Pattern Recognition Letters 31 (2010) 1348–1358

Contents lists available at ScienceDirect

Pattern Recognition Letters

journal homepage: www.elsevier.com/locate/patrec

Locality sensitive hashing: A comparison of hash function types and querying mechanisms

Loïc Paulevé [a,*], Hervé Jégou [b], Laurent Amsaleg [c]

[a] ENS Cachan, Antenne de Bretagne, Campus de Ker Lann, Avenue R. Schuman, 35170 Bruz, France
[b] INRIA, Campus de Beaulieu, 35042 Rennes Cedex, France
[c] CNRS/IRISA, Campus de Beaulieu, 35042 Rennes Cedex, France

ARTICLE INFO

Article history:
Received 16 July 2009
Received in revised form 13 January 2010
Available online 10 April 2010
Communicated by W. Pedrycz

Keywords:
Search methods
Image databases
Quantization
Database searching
Information retrieval
LSH

ABSTRACT

It is well known that high-dimensional nearest neighbor retrieval is very expensive. Dramatic performance gains are obtained using approximate search schemes, such as the popular Locality-Sensitive Hashing (LSH). Several extensions have been proposed to address the limitations of this algorithm, in particular, by choosing more appropriate hash functions to better partition the vector space. All the proposed extensions, however, rely on a *structured* quantizer for hashing, poorly fitting real data sets, limiting its performance in practice. In this paper, we compare several families of space hashing functions in a real setup, namely when searching for high-dimension SIFT descriptors. The comparison of random projections, lattice quantizers, *k*-means and hierarchical *k*-means reveal that *unstructured* quantizer significantly improves the accuracy of LSH, as it closely fits the data in the feature space. We then compare two querying mechanisms introduced in the literature with the one originally proposed in LSH, and discuss their respective merits and limitations.

1. Introduction

Nearest neighbor search is inherently expensive due to the *curse of dimensionality* (Böhm et al., 2001; Beyer et al., 1999). This operation is required by many pattern recognition applications. In image retrieval or object recognition, the numerous descriptors of an image have to be matched with those of a descriptor dataset (direct matching) or a codebook (in bag-of-features approaches). Approximate nearest neighbor (ANN) algorithms are an interesting way of dramatically improving the search speed, and are often a necessity. Several *ad hoc* approaches have been proposed for vector quantization (see Gray and Neuhoff, 1998, for references), when finding the exact nearest neighbor is not mandatory as long as the reconstruction error is limited. More specific ANN approaches performing content-based image retrieval using local descriptors have been proposed (Lowe, 2004; Lejsek et al., 2006). Overall, one of the most popular ANN algorithms is the Euclidean Locality-Sensitive Hashing (E2LSH) (Datar et al., 2004; Shakhnarovich et al., 2006). LSH has been successfully used in several multimedia applications (Ke et al., 2004; Shakhnarovich et al., 2006; Matei et al., 2006).

Space hashing functions are the core element of LSH. Several types of hash functions have recently been proposed to improve the performance of E2LSH, including the Leech lattices (Shakhnarovich et al., 2006) and E8 lattices (Jégou et al., 2008a), which offer strong quantization properties, and spherical hash functions (Terasawa and Tanaka, 2007) for unit-norm vectors. These structured partitions of the vector space are regular, i.e., the size of the regions are of equal size, regardless the density of the vectors. Their advantage over the original algorithm is that they exploit the vectorial structure of the Euclidean space, offering a partitioning of the space which is better for the Euclidean metric than the separable partitioning implicitly used in E2LSH.

One of the key problems when using such a regular partitioning is the lack of adaptation to real data. In particular the underlying vector distribution is not taken into account. Besides, simple algorithms such as a *k*-means quantizer have been shown to provide excellent vector search performance in the context of image and video search, in particular when used in the so-called *Video-Google* framework of Sivic and Zisserman (2003), where the descriptors are represented by their quantized indexes. This framework was shown to be equivalent to an approximate nearest neighbor search combined with a voting scheme in Jégou et al., 2008b. However, to our knowledge this *k*-means based partitioning has never been evaluated in the LSH framework, where multiple hash functions reduce the probability that a vector is missed, similar to what is done when using randomized trees (Muja and Lowe, 2009).

The first contribution of this paper is to analyze the individual performance of different types of hash functions on real data. For

* Corresponding author. Fax: +33 240 376 930.
E-mail addresses: loic.pauleve@ens-cachan.org (L. Paulevé), herve.jegou@inria.fr (H. Jégou), laurent.amsaleg@irisa.fr (L. Amsaleg).

0167-8655/$ - see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.patrec.2010.04.004

this purpose, we introduce the performance metrics, as the one usually proposed for LSH, namely the "ε-sensitivity", does not properly reflect objective function used in practice. For the data, we focus on the established SIFT descriptor (Lowe, 2004), which is now the standard for image local description. Typical applications of this descriptor are image retrieval (Lowe, 2004; Nistér and Stewénius, 2006; Jégou et al., 2008b), stitching (Brown and Lowe, 2007) and object classification (Zhang et al., 2007). Our analysis reveal that the k-means quantizer, i.e., an unstructured quantizer learned on a vector training set, behaves significantly better than the hash functions used in the literature, at the cost of an increased pre-processing cost. Note that several approximate versions of k-means have been proposed to improve the efficiency of pre-processing the query. We analyze the performance of one of the popular hierarchical k-means (Nistér and Stewénius, 2006) in this context.

Second, inspired by state-of-the-art methods that have proved to increase the quality of the results returned by the original LSH scheme, we propose two variants of the k-means approach offering different trade-offs in terms of memory usage, efficiency and accuracy. The relevance of these variants are shown to depend on the database size. The first one, multi-probe LSH (Lv et al., 2007), decreases the query pre-processing cost. It is therefore of interest for datasets of limited size. The second variant, query-adaptive LSH (Jégou et al., 2008a), improves the expected quality of the returned vectors at the cost of increased pre-processing. It is of particular interest when the number of vectors is huge. In that case, the query preparation cost is negligible compared to that of post-processing the vectors returned by the indexing structure.

This paper is organized as follows. Section 2 briefly describes the LSH algorithm, the evaluation criteria used to measure the performance of ANN algorithms in terms of efficiency, accuracy and memory usage, and presents the dataset used in all performance experiments. An evaluation of individual hash functions is proposed in Section 3. We finally present the full k-means LSH algorithm in Section 4, and the two variants for the querying mechanism.

2. Background

This section first briefly presents the background material for LSH that is required to understand the remainder of this paper. We then detail the metrics we are using to discuss the performance of all approaches. We finally present the dataset derived from real data that is used to perform the experiments.

2.1. Locality sensitive hashing

Indexing d-dimensional descriptors with the Euclidean version E2LSH of LSH (Shakhnarovich et al., 2006) proceeds as follows. The n vectors of the dataset to index are first projected onto a set of m directions characterized by the d-dimensional vectors $(a_i)_{1 \leqslant i \leqslant m}$ of norm 1. Each direction a_i is randomly drawn from an isotropic random generator.

The n descriptors are projected using m hash functions, one per a_i. The projections of the descriptor x are defined as

$$h_i(x) = \left\lfloor \frac{\langle x | a_i \rangle - b_i}{w} \right\rfloor, \tag{1}$$

where w is the quantization step chosen according to the data (see Shakhnarovich et al., 2006). The offset b_i is uniformly generated in the interval $[0,w)$ and the inner product $\langle x-a_i \rangle$ is the projected value of the vector onto the direction a_i.

The m hash functions define a set $\mathscr{H} = \{h_i\}_{1 \leqslant i \leqslant m}$ of scalar hashing functions. To improve the hashing discriminative power, a second level of hash functions, based on $\mathscr{H}$, is defined. This level is formed by a family of l functions constructed by concatenating several functions from $\mathscr{H}$. Hence, each function g_i of this family is defined as

$$g_j = (h_{j,1}, \ldots, h_{j,d^*}), \tag{2}$$

where the functions $h_{j,i}$ are randomly chosen from $\mathscr{H}$. Note that this hash function can be seen as a quantizer performed on a subspace of dimension d^*. At this point, a vector x is indexed by a set of l vector of integers $g_j(x) = (h_{j,1}(x), \ldots, h_{j,d^*}(x)) \in \mathbb{Z}^k$.

The next step stores the vector identifier within the cell associated with this vector value $g_j(x)$. Note additional steps aiming at avoiding the collision in hash-buckets of distant vectors are performed, but can be ignored for the remainder of this paper, see Shakhnarovich et al. (2006) for details.

At run time, the query vector q is also projected onto each random line, producing a set of l integer vectors $\{g_1(q), \ldots, g_l(q)\}$. From that set, l buckets are determined. The identifiers of all the vectors from the database lying in these buckets make the result short-list. The nearest neighbor of q is found by performing an exact search (L_2) within this short-list, though one might prefer using another criterion to adapt a specific problem, as done in (Casey and Slaney, 2007). For large datasets, this last step is the bottleneck in the algorithm. However, in practice, even the first steps of E2LSH can be costly, depending on the parameter settings, in particular on l.

2.2. Performance metrics

LSH and its derivatives (Gionis et al., 1999; Terasawa and Tanaka, 2007; Andoni and Indyk, 2006) are usually evaluated using the so-called "ε-sensitivity", which gives a intrinsic theoretical performance of the hash functions. However, for real data and applications, this measure does not reflect the objective which is of interest in practice: how costly will be the query, and how likely will the true nearest neighbor of the query point be in the result? Hereafter, we address these practical concerns through the use of the following metrics:

2.2.1. Accuracy: recall

Measuring the quality of the results returned by ANN approaches is central. In this paper, we will be measuring the impact of various hashing policies and querying mechanisms on the accuracy of the result.

To have a simple, reproducible and objective baseline, we solely measure the accuracy of the result by checking whether the nearest neighbor of each query point is in the short-list or not. This measure then corresponds to the probability that the nearest neighbor is found if an exhaustive distance calculation is performed on the elements of the short-list. From an application point of view, it corresponds for instance to the case where we want to assign a SIFT descriptor to a visual word, as done in (Sivic and Zisserman, 2003).

The recall of the nearest neighbor retrieval process is measured, on average, by aggregating this observation (0 or 1) over a large number of queries. Note that instead, we could have performed a k-nn retrieval for each query. Sticking to the nearest neighbor avoids choosing an arbitrary value for k.

2.2.2. Search complexity

Finding out how costly an ANN approach is has major practical consequences for real world applications. Therefore, it is key to measure the overall complexity of the retrieval process in terms of resource consumption. The LSH algorithm, regardless of the hashing options, has two major phases, each having a different cost:

- *Phase 1: Query preparation cost.* With LSH, some processing of the query vector q must take place before the search can probe the index. The query descriptor must be hashed into a series of l values. These values identify the hash-buckets that the algorithm will subsequently analyze in detail. Depending on the hash function type, the cost of hashing operations may vary significantly. It is a function of the number of inner products and/or comparisons with database vectors to perform in order to eventually get the l values. That *query preparation cost* is denoted by *qpc* and is given in Section 3 for several hash function types.
- *Phase 2: Short-list processing cost: selectivity.* Depending on the density of the space nearby q, the number of vectors found in the l hash-buckets may vary significantly. For a given q, we can observe the *selectivity* of the query, denoted by *sel*.

Definition. The selectivity *sel* is the fraction of the data collection that is returned in the short-list, on average, by the algorithm.

In other words, multiplying the selectivity by the number of indexed vectors gives the expected number of elements returned as potential nearest neighbors by the algorithm. The number of memory cells to read and the cost of processing the short-list are both a linear function of the short-list length, hence of *sel*.

In the standard LSH algorithm, it is possible to estimate this selectivity from the probability mass function of hash values, as discussed later in the Section 3.5.

If exhaustive distance calculation is performed on the short-list returned by LSH, the overall cost for retrieving the ANN of a query vector is expressed as

$$ocost = sel \times n \times d + qpc. \tag{3}$$

An interesting measure is the acceleration factor *ac* over exhaustive search, which is given by

$$ac = \frac{n \times d}{ocost} = \frac{1}{sel + \frac{qpc}{n \times d}}. \tag{4}$$

For very large vector collections, the selectivity term is likely to dominate the query preparation cost in this equation, as hash-buckets tends to contain many vectors. This is the rationale for using the selectivity as the main measurement.

2.2.3. Memory usage

The complexity of the search also includes usage of the main memory. In this paper, we assume that the complete LSH data structure fits in main memory. Depending on the strategy for hashing the vectors, more or less main memory is needed. As the memory occupation has a direct impact on the scalability of search systems, it is worth noticing that in LSH, this memory usage is proportional to the number of hash functions considered and to the number of database vectors:

$$\text{memory usage} = \mathcal{O}(l \times n). \tag{5}$$

The number of hash functions used in LSH will hence be used as the main measurement of the memory usage.

2.3. Dataset

Our vector dataset is extracted from the publicly available INRIA Holidays dataset,[1] which is composed of high-definition real Holiday photos. There are many series of images with a large variety of scene types (natural, man-made, water and fire effects, etc). Each series contains somehow visually similar images, differing however due to various rotations, viewpoint and illumination changes of the same scenes taken by the photographers.

[1] http://lear.inrialpes.fr/people/jegou/data.php.

These images have been described using the SIFT descriptor (Lowe, 2004), for which retrieving the nearest neighbors is a very computationally-demanding task. SIFT descriptors have been obtained on these images using the affine co-variant features extractor from Mikolajczyk and Schmid (2004). When using the standard parameters of the literature, the dimensionality of SIFT descriptors is $d = 128$.

The descriptor collection used in this paper is a subsample of 1 million descriptors randomly picked from the descriptors of the image dataset. We also randomly picked 10,000 descriptors used as queries, and another set of 1 million vectors extracted from a distinct image dataset (downloaded from Flickr) for the methods requiring a learning stage. Finally, we ran exact searches using the Euclidean distance to get the true nearest neighbor of each of these query descriptors. This ground-truth will be the one against which ANN searches will be compared.

3. Hash function evaluation

This section first discusses the key design principles behind four types of hash functions and the key parameters that matter for evaluating their performance in the context of LSH. We first recall the original design, where hash functions are based on random projections. Following recent literature (Andoni and Indyk, 2006; Jégou et al., 2008a), we then describe high-dimensional lattices used for spatial hashing. These two types of hash functions belong to the family of *structured* quantizers, and therefore do not capture the peculiarities of the data collection's distribution in space. To contrast with these approaches, we then discuss the salient features of a k-means *unstructured* quantizer for hashing, as well as one if its popular tree-based variant. Overall, choosing a hash function in LSH amounts to modifying the definition of g introduced in Section 2.1.

Enunciated these design methods, we then move to evaluate how each type of hash function performs on the real data collection introduced above. The performance is evaluated for a single hash function. This accurately reflects the intrinsic properties of each hash function, while avoiding to introduce the parameter l (number of distinct hash functions).

3.1. Random projection based

The foundations of the hash functions used in the original E2LSH approach have been presented in Section 2.1. Overall, the eventual quantization of the data space is the result of a product of unbounded scalar quantizers.

The key parameters influencing the performance of each E2LSH hash function are:

- the quantization step w;
- the number d^* of components used in the second-level hash functions g_j.

As the parameters b_i and m are provided to improve the diversity between different hash functions, they are arbitrarily fixed, as we only evaluate the performance of a single hash function. The values chosen for these parameters do not noticeably impact the selectivity, though large values of m linearly impacts the query preparation cost. This one remains low for this structured hashing method.

3.2. Hashing with lattices

Lattices have been extensively studied in mathematics and physics. They were also shown to be of high interest in quantization (Gray and Neuhoff, 1998; Conway and Sloane, 1982b). For a uniform distribution, they give better performance

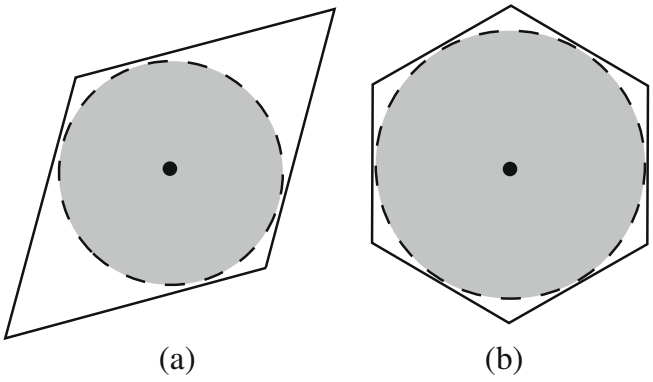


Fig. 1. Fundamental regions obtained using random projections (a) and the lattice A_2 (b). The disparity of distances between the furthest possible points in each region dramatically reduces with dense lattices. This illustrates the vectorial gain.

than scalar quantizers (Gray and Neuhoff, 1998). Moreover, finding the nearest lattice point of a vector can be performed with an algebraic method (Agrell et al., 2002). This is referred to as *decoding*, due to its application in compression.

A lattice is a discrete subset of $\mathbb{R}^{d'}$ defined by a set of vectors of the form

$$\{x = u_1 a_1 + \cdots + u_d a_d | u_1, \ldots, u_d \in \mathbb{Z}\}, \tag{6}$$

where $a_1, \ldots, a_d$ are linearly independent vectors of $\mathbb{R}^{d'}$, $d' \geqslant d$. Hence, denoting by $A = [a_1 \cdots a_d]$ the matrix whose columns are the vectors a_j, the lattice is the set of vectors spanned by $A\,u$ when $u \in \mathbb{Z}^d$. With this notation, a point of a lattice is uniquely identified by the integer vector u.

Lattices offer a regular infinite structure. The Voronoi region around each lattice points has identical shape and volume (denoted by $\mathscr{V}$) and is called the *fundamental region*.

By using lattice-based hashing, we aim at exploiting their spatial consistency: any two points decoded to the same lattice point are separated by a bounded distance, which depends only on the lattice definition. Moreover, the maximum distance between points inside a single lattice cell tends to be identical for some particular lattices. In the rest of this paper, we refer to this phenomenon as the *vectorial gain*.

Vectorial gain is strongly related to the *density* of lattices. The density of a lattice is the ratio between the volume $\mathscr{V}$ of the fundamental region and the volume of its inscribed sphere. Basically, considering Euclidean lattices, the closer to 1 the density, the closer to a sphere the fundamental region, and the greater the vectorial gain. Fig. 1 illustrates the vectorial gain for two 2-d lattices having fundamental region of identical volume. In other terms, if $L_2(x,y)$ is the Euclidean distance between x and y, and $\mathscr{V}_a$ (respectively $\mathscr{V}_b$) is the closed domain of vectors belonging to the region depicted on Fig. 1(a) (respectively Fig. 1(b)), then:

$$\max_{x_a \in \mathscr{V}_a, y_a \in \mathscr{V}_a} L_2(x_a, y_a) \gg \max_{x_b \in \mathscr{V}_b, y_b \in \mathscr{V}_b} L_2(x_b, y_b), \tag{7}$$

where $\int_{x_a \in \mathscr{V}_a} dx_a = \int_{x_b \in \mathscr{V}_b} dx_b$ (i.e., for identical volumes).

In this paper, we will focus on some particular lattices for which fast decoding algorithms are known. These algorithms take advantage of the simplicity of the lattice definition. We briefly introduce the lattices D_d, D_d^+ and A_d. More details can be found in Conway et al. (1987, chap. 4).

- **Lattice** D_d is the subset of vectors of $\mathbb{Z}^d$ having an even sum of the components:

$$D_d = \left\{ (x_1, \ldots, x_d) \in \mathbb{Z}^d : \sum_{i=1}^{d} x_i \text{ even} \right\}, \quad d \geqslant 3. \tag{8}$$

- **Lattice** D_d^+ is the union of the lattice D_d with the lattice D_d translated by adding $\frac{1}{2}$ to each coordinate of lattice points. That translation is denoted by $\frac{\mathbf{1}}{\mathbf{2}} + D_d$.

$$D_d^+ = D_d \cup \left(\frac{\mathbf{1}}{\mathbf{2}} + D_d \right), \tag{9}$$

  when $d = 8$, this lattice is also known as E_8, which offers the best quantization performance for uniform 8-dimensional vectors.
- **Lattice** A_d is the subset of vectors of $\mathbb{Z}^{d+1}$ living on the d-dimensional hyper-plane where the sum of the components is null:

$$A_d = \left\{ (x_0, x_1, \ldots, x_d) \in \mathbb{Z}^{d+1} : \sum_{i=0}^{d} x_i = 0 \right\}. \tag{10}$$

A vector q belonging to $\mathbb{R}^d$ can be mapped to its $d+1$-dimensional coordinates by multiplying it on the right by the n lines $\times$ $n+1$ columns matrix:

$$\begin{pmatrix} -1 & 1 & 0 & \cdots & 0 & 0 \\ 0 & -1 & 1 & \cdots & 0 & 0 \\ . & . & . & \ldots & . & . \\ 0 & 0 & 0 & \cdots & -1 & 1 \end{pmatrix}. \tag{11}$$

For these lattices, finding the nearest lattice point of a given query vector is done in a number of steps that is linear with its dimension (Conway and Sloane, 1982a).

The main parameters of a lattice hash function are:

- the scale parameter w, which is similar to the quantization step for random projections;
- the number d^* of components used.

Hashing the data collection using a lattice asks first to randomly pick d^* components among the original d dimensions—the natural axes are preserved. Then, given w, the appropriate lattice point is assigned to each database vector. The index therefore groups all vectors with the same lattice point identifier into a single bucket.

Remark. The Leech lattice used in (Shakhnarovich et al., 2006) has not been considered here for two reasons. First, it is defined for $d^* = 24$ only, failing to provide any flexibility when optimizing the choice of d^* for performance. Second, its decoding requires significantly more operations compared to the others lattices: 3595 operations per lattice point (Vardy and Be'ery, 1993)[2].

3.3. k-means vector quantizer

Up to now, we have only considered structured quantizers which do not take into account the underlying statistics of the data, except by the choice of the parameters w and d^*. To address this problem, we propose to use an unstructured quantizer learned on a representative set of the vectors to index. Formally, an unstructured quantizer g is defined as a function

$$\begin{aligned} &\mathbb{R} \to [1, \ldots, k], \\ &x \to g(x) = \arg\min_{i=1..k} L_2(x, c(i)) \end{aligned} \tag{12}$$

mapping an input vector x to a cell index $g(x)$. The integer k is the number of possible values of $g(x)$. The vectors $c(i)$, $1 \leqslant i \leqslant k$ are called *centroids* and suffice to define the quantizer.

To construct a good unstructured quantizer, a nice choice is to use the popular k-means clustering algorithm. In that case, k

[2] Note, however, that this number is small compared to what is needed for unstructured quantizers.

corresponds to the number of clusters. This algorithm minimizes[3] the overall distortion of reconstructing a given vector of the learning set using its nearest centroid from the codebook, hence exploiting the underlying distribution of the vectors. Doing so, the potential of vector quantization is fully beneficial since it is able to exploit the vectorial gain. Note that, by contrast to the structured quantizers, there is no random selection of the vector components. Hence, the hashing dimension d^* is equal to the vector dimension d, as the quantizer is learned directly on the vector space.

However, learning a k-means quantizer may take a long time when k is large. In practice, bounding the number of iterations improves the learning stage without significantly impacting the results. In the following, we have set the maximum number of iterations to 20 for SIFT descriptors, as higher values provide comparable results.

3.4. Hierarchical k-means

Approximate variants of the k-means quantizer and the corresponding centroid assignment have been proposed (Nistér and Stewénius, 2006; Philbin et al., 2007) to reduce both the learning stage and the query preparation costs. We evaluate the hierarchical k-means (HKM) of Nistér and Stewénius (2006), which is one of the most popular approach.

The method consists of computing a k-means with k relatively small, and to recursively computes a k-means for the internal nodes until obtaining a pre-defined tree height. This produces a balanced tree structure, where each internal node is connected to a fixed number of centroids. The search is performed top-down by recursively finding the nearest centroid until a leaf is reached. The method uses two parameters:

- the height h_t of the tree;
- the branching factor b_f.

The total number of centroids (leaves) is then obtained as $(b_f)^{h_t}$.

Remark. The method used in (Philbin et al., 2007) relies on randomized trees. This method was improved in (Muja and Lowe, 2009) by automatic tuning of the parameters. The method was shown to outperform HKM, leading to results comparable to that of standard k-means. Therefore, the results we give here for the k-means give a good approximation of the selectivity/recall trade-off that the package of Muja and Lowe (2009) would provide, with a lower query preparation cost, however.

3.5. Experiments and discussion

Fig. 2 gives the evaluation of the different types of hash function introduced in this section. For both random projection and lattices, the two parameters w and d^* are optimized. Fig. 2 only presents the optimal ones, which are obtained as follows. Given a couple of parameters w and d^*, we compute the nearest neighbor recall at a given selectivity. This process is repeated for a set of varying couples of parameters, resulting in a set of tuples associating a selectivity to a nearest neighbor recall. Points plotted on the curves belong to the roof of the convex hull of these numbers. Therefore, a point on the figure corresponds to an optimal parameter setting, the one that gives the best performance obtained for a given selectivity.

For the k-means hash function, only one parameter has to be fixed: the number of centroids k, which gives the trade-off between recall and selectivity. This simpler parametrization is an advantage

[3] This minimization only ensures to find a local minimum.

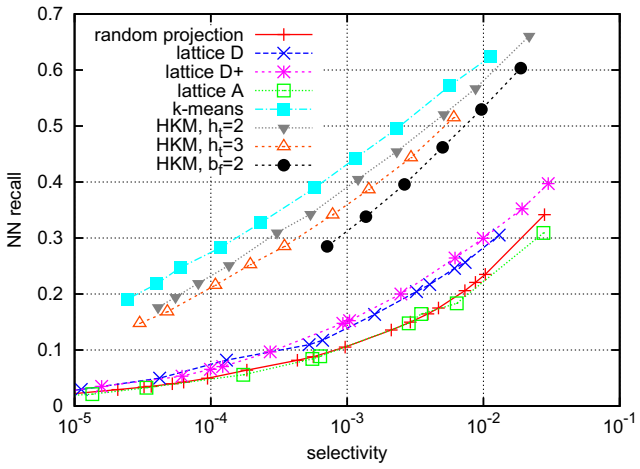


Fig. 2. Evaluation of the different types of hash functions on the SIFT dataset.

in practice. HKM is parametrized by two quantities: the branching factor b_f and the height h_t of the k-means tree. We evaluate the extremal cases, i.e.,

- a fixed height ($h_t = 2,3$) with a varying branching factor
- and a binary tree ($b_f = 2$) with a varying tree height.

3.5.1. Vectorial gain

Fig. 2 clearly shows that the lattice quantizers provide significantly better results than random projections, due to the vectorial gain. These results confirm that the random projections used in E2LSH are unable to exploit the spatial consistency. Note that this phenomenon was underlined in (Andoni and Indyk, 2006; Jégou et al., 2008a). However, by contrast to these works, the lattices we are evaluating are more flexible, as they are defined for any value of d^*. In particular, the lattice E_8 used in (Jégou et al., 2008a) is a special case of the D^+ lattice.

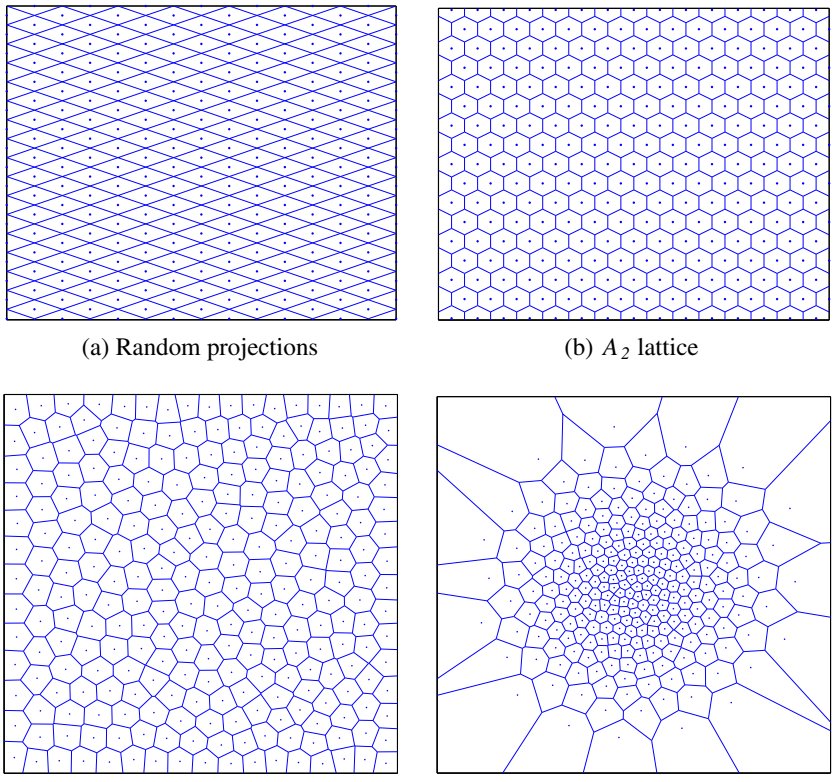
Fig. 2 also shows that the various types of lattice perform differently. We observe an improvement of the nearest neighbor recall with lattices D and D^+ compared to random projections whereas lattice A gives similar performance. The density of D^+ is known to be twice the density of D. In high dimensions, the density of A is small compared to that of D. Overall, density clearly affects the performance of lattices. However, density is not the only crucial parameter. The shape of the fundamental region and its orientation may also be influential, depending on the distribution of the dataset.

Before discussing the performance of the unstructured quantizers evaluated in this paper and shown on Fig. 2, it is necessary to put some emphasis on the behavior of quantization mechanisms with respect to the distribution of data and the resulting cardinality in Voronoi cells.

3.5.2. Structured vs unstructured quantizers

Hashing with lattices intrinsically defines Voronoi cells that all have the same size, that of the fundamental region. This is not relevant for many types of high-dimensional data, as some regions of the space are quite populated, while most are void. This is illustrated by Fig. 3, which shows how well the k-means is able to fit the data distribution. Fig. 3 depicts the Voronoi diagram associated with the different hash functions introduced in this section, and consider two standard distributions. The dimensions $d = d^* = 2$ are chosen for the sake of presentation.

As mentioned above, by construction the structured quantizers (see Fig. 3(a) and (b)) introduced above lead to Voronoi cells of equal sizes. This property is not desirable in the LSH context,



(a) Random projections (b) A_2 lattice

(c) k-means Uniform distribution (d) k-means Gaussian distribution

Fig. 3. Voronoi regions associated with random projections (a) lattice A_2 (b) and a k-means quantizer (c,d).

because the number of retrieved points is too high in dense regions and too small in regions yielding small vector density.

Considering the k-means quantizer in Fig. 3(c), we first observe that for a uniform distribution, the shape of the cells is close to the one of the A_2 lattice, which is optimal for this distribution. But k-means is better for other distributions, as the variable volume of the cells adapts to the data distribution, as illustrated for a Gaussian distribution in Fig. 3(d). The cell size clearly depends on the vector density. Another observation is that k-means exploits the prior on the bounds of the data, which is not the case of the A_2 lattice, whose optimality is satisfied only in the unrealistic setup of unbounded uniform vectors.

As a result, for structured quantizers, the cell population is very unbalanced, as shown by Fig. 4. This phenomenon penalizes the selectivity of the LSH algorithm. In contrast to these quantizers, the k-means hash function exploits both the vectorial gain and the empirical probability density function provided by the learning set. Because the Voronoi cells are quite balanced, the variance of the number of vectors returned for a query is small compared to that of structured quantizers.

Turning back to Fig. 2, one can clearly observe the better performance of the k-means hash function design in terms of the trade-off between recall and selectivity. For the sake of fairness, the codebook (i.e., the centroids) has been learned on a distinct set: k-means being an unsupervised learning algorithm, learning the quantizer on the set of data would overestimate the quality of the algorithm for a new set of vectors. The improvement obtained by using this hash function construction method is very significant: the selectivity is about two order of magnitude smaller for the same recall.

Although HKM is also learned to fit the data, it is inferior to k-means, due to poorer quantization quality. The lower the branching factor is, the closer the results are compared to those of k-means. The two extremal cases depicts in Fig. 2, i.e., (1) a fixed tree height of 2 with varying branching factor and (2) the binary tree ($b_f = 2$) delimits the regions in which all other settings lie. As expected, the performance of HKM in terms of selectivity/recall is the inverse of the one in terms of the query preparation cost. Therefore, considering Eq. (3), the trade-off between b_f and h_t appears to be a function of the vector dataset size.

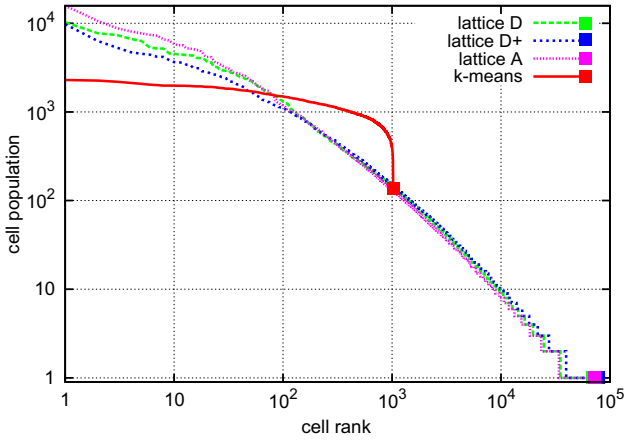


Fig. 4. Population of cells by decreasing order for different lattices and k-means.

3.5.3. Query preparation cost

Table 1 shows the complexity of the query preparation cost qpc associated with the different hash functions we have introduced.

Table 1
Query preparation cost associated with the different hash functions.

Hash function	Query preparation cost
Random projection (E2LSH)	$m \times d + d^* \times l$
Lattice D_{d^*}	$d^* \times l$
Lattice $D^+_{d^*}$	$d^* \times l$
Lattice A_{d^*}	$d^* \times l$
k-means	$k \times d \times l$
HKM	$b_f \times h_t \times l$

Note that this table reflects the typical complexity in terms of the number of operations. It could clearly be refined by considering the respective costs of these knowing the architecture on which the hashing is performed.

Lattices are the most efficient quantizers, even compared with random projections. Using the k-means hash function is slower than using random projection for typical parameters. HKM is a good compromise, as it offers a relatively low query preparation cost while adapting to the data.

4. Querying mechanisms

In this section, we detail how the k-means approach is used to build a complete LSH system, and analyze the corresponding search results. The resulting algorithm is referred to as KLSH in the following. We then build upon KLSH by proposing and evaluating more sophisticated strategies, somehow similar in spirit to those recently introduced in the literature, namely multi-probing and query-adaptive querying.

4.1. KLSH

Indexing d-dimensional descriptors with KLSH proceeds as follows. First, it is necessary to generate l different k-means clustering using the same learning set of vectors. This diversity is obtained by varying initializations[4] of the k-means. Note that it is very unlikely that these different k-means gives the same solution for k high enough, as the algorithm converges to a local minimum only. Once these l codebooks are generated, each one being represented by its centroids $\{c_{j,1},\ldots,c_{j,k}\}$, all the vectors to index are read sequentially. A vector to index is assigned to the nearest centroid found in one codebook. All codebooks are used in turn for doing the l assignments for this vector before moving to the next vector to index. Note this mechanism replaces the standard E2LSH $\mathscr{H}$ and g_j hash functions from Section 2.

At search time, the nearest centroid for each of the l k-means codebooks is found for the query descriptor. The database vectors assigned to these same centroids are then concatenated in the short-list, as depicted by Algorithm 1. From this point, the standard LSH algorithm takes on for processing the short-list.

Algorithm 1: KLSH, search procedure

Input: query vector q
Output: short-list sl
sl = **emptyset**
for $j = 1$ to l **do**
  //find the nearest centroid of q from codebook j:
  $i^* = \text{argmin}_{i=1,\ldots,k} L_2(q, c_{j,i})$
  $sl = sl \cup \{x \in \texttt{cluster}(c_{j,i^*})\}$
end for

[4] This is done by modifying the `seed` when randomly selecting the initial centroids from the learning set.

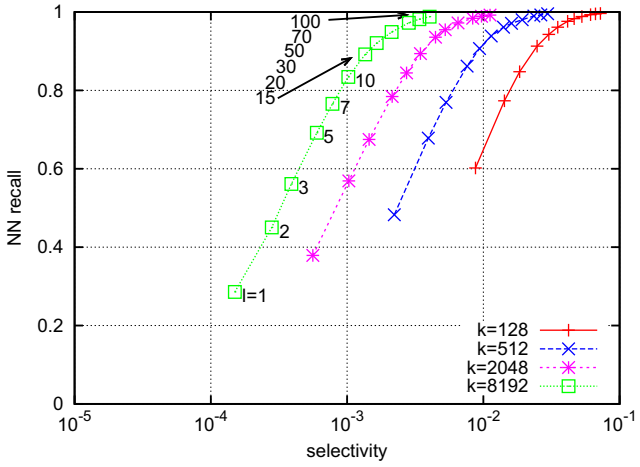


Fig. 5. Performance LSH with k-means hash functions for a varying number l of hash functions.

The results for KLSH are displayed Fig. 5. One can see that using a limited number of hash functions is sufficient to achieve high recall. A higher number of centroids leads to the best trade-off between search quality and selectivity. However, as indicated Section 2.2, the selectivity measures the asymptotic behavior for large datasets, for which the cost of this *qpc* stage is negligible compared to that of treating the set of vectors returned by the algorithm.

For small datasets, the selectivity does not solely reflect the "practical" behavior of the algorithm, as it does not take into account *qpc*. For KLSH, the overall cost is:

$$ocost = sel \times n \times d + k \times l \times d. \tag{13}$$

The acceleration factor therefore becomes:

$$ac = \frac{1}{sel + \frac{k \times l}{n}}. \tag{14}$$

Fig. 6 shows the acceleration factor obtained for a dataset of one million vectors, assuming that a full distance calculation is performed on the short-list. This factor accurately represents the true gain of using the ANN algorithm when the vectors are stored in main memory. Unlike what observed for asymptotically large datasets, for which the selectivity is dominant, one can observe that there is an optimal value of the quantizer size obtained for

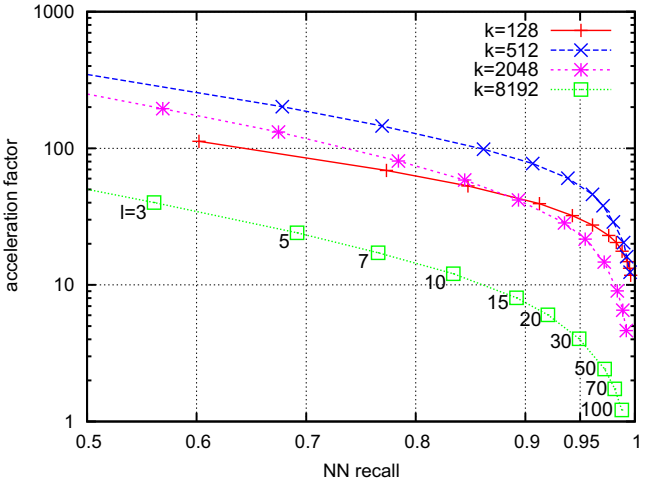


Fig. 6. Acceleration factor of LSH over exhaustive search. Both the query preparation cost and the final distance calculation are included.

$k = 512$. It offers the best trade-off between the query preparation cost and the post-processing of the vectors. Note that this optimum depends on the database size: the larger the database, the larger should be the number of centroids.

Algorithm 2: Multi-probe KLSH, search procedure

Input: query vector q
Output: short-list sl
sl = **emptyset**
for $j = 1$ to l **do**
//find the m_p nearest centroids of q from codebook j:
$(i^*_1, \ldots, i^*_m) = m_p\text{-argmin}_{i=1,\ldots,k} L_2(q, c_{j,i})$
$sl = sl \cup \bigcup_{i*=i*_1,\ldots,i*_m} \{x \in \texttt{cluster}(c_{j,i*})\}$
end for

As a final comment, in order to reduce the query preparation cost for small databases, a approximate k-means quantizer could advantageously replace the standard k-means, as done in (Philbin et al., 2007). Such quantizers assign vectors to cell indexes in logarithmic time with respect to the number of cells k, against linear time for standard k-means. This significantly reduces the query preparation cost, which is especially useful for small datasets.

4.2. Multi-probe KLSH

Various strategies have been proposed in the litterature to increase the quality of the results returned by the original LSH approach. One series of mechanisms extending LSH uses a so-called multi-probe approach. In this case, at query time, several buckets per hash function are retrieved, instead of one (see Lv et al. (2007) and Joly and Buisson (2008)). Probing multiple times the index increases the scope of the search, which, in turn, increases both recall and precision.

Originally designed for structured quantizers, this multi-probe approach can equally be applied to our unstructured scheme with the hope of also improving precision and recall. For the k-means hash functions, multi-probing can be achieved as follows. Having fixed the number m_p of buckets that we want to retrieve, for each of the l hash function, we select the m_p closest centroids of the unstructured quantizer $g_j = \{c_{j,1}, \ldots, c, c_{j,k}\}$. Algorithm 2 briefly presents the procedure.

The vectors associated with the selected m_p buckets are then returned for the l hash functions. Note that choosing $m_p = 1$ is equivalent to using the basic KLSH approach. The total number of buckets retrieved is $l \times m_p$. Therefore, for a fixed number of buckets, the number of hash functions is reduced by a factor m_p. The memory usage and the query preparation cost are thus divided by this factor.

Fig. 7 shows the results obtained when using $l = 1$ and varying values of m_p, i.e., for a single hash function. The results are reasonably good, especially considering the very low memory usage associated with this variant. However, comparing Figs. 5 and 7, the recall is lower for the same selectivity. This is not surprising, as in KLSH, the vectors which are returned are localized in the same cell, whereas the multi-probe variant returns some vectors that are not assigned to the same centroid.

For small datasets, for which the query preparation cost is not negligible, this multi-probe variant is of interest. This is the case for our one million vectors dataset: Fig. 8 shows the better performance of the multi-probe algorithm compared to the standard querying mechanism (compare to Fig. 6). This acceleration factor compares favorably against state-of-the-art methods of the literature. In a similar experimental setup (dataset of 1 million SIFT descriptors), Muja and Lowe (2009) reports, for a recall of 0.90,

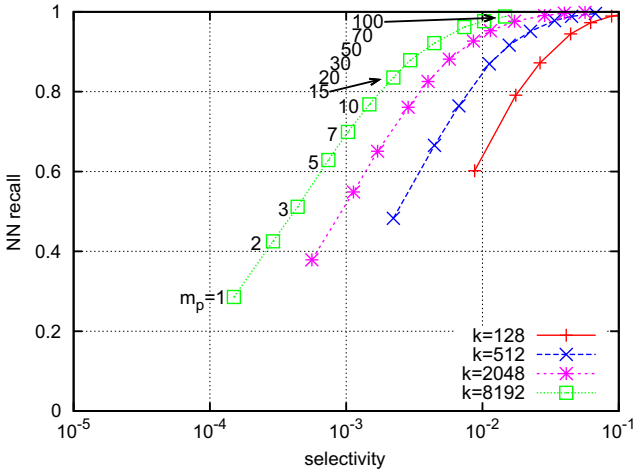


Fig. 7. Multi-probe KLSH for a single hash function ($l = 1$) and varying numbers of visited cells m_p.

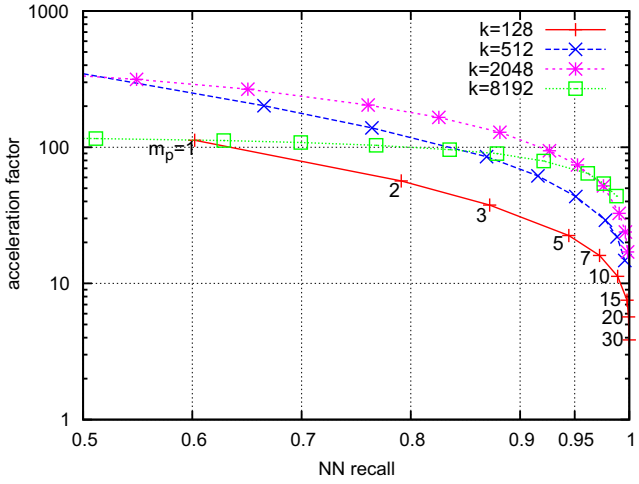


Fig. 8. Multi-probe KLSH: Acceleration factor of LSH over exhaustive search.

an acceleration factor lower than 100, comparable to our results but for a higher memory usage: the multi-probe KLSH structure only uses 4 bytes per descriptor for $m_p = 1$.

4.3. Query-adaptive KLSH

While multi-probing is one direction for improving the quality of the original structured LSH scheme, other directions exist, like the query-adaptive LSH by Jégou et al. (2008a). In a nutshell, this method adapts its behavior because it picks from a large pool of existing random hash functions the ones that are the most likely to return the nearest neighbors, on a per-query basis.

As it enhances result quality, this principle can be applied to our unstructured approach. Here, instead of using a single k-means per hash function, it is possible to maintain a poll of independent k-means. At query time, the best k-means can be selected for each hash function, increasing the likelyhood of finding good neighbors.

Before developping the query-adaptive KLSH, we must describe the original query-adaptive LSH to facilitate the understanding of the remainder. Query-adaptive LSH as described in Jégou et al., 2008a proceeds as follows (this is also summarized in Algorithm 3):

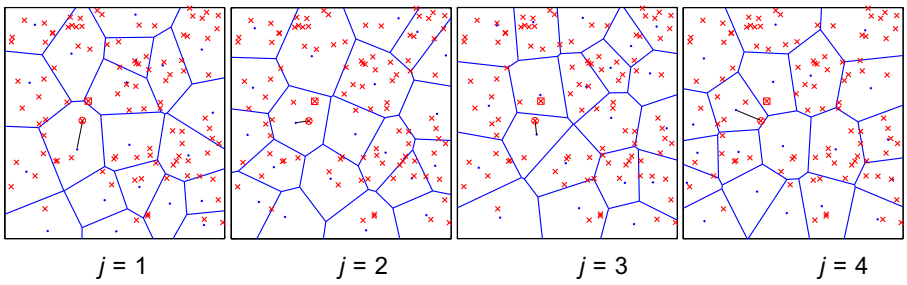


Fig. 9. Toy example: hash function selection process in query-adaptive KLSH. The length of the segment between the query vector (circled) and its nearest centroid corresponds to the relevance criterion λ_j ($j = 1..4$). Here, for $p = 1$, the second hash function ($j = 2$) is used and returns the correct nearest neighbor (squared).

Algorithm 3: Query-adaptive KLSH, search procedure

Input: query vector q
Output: short-list sl
sl = **emptyset**
//select the p hash functions minimizing $\lambda(g_j(q))$:
$(j_1,\ldots,j_p) = p\text{-argmin}_{j=1,\ldots,l}\lambda(g_j(q))$
for $j \in (j_1,\ldots,j_p)$ **do**
  //find the nearest centroid of q from codebook j:
  $i^* = \text{argmin}_{i=1,\ldots,k} L_2(q, c_{j,i})$
  $sl = sl \cup \{x \in \texttt{cluster}(c_{j,i}{}^*)\}$
end for

- The method defines a pool l of hash functions, with l larger than in standard LSH.
- For a given query vector, a relevance criterion λ_j is computed for each hash function g_j. This criterion is used to identify the hash functions that are most likely to return the nearest neighbor(s).
- Only the buckets associated with the p most relevant hash functions are visited, with[5] $p \leqslant l$.

The relevance criterion proposed in (Jégou et al., 2008a) corresponds, for the E_8 lattice, to the distance between the query point and the center of the Voronoi cell. We use the same criterion for our KLSH variant. For the query vector q,λ is defined as

$$\lambda(g_j) = \min_{i=1,\ldots c,k} L_2(q, c_{j,i}). \quad (15)$$

It turns out that this criterion is a by-product of finding the nearest centroid. Therefore, for a fixed number l of hash functions, the pre-processing cost is the same as in the regular querying method of KLSH. These values are obtained to select the p best hash function as

$$p - \arg\min_{j=1,\ldots c,l} \lambda(g_j). \quad (16)$$

The selection process is illustrated by the toy example of Fig. 9, which depicts a structure comprising $l = 4$ k-means hash functions. Intuitively, one can see that the location of a descriptor x in the cell has a strong impact on the probability that its nearest neighbor is hashed in the same bucket. On this example, only the second clustering ($j = 2$) puts the query vector and its nearest neighbor in the same cell.

I n order for the query-adaptive KLSH to have interesting properties, one should use a large number l of hash functions. This yields two limitations for this variant:

- the memory required to store the hash tables is increased;
- the query preparation cost is higher, which means that this variant is interesting only for very large datasets, for which the dominant cost is the processing of the vectors returned by the algorithm.

The selection of the best hash functions is not time consuming since the relevance criterion is obtained as a by-product of the vector quantization for the different hash functions. However, this variant is of interest if we use more hash functions than in regular LSH, hence in practice its query preparation cost is higher. For a reasonable number of hash functions and a large dataset, the bottleneck of this query adaptive variant is the last step of the "exact" LSH algorithm. This is true only when the dominant cost is the processing cost of the search for the exact nearest neighbors within the short-list obtained by parsing the buckets. This is not the case in our experiments on one million vectors, in which the acceleration factor obtained for this variant is not as good as those of KLSH and multi-probe KLSH.

Fig. 10 gives the selectivity obtained, using only *one* voting hash function ($p = 1$), for varying sizes l of the set of hash functions. Unsurprisingly, the larger l, the better the results. However, most of the gain is attained by using a limited number of hash functions. For this dataset, choosing $l = 10$ seems a reasonable choice.

Now, using several voting hash functions, i.e., for $p > 1$, one can observe in Fig. 11 that the query-adaptive mechanism significantly outperforms KLSH in terms of the trade-off between selectivity

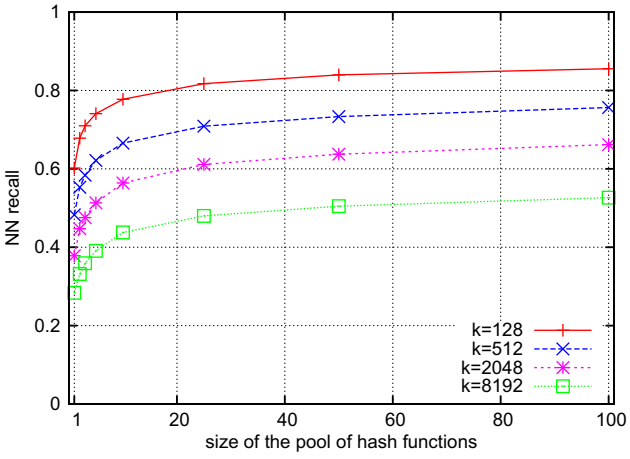


Fig. 10. Query-adaptive KLSH: performance when using **a single** hash function among of pool of l hash functions, $l = 1,2,3,5,10,20,25,50,100$. For a given number k of clusters, the selectivity is very stable and close to $1/d$: 0.0085 for $k = 128$, 0.0021 for $k = 512$, 0.00055 for $k = 2048$ and 0.00014 for $k = 8192$.

[5] For $p = l$, the algorithm is equivalent to the KLSH.

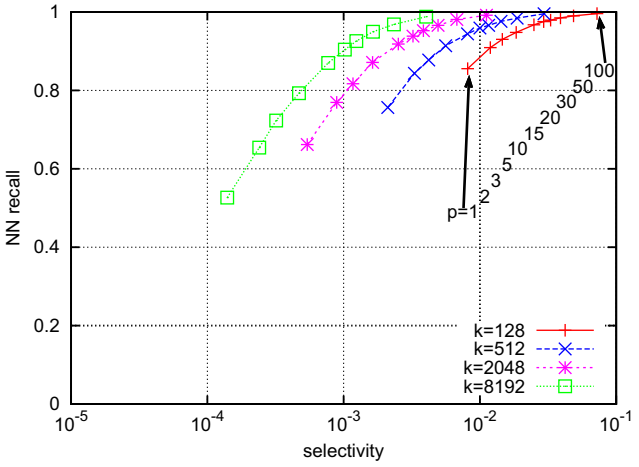


Fig. 11. Query-adaptive KLSH: performance for a fixed pool of $l = 100$ k-means hash functions and a varying number p of selected hash functions.

and efficiency. In this experiment the size of the pool is fixed to $l = 100$. However, on our "small" dataset of one million descriptors, this variant is not interesting for this large number of hash functions: the cost of the query preparation stage ($k \times l \times d$) is too high with respect to the post-processing stage of calculating the distances on the short-list. This contradictory result stems from the indicator: the selectivity is an interesting complexity measurement for large datasets only, for which the query preparation cost is negligible compared to that of processing the vectors returned by the algorithm.

4.4. Discussion

4.4.1. Off-line processing and parametrization

One of the drawbacks of k-means hash functions over structured quantizers is the cost of creating the quantizer. This is especially true for the query-adaptive variant, where the number l of hash functions may be high. However, in many applications, this is no a critical point, as this clustering is performed off-line. Moreover, this is somewhat balanced by the fact that KLSH admits a simpler optimization procedure to find the optimal parametrization, as we only have to optimize the parameter k, against two parameters w and d^* for the structured quantizer. Finally, approximate k-means reduces the cost of both the learning stage and the query preparation. It is therefore not surprising that the emerging state-of-the-art ANN methods, e.g., (Muja and Lowe, 2009), relies on such partitioning methods.

4.4.2. Which querying mechanism should we use?

It appears that each of the querying mechanism proposed in this section may be the best, depending on the context: dataset size, vector properties and resource constraints. The less memory-demanding method is the multi-probe version. For very large datasets and with no memory constraint, query-adaptive KLSH gives the highest recall for a fixed selectivity. If, for the fine verification, the vectors are read from a low-efficiency storage device, e.g., a mechanical hard drive, then the query-adaptive version is also the best, as in that case the bottleneck is to read the vectors from the disk. As a general observation, multi-probe and query-adaptive KLSH offer opposite properties is terms of selectivity, memory usage and query preparation cost. The "regular" KLSH is in between, offering a trade-off between these parameters. Overall, the three methods are interesting, but for different operating points.

5. Conclusion

In this paper, we have focused on the design of the hash functions and the querying mechanisms used in conjunction with the popular LSH algorithm. First, confirming some results of the literature in a real application scenario, we have shown that using lattices as stronger quantizers significantly improve the results compared to the random projections used in the Euclidean version of LSH. Second, we have underlined the limitations of structured quantizers, and show that using unstructured quantizers as a hash functions offer better performance, because it is able to take into account the distribution of the data. The results obtained by k-means LSH are appealing: very high recall is obtained by using only a limited number of hash functions. The speed-up over exhaustive distance calculation is typically greater than 100 on a one million vector dataset for a reasonable recall. Finally, we have adapted and evaluated two recent variants of the literature, namely multi-probe LSH and query-adaptive LSH, which offer different trade-offs in terms of memory usage, complexity and recall.

Acknowledgement

The authors would like to thank the project Quaero for its financial support.

References

Agrell, E., Eriksson, T., Vardy, A., Zeger, K., 2002. Closest point search in lattices. IEEE Trans. Inform. Theory 48 (8), 2201–2214.
Andoni, A., Indyk, P., 2006. Near-optimal hashing algorithms for near neighbor problem in high-dimensions. In: Proc. of the Symposium on the Foundations of Computer Science, pp. 459–468.
Beyer, K., Goldstein, J., Ramakrishnan, R., Shaft, U., 1999. When is nearest neighbor meaningful? In: Int. Conf. on Database Theory, August, pp. 217–235.
Böhm, C., Berchtold, S., Keim, D., 2001. Searching in high-dimensional spaces: Index structures for improving the performance of multimedia databases. ACM Comput. Surv. 33 (3), 322–373.
Brown, M., Lowe, D.G., 2007. Automatic panoramic image stitching using invariant features. Internat. J. Comput. Vision 74 (1), 59–73.
Casey, M., Slaney, M. 2007. Fast recognition of remixed music audio. In: Internat. Conf. on Acoustics, Speech and Signal Process, April, vol. 4, pp. 1425–1428.
Conway, J., Sloane, N., 1982a. Fast quantizing and decoding algorithms for lattice quantizers and codes. IEEE Trans. Inform. Theory 28 (2), 227–232.
Conway, J., Sloane, N., 1982b. Voronoi regions of lattices, second moments of polytopes, and quantization. IEEE Trans. Inform. Theory 28 (2), 211–226.
Conway, J., Sloane, N., Bannai, E., 1987. Sphere-packings, Lattices, and Groups. Springer-Verlag New York, Inc., New York, NY, USA.
Datar, M., Immorlica, N., Indyk, P., Mirrokni, V., 2004. Locality sensitive hashing scheme based on p-stable distributions. In: Proc. of the Symposium on Computational Geometry, pp. 253–262.
Gionis, A., Indyk, P., Motwani, R., 1999. Similarity search in high-dimension via hashing. In: Proc. of the Internat. Conf. on Very Large DataBases, pp. 518–529.
Gray, R.M., Neuhoff, D.L., 1998. Quantization. IEEE Trans. Inform. Theory 44, 2325–2384.
Jégou, H., Amsaleg, L., Schmid, C., Gros, P., 2008a. Query-adaptive locality sensitive hashing. In: Internat. Conf. Acoustics, Speech, and Signal Processing.
Jégou, H., Douze, M., Schmid, C., 2008b. Hamming embedding and weak geometric consistency for large scale image search. In: European Conf. on Computer Vision, October.
Joly, A., Buisson, O., 2008. A posteriori multi-probe locality sensitive hashing. In: ACM Conf. on Multimedia, pp. 209–218.
Ke, Y., Sukthankar, R., Huston, L., 2004. Efficient near-duplicate detection and sub-image retrieval. In: ACM Conf. on Multimedia, pp. 869–876.
Lejsek, H., Ásmundsson, F., Jónsson, B., Amsaleg, L., 2006. Scalability of local image descriptors: A comparative study. In: ACM Conf. on Multimedia, pp. 589–598.
Lowe, D., 2004. Distinctive image features from scale-invariant keypoints. Internat. J. Comput. Vision 60 (2), 91–110.
Lv, Q., Josephson, W., Wang, Z., Charikar, M., Li, K., 2007. Multi-probe LSH: Efficient indexing for high-dimensional similarity search. In: Proc. Internat. Conf. on Very Large DataBases, pp. 950–961.
Matei, B., Shan, Y., Sawhney, H., Tan, Y., Kumar, R., Huber, D., Hebert, M., 2006. Rapid object indexing using locality sensitive hashing and joint 3D-signature space estimation. IEEE Trans. Pattern Anal. Machine Intell. 28 (7), 1111–1126.
Mikolajczyk, K., Schmid, C., 2004. Scale and affine invariant interest point detectors. Internat. J. Comput. Vision 60 (1), 63–86.

Muja, M., Lowe, D.G., 2009. Fast approximate nearest neighbors with automatic algorithm configuration. In: Internat. Computer Graphics Theory and Applications.

Nistér, D., Stewénius, H., 2006. Scalable recognition with a vocabulary tree. In: Proc. IEEE Conf. on Computer Vision and Pattern Recognition, pp. 2161–2168.

Philbin, J., Chum, O., Isard, M., Sivic, J., Zisserman, A., 2007. Object retrieval with large vocabularies and fast spatial matching. In: Proc. IEEE Conf. on Computer Vision and Pattern Recognition.

Shakhnarovich, G., Darrell, T., Indyk, P., 2006. Nearest-Neighbor Methods in Learning and Vision: Theory and Practice. MIT Press. Ch. 3.

Sivic, J., Zisserman, A., 2003. Video-Google: A text retrieval approach to object matching in videos. In: Proc. IEEE Internat. Conf. on Computer Vision, pp. 1470–1477.

Terasawa, K., Tanaka, Y., 2007. Spherical LSH for Approximate Nearest-Neighbor Search on Unit Hypersphere. Springer. pp. 27–38.

Vardy, A., Be'ery, Y., 1993. Maximum likelihood decoding of the leech lattice. IEEE Trans. Inform. Theory 39 (4), 1435–1444.

Zhang, J., Marszalek, M., Lazebnik, S., Schmid, C., 2007. Local features and kernels for classification of texture and object categories: A comprehensive study. Internat. J. Comput. Vision 73, 213–238.