K-means Hashing: an Affinity-Preserving Quantization Method for Learning Binary Compact Codes

Kaiming He Fang Wen Jian Sun
Microsoft Research Asia

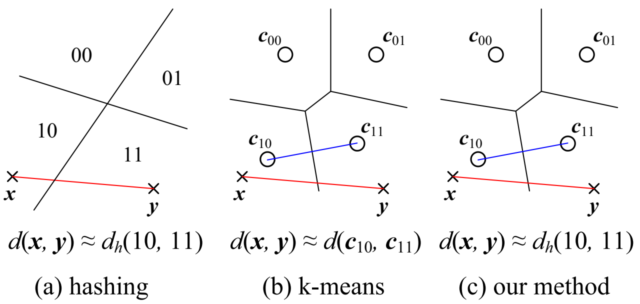
Abstract

In computer vision there has been increasing interest in learning hashing codes whose Hamming distance approximates the data similarity. The hashing functions play roles in both quantizing the vector space and generating similarity-preserving codes. Most existing hashing methods use hyper-planes (or kernelized hyper-planes) to quantize and encode. In this paper, we present a hashing method adopting the k-means quantization. We propose a novel Affinity-Preserving K-means *algorithm which simultaneously performs k-means clustering and learns the binary indices of the quantized cells. The distance between the cells is approximated by the Hamming distance of the cell indices. We further generalize our algorithm to a product space for learning longer codes. Experiments show our method, named as* K-means Hashing *(*KMH*), outperforms various state-of-the-art hashing encoding methods.*

1. Introduction

Approximate nearest neighbors (ANN) search is widely applied in image/video retrieval [27, 11], recognition [28], image classification [23, 2], pose estimation [25], and many other computer vision problems. An issue of particular interest in ANN search is to use compact representations to approximate the data distances (or their ranking orders). Studies on compact encoding are roughly in two streams, categorized by their ways of *distance computation*: *Hamming*-based methods like locality sensitive hashing (LSH) [9, 1, 28] and others [31, 14, 29, 5, 19, 16, 15], and *lookup*-based methods like vector quantization [7] and product quantization [10, 11, 3].

We observe these compact encoding methods commonly involve two stages: (i) *quantization* - the feature space is partitioned into a number of non-overlapping cells with a unique index (code) for each cell; and (ii) *distance computation* based on the indices. Existing *Hamming*-based methods (mostly termed as hashing) quantize the space using hyperplanes [1, 13, 29, 5, 19] or kernelized hyperplanes [31, 14, 15]. One hyperplane encodes one bit of the code and is often determined by the sign of one hashing function. The distance between two samples is approximated by the Hamming distance between their indices (Fig. 1(a)): this computation can be extremely fast in modern CPUs with merely two instructions[1].



(a) hashing (b) k-means (c) our method

Figure 1: Compact encoding methods. A black line denotes a partition boundary, a circle denotes a k-means center, and a cross denotes a sample vector. Here d denotes Euclidean distance, and d_h denotes Hamming-based distance. The indices are denoted in binary forms.

On the other hand, *lookup*-based methods [7, 10, 11, 3] quantize the space into cells via k-means [18]. K-means is a more adaptive quantization method than those using hyperplanes, and is optimal in the sense of minimizing quantization error [7]. The distance between two samples is approximated by the distance between the k-means centers (Fig. 1(b)), which can be read from pre-computed lookup tables given the center indices. The product quantization [7, 10] is a way of applying k-means-based quantization for a larger number of bits (*e.g.*, 64 or 128).

Both Hamming-based methods and lookup-based methods are of growing interest recently, and each category has its benefits depending on the scenarios. The lookup-based methods like [3, 10] has been shown more accurate than some Hamming methods with the same code-length, thanks

[1] The Hamming distance is defined as the number of different bits between two binary codes. Given two codes i and j, it is computed by `__popcnt(`i ^ j`)` in C++, where ^ is bitwise xor and `__popcnt` is the instruction counting non-zero bits. This commend takes about 10^{-9} s.

to the adaptive k-means quantization and the more flexible distance lookup. However, the lookup-based distance computation is slower than the Hamming distance computation[2]. Hamming methods also have the advantage that the distance computation is problem-independent: they involve only an encoding stage but no decoding stage (*i.e.*, index-based lookup). This property is particularly favored, *e.g.*, in mobile product search [8] and built-in-hardware systems.

In this paper, we focus on learning binary compact codes with Hamming distance computation. We propose a novel scheme: we partition the feature space by k-means-based quantization, but approximate the distance by the Hamming distance between the cell indices (Fig. 1(c)). We desire the Hamming distances to preserve the Euclidean distances between the k-means centers. A naive solution would be first to quantize the space using k-means and then assign distance-preserving indices to the cells. But this two-step solution can only achieve suboptimal results, and the assignment problem is not feasible in practice.

To this end, we propose a novel quantization algorithm called *Affinity-Preserving K-means* which simultaneously takes both quantization and distance approximation into account. This algorithm explicitly preserves the similarity between Euclidean and Hamming distances during the k-means clustering stage. This algorithm can be naturally generalized to a product space for learning longer codes. Our method, named as *K-means Hashing* (KMH), enjoys the benefits of adaptive k-means quantization and fast Hamming distance computation.

We notice a method called "k-means locality sensitive hashing" [22] has been proposed as an inverted file system [27]. We point out that the terminology "hashing" in [22] refers to the classical hashing strategy of distributing data into buckets so that similar data would collide in the same bucket. In this paper we follow the terminology in many other recent papers [1, 31, 14, 29, 19, 15] where "hashing" refers to compact binary encoding with Hamming distance computation. A discussion on the two terminologies of "hashing" can be found in [24].

2. Affinity-Preserving K-means

2.1. Basic Model

Our basic model is to quantize the feature space in a k-means fashion and compute the approximate distance via the Hamming distance of the cell indices.

Following the classical *vector quantization* (VQ) [7], we map a d-dimensional vector $\mathbf{x} \in \mathbb{R}^{\mathrm{d}}$ to another vector $q(\mathbf{x}) \in \mathcal{C} = \{\mathbf{c}_i \mid \mathbf{c}_i \in \mathbb{R}^{\mathrm{d}}, 0 \leq i \leq k-1\}$. The set $\mathcal{C}$ is known as a *codebook* [7], $\mathbf{c}_i$ is a *codeword*, and k is the number of codewords. Given b bits for indexing, there are at most $k = 2^b$ codewords. VQ assigns any vector to its nearest codeword in the codebook. Usually the codewords are given by the k-means centers, as they provide minimal quantization error [7].

VQ approximates the distance between any two vectors $\mathbf{x}$ and $\mathbf{y}$ by the distance of their codewords:

$$d(\mathbf{x}, \mathbf{y}) \simeq d(q(\mathbf{x}), q(\mathbf{y})) = d(\mathbf{c}_{i(\mathbf{x})}, \mathbf{c}_{i(\mathbf{y})}), \quad (1)$$

Here we use $d(\mathbf{x}, \mathbf{y}) = \|\mathbf{x} - \mathbf{y}\|$ to denote the Euclidean distance between two vectors , and $i(\mathbf{x})$ denotes the index of the cell containing $\mathbf{x}$. The above notation highlights that the distance computation solely depends on the indices: it can be read from a k-by-k pre-computed lookup table $d(\cdot, \cdot)$.

To get rid of the lookup tables and take advantage of fast Hamming distance computation, we approximate the lookup-based distance using the Hamming distance:

$$d(\mathbf{c}_{i(\mathbf{x})}, \mathbf{c}_{i(\mathbf{y})}) \simeq d_h(i(\mathbf{x}), i(\mathbf{y})) \quad (2)$$

where d_h is defined as a Hamming-based distance between any two indices i and j:

$$d_h(i, j) \triangleq s \cdot h^{\frac{1}{2}}(i, j). \quad (3)$$

Here s is a constant scale, h denotes the Hamming distance, and $h^{\frac{1}{2}}$ is its square root. The square root is essential: it relates our method to orthogonal hashing methods (Sec. 2.4), and it enables to generalize this approximation to longer codes (Sec. 3.1). The usage of s is because the Euclidean distance d can be in arbitrary range, while the Hamming distance h is constrained in $[0, b]$ given b bits. We will discuss how to determine s in Sec. 2.4.

In sum, given a codebook $\mathcal{C}$ we approximate the distance $d(\mathbf{x}, \mathbf{y})$ through $d_h(i(\mathbf{x}), i(\mathbf{y}))$ (see Fig. 1(c)). It is obvious that the *indexing* of the codewords affects the approximate distance.

2.2. A Naive Two-step Method

A naive two-step solution to the above model would be: first quantize via k-means with $k = 2^b$ codewords, and then assign optimal indices to the codewords. We term an *index-assignment* $\mathcal{I} = \{i_0, i_1, ..., i_{k-1}\}$ is a permutation of the integers $\{0, 1, ..., k-1\}$. Given a fixed sequence of codewords $\{\mathbf{c}_i\}$ leaned from k-means, we consider an optimal index-assignment as the one minimizing the error introduced by the Hamming approximation in Eqn.(2):

$$\min_{\mathcal{I}} \sum_{a=0}^{k-1} \sum_{b=0}^{k-1} \left(d(\mathbf{c}_{i_a}, \mathbf{c}_{i_b}) - d_h(i_a, i_b)\right)^2 . \quad (4)$$

This equation minimizes the difference between two k-by-k *affinity matrices* $d(\cdot, \cdot)$ and $d_h(\cdot, \cdot)$.

If we optimize (4) by exhausting all possible assignments $\mathcal{I}$, the problem is combinatorially complex: there are $(2^b)!$

[2] In our experiments, it is 10-20 times slower per distance computation in product quantization than in hashing methods.

possibilities with "!" denoting factorial[3]. Only when $b \leq 3$ bits this problem is feasible. When $b = 4$ it takes over one day for exhausting, and if $b > 4$ it is highly intractable.

More importantly, even with the above exhaustive optimization we find this two-step method does not work well (evidenced in Sec. 4). This is because the k-means quantization would generate an affinity matrix $d(\cdot,\cdot)$ of arbitrary range. Even optimally fitting such a matrix would possibly lead to large error, because the Hamming distance only takes a few discrete values in a limited range.

2.3. Affinity-Preserving K-means

The above discussion indicates a two-step method could only achieve a suboptimal solution. The affinity fitting error as in (4) is a quantity not concerned in the first-step k-means quantization. This motivates us to simultaneously minimize the quantization error and the affinity error.

The classical k-means algorithm [18] minimizes the average *quantization error* E_{quan} of the training samples:

$$E_{\text{quan}} = \frac{1}{n} \sum_{\mathbf{x} \in \mathcal{S}} \|\mathbf{x} - \mathbf{c}_{i(\mathbf{x})}\|^2, \tag{5}$$

where $\mathcal{S}$ is the training set with n samples. In the classical k-means this error is minimized by an Expectation-Maximization (EM) alike algorithm: alternatively assign the sample indices $i(\mathbf{x})$ and update the codewords $\{\mathbf{c}_i\}$.

We also want to minimize the error due to the distance approximation in Eqn.(2). We consider the *affinity error* E_{aff} between all sample pairs:

$$E_{\text{aff}} = \frac{1}{n^2} \sum_{\mathbf{x} \in \mathcal{S}} \sum_{\mathbf{y} \in \mathcal{S}} \left(d(\mathbf{c}_{i(\mathbf{x})}, \mathbf{c}_{i(\mathbf{y})}) - d_h(i(\mathbf{x}), i(\mathbf{y})) \right)^2 .$$

The computation is infeasible because it has n^2 terms. Fortunately, it is easy to show that E_{aff} can be written as:

$$E_{\text{aff}} = \sum_{i=0}^{k-1} \sum_{j=0}^{k-1} w_{ij} \left(d(\mathbf{c}_i, \mathbf{c}_j) - d_h(i,j) \right)^2 , \tag{6}$$

Here $w_{ij} = n_i n_j / n^2$, and n_i and n_j are the number of samples having index i and j. Intuitively, E_{aff} is the weighted difference between two k-by-k affinity matrices $d(\cdot,\cdot)$ and $d_h(\cdot,\cdot)$.

Putting the quantization error and affinity error together, we minimize the following objective function:

$$E = E_{\text{quan}} + \lambda E_{\text{aff}}, \tag{7}$$

where λ is a fixed weight (in this paper we use 10). We minimize this function in an alternating fashion:

Assignment step: fix $\{\mathbf{c}_i\}$ and optimize $i(\mathbf{x})$. This step is the same as the classical k-means algorithm: each sample $\mathbf{x}$ is assigned to its nearest codeword in the codebook $\{\mathbf{c}_i\}$.

Update step: fix $i(\mathbf{x})$ and optimize $\{\mathbf{c}_i\}$. Unlike the classical k-means algorithm, the update of any codeword depends on all the others due to the pairwise affinity $d(\mathbf{c}_i, \mathbf{c}_j)$ in (6). So we sequentially optimize each individual codeword $\mathbf{c}_j$ with other $\{\mathbf{c}_i\}_{i \neq j}$ fixed:

$$\mathbf{c}_j = \arg\min_{\mathbf{c}_j} \Big(\frac{1}{n} \sum_{\mathbf{x}; i(\mathbf{x}) = j} \|\mathbf{x} - \mathbf{c}_j\|^2 + 2\lambda \sum_{i; i \neq j} w_{ij} \left(d(\mathbf{c}_i, \mathbf{c}_j) - d_h(i,j) \right)^2 \Big). \tag{8}$$

This problem can be solved by the quasi-Newton method [26] (we use the Matlab commend `fminunc` for simplicity). We update each codeword once in this step.

Initialization. The above iterative algorithm needs to initialize the indices $i(\mathbf{x})$, codebook $\mathcal{C}$, and scale s in (3). In practice we initialize the indices using the binary codes learned by PCA-hashing [29, 5]. To obtain the corresponding codebook $\mathcal{C}$ and the scale s, we need to build a relation between existing hashing methods and our method. We discuss this problem in Sec. 2.4.

We name the above algorithm as *Affinity-Preserving K-means*. A pseudo-code is in Algorithm 1. Empirically the algorithm converges in 50-200 iterations.

Algorithm 1 Affinity-Preserving K-means.

Input: a training set $\mathcal{S} = \{\mathbf{x}\}$, the bit number b
Output: an *ordered* set $\mathcal{C} = \{\mathbf{c}_i \mid i = 0, 1, ..., 2^b - 1\}$

```
Initialize $i(\mathbf{x})$, $\mathcal{C}$, $s$.
repeat
    Assignment: for $\forall \mathbf{x} \in \mathcal{S}$ update $i(\mathbf{x})$ by $\mathbf{x}$'s nearest codeword's index.
    Update: for $j = 0$ to $2^b - 1$, update $\mathbf{c}_j$ using (8).
until convergence
```

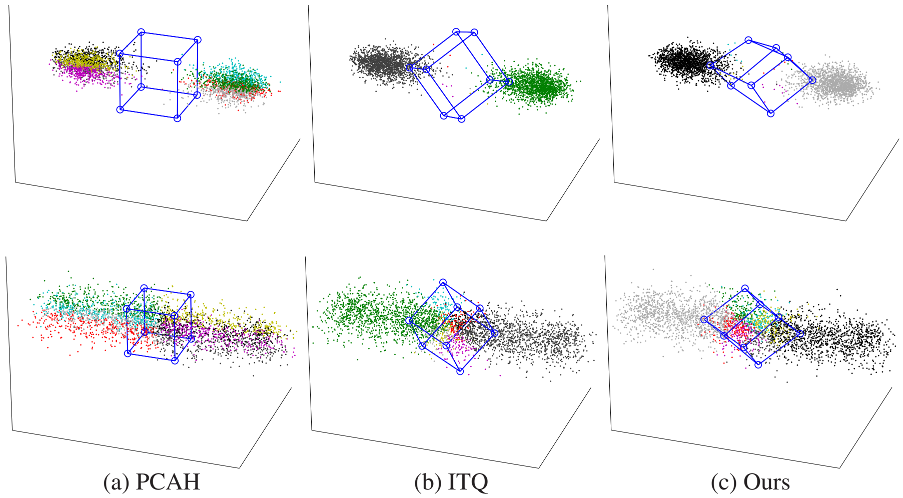
2.4. Relation to Existing Methods

Our method would become classical vector quantization methods or hashing methods if we relax or strengthen some constraints. Actually, our method trades off these two kinds of methods.

Vector Quantization [7].

If we allow to use pre-computed lookup tables $d(\cdot,\cdot)$, we can remove the affinity error term E_{aff} in (7) by setting $\lambda = 0$. As a result, the update step in (8) is simply solved by the sample mean. Thus our method degrades to the classical k-means algorithm.

[3] If only considering distinct affinity matrices, we can prove there are (2^b-1)!/b! possibilities.



(a) PCAH (b) ITQ (c) Ours

	PCAH	ITQ	Ours
E_{quan} ($\times 10^3$)	28.7	8.6	***8.2***
E_{dist} ($\times 10^3$)	29.4	16.2	***16.1***

	PCAH	ITQ	Ours
E_{quan} ($\times 10^3$)	7.72	4.51	***4.31***
E_{dist} ($\times 10^3$)	8.11	7.35	***7.14***

Figure 3: A geometric view of Hamming distance: (a) PCAH, (b) ITQ, (c) ours. A circle denotes a codeword, and a line linking two circles means their Hamming distance is 1. A dot denotes a data point, with different colors indicating different encoded indices. Two synthetic datasets are shown here. Each database consists of 3 randomly selected principal components from the SIFT1M datasets [10]. This figure is best viewed in color version.

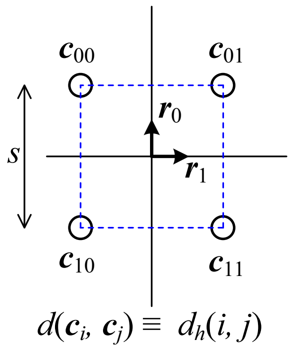


Figure 2: Relation between vector quantization and orthogonal hashing. The vertexes of a hyper-cube are used as codewords. In this 2-d example the hyper-cube is a square. The partition boundaries of the cells are orthogonal hyperplanes (lines in 2-d).

Orthogonal Hashing and Iterative Quantization [5].

If we set $\lambda = \infty$ in (7) so $d(\cdot,\cdot)$ and $d_h(\cdot,\cdot)$ must be identical, minimizing (7) is equivalent to the hashing method called Iterative Quantization (ITQ) [5].

For simplicity we suppose that the data are b-dimensional (*e.g.*, via PCA as in [5]). If the two lookup tables $d(\cdot,\cdot)$ and $d_h(\cdot,\cdot)$ are identical, the $k = 2^b$ codewords must be taken from the vertexes of a b-dimensional hyper-cube. The 2^b-word codebook is given by:

$$\mathcal{C}_{\text{cube}} = \{\mathbf{c} \mid \mathbf{c} \in \mathbb{R}^b, \mathbf{c}{\cdot}\mathbf{r}_t = \pm\frac{1}{2}s, \forall t = 0, 1, ..., b{-}1\}, \quad (9)$$

where s is the side-length of the hyper-cube, and the vectors $\{\mathbf{r}_t\}$ are b-dimensional orthogonal bases (Fig. 2). It is easy to see that the partition boundaries of the resulting cells are orthogonal hyperplanes (*e.g.*, the two orthogonal lines in the 2-d example in Fig. 2). The Euclidean distance between *any* two codewords equals to the Hamming-based distance: $d^2(\mathbf{c}_i, \mathbf{c}_j) = s^2 \cdot h(i,j) = d_h^2(i,j)$. It is easy to see this fact in Fig. 2. The above equivalence was noticed by [5].

Suppose the data have been zero-centered. The only freedom in (9) is a rotational matrix $\mathbf{R}$ whose columns are $\{\mathbf{r}_t\}$. As a result, minimizing the objective function in (7) with $\lambda = \infty$ is equivalent to minimizing the quantization error w.r.t. a rotational matrix $\mathbf{R}$. This is exactly the same cost function as ITQ [5].

This relation suggests a way of determining s in Eqn.(3). We initialize the codebook by $\mathcal{C}_{\text{cube}}$, using the PCA bases as $\{\mathbf{r}_t\}$. Putting (9) into (5), it is easy to show that E_{quan} is a quadratic function on s. We initial s by minimizing E_{quan} with respect to s. We fix s after initialization. Theoretically we can update s after each iteration, but we observe marginal influence in practice.

2.5. A Geometric View

As discussed above, any orthogonal hashing method (*e.g.*, ITQ [5] and PCAH [29, 5]) can be considered as a vector quantization method using the vertexes of a *rotated* hyper-cube as the codewords. Fig. 3 (a) and (b) show the geometric view when $b = 3$ bits.

Our method allows to "*stretch*" the hyper-cube while rotating it, as in Fig. 3 (c). The stretching distortion is controlled by E_{aff}. Although ITQ has the minimal quantization error E_{quan} among orthogonal hashing methods, our method achieves smaller E_{quan} thanks to the stretching, as shown in the tables in Fig. 3. We also evaluate the empirical mean *distance error* E_{dist}:

$$E_{\text{dist}} = \frac{1}{n^2} \sum_{\mathbf{x} \in \mathcal{S}} \sum_{\mathbf{y} \in \mathcal{S}} \left(d(\mathbf{x}, \mathbf{y}) - d_h(i(\mathbf{x}), i(\mathbf{y}))\right)^2. \quad (10)$$

The tables in Fig. 3 show our method has the smallest E_{dist}.

Notice the scale s impacts in measures in the tables in Fig. 3. Here for each dataset we use the same s among all methods, with s computed as in Sec. 2.4 using the PCA bases. We observe similar comparisons if we initialize the s using the bases of ITQ.

Each dataset in Fig. 3 consists of 3 randomly selected principal components in the SIFT1M dataset [10]. Interestingly, we notice that in the first dataset (Fig. 3 top, containing the largest principal component of SIFT) there are roughly two clusters. Though the methods are given 3 bits and at most 2^3=8 clusters, both ITQ and our method divide the data into roughly two clusters (colored in Fig. 3 top). The codewords of these two clusters are on the diagonal of the hyper-cube, having the max possible Hamming distance (=3 here). Although it seems inefficient to use 3 bits to encode two clusters, it is worthwhile if preserving distance is the concern. On the contrary, though PCAH divides the data into 8 more balanced clusters, the Euclidean distance is harder to be preserved by the Hamming distance. The table in Fig. 3 (top) shows it has the largest E_{dist}.

3. Generalization to a Product Space

Like classical k-means, the Affinity-Preserving K-means is not practical if the bit number b is large, because the algorithm needs to compute and store 2^b codewords. The product quantization (PQ) method [10] addresses this issue by separately training k-means in the subspaces of a product space. We show that our algorithm can be naturally generalized to a product space.

3.1. From Product Quantization to Hamming Distance Approximation

The PQ method in [10] decomposes the space $\mathbb{R}^D$ into a Cartesian product of subspaces. Specifically, a vector $\mathbf{x} \in \mathbb{R}^D$ is represented as a concatenation of M subvectors: $\mathbf{x} = [\hat{\mathbf{x}}^1, ...\hat{\mathbf{x}}^m, ..., \hat{\mathbf{x}}^M]$, where the superscript m in $\hat{\mathbf{x}}^m$ denotes the m-th subvector. A sub-codebook $\hat{\mathcal{C}}^m$ with k=2^b sub-codewords is independently trained in the m-th subspace. Any codeword $\mathbf{c}$ in the product space $\mathbb{R}^D$ is a concatenation of M sub-codewords drawn from the M sub-codebooks. In this way, there are essentially $K = k^M = 2^{Mb}$ distinct codewords in $\mathbb{R}^D$, indexed by $B = Mb$ bits. But the algorithm only needs to compute and store $M{\cdot}2^b$ sub-codewords.

As in VQ (*c.f.* Eqn.(1)), PQ approximates the distance between two vectors by the distance between codewords[4]:

$$\begin{aligned} d(\mathbf{x}, \mathbf{y}) &\simeq d(q(\mathbf{x}), q(\mathbf{y})) \\ &= \sqrt{\sum_{m=1}^{M} \left(d(q^m(\hat{\mathbf{x}}^m), q^m(\hat{\mathbf{y}}^m))\right)^2}, \quad (11) \end{aligned}$$

where q^m denotes the quantizer in the m-th subspace. In PQ the distance in (11) is computed through M separate k-by-k lookup tables .

Driven by the same motivation as Eqn.(2) in Sec. 2.1, we approximate the lookup-based distance in Eqn.(11) by Hamming distance:

$$d(q(\mathbf{x}), q(\mathbf{y})) \simeq \sqrt{\sum_{m=1}^{M} \left(d_h(\hat{i}^m(\hat{\mathbf{x}}^m), \hat{i}^m(\hat{\mathbf{y}}^m))\right)^2}, \quad (12)$$

where the notation $\hat{i}^m$ indicates this index is from the m-th sub-codebook $\hat{\mathcal{C}}^m$. Putting Eqn.(2) into (11), we have:

$$\begin{aligned} d(q(\mathbf{x}), q(\mathbf{y})) &\simeq \sqrt{\sum_{m=1}^{M} s^2 \cdot h(\hat{i}^m(\hat{\mathbf{x}}^m), \hat{i}^m(\hat{\mathbf{y}}^m))} \\ &= s \cdot \sqrt{h(i(\mathbf{x}), i(\mathbf{y}))}. \quad (13) \end{aligned}$$

The equality holds if the index $i(\mathbf{x})$ is a concatenation of M sub-indices.

This equation means that we can apply Algorithm 1 independently to each subspace, and use the concatenation of the M sub-indices as the final index $i(\mathbf{x})$. In this case the Hamming distance still approximates the Euclidean distance. If the ranking of the distances is the only concern, the scale s and the square root in (13) are ignorable as they are monotonic operations (*i.e.*, ranking $s{\cdot}\sqrt{h}$ is equivalent to ranking h).

Notice that Eqn.(13) requires the scale s to remain the same across all subspaces. We can initial s via PCAH in the full space (this is well tractable). But in practice we find it performs as good when s is independently initialized for each subspace.

3.2. Decomposing the Product Space

To decompose the space $\mathbb{R}^D$ into a product of M subspaces, we adopt the simple criteria as below.

Following the common criterion in [31, 29, 3, 5] that the bits should be independent, we expect the subspaces in our method to be independent. To this end, we preprocess the

[4] More precisely, Eqn.(11) is the *Symmetric Distance Computation* (SDC) in [10]. SDC is solely index-dependent. [10] further proposes the *Asymmetric Distance Computation* (ADC). ADC depends on the query vector as a real-number vector.

data by PCA projection (without dimension reduction). Following another common criterion in PQ methods [11, 12], we expect the variance of each subspace to be balanced. We define *variance* as the product of the eigenvalues of a subspace. We adopt a simple greedy algorithm to achieve balance: we sort all the principal components in the descending order of their eigenvalues, and sequentially assign each of them to one out of M buckets that has the smallest variance. The principal components in the a bucket will form a subspace.

This decomposing method is called ***Eigenvalue Allocation***, with its theoretical foundation in [4].

3.3. Algorithm Summary

With the above subspaces decomposition, we separately apply Affinity-Preserving K-means to each subspace. This step generates M sub-codebooks, and each sub-codebook has 2^b $\frac{D}{M}$-dimensional codewords.

To encode any vector, we first divide it into M sub-vectors. Each sub-vector will find the nearest sub-codeword in its sub-codebook. These M sub-indices are concatenated into a binary code with $B{=}M{\cdot}b$ bits. *On-line*, the complexity of encoding a query is $O(D^2 + 2^b D) = O(D^2 + 2^{\frac{B}{M}} D)$ where D^2 is due to the PCA projection. In practice we use $b \leq 8$ and the encoding cost is ignorable.

We publish the Matlab codes of our algorithm for the ease of understanding the details[5].

4. Experimental Results

We evaluate the ANN search performances on two public datasets. The first dataset is SIFT1M from [10], containing 1 million 128-d SIFT features [17] and 10,000 independent queries. The second dataset is GIST1M, containing 1 million 384-d GIST features [21] we randomly sample from the 80M dataset [28] with extra 10,000 randomly sampled queries. We considered the groundtruth as each query's K Euclidean nearest neighbors. We set K=10 in the experiments. We will have a discussion on K later.

We follow the search strategy of *Hamming ranking* commonly adopted in many hashing methods [1, 31, 29, 19, 5, 6]. Hamming ranking is to sort the data according to their Hamming distances to the query. The first N samples will be retrieved. Hamming ranking is an exhaustive linear search method, but very fast in practice: it takes about 1.5ms to scan 1 million 64-bit codes in a single-core C++ implementation (Intel Core i7 2.93GHz CPU and 8GB RAM). Hamming ranking can be further sped up by a recent method [20]. We evaluate the recall at the first N Hamming neighbors. The recall is defined as the fraction of retrieved true nearest neighbors to the total number of true nearest neighbors.

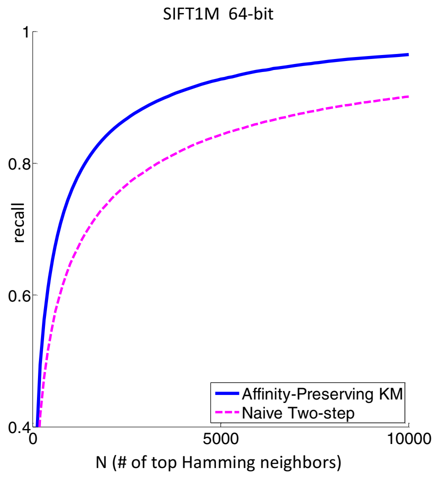


Figure 4: Comparison with the naive two-step method in SIFT1M with 64 bits.

We use cross-validation to choose the parameter b (bit number per subspace) from $\{2, 4, 8\}$. The subspace number M is then given by $\frac{B}{b}$ with the code length B. *Off-line*, the training time of our method when $B = 64$ bits is about 3 minutes in SIFT1M ($b = 4$) and 20 minutes in GIST1M ($b = 8$), using an unoptimized Matlab code. *On-line*, the query encoding time ($<$ 0.01ms) is comparable with other hyperplane-based hashing methods, and is ignorable compared with the Hamming ranking time (1.5ms for one million data).

Comparisons with the naive two-step method

The naive two-step method in Sec. 2.2 can also be applied to the same product space as our method. We have implemented this two-step method in Sec. 2.2 for b=4, and it takes more than one day to train. Fig. 4 shows the comparison between this method and our method, in SIFT1M with 64 bits ($M = 16$ subspaces).

Fig. 4 shows that the naive method is inferior even it exhaustively fits the two affinity matrices. This implies that the two-step method could only achieve sub-optimality. The affinity error can be large when fitting an Hamming affinity matrix to an arbitrary matrix.

Comparisons with state-of-the-arts hashing methods

We compare our k-means hashing (KMH) method with the follow state-of-the-arts unsupervised hashing methods - locality sensitive hashing (LSH) [1], spectral hashing (SH) [31], principal component analysis hashing (PCAH) [29, 5], and iterative quantization (ITQ) [5]. We also compare with a semi-supervised method - minimal loss hashing (MLH) [19], for which we use 10,000 samples to generate the pseudo-labels. All methods have publicly available codes and we use their default settings.

Fig. 5 and Fig. 6 show the comparisons in the two

[5] `http://research.microsoft.com/en-us/um/people/kahe/`

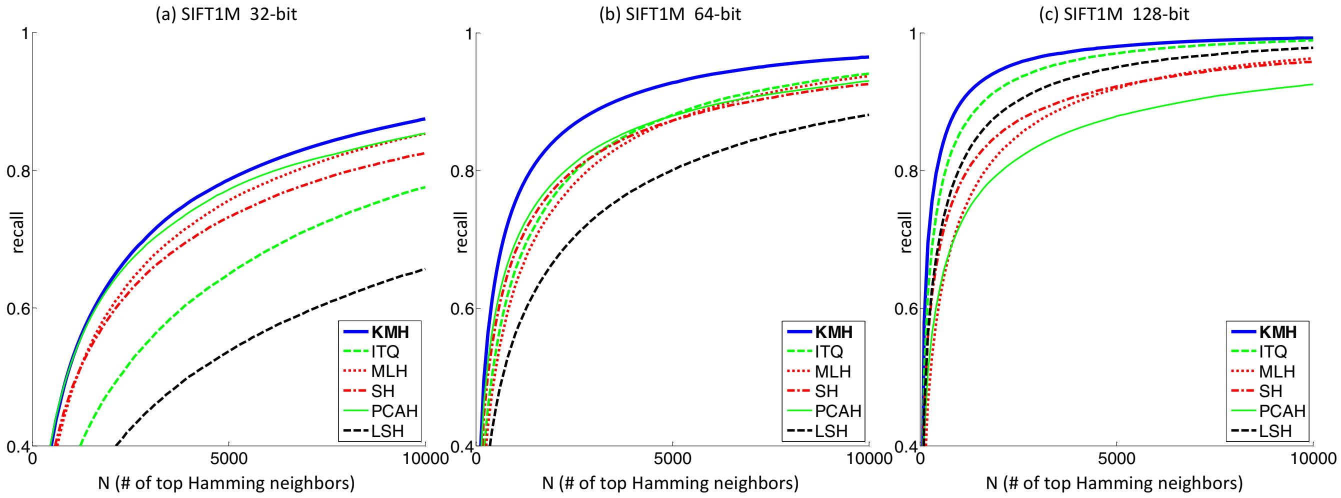


Figure 5: ANN search performance of six hashing methods on SIFT1M using 32, 64, and 128-bit codes. In this figure, K=10 Euclidean nearest neighbors are considered as the ground truth. Our method uses b=2 in the 32-bit case, and b=4 in the 64/128-bit cases.

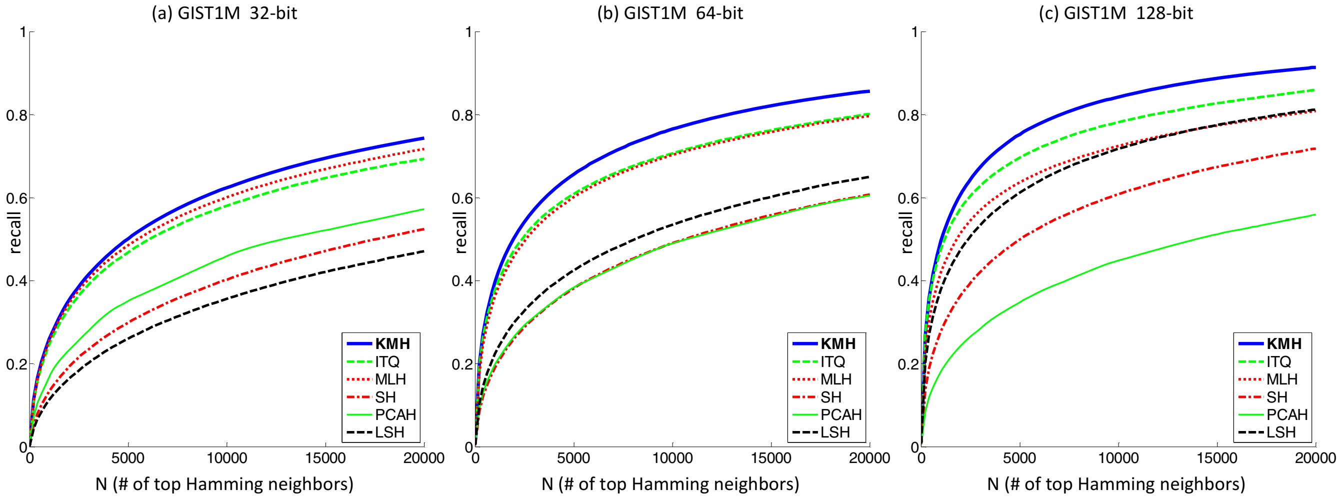


Figure 6: ANN search performance of six hashing methods on GIST1M using 32, 64, and 128-bit codes. In this figure, K=10 Euclidean nearest neighbors are considered as the ground truth. Our method uses b=8 in all cases.

datasets. We have tested B=32, 64, and 128 bits. Our method consistently outperforms all competitors in all bit numbers in both datasets. We also notice that besides our method, there is no method that outperforms the remaining competitors. ITQ is competitive in most settings, typically using 64 and 128 bits. This implies that reducing the quantization error is a reasonable objective. PCAH performs surprisingly well in SIFT1M with 32 bits, but it is inferior in other cases.

Performance under various K

In a recent paper [30] it has been noticed that when evaluating hashing methods, the threshold (like K nearest neighbors in our experiment setting) determining the ground truth nearest neighbors can be of particular impact. In Fig. 7 we show the evaluation in a wide range of different K (1 to 1000). Due to the limited space we only show ITQ and MLH, because we find them superior among the competitors for larger K. In Fig. 7 we see that in a wide range of K our method consistently performs better. This shows the robustness of our method.

5. Discussion and Conclusion

We have proposed a k-means-based binary compact encoding method. Unlike most hashing methods using hyperplanes to quantize, our method enjoys the adaptivity of k-

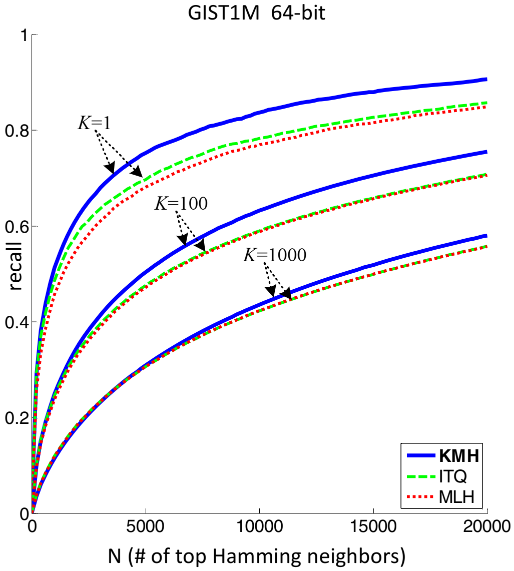


Figure 7: Comparisons in GIST1M with 64 bits under different metric of ground truth nearest neighbors (K=1, 100, 1000). The result of K=10 is in Fig. 6.

means. Our Affinity-Preserving K-means algorithm allows to approximate the Euclidean distance between codewords without lookup tables, so our method also enjoys the advantages of Hamming distance computation. Experiments have shown that our method outperforms many hashing methods.

References

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

[2] O. Boiman, E. Shechtman, and M. Irani. In defense of nearest-neighbor based image classification. In *CVPR*, pages 1–8, 2008.

[3] J. Brandt. Transform coding for fast approximate nearest neighbor search in high dimensions. In *CVPR*, pages 1815–1822, 2010.

[4] T. Ge, K. He, Q. Ke, and J. Sun. Optimized product quantization for approximate nearest neighbor search. In *CVPR*, 2013.

[5] Y. Gong and S. Lazebnik. Iterative quantization: A procrustean approach to learning binary codes. In *CVPR*, pages 817–824, 2011.

[6] K. Grauman and R. Fergus. Learning binary hash codes for large-scale image search. *Machine Learning for Computer Vision*, pages 49–87, 2013.

[7] R. Gray. Vector quantization. *ASSP Magazine, IEEE*, 1(2):4–29, 1984.

[8] J. He, J. Feng, X. Liu, T. Cheng, T.-H. Lin, H. Chung, and S.-F. Chang. Mobile product search with bag of hash bits and boundary reranking. In *CVPR*, pages 3005–3012, 2012.

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

[10] H. Jegou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *TPAMI*, 33, 2011.

[11] H. Jegou, M. Douze, C. Schmid, and P. Perez. Aggregating local descriptors into a compact image representation. In *CVPR*, pages 3304–3311, 2010.

[12] H. Jegou, R. Tavenard, M. Douze, and L. Amsaleg. Searching in one billion vectors: re-rank with source coding. In *ICASSP*, pages 861–864, 2011.

[13] B. Kulis and T. Darrell. Learning to hash with binary reconstructive embeddings. *Advances in neural information processing systems*, 22:1042–1050, 2009.

[14] B. Kulis and K. Grauman. Kernelized locality-sensitive hashing for scalable image search. In *ICCV*, 2009.

[15] W. Liu, J. Wang, R. Ji, Y.-G. 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] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *IJCV*, 60:91–110, 2004.

[18] J. B. MacQueen. Some methods for classification and analysis of multivariate observations. In *Proceedings of 5th Berkeley Symposium on Mathematical Statistics and Probability*, pages 281–297. University of California Press, 1967.

[19] M. E. Norouzi and D. J. Fleet. Minimal loss hashing for compact binary codes. In *ICML*, pages 353–360, 2011.

[20] M. E. Norouzi, A. Punjani, and D. J. Fleet. Fast search in hamming space with multi-index hashing. In *CVPR*, 2012.

[21] A. Oliva and A. Torralba. Modeling the shape of the scene: a holistic representation of the spatial envelope. *IJCV*, 42:145–175, 2001.

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

[23] F. Perronnin and C. Dance. Fisher kernels on visual vocabularies for image categorization. In *CVPR*, 2007.

[24] F. Perronnin and H. Jegou. Large-scale visual recognition. In *CVPR tutorial*, 2012.

[25] G. Shakhnarovich, T. Darrell, and P. Indyk. *Nearest-neighbor methods in learning and vision: theory and practice*. The MIT Press, 2006.

[26] D. Shanno et al. Conditioning of quasi-newton methods for function minimization. *Mathematics of computation*, 24(111):647–656, 1970.

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

[28] A. B. Torralba, R. Fergus, and W. T. Freeman. 80 million tiny images: A large data set for nonparametric object and scene recognition. *TPAMI*, 30:1958–1970, 2008.

[29] J. Wang, S. Kumar, and S.-F. Chang. Semi-supervised hashing for scalable image retrieval. In *CVPR*, 2010.

[30] Y. Weiss, R. Fergus, and A. Torralba. Multidimensional spectral hashing. In *ECCV*, 2012.

[31] Y. Weiss, A. Torralba, and R. Fergus. Spectral hashing. In *NIPS*, pages 1753–1760, 2008.