{q}_1 = \bar{\mathbf{K}}^T \mathbf{p}_k - \mathbf{1} b_k,$$
$$\mathbf{q}_2 = \varphi(\mathbf{q}_1),$$
$$\mathbf{q}_3 = \frac{1}{2}(\mathbf{1} - \mathbf{q}_2 \odot \mathbf{q}_2),$$
$$\mathbf{q}_4 = \varphi(\mathbf{1}\varepsilon - \mathbf{q}_1 \odot \mathbf{q}_2)$$

The symbol $\odot$ represents the Hadamard product(i.e., element-wise product).

In the gradient descent procedure, we enforce $\|\mathbf{p}_k\|^2 = 1$ and apply the Nesterov's gradient method [19] for the fast convergence. The algorithm procedure of CPH is summarized in Algorithm 1.

3.6. Computational Complexity Analysis

Given n data points with the dimensionality d, we select $m (m \ll n)$ samples and train c-bit hash function, the

computational complexity of CPH in the training stage is as follows:

1.$O(nmd)$: Obtain the centralized kernel matrix $\bar{\mathbf{K}}$(Step 1 in Alg. 1).

2.$O(nm)$: Compute complementary informations(Step 3 in Alg. 1).

3.$O(m^3 + m^2n + mn)$: Compute the initial coefficient vector(Step 4 in Alg. 1).

4.$O(nm)$: Compute the optimal coefficient vector and threshold(Step 5 in Alg. 1).

5.$O(nmc)$: Create binary codes of $\mathbf{X}$(Step 7 in Alg. 1).

So, the total computational complexity of training process is: $O(nmd+(nm+m^3+m^2n)c)$. In the testing stage, given a query point, CPH needs $O(dm + mc)$ to compress the query point into a c-bit binary code.

4. Experiments

In this section, we evaluate our CPH algorithm on the high dimensional nearest neighbors search problem. Three large scale real-world data sets are used in our experiments.

- **CIFAR-10**: It consists of 60,000 images and each image is represented by a 3072-dim vector. This data set is publicly available [3] and has been used in [7, 25, 15].

- **GIST-1M**: It contains one million GIST descriptors and each descriptor is represented by a 384-dim vector. This data set is publicly available [4] and has been used in [25, 9, 15].

- **SIFT-1M**: It contains one million SIFT descriptors and each descriptor is represented by a 128-dim vector. This data set has been used in [25, 24, 12] and is provided by those authors.

As suggested in [7], all the data is centralized to produce a better result. For each data set, we randomly select 2k data points as the queries and use the remaining to form the gallery database. Following [9, 12], a returned point is considered to be a true neighbor if it lies in the 1000 closest neighbors (measured by the Euclidian distance in the original space) of the query.

Following [25, 16, 9], we used three criteria to evaluate different aspects of hashing algorithms as follows:

- **Mean Average Precision (MAP)**: This is a classical metric in IR community [6]. MAP approximates the area under precision-recall curve [3] and evaluates the overall performance of a hashing algorithm. This metric has been widely used to evaluate the performance of various hashing algorithms [25, 24, 7, 16, 9, 15].

- **Hash Lookup Precision (HLP)**: Given a query, all the points in the buckets that fall within a small hamming radius r of the hamming code of the query will be retrieved and a linear scan over these points is performed to return the results. If c-bits code is used, the number of buckets one should examine is $\sum_{i=0}^{r} \binom{c}{i}$. Considering the linear scan time, r cannot be very large. Comparing with MAP, it is more meaningful to evaluate the precision with a predefined hamming radius in real scenarios. The HLP is defined as the precision over all the points in the buckets that fall within hamming radius r of the hamming code of the query [24]. Following [25, 24, 16, 15], we fixed $r = 2$ in our evaluation.

- **Recall Curve**: Direct comparison of running time for each algorithm is not practical, since different implementation may result in varied search times of the same method. Given a fix number of corrected neighbors, the search time can be measured through the number of samples one algorithm should examined [8]. This is exactly the recall curve and has been also used widely in [8, 9, 25, 24, 16, 15].

4.1. Compared methods

Seven state-of-the-art hashing algorithms for high dimensional nearest neighbors search are compared as follows:

- **LSH**: Locality Sensitive Hashing [1], which is fundamentally based on the random projection.

- **KLSH**: Kernelized locality sensitive hashing [13], which generalizes the LSH method to the kernel space.

- **ITQ**: Iterative quantization [7], which finds a rotation of zero-centered data so as to minimize the quantization error of mapping this data to the vertices of a zero-centered binary hypercube.

- **SH**: Spectral Hashing [27], which is based on quantizing the values of analytical eigenfunctions computed along PCA directions of the data.

- **AGH**: Anchor Graph Hashing [16], which constructs an anchor graph to speed up the spectral analysis.

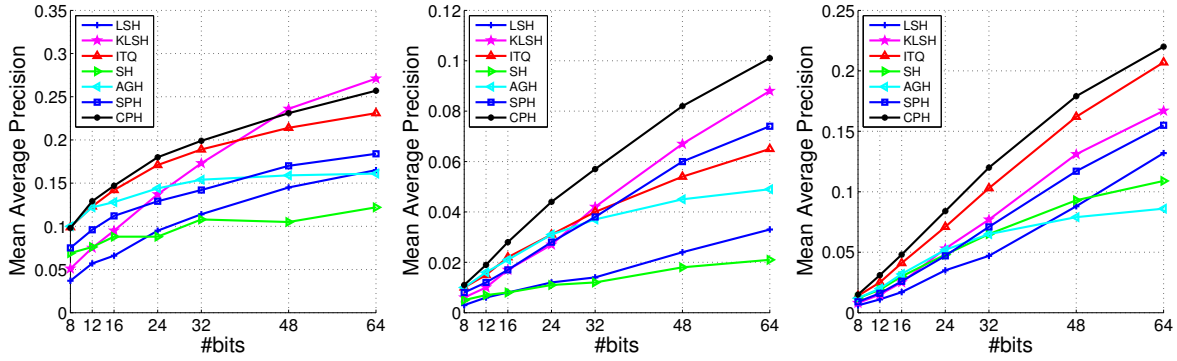
- **SPH**: Spherical hashing [9], which uses a hypersphere-based hash function to map data points into binary codes.

- **CPH**: Complementary Projection Hashing, which is the proposed method in this paper.

All the codes of compared methods are provided by the original authors.

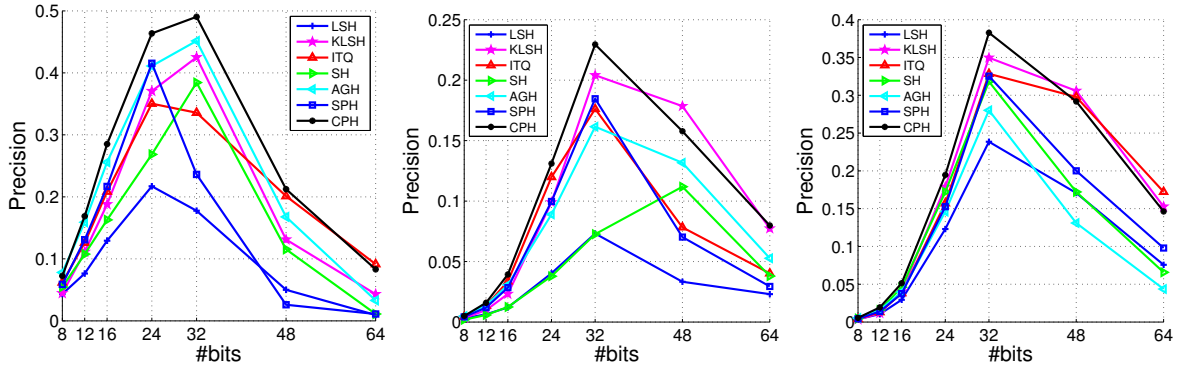
[3] http://www.cs.utoronto.ca/∼kriz/cifar.html

[4] http://horatio.cs.nyu.edu/mit/tiny/data/index.html



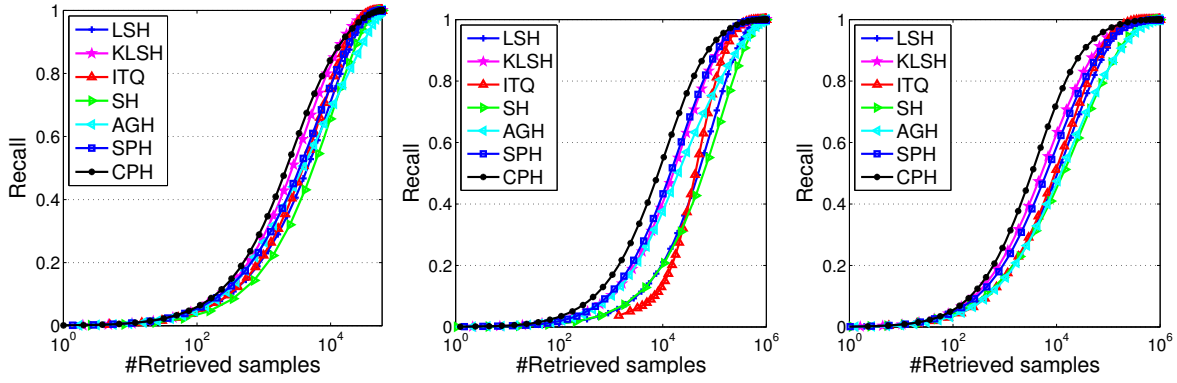
(a) CIFAR-10 (b) GIST-1M (c) SIFT-1M

Figure 3. The mean average precision of all the algorithms on the three data sets.



(a) CIFAR-10 (b) GIST-1M (c) SIFT-1M

Figure 4. The hash lookup Precision @ hamming radius 2 of all the algorithms on the three data sets.



(a) CIFAR-10 (b) GIST-1M (c) SIFT-1M

Figure 5. The recall curves of all the algorithms on the three data sets with 64 bits. Given a fixed recall, the smaller of the number of the retrieved samples, the better of the algorithm.

It is important to note that both LSH and ITQ are *linear* methods while the remaining five methods are *nonlinear* methods. Specifically, KLSH, AGH, and CPH use the kernel trick to learn the nonlinear hashing function. We use the Gaussian kernel $\mathrm{K}(\mathbf{x}, \mathbf{y}) = \exp(-\|\mathbf{x} - \mathbf{y}\|^2/2\sigma^2)$ and the width parameter σ is estimated by randomly choosing 3000 samples and let σ equal to the average of the pair-wise distances. All the three algorithms need to select m supporting samples and we use the same m samples (random selected) for all the three algorithms. m is fixed to be 300 throughout the experiment.

CPH has two essential parameters ε and α. ε controls the size of the boundary region of each hashing function. In the experiment, we randomly choose a hyperplane which can evenly separate the data. Then we compute the average distance s of all the samples to this hyperplane. We then empirically set $\varepsilon = 0.01s$. α is the balancing parameter (used to find a good tradeoff between the two requirements)

and was empirically set as 0.1.

4.2. Results

Fig. 3 shows the MAP curves for all the algorithms on the three data sets. When the code length is short, the random projection based methods (LSH and KLSH) have a low MAP while the learning based methods (ITQ, SH, AGH, SPH and CPH) have a relatively high MAP. As the code length increases, the performance of LSH and KLSH consistently increase because of the theoretical guarantee [1]. By explicitly taking into account the two requirements (crossing the sparse region and balanced buckets) of a good hashing method, our CPH consistently outperforms its competitors almost on all the cases.

Fig. 4 shows the hash lookup precision within hamming radius 2 of all the algorithms. The precisions peak at 32 bits for almost all the methods and decrease sharply as the code length increases. This mainly because many buckets become empty as the code length increase. Our CPH achieves the best performance almost on all the cases.

Fig. 5 shows recall curves of different methods with 64 bits. Given a fixed recall, the smaller of the number of the retrieved samples, the better of the algorithm. Fig. 5 clearly shows the superiority of CPH over other hashing methods.

5. Conclusions

In this paper, we propose a novel hashing algorithm named Complementary Projection Hashing (CPH) to obtain high search accuracy and high search speed simultaneously. By learning complementary bits, CPH learns a series of hashing functions which cross the sparse data region and generate balanced hash buckets. Extensive experiments on three real world data sets have demonstrated the effectiveness of the proposed method.

6. Acknowledgments

This work was supported by the National Basic Research Program of China (973 Program) under Grant 2013CB336500, National Natural Science Foundation of China (Grant Nos: 61222207, 61125106, 91120302) and Shaanxi Key Innovation Team of Science and Technology (Grant No.: 2012KCT-04).

References

[1] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. In *Communications of the ACM*, 2008.

[2] L. Arge, M. Berg, H. Haverkort, and K. Yi. The priority r-tree: a practically efficient and worst-case optimal r-tree. In *SIGMOD*, 2004.

[3] A.Turpin and F.Scholer. User performance versus precision measures for simple search tasks. In *SIGIR*, 2006.

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

[5] K. Beyer, J. Goldstein, R. Ramakrishnan, and U. Shaft. When is "nearest neighbor" meaningful? In *ICDT*, 1999.

[6] C. D.Manning, P. Raghavan, and H. Schutze. *An Introduction to Information Retrieval*. Cambridge University Press, 2008.

[7] Y. Gong and S. Lazebnik. Iterative quantization: A procrustean approach to learning binary codes. In *CVPR*, 2011.

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

[9] J.-P. Heo, Y. Lee, J. He, S. F.Chang, and S. E.Yoon. Spherical hashing. In *CVPR*, 2012.

[10] P. Jain, B. Kulis, and K. Grauman. Fast similarity search for learned metrics. *TPAMI*, 2009.

[11] A. Joly and O. Buisson. A posteriori multi-probe locality sensitive hashing. In *ACM Multimedia*, 2008.

[12] A. Joly and O. Buisson. Random maximum margin hashing. In *CVPR*, 2011.

[13] B. Kulis and K. Grauman. Kernelized locality-sensitive hashing. *TPAMI*, 2012.

[14] Y. Lin, R. Jin, D. Cai, S. Yan, and X. Li. Compressed hashing. In *CVPR*, 2013.

[15] W. Liu, J. Wang, R. Ji, Y. Jiang, and S. F.Chang. Supervised hashing with kernels. In *CVPR*, 2012.

[16] W. Liu, J. Wang, S. Kumar, and S. F.Chang. Hashing with graphs. In *ICML*, 2011.

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

[18] Y. Mu, J. Shen, and S. Yan. Weakly-supervised hashing in kernel space. In *CVPR*, 2010.

[19] Y. Nesterov. Introductory lectures on convex optimization: A basic course. *Kluwer Academic Publishers*, 2003.

[20] M. Norouzi and D. J.Fleet. Minimal loss hashing for compact binary codes. In *ICML*, 2011.

[21] R. O.Duda, P. E.Hart, and D. G.Stock. *Pattern Classification 2nd Edition*. John Wiley & Sons, Inc, 2001.

[22] R. Panigrahy. Entropy based nearest neighbor search in high dimensions. In *SODA*, 2006.

[23] J. S.Beis and D. G.Lowe. Shape indexing using approximate nearest-neighbour search in high-dimensional spaces. In *CVPR*, 1997.

[24] J. Wang, S. Kumar, and S.-F. Chang. Sequential projection learning for hashing with compact codes. In *ICML*, 2010.

[25] J. Wang, S. Kumar, and S. F.Chang. Semi-supervised hashing for large scale search. *TPAMI*, 2012.

[26] X.-J. Wang, L. Zhang, F. Jing, and W.-Y. Ma. Annosearch: Image auto-annotation by search. In *CVPR*, 2006.

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

[28] C. Wu, J. Zhu, D. Cai, C. Chen, and J. Bu. Semi-supervised nonlinear hashing using bootstrap sequential projection learning. *TKDE*, 2013.

[29] H. Xu, J. Wang, Z. Li, G. Zeng, S. Li, and N. Yu. Complementary hashing for approximate nearest neighbor search. In *ICCV*, 2011.