Learning Binary Hash Codes for Large-Scale Image Search

Kristen Grauman and Rob Fergus

Abstract Algorithms to rapidly search massive image or video collections are critical for many vision applications, including visual search, content-based retrieval, and non-parametric models for object recognition. Recent work shows that *learned binary projections* are a powerful way to index large collections according to their content. The basic idea is to formulate the projections so as to approximately preserve a given similarity function of interest. Having done so, one can then search the data efficiently using hash tables, or by exploring the Hamming ball volume around a novel query. Both enable sub-linear time retrieval with respect to the database size. Further, depending on the design of the projections, in some cases it is possible to bound the number of database examples that must be searched in order to achieve a given level of accuracy.

This chapter overviews data structures for fast search with binary codes, and then describes several supervised and unsupervised strategies for generating the codes. In particular, we review supervised methods that integrate metric learning, boosting, and neural networks into the hash key construction, and unsupervised methods based on spectral analysis or kernelized random projections that compute affinity-preserving binary codes. Whether learning from explicit semantic supervision or exploiting the structure among unlabeled data, these methods make scalable retrieval possible for a variety of robust visual similarity measures. We focus on defining the algorithms, and illustrate the main points with results using millions of images.

Kristen Grauman
Dept. of Computer Science, University of Texas, Austin e-mail: grauman@cs.utexas.edu

Rob Fergus
Courant Institute, New York University e-mail: fergus@cs.nyu.edu

1 Introduction

Huge collections of images and video are increasingly available, arising in domains as diverse as community photo collections, scientific image data of varying modalities, news media, consumer catalogs, or surveillance archives. In the last decade in particular, user-generated content is widely stored and shared on the Web. Numbering in to the tens or hundreds of billions, their sheer size poses a challenge for conventional computer vision techniques. For example, every *minute* more than 20 hours of new video are uploaded to YouTube and 100,000 photos are uploaded to Facebook. Clearly, even real-time methods would be incapable of coping with this deluge of data. Consequently, researchers are exploring new representations and approaches to search large-scale datasets.

At the core of visual search is the nearest neighbor problem: given a query, which items in the database are most like it? Despite the simplicity of this problem statement, fast and accurate nearest neighbor search can enable a spectrum of important applications. See Figure 1.

Efficient algorithms to address the basic similarity search task have received much attention over the years, yielding a variety of tree-based and hashing-based algorithms [22, 9, 66, 44, 47]. However, while applicable to visual data in certain cases, traditional methods often fall short of technical demands inherent to our setting:

- **High-dimensional data.** First of all, good descriptors for images or videos typically live in a high-dimensional space, easily numbering thousands or more dimensions. At the same time, the volume of data quickly strains memory, and disk access is slow. Both aspects argue for mapping to a more compact representation, and/or developing approximate search methods whose error and storage requirements do not blow up with the input dimensionality.

- **Structured input spaces and specialized similarity functions.** Secondly, visual data need not fit neatly within a vector space representation at all. More sophisticated descriptors built on graphs, sets, or trees are often appealing, as they more more closely model the real structure at hand. For example, an image might be described by a planar graph over its component regions, or a video clip may be encoded as a set of loosely ordered keyframes. Alongside such structured representations, researchers have developed specialized affinity or *kernel* functions to accommodate them that are accurate, but would be prohibitively costly to apply naively in the search setting. Thus, there is a clear need for flexibility in the similarity measures supported.

- **Availability of external supervision.** Finally, the complex relationships intrinsic to visual data can be difficult to capture with manually-defined features and metrics. There is a well-known gap between the low-level cues one might pull from an image or video, and the high-level semantics one would like preserved in a content-based search. Access to external knowledge—in the form of labeled

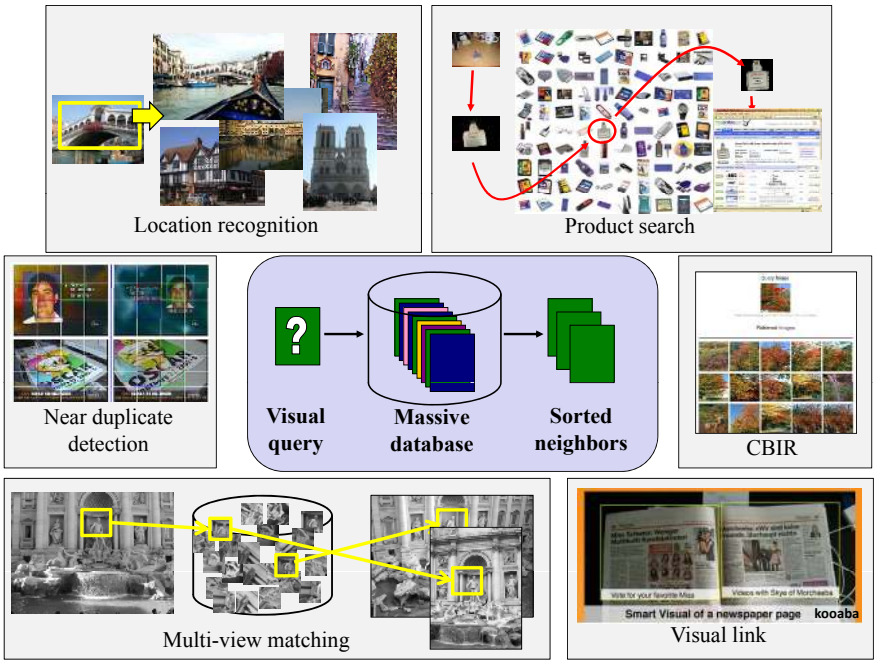

Fig. 1 Large-scale visual search underlies many interesting applications. *Figure credits: Product search image is courtesy of Yeh et al. [75], CBIR image is courtesy of Iqbal and Aggarwal [35], near duplicate detection image is courtesy of Xu et al. [74], and visual link image is from kooaba.com.*

instances or target similarity constraints—would ideally inform the comparisons. This suggests learning should somehow be integrated into the indexing approach or representation.

The central theme of this chapter is the construction and use of binary representations for visual search, as a means to address the above requirements. The main idea is to compute binary projections such that a given similarity function of interest is approximately preserved in Hamming space. Having done so, one can then search the data efficiently using hash tables, or by exploring the Hamming ball volume around a novel query. Depending on the design of the projections, guarantees on the relative retrieval cost and error are sometimes possible.

Why do we explicitly target binary representations? Not only do they fit well with hashing and Hamming-ball search strategies, but also we will see that by carefully maximizing the information carried by each bit, we can achieve far more compact representations for a given storage budget than using real-valued descriptors (which effectively use groups of bits, i.e., 32 for a single-precision real). Clearly, when the datasets are $O(10^{11})$ in size, minimizing the storage overhead is vital, as discussed above.

The algorithms for learned binary projections that we discuss have important ties and contrasts to previous work in dimensionality reduction and distance learning. Dimensionality reduction is a classic problem in machine learning, but the variety of existing techniques are not necessarily well-suited to this domain. Many approaches (such as Isomap [64], LLE [55], or t-SNE [67]) do not scale to large datasets since their complexity is quadratic (or worse) in the number of data points. Furthermore, with large datasets, the ability to project new instances is vital, since it allows parallelization and asynchronous processing of the data. Yet many classic approaches [64, 55, 67] lack an explicit mapping function to the low-dimensional space, making "out-of-sample" projections problematic. In contrast, the techniques we describe have an explicit mapping to the low-dimensional space, allowing the binary representation to be quickly computed.

Effective retrieval requires a representation that places images of semantically similar content close and dissimilar content far apart. Traditional approaches from content-based image retrieval (CBIR) rely on simple representations—for example, using color or edge histograms (see [18] for a review of such methods). However, significant improvements over these representations have proven difficult to achieve through hand-crafting alone. Recent efforts have therefore focused on *learning* good representations using labeled training data (see text below for references). In practice, this can be viewed as a supervised form of dimensionality reduction or distance learning. The work we describe fits into this area, but with a focus on deriving techniques that produce binary embeddings instead of real-valued ones.

In the following section, we first briefly review primary data structures for fast search with binary codes. Having established the search protocols, we then devote the majority of the text to explaining several strategies to generate binary codes. We organize them roughly around the degree of supervision assumed: in Section 3 we describe supervised methods that integrate metric learning, boosting, or neural networks into the hash key construction, and in Section 4 we describe unsupervised methods that use spectral analysis or kernelized random projections to compute affinity-preserving binary codes. While the former exploit explicit supervision from annotated training data, the latter exploit the structure among a sample of unlabeled data to learn appropriate embeddings. We include some example empirical results to illustrate key points.

We refer the reader to the original publications that introduced the algorithms for more details, particularly [37, 40, 57, 60, 72, 39]. While we present the methods in the context of visual retrieval tasks, in fact the core methods are not specific to images in any way, and could be applied to search other forms of data as well.

2 Search Algorithms for Binary Codes

The majority of this chapter is devoted to the construction of compact binary codes, either using label information (Section 3) or just by mimicking the neighborhood structure of the original input space (Section 4). However, mindful of our overall

Dataset	LabelMe	Web
# datapoints	2×10^4	1.29×10^7
Gist vector dim.	512	384
Method	**Time (s)**	**Time (s)**
Spill tree - Gist vector	1.05	-
Brute force - Gist vector	0.38	-
Brute force - 30 bit binary	4.3×10^{-4}	0.146
″ - 30 bit binary, M/T	2.7×10^{-4}	0.074
Brute force - 256 bit binary	1.4×10^{-3}	0.75
″ - 256 bit binary, M/T	4.7×10^{-4}	0.23
Sem. Hashing - 30 bit binary	6×10^{-6}	6×10^{-6}

Fig. 2 Performance of brute-force exhaustive search and Semantic Hashing [56] on two datasets: LabelMe (20,000 images, 512 dimensional Gist input) and Tiny Images (12.9 million, 384 dimensions). The timings for a single query are shown, using a range of different approaches. A spill tree (*kd*-tree variant) [44] is worse than linear search due to the high dimensionality of the input. Reducing the Gist descriptors to binary codes (30-bit and 256-bits) makes linear search very fast ($<$ 1ms/query) on small datasets, but only moderately quick for the larger Tiny Images ($<$ 1s/query). Linear search is easily parallelized (e.g., by using multiple CPU cores—see entries labeled M/T). Semantic hashing is extremely quick ($< 1\mu$s), regardless of dataset size, but can only be applied to compact codes. In contrast, the query time for linear search simply scales linearly with the code length (and database size).

goal, we now explain how these binary codes can be used to perform the nearest-neighbor search.

In general for large-scale retrieval, the most important property is that the search time complexity be sub-linear in the number of database examples. Additionally, given the distributed nature of large-scale computing, the ability to parallelize the search is important for practical applications. In the particular context of binary codes, as we consider here, retrieval involves finding all examples that have a zero or small Hamming distance from the query, where the Hamming distance between two binary vectors is the number of bits that differ between them.

To satisfy these requirements, we consider variants of *hashing*. Hashing is quick and has minimal storage requirements beyond the binary data vectors themselves. It uses all dimensions of the binary codes (bits) in parallel to perform retrieval. This is in contrast to tree-based algorithms such as *kd*-trees, where each example is found by making a series of binary decision to traverse the tree, each decision (bit) being conditional on the choices above.

2.1 Linear Scan in Hamming Space

The most straightforward solution is a brute-force linear scan—that is, to compute the Hamming distance between the query vector and every vector in the database. Although this scales linearly with the size of the database, the constant is typically very small, and thus it can be practical for surprisingly large datasets. It is also worth noting that the memory overhead for this approach is virtually nil.

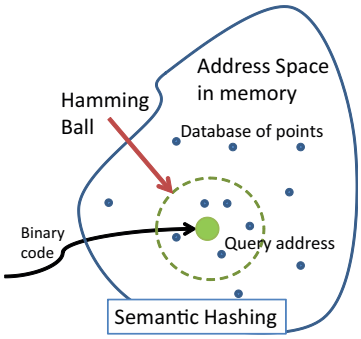

Fig. 3 Semantic Hashing [57] treats the query vector as an address in memory. A Hamming ball is explored by perturbing bits within the query. Nearby points will thus be found by direct memory look-up.

Computing the Hamming distance between two vectors requires two steps: (i) compute the XOR and (ii) count the number of 1's in the resulting vector. Both operations can be performed extremely quickly on modern CPUs, with parallel XOR operations being part of the SSE instruction set. The parallelism can also be trivially extended at the machine level, with each server in a cluster searching a different subset of the database.

In Figure 2 we show timings for brute-force evaluation using 30- and 256-bit codes on two datasets, one of 20,000 vectors, the other 12.7 million. For the 30-bit codes, a 2.0Ghz CPU is able to compare 50 million pairs per second, while for 256-bit codes this increases to 120 million per second. Thus, the brute-force approach scales gracefully to longer code lengths, as we would expect. Figure 2 also shows the use of two cores instead of one nearly doubles the speed, confirming that it is easy parallelized. These results demonstrate that it is a viable approach for moderately large datasets.

2.2 Semantic Hashing

Salakhutdinov and Hinton proposed a nearest-neighbors technique for binary vectors called *Semantic Hashing* whose speed is *independent of the number of data points* [57]. Each binary vector corresponds to an address in memory. Matches to a query vector are found by taking the query vector and systematically perturbing bits within it, so exploring a Hamming ball around the original vector. Any neighbors in the database that fall within this ball will be returned as neighbors. See Figure 3 for an illustration.

The approach has two main advantages: (i) provided the radius of the Hamming ball is small, it is extremely quick (i.e. μs, see Figure 2, bottom row); and (ii) constructing the database is also very fast, e.g., compared to *kd*-tree type data structures.

However, a major drawback is that it breaks down for long code vectors, since the mean Hamming distance between points becomes large and the volume of the Ham-

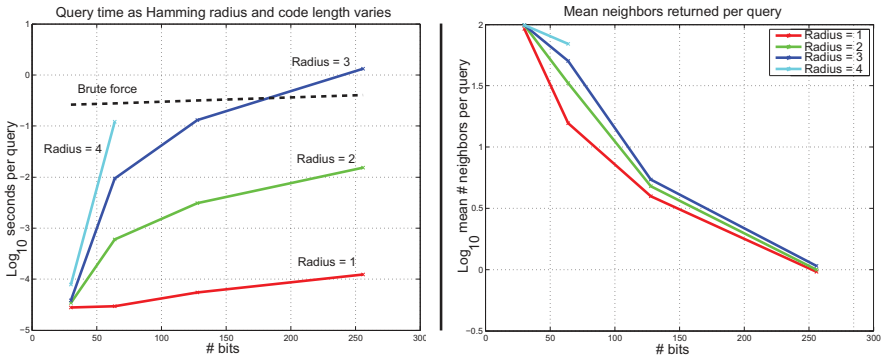

Fig. 4 Left: Query time as a function of code length, for different radii of the Hamming ball. Right: Mean number of returned neighbors as a function of #bits and radius of Hamming ball. Beyond 64 bits or so, the method is no longer significantly faster than brute force methods, and there is a risk that many queries will not find any neighbors. Increasing the radius slows the method down significantly, but does not significantly improve the number of neighbors found.

ming ball (which is `nchoosek`(dimension,radius)) becomes prohibitive to explore. To be concrete, suppose we have a code length of 100 bits. The mean distance to a query's closest neighbor may well be quite large, e.g., differing in 7 bits or more. However, if one can afford only a Hamming ball radius search of 3, then very often a query will not find any neighbors within that restricted search volume (see Figure 4, right). Another problem is that Semantic Hashing requires a contiguous block of memory, which becomes impractical for vectors beyond $l = 32$ bits (corresponding to a 4Gb block).[1]

This practical restriction motivates using sophisticated embedding methods to preserve information within a very compact code. For example, neural network-based methods that we will describe in Section 3.3.2 can be used to train a 30-bit descriptor that yields μs retrieval times with Semantic Hashing, as shown in Figure 2 (bottom row). Alternatively, one can employ hashing algorithms that are designed to map similar instances to the *same* bin, thus avoiding the need for Hamming ball exploration or contiguous memory. We discuss such an approach next.

2.3 Locality Sensitive Hashing

While the Semantic Hashing approach is simple and extremely effective for very compact codes, it has the practical limitations discussed above, and also lacks formal guarantees on the search quality obtained. *Locality Sensitive Hashing* is a randomized hashing framework introduced earlier by Gionis, Indyk, and Motwani that

[1] This can be circumvented by introducing a second randomized hash function that maps the 2^l block of memory down to the permissible memory budget. An appropriate randomized hash function will introduce few collisions and be quick, and thus will not impact query accuracy or speed.

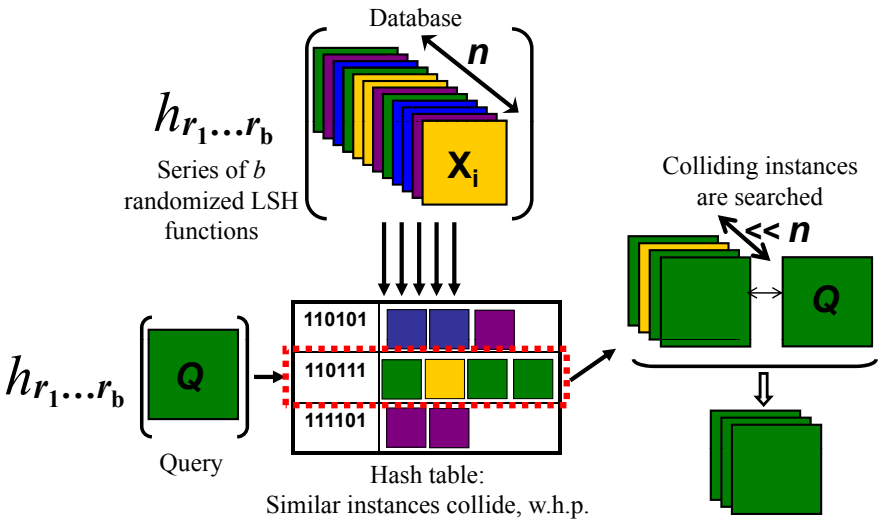

Fig. 5 Locality Sensitive Hashing (LSH) uses hash keys constructed so as to guarantee collision is more likely for more similar examples [33, 23]. Once all database items have been hashed into the table(s), the same randomized functions are applied to novel queries. One exhaustively searches only those examples with which the query collides.

counters some of these shortcomings, and allows a user to explicitly control the similarity search accuracy and search time tradeoff [23].

The main idea in Locality Sensitive Hashing (LSH) is to insert database items into a hash table such that similar things fall in the same bucket, with high probability [33, 23, 1, 12]. Intuitively, if only highly similar examples collide in the hash table (i.e., are assigned the same *hash key*), then at query time, directly hashing to a stored bucket will reveal the most similar examples, and only those need to be searched. See Figure 5 for a visualization of the method. The hash keys generally consist of low-dimensional binary strings; each database item is mapped to b bits, where each bit is generated independently by a valid locality-sensitive hash function.

Assuming such projections can be appropriately formed, there are algorithms to retrieve the approximate neighbors for a query using hash tables or related data structures [23, 1, 12]. The neighbors are "approximate" in that they are within some ε error of the true near neighbor, and the bounds for the search time are tied to this approximation error. For example, the query time for retrieving $(1+\varepsilon)$-near neighbors can be bounded by $O(n^{1/(1+\varepsilon)})$ for the Hamming distance using appropriate hash functions [23]. This allows a trade-off between the sub-linear retrieval time achieved and the extent to which the returns mimic an exhaustive linear scan, even for high-dimensional input data. Thus, the hashing strategy has important advantages over tree-based techniques that are known to perform poorly in practice for high-dimensional data (e.g., *kd*-trees). Further, because LSH aims to have all relevant instances collide in the same bucket (hash code), its query-time cost depends

only linearly on the number of bits used, and does not require a scan of neighboring hash codes (in contrast to Semantic Hashing above).

Note that while early definitions of LSH functions were designed to preserve a given geometric distance, work since has explored ways to formulate *learned* locality-sensitive functions amenable to a target task, or functions that are sensitive to a family of kernel functions, as we will see later in the chapter in Sections 3.2 and 4.3.

More formally, suppose we have a database consisting of data points $\boldsymbol{x}_1, ..., \boldsymbol{x}_n$. Given an input query $\boldsymbol{q}$, we are interested in finding those items in the database that are most similar to the query, under some defined measure of similarity or distance (which we will discuss in more detail below). The hash keys are constructed by applying b binary-valued hash functions $h_1, \ldots, h_b$ to each of the database objects, where each h_i is a random sampling from an LSH function family $\mathscr{H}$.

Nearest Neighbor-based LSH Definition One formulation of LSH [12] describes valid locality-sensitive functions by equating collision probabilities with a similarity score; that is, each hash function $h_{\mathscr{H}}$ drawn from the distribution $\mathscr{H}$ must satisfy:

$$\Pr[h_{\mathscr{H}}(\boldsymbol{x}_i) = h_{\mathscr{H}}(\boldsymbol{x}_j)] = \text{sim}(\boldsymbol{x}_i, \boldsymbol{x}_j), \tag{1}$$

where $\text{sim}(\boldsymbol{x}_i, \boldsymbol{x}_j) \in [0,1]$ is the similarity function of interest.

The preprocessing of the database items is as follows. After computing the projections for all n database inputs, one then forms $M = 2n^{1/(1+\varepsilon)}$ random permutations of the bits. If we think of the database hash keys as an $n \times b$ matrix, that means we randomly permute the vector $[1, 2, \ldots, b]$ M times, and use each permutation as indices to reorder the columns of the hash key matrix. Then each list of permuted hash keys is sorted lexicographically to form M "sorted orders". Given a novel query, its hash key indexes into each sorted order with a binary search, and the $2M$ nearest examples found contain the approximate nearest neighbors. This procedure requires searching $O(n^{1/(1+\varepsilon)})$ examples using the original distance function of interest to obtain the $k = 1$ approximate nearest neighbor (NN). See [12] for more details. We will return to this definition below when discussing forms of supervised and unsupervised LSH function generation.

Radius-based LSH Definition While the above provides guarantees for approximating nearest neighbor search for a similarity function, another related formulation of LSH provides guarantees in terms of the likelihood of collision with a query's r-radius neighbors (i.e., where the goal is to retrieve a database item within a given radius of the query). Let $d(\cdot,\cdot)$ be a distance function over items from a set S, and for any item $\boldsymbol{p} \in S$, let $B(\boldsymbol{p}, r)$ denote the set of examples from S within radius r from $\boldsymbol{p}$. Let $h_{\mathscr{H}}$ denote a random choice of a hash function from the family $\mathscr{H}$. The family $\mathscr{H}$ is called $(r, r(1+\varepsilon), p_1, p_2)-$sensitive [23] for $d(\cdot,\cdot)$ when, for any $\boldsymbol{q}, \boldsymbol{p} \in S$,

- if $\boldsymbol{p} \in B(\boldsymbol{q}, r)$ then $\Pr[h_{\mathscr{H}}(\boldsymbol{q}) = h_{\mathscr{H}}(\boldsymbol{p})] \geq p_1$,
- if $\boldsymbol{p} \notin B(\boldsymbol{q}, r(1+\varepsilon))$ then $\Pr[h_{\mathscr{H}}(\boldsymbol{q}) = h_{\mathscr{H}}(\boldsymbol{p})] \leq p_2$.

For a family of functions to be useful, it must satisfy $p_1 > p_2$. Note that the probability of collision for close points is thus at least p_1^k, while for dissimilar points it is at most p_2^k.

During a preprocessing stage, all database points are mapped to a series of l hash tables indexed by independently constructed $g_1, \ldots, g_l$, where each g_i is a b-bit function. Then, given a query $\boldsymbol{q}$, an exhaustive search is carried out only on those examples in the union of the l buckets to which $\boldsymbol{q}$ hashes. These candidates contain the (r, ε)-nearest neighbors (NN) for $\boldsymbol{q}$, meaning if $\boldsymbol{q}$ has a neighbor within radius r, then with high probability some example within radius $r(1+\varepsilon)$ is found.[2]. More recent work includes an LSH formulation for data in Euclidean space, with improved query time and data structures [1].

Additional notes on LSH Intuitively the concatenation of b bits into hash keys decreases the false positive rate (we are more selective in what things will collide), whereas the aggregation of search candidates from l independently generated tables increases the recall (we are considering more randomized instances of the functions).

Early work by researchers in the theory community designated LSH function families for Hamming distance, ℓ_p norms, and the inner product [17, 12], as well as embedding functions to map certain metrics into Hamming space (e.g., the Earth Mover's Distance [34]). Given the attractive guarantees of LSH and the relatively simple implementation, vision and machine learning researchers have also explored novel hash function families so as to accommodate fast retrieval for additional metrics of interest. In particular, in this chapter we highlight hash functions for learned Mahalanobis metrics (Section 3.2) and kernel functions (Section 4.3).

2.4 Recap of Search Strategy Tradeoffs

Whereas the Semantic Hashing technique discussed above essentially takes an *embedding* strategy, where similarity ought to fall off smoothly as one looks at more distant codes in Hamming space, Locality Sensitive Hashing takes a direct *hashing* strategy, where similar items ought to map to the same hash key (i.e., Hamming distance = 0). LSH does not entail the bit-length restriction of Semantic Hashing. Memory usage with LSH is typically greater, however, assuming one opts to mitigate the 0-threshold Hamming distance by expanding the search to multiple independently generated hash tables.[3] Furthermore, whereas a user of Semantic Hashing specifies a radius of interest in the embedded Hamming space, a user of LSH (for

[2] For example, in [23] an LSH scheme using projections onto single coordinates is shown to be locality-sensitive for the Hamming distance over vectors. For that hash function, $\rho = \frac{\log p_1}{\log p_2} \leq \frac{1}{1+\varepsilon}$, and using $l = n^\rho$ hash tables, a $(1+\varepsilon)$-approximate solution can be retrieved in time $O(n^{\frac{1}{(1+\varepsilon)}})$

[3] In practice, a common implementation hack is to simply look at nearby bins according to Hamming distance, similar to Semantic Hashing, even if not necessarily using addresses as the bin index.

the radius-based search variant) specifies the radius of interest in the original feature space.

Perhaps the main distinction between the methods, however, is the degree to which the search and embedding procedures are integrated. In Semantic Hashing, the procedure to construct the binary embedding is performed independently (without knowledge of) the ultimate search data structure, whereas in LSH, the binary projections and search structure are always intertwined. This distinction can be viewed as a pro or con for either hashing implementation. The elegance and bounds of LSH are a potential advantage, but the restriction of designating appropriate LSH functions limits its flexibility. On the other hand, Semantic Hashing has the flexibility of choosing various learning algorithms to form the binary projections, but its behavior is less predictable with respect to an exhaustive linear scan.

While we will discuss the hash code construction techniques below in the context of one hashing implementation or the other, in practice a user could incorporate either one (or the linear scan), simply keeping the above tradeoffs in mind.

3 Supervised Methods for Learning Binary Projections

The quality of retrieval results clearly will depend on the chosen image representation as well as the distance metric used to compare examples. Ideally, these two components would together accurately reflect the instances' true relationships, such that relevant database items have a small distance to the query, and irrelevant items have a large distance. While a generic distance function (such as an L_p norm) may be more manageable computationally for large-scale search, it may or may not nicely match the desired relationships for a given application. Instead, if we have access to some form of supervision on a subset of examples, then we can attempt to *learn* how to compare them. General supervised classification methods as well as advances in metric learning over the last several years make it possible to fine-tune parametric distance functions [73, 5, 26, 59, 30, 71, 24, 19, 31, 16, 4].

Furthermore, we can attempt to simultaneously learn binary projections that reflect those specialized comparisons, thereby enabling fast Hamming space comparisons. Addressing both aspects generally entails optimizing the metric parameters according to data labeled by their classes or known distance relationships, while also balancing a preference for compact projections.

In this section, we describe two such approaches in detail. The first approach generates randomized hash functions that are locality-sensitive for learned Mahalanobis metrics, exploiting a sparse set of similarity constraints on tuples of points (Section 3.2). The second approach learns a Hamming embedding from a set of labeled training images (Section 3.3). Both map similar examples to be close-by in a binary space, while keeping dissimilar examples far apart. They differ primarily in the types of learning algorithms integrated to preserve those constraints, and secondarily in the hashing implementation employed alongside.

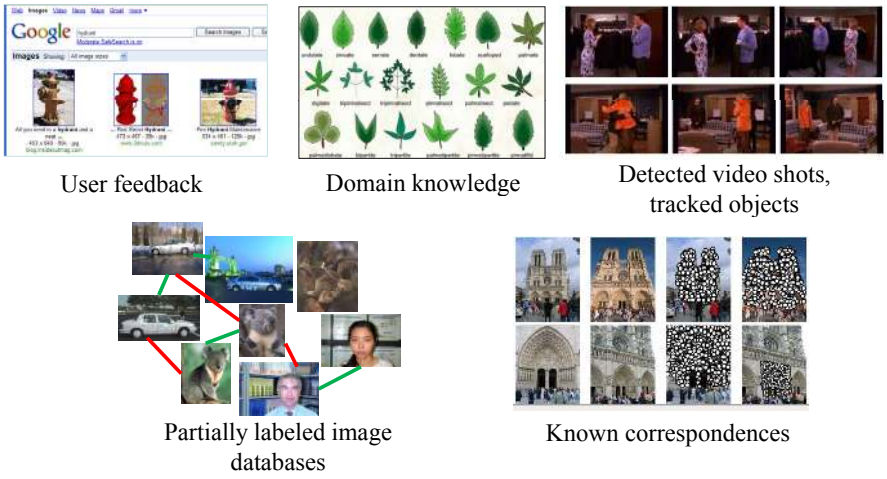

Fig. 6 Supervision about which instances should be near or far from one another could come from a variety of sources.

3.1 Forms of Supervision to Define Semantic Similarity

In the various supervised methods for code construction we discuss, the external supervision can take a variety of forms: labels or feedback on instances can specify those which ought to cluster together, relative judgments on triples of instances can specify their ideal relationships, or the desired nearest neighbor lists for a sample of points can specify those that need to remain close. As such, the techniques are suitable for enhancing nearest neighbor categorization as well as similarity search for content-based retrieval. Figure 6 highlights some possible sources of similarity constraints in the visual search domain.

When similarity constraints are non-exhaustive across a set of training data, we will represent them as sets of pairs: a set $\mathscr{S}$ containing pairs that should remain close (similar), and a set $\mathscr{D}$ containing pairs that should remain far (dissimilar). Alternatively, if we have complete pairwise information on all N training instances, we can think of the semantic information stored in an $N \times N$ matrix. To represent binary similarity constraints, the i, j-th entry in that matrix is 1 if two instances are meant to be close, 0 if far (e.g., with a discrete set of class labels, the entry is 1 for any same-class pairs of points).

However, the desired relationships need not be discrete; work in this area also derives continuous semantic *distance* functions from class information or other annotations, specifying a richer set of constraints that ought to be preserved by the binary codes. For example, Fergus et al. explore using the distance between classes in WordNet to quantify their semantic distance [20]. Or, rather than enforce similarity only between pairs of classes, one can incorporate a desired similarity between individual images (e.g., by collecting image-level constraints from online annotators).

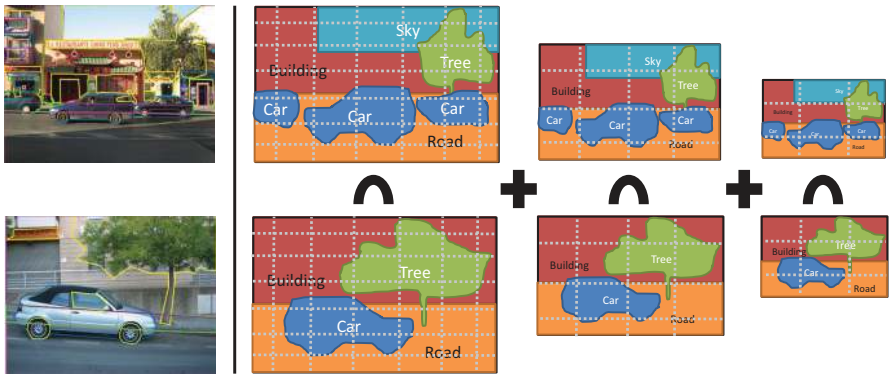

Fig. 7 Left: Where dense annotations are available, they can be used to define a semantic similarity metric between images. In this pair of images from LabelMe, users have labeled pixels as belonging to different objects like cars, roads, tree, sky and so on. Right: The semantic similarity between training images is obtained by computing the intersection of the spatial pyramid histogram built on the object category *label maps* of the two images. The same objects in the same positions produce the highest similarity, and the score degrades smoothly as the locations of the objects differ. See [65] for details.

Additionally, if pixel-level labels exist, as opposed to image-level ones, then more fine-grained measures can be used. Torralba et al. [65] define ground truth semantic similarity based a spatial pyramid matching [27, 41] scheme on the object label maps, as illustrated in Figure 7. This results in a simple similarity measure that takes into account the objects present in the image as well as their spatial organization: two images that have the same object labels in similar spatial locations are rated as closer than two images with the same objects but in different spatial locations, and either case is rated closer than two images with different object classes.

3.2 Hash Functions for Learned Mahalanobis Kernels

Having defined possible sources of supervision, we now describe how those semantics are integrated into binary code learning. We first review a hashing-based algorithm for learned Mahalanobis metrics introduced by Jain and colleagues in [37, 40]. The main idea is to learn a parameterization of a Mahalanobis metric (or kernel) based on provided labels or paired constraints for some training examples, while simultaneously encoding the learned information into randomized hash functions. These functions will guarantee that the more similar inputs are under the learned metric, the more likely they are to collide in a hash table. After indexing all of the database examples with their learned hash keys, those examples similar to a new instance are found in sub-linear time via hashing.

Learned Mahalanobis Metrics and Kernels The majority of work in metric learning focuses on learning Mahalanobis metrics (e.g., [73, 71, 26, 5, 19]). Given

N points $\{\boldsymbol{x}_1,\ldots,\boldsymbol{x}_N\}$, with all $\boldsymbol{x}_i \in \Re^d$, the idea is to compute a positive-definite (p.d.) $d \times d$ matrix A to parameterize the squared Mahalanobis distance:

$$d_A(\boldsymbol{x}_i,\boldsymbol{x}_j) = (\boldsymbol{x}_i - \boldsymbol{x}_j)^T A(\boldsymbol{x}_i - \boldsymbol{x}_j), \tag{2}$$

for all $i, j = 1,\ldots,N$. Note that a generalized inner product (kernel) measures the pairwise similarity associated with that distance:

$$s_A(\boldsymbol{x}_i,\boldsymbol{x}_j) = \boldsymbol{x}_i^T A \boldsymbol{x}_j. \tag{3}$$

The Mahalanobis distance is often used with A as the inverse of the sample covariance when data is assumed to be Gaussian, or with A as the identity matrix if the squared Euclidean distance is suitable.

Let $\mathscr{S}$ and $\mathscr{D}$ denote sets containing pairs of points constrained to be similar and dissimilar, respectively. Given these similarity constraints, one can *learn* the matrix A to yield a measure that is more accurate for a given problem.

For example, Xing et al. learn a Mahalanobis metric by using semidefinite programming to minimize the sum of squared distances between similarly labeled examples, while requiring a certain lower bound on the distances between examples with different labels [73]. In related techniques, Globerson and Roweis [24] constrain within-class distances to be zero and maximize between-class distances, Weinberger et al. formulate the problem in a large-margin k-nearest-neighbors setting [71], while Goldberger et al. maximize a stochastic variant of leave-one-out KNN score on the training set [26]. In addition to using labeled data, research has shown how metric learning can proceed with weaker supervisory information, such as equivalence constraints or relative constraints. For example, equivalence constraints are exploited in the Relevant Component Analysis method of Bar-Hillel et al. [5]; the method of Hadsell et al. [29] learns a global non-linear mapping of the input data; the Support Vector Machine-based approach of Schultz and Joachims [59] incorporates relative constraints over triples of examples. Davis et al. develop an information-theoretic approach that accommodates any linear constraints on pairs of examples, and provide an efficient optimization solution that forgoes eigenvalue decomposition [19].

Main Idea To use a learned Mahalanobis metric for search, we want to retrieve examples $\boldsymbol{x}_i$ for an input $\boldsymbol{x}_q$ for which the value $d_A(\boldsymbol{x}_i,\boldsymbol{x}_q)$ resulting from Eqn. (2) is small—or, in terms of the kernel form, for which the value of $s_A(\boldsymbol{x}_i,\boldsymbol{x}_q) = \boldsymbol{x}_q^T A \boldsymbol{x}_i$ is high. We next describe how to generate hash functions for the Mahalanobis similarity (1) in the *explicit* case, where the dimensionality of the data is low enough that the $d \times d$ matrix A can be handled in memory, and (2) in the *implicit* case, where A cannot be accessed directly and we want to use a kernelized form of metric learning.

Explicit Formulation In [12], Charikar proposes a hash function family that is locality-sensitive for the normalized inner product (cosine similarity):

$$\mathrm{sim}(\boldsymbol{x}_i,\boldsymbol{x}_j) = \frac{\boldsymbol{x}_i^T \boldsymbol{x}_j}{\|\boldsymbol{x}_i\|_2 \|\boldsymbol{x}_j\|_2}. \tag{4}$$

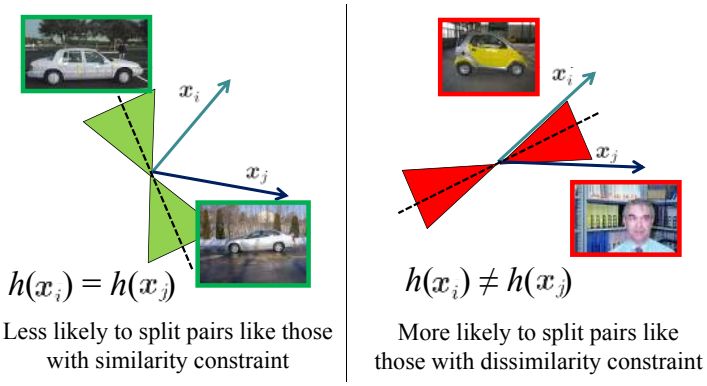

Fig. 8 Whereas traditional unsupervised LSH functions would generate a hyperplane uniformly at random to separate instances, randomized hash functions for a learned kernel are biased so as to ensure similar things become more likely to collide, while dissimilar things become less likely to collide. The hourglass-shaped regions denote that the hash function will be more likely to drawn as such. In the left example, even though the measured angle between x_i and x_j is somewhat wide, the learned hash functions are unlikely to split them into different buckets since the constraints indicate that they (and pairs like them) should be treated as similar.

Each hash function simply rounds the output of a product with a random hyperplane:

$$h_{\boldsymbol{r}}(\boldsymbol{x}) = \begin{cases} 1, & \text{if } \boldsymbol{r}^T\boldsymbol{x} \geq 0 \\ 0, & \text{otherwise} \end{cases}, \tag{5}$$

where $\boldsymbol{r}$ is sampled from a zero-mean multivariate Gaussian $\mathcal{N}(0,I)$ of the same dimensionality as the input $\boldsymbol{x}$. The fact that this hash function satisfies the LSH requirement $\Pr[h(\boldsymbol{x}_i) = h(\boldsymbol{x}_j)] = \text{sim}(\boldsymbol{x}_i, \boldsymbol{x}_j)$ relies on a result from Goemans and Williamson [25], who showed that

$$\Pr[\text{sign}(\boldsymbol{x}_i^T\boldsymbol{r}) = \text{sign}(\boldsymbol{x}_j^T\boldsymbol{r})] = 1 - \frac{1}{\pi}\cos^{-1}\left(\frac{\boldsymbol{x}_i^T\boldsymbol{x}_j}{\|\boldsymbol{x}_i\|\|\boldsymbol{x}_j\|}\right), \tag{6}$$

for vectors on the unit sphere. This relationship is quite intuitive: the wider the angle between two vectors $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$, the more likely a randomly selected hyperplane will fall between them, and vice versa.

As shown by Jain et al. [37], this is easily extended to accommodate learned Mahalanobis distances. Given the p.d. matrix A, with $A = G^TG$, we generate the following randomized hash functions $h_{\boldsymbol{r},A}$, which accept an input point and return a single hash key bit:

$$h_{\boldsymbol{r},A}(\boldsymbol{x}) = \begin{cases} 1, & \text{if } \boldsymbol{r}^TG\boldsymbol{x} \geq 0 \\ 0, & \text{otherwise} \end{cases}, \tag{7}$$

where the vector $\boldsymbol{r}$ is again chosen at random from a d-dimensional Gaussian distribution with zero mean and unit variance.[4] By parameterizing the hash functions by not only $\boldsymbol{r}$ but also A, we obtain the following relationship:

$$\Pr[h_{\boldsymbol{r},A}(\boldsymbol{x}_i) = h_{\boldsymbol{r},A}(\boldsymbol{x}_j)] = 1 - \frac{1}{\pi}\cos^{-1}\left(\frac{\boldsymbol{x}_i^T A \boldsymbol{x}_j}{\sqrt{|G\boldsymbol{x}_i||G\boldsymbol{x}_j|}}\right),$$

which fulfills the LSH requirement of Eqn. (1) for a metric obtained with any of the Mahalanobis metric learning algorithms. Essentially we have biased the selection of the random hyperplane according to the learned parameters, and by factoring it by G we allow the random hash function itself to "carry" the information about the learned metric. See Figure 8. The denominator in the cosine term normalizes the learned kernel values.

Implicit Formulation Beyond the explicit formulation given above, we are also interested in the case where the dimensionality d may be very high—say on the order of 10^4 to 10^6—but the examples are sparse and therefore can be stored efficiently. For example, bag of visual word descriptors or histogram pyramids often require millions of dimensions [63, 27, 51]. Even though the examples are themselves sparse and therefore compactly represented, the matrix A can be dense.

In this case, we turn to a particular information-theoretic metric learning (ITML) algorithm developed by Davis et al. [19]. In contrast to most other Mahalanobis metric learning approaches, it is kernelizable. It takes an initial "base" parameterization A_0 as input, and then during the learning process it computes *implicit* updates to those parameters, using weighted kernel evaluations between pairs of points involved in the similarity constraints (as opposed to explicit multiplication with A). We briefly summarize the relevant portions of the ITML approach; see [19] for more details.

Information-Theoretic Metric Learning Given an initial $d \times d$ p.d. matrix A_0 specifying prior knowledge about inter-point distances, the learning task is posed as an optimization problem that minimizes the LogDet loss between A_0 and the ultimate learned parameters A, subject to a set of constraints specifying pairs of examples that are similar or dissimilar (listed in the sets $\mathscr{S}$ and $\mathscr{D}$):

$$\begin{aligned} \min_{A \succeq 0} \quad & D_{\ell\mathrm{d}}(A, A_0) \\ \text{s. t.} \quad & d_A(\boldsymbol{x}_i, \boldsymbol{x}_j) \leq u \quad (i,j) \in \mathscr{S}, \\ & d_A(\boldsymbol{x}_i, \boldsymbol{x}_j) \geq \ell \quad (i,j) \in \mathscr{D}, \end{aligned} \tag{8}$$

where $D_{\ell\mathrm{d}}(A, A_0) = \mathrm{tr}(AA_0^{-1}) - \log\det(AA_0^{-1}) - d$, d is the dimensionality of the data points, $d_A(\boldsymbol{x}_i, \boldsymbol{x}_j)$ is the Mahalanobis distance between $\boldsymbol{x}_i$ and $\boldsymbol{x}_j$ as defined

[4] In this case—where A can be explicitly handled in memory—we could equivalently transform all the data according to A *prior* to hashing; however, the choice of presentation here helps set up the formulation presented next.

in Eqn. (2), and ℓ and u are large and small values, respectively.[5] The objective is "information-theoretic" in that it corresponds to minimizing the relative entropy between the associated Gaussians whose covariance matrices are parameterized by A and A_0.

The LogDet loss leads to an efficient algorithm for optimizing Eqn. (8), which involves repeatedly projecting the current solution onto a single constraint, via the explicit update [19]:

$$A_{t+1} = A_t + \beta_t A_t (\boldsymbol{x}_{i_t} - \boldsymbol{x}_{j_t})(\boldsymbol{x}_{i_t} - \boldsymbol{x}_{j_t})^T A_t, \tag{9}$$

where $\boldsymbol{x}_{i_t}$ and $\boldsymbol{x}_{j_t}$ are the constrained data points for iteration t, and β_t is a projection parameter computed (in closed form) by the algorithm.

However, when the dimensionality of the data is very high, one cannot explicitly work with A, and so the update in Eqn. (9) is impractical. Instead, it is replaced with updates in kernel space for an equivalent kernel learning problem in which $K = X^T A X$ for $X = [\boldsymbol{x}_1, \ldots, \boldsymbol{x}_c]$, for a small set of c of the points involved in similarity constraints (see [40]). If K_0 is the input kernel matrix for the data ($K_0 = X^T A_0 X$), then the appropriate update is:

$$K_{t+1} = K_t + \beta_t K_t (\boldsymbol{e}_{i_t} - \boldsymbol{e}_{j_t})(\boldsymbol{e}_{i_t} - \boldsymbol{e}_{j_t})^T K_t, \tag{10}$$

where the vectors $\boldsymbol{e}_{i_t}$ and $\boldsymbol{e}_{j_t}$ refer to the i_t-th and j_t-th standard basis vectors, respectively. This update is derived by multiplying Eqn. (9) on the left by X^T and on the right by X. If $A_0 = I$, then the initial kernel matrix is $K_0 = X^T X$; this matrix may be formed using any valid kernel function, and the result of the algorithm is to learn a distance metric on top of this input kernel. By performing the updates in kernel space, the storage requirements change from $O(d^2)$ to $O(c^2)$.

Simultaneous Metric and Hash Function Updates In order to permit large-scale search with such metrics, the goal is to use the same hash functions as defined above in Eqn. (7), but to express them in a form that is amenable to computing the hash bit with high-dimensional input data. In other words, we want to insert the learned parameters into the hash function and compute $\boldsymbol{r}^T G \boldsymbol{x}$, but now we must do so without working directly with G. To this end, we describe next how to simultaneously make implicit updates to both the hash functions and the metric.

In [37], Jain et al. show how to express G in terms of the initially chosen c data points. Let $X = [\boldsymbol{x}_1, \ldots, \boldsymbol{x}_c]$ be the $d \times c$ matrix of an initial c data points participating in (dis)similarity constraints, and let $\boldsymbol{x}_i^T \boldsymbol{x}_j$ be the initial (non-learned) Mahalanobis similarity value between example $\boldsymbol{x}_i$ and the input $\boldsymbol{x}_j$. Recall the update rule for A from Eqn. (9): $A_{t+1} = A_t + \beta_t A_t \boldsymbol{v}_t \boldsymbol{v}_t^T A_t$, where $\boldsymbol{v}_t = \boldsymbol{y}_t - \boldsymbol{z}_t$, if points $\boldsymbol{y}_t$ and $\boldsymbol{z}_t$ are involved in the constraint under consideration at iteration t. Just as this update must be implemented implicitly via Eqn. (10), so too we must derive an *implicit* update for the G_t matrix required by our hash functions. Since A_t is p.d., we can factorize

[5] Note that alternatively the constraints may also be specified in terms of relative distances, i.e., $d_A(\boldsymbol{x}_i, \boldsymbol{x}_j) < d_A(\boldsymbol{x}_i, \boldsymbol{x}_k)$. To guarantee the existence of a feasible A, slack variables are also included, but omitted here for brevity.

it as $A_t = G_t^T G_t$, which allows us to rewrite the update as:

$$A_{t+1} = G_t^T (I + \beta_t G_t \boldsymbol{v}_t \boldsymbol{v}_t^T G_t^T) G_t.$$

As a result, factorizing $I + \beta_t G_t \boldsymbol{v}_t \boldsymbol{v}_t^T G_t^T$, we can derive an update for G_{t+1}:

$$\begin{aligned} G_{t+1} &= (I + \beta_t G_t \boldsymbol{v}_t \boldsymbol{v}_t^T G_t^T)^{1/2} G_t \\ &= (I + \alpha_t G_t \boldsymbol{v}_t \boldsymbol{v}_t^T G_t^T) G_t, \end{aligned} \tag{11}$$

where the second equality follows from Lemma 1 in [40] using $\boldsymbol{y} = G_t \boldsymbol{v}_t$, and α_t is defined accordingly.

Using Eqn. (11) and Lemma 2 in [40], G_t can be expressed as $G_t = I + X S_t X^T$, where S_t is a $c \times c$ matrix of coefficients that determines the contribution of each of the c points to G. Initially, S_0 is set to be the zero matrix, and from there every S_{t+1} is iteratively updated in $O(c^2)$ time via

$$S_{t+1} = S_t + \alpha_t (I + S_t K_0)(\boldsymbol{e}_{i_t} - \boldsymbol{e}_{j_t})(\boldsymbol{e}_{i_t} - \boldsymbol{e}_{j_t})^T (I + K_0 S_t^T)(I + K_0 S_t).$$

Using this result, at convergence of the metric learning algorithm we can compute $G\boldsymbol{x}$ in terms of the c^2 input pairs $(\boldsymbol{x}_i, \boldsymbol{x}_j)$ as follows:

$$\begin{aligned} G\boldsymbol{x} &= \boldsymbol{x} + X S X^T \boldsymbol{x} \\ &= \boldsymbol{x} + \sum_{i=1}^{c} \sum_{j=1}^{c} S_{ij} \boldsymbol{x}_j \boldsymbol{x}_i^T \boldsymbol{x}. \end{aligned}$$

Therefore, we have

$$\boldsymbol{r}^T G\boldsymbol{x} = \boldsymbol{r}^T \boldsymbol{x} + \sum_{i=1}^{c} \sum_{j=1}^{c} S_{ij} \boldsymbol{r}^T \boldsymbol{x}_j \boldsymbol{x}_i^T \boldsymbol{x}, \tag{12}$$

and the final implicit hash function $h_{\boldsymbol{r},A}$ for an input $\boldsymbol{x}$ can be defined as:

$$h_{\boldsymbol{r},A}(\boldsymbol{x}) = \begin{cases} 1, & \text{if } \boldsymbol{r}^T \boldsymbol{x} + \sum_{i=1}^{c} \gamma_i^r \boldsymbol{x}_i^T \boldsymbol{x} \geq 0 \\ 0, & \text{otherwise} \end{cases}, \tag{13}$$

where each $\gamma_i^r = \sum_j S_{ij} \boldsymbol{r}^T \boldsymbol{x}_j$.

There are several important things to notice about the ultimate hash function definition. First, we see that the values of each γ_i^r rely only on the basis points, and thus can be efficiently computed in the training phase, prior to hashing anything into the database. Second, the summation consists of as many terms as there are *basis* constrained points c—not the total number of constraints used during metric learning, nor the number of total database points. This is particularly important at query time, when we want the overhead of computing the query's hash key to be low (certainly, it must not require comparing to each database point!) Third, we emphasize that while G is dense and therefore $\boldsymbol{r}^T G$ is not manageable, this method does assume that computing $\boldsymbol{r}^T \boldsymbol{x}$ *is* manageable; for sparse data, only the entries of

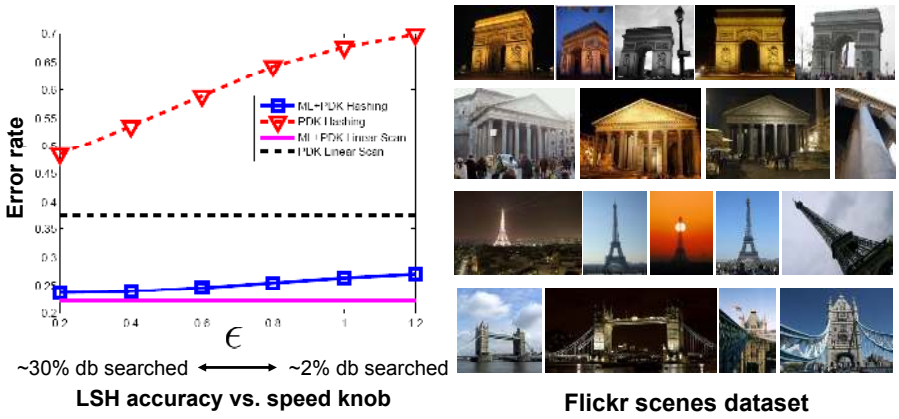

Fig. 9 Hashing error relative to an exhaustive linear scan as a function of ε, which controls the search time required. The error rate of exhaustive search decreases substantially once the similarity constraints on labeled data are used to learn a matching kernel (compare flat dotted black line to flat solid pink line). When these constraints are integrated into the hash function selection, we can obtain similar results to the linear scan, but searching just a fraction of the data (compare ML+PDK Hashing curve to ML+PDK Linear Scan line). Result is from [40].

$\boldsymbol{r}$ corresponding to non-zero entries in $\boldsymbol{x}$ need to be generated. [6] Finally, in practice, it is a strength of the ITML metric learning formulation that one may provide an initial parameterization A_0 based on any *a priori* knowledge, refining it with the external similarity constraints. For example, one could initialize with a pyramid match kernel, and refine it with the similarity constraints.

Figure 9 shows an example result using these supervised LSH functions to index Flickr images of 18 different tourist sites, where the images are represented with sets of local SIFT [45] features. Using a base matching kernel [43] as input, the algorithm learns a Mahalanobis kernel on top of it, and simultaneously updates the LSH function parameterization. Then a nearest-neighbor scene classification task is posed. The results illustrate the nice control one has on the balance between (i) the search time required and (ii) the accuracy guaranteed relative to a linear scan, as determined by the LSH parameter ε (discussed above in Section 2.3). Searching only about 2% of the database, we see error levels similar to that of an exhaustive linear scan with the learned metric.

Interestingly, this plot also reveals that the learning stage has a dimensionality reduction effect. While both hashing curves use the same number of hash bits b, the learned hash functions more closely approximate the associated linear scan result. (Compare the relative gaps between the top two curves and the bottom two curves, respectively.) We can attribute this to the fact that the learned hash functions usefully focus the partitions' placement in the original feature space, requiring fewer bits for the same representational power.

[6] Thus it is particularly efficient when the inputs are sparse. If they are high-dimensional but dense, the implicit form is still valuable, as it bypasses computing $O(d^2)$ products with G and requires only $O(d)$ inner products for $\boldsymbol{r}^T\boldsymbol{x}$.

3.3 Learned Binary Embeddings for Semantic Similarity

The previous section reviewed methods for hashing with learned Mahalanobis metrics, where the similarity constraints are used to establish a linear transformation of the original input space (possibly implicitly computed). In this section we consider other more general types of transformations that can be constructed using labeled training examples. The main idea is to exploit supervised learning algorithms to aggregate a set of functions that jointly preserve the desired neighborhood structure. Whereas the learned kernel technique above generates locality-sensitive hash functions, these techniques generate Hamming embeddings (refer back to Section 2 for contrasts).

Specifically, the methods in this section address the following learning problem: given a database of images $\{\boldsymbol{x}_i\}$ and a distance function $d(\boldsymbol{x}_i,\boldsymbol{x}_j)$ we seek a binary feature vector $\boldsymbol{y}_i = f(\boldsymbol{x}_i)$ that preserves the nearest neighbor relationships using a Hamming distance. Formally, for a point $\boldsymbol{x}_i$, denote by $\mathcal{N}_{100}(\boldsymbol{x}_i)$ the indices of the 100 nearest neighbors of $\boldsymbol{x}_i$ according to a semantic distance function $d(\boldsymbol{x}_i,\boldsymbol{x}_j)$ derived using one of the supervision forms described in Section 3.1. Similarly, define $\mathcal{N}_{100}(\boldsymbol{y}_i)$ to be the set of indices of the 100 descriptors $\boldsymbol{y}_j$ that are closest to $\boldsymbol{y}_i$ in terms of Hamming distance. Ideally, we would like $\mathcal{N}_{100}(\boldsymbol{x}_i) = \mathcal{N}_{100}(\boldsymbol{y}_i)$ for all examples in our training set.

We discuss two general approaches to learning the explicit mapping functions. The first is a variant of the *Parameter Sensitive Hashing* (PSH) algorithm of Shakhnarovich et al., which uses boosting and a rounding-based LSH function to select feature dimensions that are most indicative of similarity in some parameter space of interest (e.g., human pose joint angles in their application) [61, 60]. The second is a neural network-based approach explored by Salakhutdinov and Hinton [57] and Torralba et al. [65], the former being used for document retrieval. These models utilize a form of unsupervised pre-training using a stack of restricted Boltzmann machines (RBMs).

3.3.1 Boosting-based Embedding

In Shakhnarovich et al. [61], each image is represented by a binary vector with b bits $\boldsymbol{y}_i = [h_1(\boldsymbol{x}_i), h_2(\boldsymbol{x}_i), ..., h_b(\boldsymbol{x}_i)]$, so that the distance between two images is given by a weighted Hamming distance $d(\boldsymbol{x}_i,\boldsymbol{x}_j) = \sum_{l=1}^{b} \alpha_l |h_l(\boldsymbol{x}_i) - h_l(\boldsymbol{x}_j)|$. The weights α_i and the functions $h_l(\boldsymbol{x}_i)$ are binary regression stumps that map the input vector $\boldsymbol{x}_i$ into binary features and are learned using Boosting.

For the learning stage, positive examples are pairs of images $\boldsymbol{x}_i,\boldsymbol{x}_j$ so that $\boldsymbol{x}_j$ is one of the nearest neighbors of $\boldsymbol{x}_i$, $j \in \mathcal{N}(\boldsymbol{x}_i)$. Negative examples are pairs of images that are not neighbors. In our implementation we use GentleBoost with regression stumps to minimize the exponential loss. In PSH, each regression stump has the form:

$$f_l(\boldsymbol{x}_i,\boldsymbol{x}_j) = \alpha_l \left[(e_l^T \boldsymbol{x}_i > T_l) = (e_l^T \boldsymbol{x}_j > T_l) \right] + \beta_l. \quad (14)$$

At each iteration l, we select the parameters of f_l, the regression coefficients (α_l, β_l), the stump parameters (where e_l is a unit vector, so that $e_l^T \boldsymbol{x}$ returns the lth component of $\boldsymbol{x}$, and T_l is a threshold), to minimize the square loss:

$$\sum_{n=1}^{N} w_l^n (z_n - f_l(\boldsymbol{x}_i^n, \boldsymbol{x}_j^n))^2, \tag{15}$$

where N is the number of training pairs, z_n is the neighborhood label ($z_n = 1$ if the two images are neighbors and $z_n = -1$ otherwise), and w_l^n is the weight for each training pair at iteration l given by $w_l^n = \exp(-z_n \sum_{t=1}^{l-1} f_l(\boldsymbol{x}_i^n, \boldsymbol{x}_j^n))$.

In Torralba et al. [65] the authors constrain the metric to be a Hamming distance, restricting the class of weak learners so that all the weights are the same for all the features $\alpha_l = \alpha$. (The values of β_l do not need to be constrained as they only contribute to final distance as a constant offset, independent of the input pair.) This small modification is important as it permits standard Hashing techniques to be used. The parameter α has an effect in the generalization of the final function.

Once the learning stage is finished, every image can be compressed into b bits, where each bit is computed as $h_l(\boldsymbol{x}_i) = e_l^T \boldsymbol{x}_i > T_l$. The algorithm is simple to code, and relatively fast to train.

3.3.2 Restricted Boltzmann Machines-based Embedding

The second approach uses the dimensionality reduction framework of Salakhutdinov and Hinton [32, 57], based on multiple layers of restricted Boltzmann machines (RBMs). We first give a brief overview of RBM's, before describing their use in Torralba et al. [65] where they are applied to images.

An RBM models an ensemble of binary vectors with a network of stochastic binary units arranged in two layers, one visible, one hidden. Units $\mathbf{v}$ in the visible layers are connected via a set of symmetric weights W to units $\mathbf{h}$ in the hidden layer. The joint configuration of visible and hidden units has an energy:

$$E(\mathbf{v}, \mathbf{h}) = -\sum_{i \in \text{visible}} b_i v_i - \sum_{j \in \text{hidden}} b_j h_j - \sum_{i,j} v_i h_i w_{ij}, \tag{16}$$

where v_i and h_j are the binary states of visible and hidden units i and j. The weights are denoted by w_{ij}, and b_i and b_j are bias terms, also model parameters. Using this energy function, a probability can be assigned to a binary vector at the visible units:

$$p(\mathbf{v}) = \sum_{\mathbf{h}} \frac{e^{-E(\mathbf{v},\mathbf{h})}}{\sum_{\mathbf{u},\mathbf{g}} e^{-E(\mathbf{u},\mathbf{g})}}. \tag{17}$$

RBMs lack connections between units within a layer, hence the conditional distributions $p(\mathbf{h}|\mathbf{v})$ and $p(\mathbf{v}|\mathbf{h})$ have a convenient form, being products of Bernoulli distributions:

$$p(h_j = 1|\mathbf{v}) = \sigma(b_j + \sum_i w_{ij} v_i)$$
$$p(v_i = 1|\mathbf{h}) = \sigma(b_i + \sum_j w_{ij} h_j), \quad (18)$$

where $\sigma(u) = 1/(1+e^{-u})$, the logistic function. Using Eqn. 18, parameters w_{ij}, b_i, b_j can be updated via a contrastive divergence sampling scheme (see [32] for details). This ensures that the training samples have a lower energy than nearby hallucinations, samples generated synthetically to act as negative examples.

Hinton and colleagues have demonstrated methods for stacking RBMs into multiple layers, creating "deep" networks which can capture high order correlations between the visible units at the bottom layer of the network. By choosing an architecture that progressively reduces the number of units in each layer, a high-dimensional binary input vector can be mapped to a far smaller binary vector at the output. Thus each bit at the output maps through multiple layers of non-linearities to model the complicated subspace of the input data.

Since the input descriptors will typically be real-valued, rather than binary (e.g. Gist or SIFT descriptors), the first layer of visible units are modified to have a Gaussian distribution.[7]

The deep network is trained into two stages: first, an unsupervised *pre-training* phase which sets the network weights to approximately the right neighborhood; second, a *fine-tuning* phase where the network has its weights moved to the local optimum by back-propagation on labeled data.

In pre-training, the network is trained from the visible input layer up to the output layer in a greedy fashion. Once the parameters of the first layer have converged using contrastive divergence, the activation probabilities (given in Eqn. 18) of the hidden layer are fixed and used as data for the layer above—the hidden units becoming the visible ones for the next layer up, and so on up to the top of the network.

In fine-tuning, the units are made deterministic, retaining the weights and biases from pre-training and performing gradient descent on them using back-propagation. One possible objective function is Neighborhood Components Analysis (NCA) [26, 56]. This attempts to preserve the semantic neighborhood structure by maximizing the number of neighbors around each query that have the same class labels. Given N labeled training cases $(\mathbf{x}^n, c^n)$, denote the probability that point n is assigned the class of point m as p_{nm}. The objective O_{NCA} attempts to maximize the expected number of correctly classified points on the training data:

$$O_{\text{NCA}} = \sum_{n=1}^{N} \sum_{l:c^n=c^l} p_{nm}, \; p_{nm} = \frac{e^{-||f(\mathbf{x}^n|W) - f(\mathbf{x}^l|W)||^2}}{\sum_{m \neq l} e^{-||f(\mathbf{x}^m|W) - f(\mathbf{x}^l|W)||^2}},$$

where $f(\mathbf{x}|W)$ is the projection of the data point $\mathbf{x}$ by the multi-layered network with parameters W. This function can be minimized by taking derivatives of O_{NCA} with

[7] In Eqn. 18, $p(v_i = u|\mathbf{h})$ is modified to be a Gaussian with a mean determined by the hidden units; see [56].

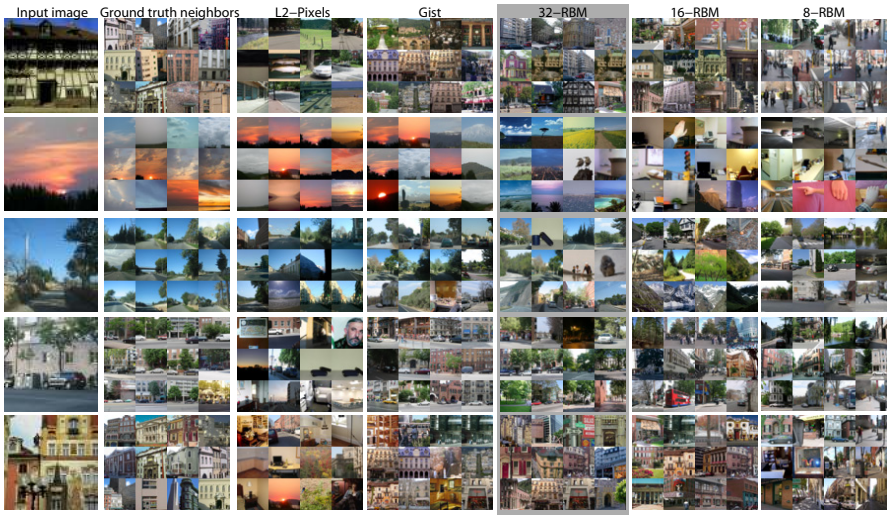

Fig. 10 Each row shows the input image and the 12 nearest neighbors using, a) ground truth distance using the histograms of objects present on each image (see text), b) L2 distance using RGB values, c) L2 distance using Gist descriptors, d) Gist features compressed to 8 bits using an RBM and Hamming distance, e) 16 bits RBM and f) 32 bits RBM.

respect to W and using conjugate gradient descent. Alternative objective functions include the DrLIM objective introduced by Hadsell et al. [29].

Figure 10 shows representative retrieval results on a 20,000 LabelMe dataset. Gist descriptors [50] are used as the high-dimensional input representation for each image (single descriptor per image). Figure 11 provides a quantitative analysis of the retrieval performance on 2,000 test images. Figure 11(a) displays the percentage of the first true 50 nearest neighbors that are included in the retrieved set as a function of the number of the images retrieved (M). Figure 11(b) shows a section of Figure 11(a) for 500 retrieved images. The figures compare LSH (with no learning), PSH and RBMs. Figure 11(b) shows the effect of increasing the number of bits. Top performance is reached with around 30 bits for RBMs, with the other methods requiring more bits. However, given enough bits, all the approaches converge to similar retrieval performance. The matching speed using the binary codes and the Gist descriptors is shown in Figure 2, where the compact codes facilitate extremely fast retrieval.

Figure 11(a) also shows a comparison with the more conventional *kd*-tree based methods. Here the FLANN [47] *kd*-tree implementation was applied to the Gist descriptors (converted to uint8), both with (black) and without (cyan) a preliminary PCA projection down to 128 dimensions. To give a fair comparison, the *kd*-tree parameters were adjusted to give a comparable retrieval time to the other methods. The performance can be seen to be considerably worse than the approaches using binary codes. This is due to the poor performance of *kd*-tree type approaches in high dimensional spaces.

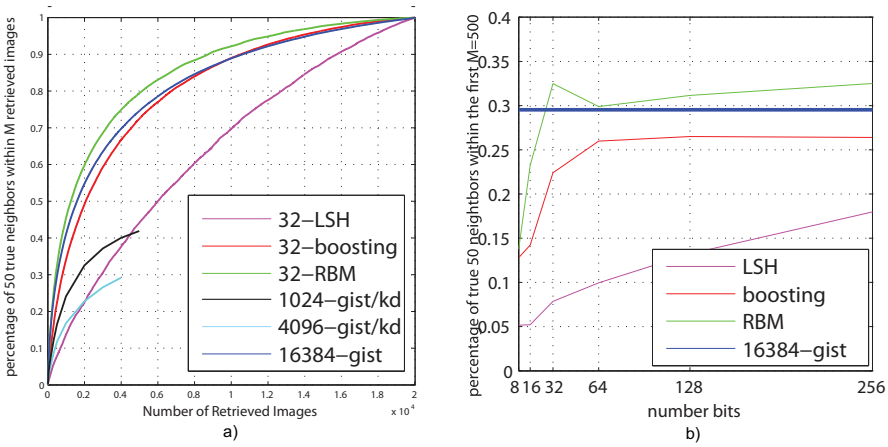

Fig. 11 **(a):** For a fixed number of true neighbors ($|\mathcal{N}| = 50$), we plot the percentage of true nearest neighbors that are retrieved as a function of the total number of images retrieved. True neighbors are defined in terms of object label histograms (see Figure 7). The original Gist descriptors (blue) perform well but a slow to match due to high dimensionality. LSH (magenta), which does not use learning, performs less well than Boosting (red) and the RBM-based (green) methods. The Boosting and RBM-based embeddings, despite using only 32-bits per descriptor match the performance of the original Gist. **(b):** Varying the number of bits for 500 retrieved images.

3.4 Other Supervised Methods

Building on the ideas presented thus far, recent work has explored alternative methods to learn hash functions. Wang et al. propose a supervised form of PCA, where pairwise binary labels act as constraints on the projection [70]. Projecting new examples with this approach requires a Nystrom-based out-of-sample projection. Mu and colleagues develop a kernel-based maximum margin approach to select hash functions [46], and a semi-supervised approach that minimizes empirical error on a labeled constraint set while promoting independence between bits and balanced partitions is described in [69]. The SPEC hashing approach [42] uses a conditional entropy measure to add binary functions in a way that matches a desired similarity function, but approximately so as to ensure linear run-time.

Jain and colleagues develop a dynamic hashing idea to accommodate metrics learned in an online manner, where similarity constraints are accumulated over time rather than made available at once in batch [36]. Bronstein and colleagues extend the idea of learning binary similarity functions with boosting to the cross-modal case, where data comes from two different input spaces (e.g., we want to judge the similarity between a CT and a PET image) [11].

Whereas the technique in Section 3.2 above connects Mahalanobis and ITML-learned kernels to LSH, the Kernelized LSH approach developed by Kulis & Grauman provides a connection for *arbitrary* kernel functions [39], which includes ker-

nels learned with ITML or otherwise. We will review this method in Section 4.3, since it can be applied in both supervised and unsupervised settings.

4 Unsupervised Methods for Defining Binary Projections

In the previous section, we reviewed supervised algorithms that require some form of label information to define which points should be close by and which should be far apart in the binary space. We now look at unsupervised techniques that simply try to preserve the neighborhood structure between points in the input space, and thus require no labels.

The goal is to compute a binary representation of the original representation for each image, so that similar instances have similar binary codes. Typically the original feature space and distance are Euclidean, but alternatives are possible, as discussed in Sections 4.1 and 4.3 below. Additionally, we want the code to be easily computed for a novel input and to be compact in length, thus enabling efficient methods such as Semantic Hashing (see Section 2.2) to be used.

We first briefly summarize several methods for specific similarity functions (Section 4.1), then discuss a spectral approach for coding the Euclidean distance on real-valued vector data (Section 4.2), and finally review an approach to generate codes for kernel functions, including those over non-vector data (Section 4.3).

4.1 Binary Codes for Specific Similarity Functions

Several embedding functions that map a specialized distance into a generic space (e.g., Euclidean) have been developed to exploit either hashing or Hamming space search for particular metrics of interest. In order to exploit known LSH functions [17], Indyk and Thaper design a low-distortion L_1 embedding for the bijective match distance between two sets of feature vectors [34]. Grauman and Darrell construct a related embedding for the normalized partial match, showing that an implicit unary encoding with a linear kernel is equivalent to the pyramid match kernel on feature sets [28], thereby allowing hashing with the function in Eqn. (5). A related embedding is given in [40] for the proximity distribution kernel [43], which is an image matching kernel that accounts for both the correspondence between features and their relative spatial layout.

The Min-Hash technique introduced by Broder [10] is a randomized embedding designed to capture the normalized set overlap: $\text{sim}(S_1,S_2) = \frac{|S_1 \bigcap S_2|}{|S_1 \bigcup S_2|}$, for sets S_1 and S_2. Assuming we have a discrete set (vocabulary) of tokens that may appear in any set, we generate a random permutation of those unique tokens. Then, given any input set, its Min-Hash key is the token present in the set that has the minimum value (appears first) in the permutation. The probability that two sets receive the same Min-Hash value is equal to the overlap similarity. Intuitively, the permutations

correspond to picking some token from the sets' union at random, and checking whether both contain it. The higher the overlap between two sets, the more likely we draw a token they share. Similar to the hash table construction process described in Section 2.3, one concatenates multiple such hash values to generate a hash key, and aggregates results over multiple independently generated functions. The set overlap is meaningful for gauging document similarity, and Min-Hash was initially used for efficient near-duplicate detection among Web pages. Chum and colleagues have shown its suitability for near-duplicate detection for bag of words image representations as well, and adapt the idea to include the effective tf-idf weighting [14], and to integrate spatial layout of features into the hash selection [13].

Notably, all of the above projection techniques cater to *sets* of features, where each instance is comprised of some variable number of descriptors, and the desired distance computes some matching or overlap between them. Such techniques' success for image search applications is a result of the strong *local feature* representations used widely in the recognition and CBIR communities in the last decade.

Aside from set-based metrics, Rahimi and Recht design embeddings for a particular form of shift-invariant kernel. They propose randomized mappings into a real-valued low-dimensional feature space such that an inner product approximates a given shift-invariant kernel, such as the Gaussian or Laplacian kernel [53]. Note that while the intent in that work is to exploit linear machine learning algorithms that permit fast training with large-scale data (as opposed to search), one could take such an embedding and again use the randomized hyperplane hash functions (Eqn. (5)). Raginsky and Lazebnik also show how to convert those real-valued mappings to binary outputs so that Hamming space search is applicable, with bounds on the expected normalized Hamming distance relative to the original shift-invariant kernel value [52].

4.2 Spectral Hashing

Whereas the above section addresses unsupervised codes developed for particular similarity functions of interest, we now examine a technique that not only aims to preserve the given similarities, but also attempts to satisfy generic properties that make compact binary codes effective. This is the Spectral Hashing framework developed by Weiss et al. [72].

In formalizing the requirements for a good code, we see that they are equivalent to a particular form of graph partitioning. This means that even for a single bit, the problem of finding optimal codes is NP hard. On the other hand, the analogy to graph partitioning suggests a relaxed version of the problem that leads to very efficient eigenvector solutions. These eigenvectors are exactly the eigenvectors used in many spectral algorithms including spectral clustering and Laplacian eigenmaps [6, 49], hence the name "Spectral Hashing" [72].

We have already discussed several basic requirements of a good binary code: it should (1) be easily computed for a novel input; (2) require a small number of

bits to code the full dataset and (3) map similar items to similar binary codewords. Beyond these basic requirements, however, the Spectral Hashing approach also aims to form codes that more efficiently utilize each bit. Specifically, we require that each bit have a 50% chance of being one or zero, and that different bits be independent of each other. Among all codes that have this property, we will seek the ones where the average Hamming distance between similar points is minimal.

Let $\{\boldsymbol{y}_i\}_{i=1}^N$ be the list of codewords (binary vectors of length b) for N data points and $W_{N\times N}$ be the affinity matrix. Assuming the inputs are embedded in R^d so that Euclidean distance correlates with similarity, a suitable affinity is $W(i,j) = \exp(-\|\boldsymbol{x}_i - \boldsymbol{x}_j\|^2/\varepsilon^2)$. Thus the parameter ε defines the distance in R^d which corresponds to similar items. Using this notation, the average Hamming distance between similar neighbors can be written: $\sum_{ij} W_{ij}\|\boldsymbol{y}_i - \boldsymbol{y}_j\|^2$. By relaxing the independence assumption and requiring the bits to be *uncorrelated* the following problem is obtained:

$$\begin{aligned} minimize: & \sum_{ij} W_{ij}\|\boldsymbol{y}_i - \boldsymbol{y}_j\|^2 \\ subject\ to: & \boldsymbol{y}_i \in \{-1,1\}^b \\ & \sum_i \boldsymbol{y}_i = 0 \\ & \frac{1}{N}\sum_i \boldsymbol{y}_i \boldsymbol{y}_i^T = I, \end{aligned} \tag{19}$$

where the constraint $\sum_i \boldsymbol{y}_i = 0$ requires each bit to fire 50% of the time, and the constraint $\frac{1}{N}\sum_i \boldsymbol{y}_i \boldsymbol{y}_i^T = I$ requires the bits to be uncorrelated. For a single bit, solving problem 19 is equivalent to balanced graph partitioning and is NP hard (see [72] for proof), thus the problem must be relaxed in some way to make it tractable.

Spectral Relaxation By introducing an $N \times b$ matrix Y whose jth row is $\boldsymbol{y}_j^T$ and a diagonal $N \times N$ matrix $D(i,i) = \sum_j W(i,j)$, the problem can be rewritten as:

$$\begin{aligned} minimize: & trace(Y^T(D-W)Y) \\ subject\ to: & Y(i,j) \in \{-1,1\} \\ & Y^T 1 = 0 \\ & Y^T Y = I \end{aligned} \tag{20}$$

This is of course still a hard problem, but by removing the constraint that $Y(i,j) \in \{-1,1\}$ an easier problem is obtained whose solutions are simply the b eigenvectors of $D-W$ with minimal eigenvalue (after excluding the trivial eigenvector 1 which has eigenvalue 0).

Out-of-Sample Extension The out-of-sample extension of spectral methods is often solved using the Nystrom method [8, 21]. However, note that the cost of calculating the Nystrom extension of a new datapoint is *linear* in the size of the dataset.

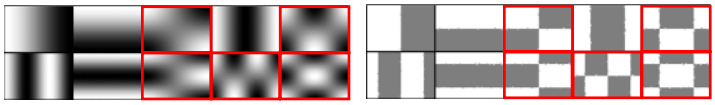

Fig. 12 Left: Eigenfunctions for a uniform rectangular distribution in 2D. Right: Thresholded eigenfunctions. Outer-product eigenfunctions have a red frame. The eigenvalues depend on the aspect ratio of the rectangle and the spatial frequency of the cut—it is better to cut the long dimension first, and lower spatial frequencies are better than higher ones.

With millions of items in the dataset this is impractical. In fact, calculating the Nystrom extension is as expensive as doing exhaustive nearest neighbor search.

In order to enable an efficient out-of-sample extension, the data points $\boldsymbol{x}_i \in R^d$ are assumed to be samples from a probability distribution $p(\boldsymbol{x})$. The equations in problem 19 above are now seen to be sample averages, which can be replaced by their expectations:

$$\begin{aligned} \textit{minimize}: \int \|\boldsymbol{y}(\boldsymbol{x}_1)-\boldsymbol{y}(\boldsymbol{x}_2)\|^2 W(\boldsymbol{x}_1,\boldsymbol{x}_2)p(\boldsymbol{x}_1)p(\boldsymbol{x}_2)d\boldsymbol{x}_1\boldsymbol{x}_2 \\ \textit{subject to}: \boldsymbol{y}(\boldsymbol{x}) \in \{-1,1\}^b \\ \int \boldsymbol{y}(\boldsymbol{x})p(\boldsymbol{x})d\boldsymbol{x} = 0 \\ \int \boldsymbol{y}(\boldsymbol{x})\boldsymbol{y}(\boldsymbol{x})^T p(\boldsymbol{x})d\boldsymbol{x} = I, \end{aligned} \tag{21}$$

with $W(\boldsymbol{x}_1,\boldsymbol{x}_2) = e^{-\|\boldsymbol{x}_1-\boldsymbol{x}_2\|^2/\varepsilon^2}$. Relaxing the constraint that $\boldsymbol{y}(\boldsymbol{x}) \in \{-1,1\}^b$ results in a spectral problem whose solutions are *eigenfunctions* of the weighted Laplace-Beltrami operators defined on manifolds [15, 7, 8, 48].

What do the eigenfunctions $\Psi_b(\boldsymbol{x})$ look like? One important special case is when $p(\boldsymbol{x})$ is a separable distribution. A simple case of a separable distribution is a multi-dimensional uniform distribution $\Pr(\boldsymbol{x}) = \prod_i u_i(\boldsymbol{x}_i)$ where u_i is a uniform distribution in the range $[a_i, \overline{a_i}]$. In the uniform case, the eigenfunctions $\Psi_b(\boldsymbol{x})$ and eigenvalues λ_b are:

$$\Psi_b(\boldsymbol{x}) = \sin(\frac{\pi}{2} + \frac{b\pi}{\overline{a}-a}\boldsymbol{x}) \tag{22}$$

$$\lambda_b = 1 - e^{-\frac{\varepsilon^2}{2}|\frac{b\pi}{\overline{a}-a}|^2}. \tag{23}$$

Figure 12 shows the analytical eigenfunctions for a 2D rectangle in order of increasing eigenvalue. The eigenvalue (which corresponds to the cut) determines which b bits will be used. Note that the eigenvalue depends on the aspect ratio of the rectangle and the spatial frequency—it is better to cut the long dimension before the short one, and low spatial frequencies are preferred.

We distinguish between *single-dimension* eigenfunctions, which are of the form $\Psi_b(\boldsymbol{x}_1)$ or $\Psi_b(\boldsymbol{x}_2)$ and *outer-product* eigenfunctions which are of the form $\Psi_b(\boldsymbol{x}_1)\Psi_l(\boldsymbol{x}_2)$.

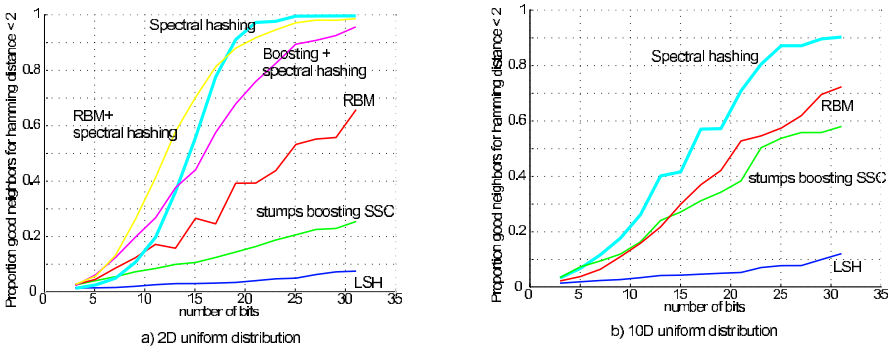

Fig. 13 Left: results on 2D rectangles with different methods. Even though Spectral Hashing is the simplest, it gives the best performance. Right: Similar pattern of results for a 10-dimensional distribution.

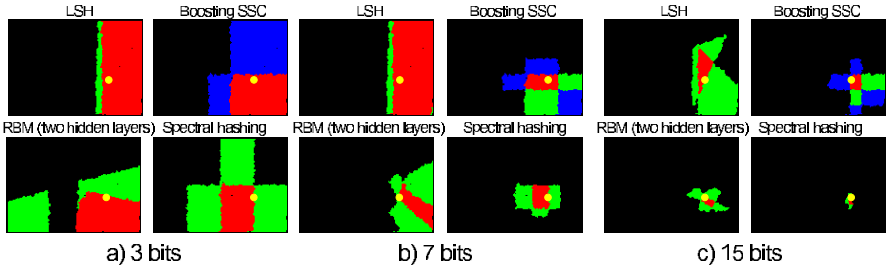

Fig. 14 Comparison of neighborhood defined by Hamming balls of different radii using codes obtained with vanilla LSH, Boosting, RBM, and Spectral Hashing when using 3, 7 and 15 bits. The yellow dot denotes a test sample. The red points correspond to the locations that are within a Hamming distance of zero. Green corresponds to a Hamming ball of radius 1, and blue to radius 2.

These outer-product eigenfunctions are shown marked with a red border in the figure. As we discuss below, these outer-product eigenfunctions should be avoided when building a hashing code.

Summary of Algorithm Recapping, given a training set of points $\{\boldsymbol{x}_i\}$ and a desired number of bits b, the steps of the Spectral Hashing algorithm are:

- Find the principal components of the data using PCA.
- Calculate the b smallest *single-dimension* analytical eigenfunctions of L_p using a rectangular approximation along every PCA direction. This is done by evaluating the b smallest eigenvalues for each direction using (Eqn. 22), thus creating a list of db eigenvalues, and then sorting this list to find the b smallest eigenvalues.
- Threshold the analytical eigenfunctions at zero, to obtain binary codes.

Illustrative Results Figure 13(a) shows a comparison between Spectral Hashing and Euclidean LSH, RBMs, and Boosting on a 2D rectangle of data. Despite the simplicity of Spectral Hashing, it outperforms the other methods. Indeed, even when

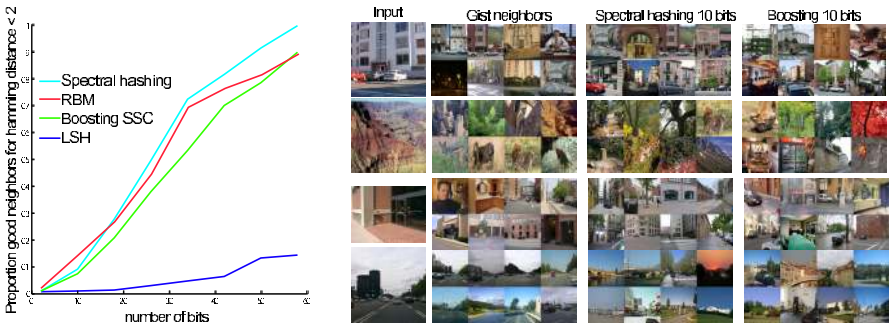

Fig. 15 Performance of different binary codes on the LabelMe dataset described in [65]. The data is certainly not uniformly distributed, and yet Spectral Hashing gives better retrieval performance than Boosting or vanilla LSH.

we apply RBMs and Boosting to the output of Spectral Hashing the performance does not improve. A similar pattern of results is shown for a 10D rectangle (Figure 13(b)). Note that the Boosting and RBM methods were trained using the approach described in Section 3.3.1 and Section 3.3.2, respectively, but using a distance matrix $D = \exp(-\|\boldsymbol{x}_i - \boldsymbol{x}_j\|^2/\varepsilon^2)$, instead of one produced by supervised label information.

Some insight into the superior performance can be gained by comparing the partitions that each bit defines on the data (see Figure 12). Recall that we seek partitions that give low cut value and are approximately independent. If simply using random linear partitions, LSH can give very unbalanced partitions. RBMs and Boosting both find good partitions, but the partitions can be highly dependent on each other. Spectral Hashing finds well balanced partitions that are more compact than those of the other methods, showing it makes efficient use of a given number of bits.

Figure 15 shows retrieval results for Spectral Hashing, RBMs, and Boosting on the LabelMe dataset [65], using Gist descriptors as the input. Note that even though Spectral Hashing uses a poor model of the statistics of the database—it simply assumes a N-dimensional rectangle, it performs better than Boosting which actually uses the distribution (the difference in performance relative to RBMs is not significant). Not only is the performance numerically better, but our visual inspection of the retrieved neighbors suggests that with a small number of bits, the retrieved images are better using Spectral Hashing than with Boosting. However, Spectral Hashing can only emulate the distance between Gist descriptors, as it has no mechanism for using label information, whereas Boosting or RBMs do (see Section 3).

4.3 Kernelized Locality Sensitive Hashing

Kernel functions are a valuable family of similarity measures, particularly since they can support structured input spaces (sets, graphs, trees) and enable connections with kernel learning algorithms. However, methods discussed thus far either assume that the data to be hashed comes from a multidimensional vector space, or require that the underlying embedding of the data be explicitly known and computable. For example, Spectral Hashing assumes uniformly distributed data in R^d; the random hyperplane LSH function expects vector data [12]; and certain specialized embeddings are manually crafted for a function of interest (Section 4.1).

This is limiting, given that many recent successful vision results employ kernels for which the underlying embedding is known only *implicitly* (i.e., only the kernel function is computable). This includes various kernels designed specifically for image comparisons (e.g., [76, 77, 68]), as well as some basic widely used functions like a Gaussian RBF kernel, or arbitrary (e.g., non-Mahalanobis) learned kernels.

Therefore, we next overview the *kernelized locality-sensitive hashing* (KLSH) approach recently introduced by Kulis and Grauman [39], which shows how to construct randomized locality-sensitive functions for arbitrary kernel functions. KLSH generalizes LSH to scenarios when the kernel-induced feature space embedding is either unknown or incomputable.

Main Idea Formally, given an arbitrary (normalized) kernel function κ, we have

$$\text{sim}(\boldsymbol{x}_i, \boldsymbol{x}_j) = \frac{\kappa(\boldsymbol{x}_i, \boldsymbol{x}_j)}{\sqrt{\kappa(\boldsymbol{x}_i, \boldsymbol{x}_i)\kappa(\boldsymbol{x}_j, \boldsymbol{x}_j)}} \tag{24}$$

$$= \frac{\phi(\boldsymbol{x}_i)^T \phi(\boldsymbol{x}_j)}{\|\phi(\boldsymbol{x}_i)\|_2 \|\phi(\boldsymbol{x}_j)\|_2}, \tag{25}$$

for some (possibly unknown) embedding function $\phi(\cdot)$. As usual, given a database of n objects, the goal is to quickly find the most similar item to a query object $\boldsymbol{q}$ in terms of the kernel function, that is, $\text{argmax}_i \kappa(\boldsymbol{q}, \boldsymbol{x}_i)$. Since we know that any Mercer kernel can be written as an inner product in some high-dimensional space [62], at a glance we might consider simply employing the random hyperplane hash functions introduced earlier in Eqn. (5), which is locality-sensitive for the inner product.

However, looking more closely, it is unclear how to do so. The random hyperplane projections assume that the vectors are represented explicitly, so that the sign of $\boldsymbol{r}^T \boldsymbol{x}$ can easily be computed. That would require referencing a random hyperplane *in the kernel-induced feature space*, but we have access to the data only through the kernel function $\kappa(\boldsymbol{x}_i, \boldsymbol{x}_j) = \phi(\boldsymbol{x}_i)^T \phi(\boldsymbol{x}_j)$. For example, the RBF kernel has an infinite-dimensional embedding, making it seemingly impossible to construct $\boldsymbol{r}$. Thus the key challenge in applying LSH to this scenario is in constructing a vector $\boldsymbol{r}$ from $\mathcal{N}(0, I)$ such that $\boldsymbol{r}^T \phi(\boldsymbol{x})$ can be computed via the kernel function.

The main idea of KLSH is to construct $\boldsymbol{r}$ as a weighted sum of a subset of the database items, drawing on the central limit theorem. In doing so, like standard LSH, hash functions are computed as random projections; however, unlike standard LSH,

these random projections will be constructed using only the kernel function and a sparse set of representative examples.

Algorithm Consider each data point $\phi(\boldsymbol{x}_i)$ from the database as a vector from some underlying distribution $\mathscr{D}$ with mean μ and covariance Σ, which are generally unknown. Given a natural number t, define

$$\boldsymbol{z}_t = \frac{1}{t}\sum_{i\in S}\phi(\boldsymbol{x}_i), \tag{26}$$

where S is a set of t database objects chosen i.i.d. from $\mathscr{D}$. According to the central limit theorem [54], for sufficiently large t, the random vector

$$\tilde{\boldsymbol{z}}_t = \sqrt{t}(z_t - \mu) \tag{27}$$

is distributed according to the multi-variate Gaussian $\mathscr{N}(0,\Sigma)$. By applying a whitening transform, the vector $\Sigma^{-1/2}\tilde{\boldsymbol{z}}_t$ will be distributed according to $\mathscr{N}(0,I)$, precisely the distribution required for hashing.

Therefore, we denote our random vector as $\boldsymbol{r} = \Sigma^{-1/2}\tilde{\boldsymbol{z}}_t$, and the desired hash function $h(\phi(\boldsymbol{x}))$ is given by

$$h(\phi(\boldsymbol{x})) = \begin{cases} 1, & \text{if } \phi(\boldsymbol{x})^T\Sigma^{-1/2}\tilde{\boldsymbol{z}}_t \geq 0 \\ 0, & \text{otherwise} \end{cases}. \tag{28}$$

Now the issue becomes how to express the product of the implicit random vector $\tilde{\boldsymbol{z}}_t$ and the matrix $\Sigma^{-1/2}$ as a weighted sum of kernel-space instances.

To do this, KLSH uses a technique similar to that used in kernel Principal Component Analysis (kPCA) [58] to project onto the eigenvectors of the covariance matrix, as follows. Both the covariance matrix Σ and the mean μ of the data are unknown, and must be approximated via a sample of the data. We choose a set of p database objects, which we denote without loss of generality as the first p items $\phi(\boldsymbol{x}_1),\ldots,\phi(\boldsymbol{x}_p)$ of the database (where $p \ll n$), and assume to be zero-centered. Now we may (implicitly) estimate the mean $\mu = \frac{1}{p}\sum_{i=1}^{p}\phi(\boldsymbol{x}_i)$ and covariance matrix Σ over the p samples. Define a kernel matrix K over the p sampled points, and let the eigendecomposition of K be $K = U\Theta U^T$. If the eigendecomposition of Σ is $V\Lambda V^T$, then $\Sigma^{-1/2} = V\Lambda^{-1/2}V^T$. Therefore, we can rewrite the hash function as follows:

$$h(\phi(\boldsymbol{x})) = \text{sign}(\phi(\boldsymbol{x})^T V\Lambda^{-1/2}V^T\tilde{\boldsymbol{z}}_t). \tag{29}$$

Note that the non-zero eigenvalues of Λ are equal to the non-zero eigenvalues of Θ. Further, denote the k-th eigenvector of the covariance matrix as $\boldsymbol{v}_k$ and the k-th eigenvector of the kernel matrix as $\boldsymbol{u}_k$. According to the derivation of kernel PCA, when the data is zero-centered, we can compute the projection

$$\boldsymbol{v}_k^T\phi(\boldsymbol{x}) = \sum_{i=1}^{p}\frac{1}{\sqrt{\theta_k}}\boldsymbol{u}_k(i)\phi(\boldsymbol{x}_i)^T\phi(\boldsymbol{x}), \tag{30}$$

where the $\phi(\boldsymbol{x}_i)$ are the sampled p data points.

We complete the computation of $h(\phi(\boldsymbol{x}))$ by performing this projection over all k eigenvectors, resulting in the following expression:

$$\phi(\boldsymbol{x})^T V \Lambda^{-1/2} V^T \tilde{\boldsymbol{z}}_t = \sum_{k=1}^{p} \frac{1}{\sqrt{\theta_k}} \boldsymbol{v}_k^T \phi(\boldsymbol{x}) \boldsymbol{v}_k^T \tilde{\boldsymbol{z}}_t. \tag{31}$$

Substituting Eqn. 30 for each of the eigenvector inner products, we have

$$\phi(\boldsymbol{x})^T V \Lambda^{-1/2} V^T \tilde{\boldsymbol{z}}_t = \sum_{k=1}^{p} \frac{1}{\sqrt{\theta_k}} \left(\sum_{i=1}^{p} \frac{1}{\sqrt{\theta_k}} \boldsymbol{u}_k(i) \phi(\boldsymbol{x}_i)^T \phi(\boldsymbol{x}) \right) \left(\sum_{i=1}^{p} \frac{1}{\sqrt{\theta_k}} \boldsymbol{u}_k(i) \phi(\boldsymbol{x}_i)^T \tilde{\boldsymbol{z}}_t \right).$$

After reordering and simplifying, this yields

$$h(\phi(\boldsymbol{x})) = \begin{cases} 1, & \text{if } \sum_{i=1}^{p} \boldsymbol{w}(i)(\phi(\boldsymbol{x}_i)^T \phi(\boldsymbol{x})) \geq 0 \\ 0, & \text{otherwise} \end{cases}, \tag{32}$$

where $\boldsymbol{w}(i) = \sum_{j=1}^{p} \sum_{k=1}^{p} \frac{1}{\theta_k^{3/2}} \boldsymbol{u}_k(i) \boldsymbol{u}_k(j) \phi(\boldsymbol{x}_j)^T \tilde{\boldsymbol{z}}_t$. See [39] for intermediate steps.

Hence, the desired Gaussian random vector can be expressed as $\boldsymbol{r} = \sum_{i=1}^{p} \boldsymbol{w}(i) \phi(\boldsymbol{x}_i)$, that is, a weighted sum over the feature vectors chosen from the set of p sampled database items.[8] Then, given any novel input, the hash bit is assigned by computing kernel values between the input and those sampled items.

Summary of Algorithm Recapping, the kernelized locality-sensitive hashing algorithm consists of the following steps:

- Select p data instances and form a kernel matrix K over this data.
- Center the kernel matrix.
- Form the hash table over the $n \gg p$ database items: for each hash function $h_j(\phi(\boldsymbol{x}))$, select t indices at random from $[1, \ldots, p]$ to sample the implicit vector $\tilde{\boldsymbol{z}}_t$, and use it to assign the next hash bit for each database instance $\boldsymbol{x}$ according to $h_j(\phi(\boldsymbol{x})) = \text{sign}(\sum_i \boldsymbol{w}(i) \kappa(\boldsymbol{x}, \boldsymbol{x}_i))$.
- For each query, form its hash key using these same hash functions (same samples of p and t indices) and employ existing LSH methods to find the approximate nearest neighbors.

Matlab code for computing KLSH functions is available from the authors' websites [39].

Computational Complexity The most expensive step in KLSH is in the single offline computation of the kernel matrix square root, which takes time $O(p^3)$. Once this matrix has been computed, each individual hash function requires $O(p^2)$ kernel function evaluations to compute its corresponding $\boldsymbol{w}$ vector (also done offline). Once $\boldsymbol{w}$ has been computed for a given hash function, the computation of the hash

[8] Note that the random vector $\boldsymbol{r}$ constructed during the KLSH routine is only *approximately* distributed according to $\mathcal{N}(0, I)$—the central limit theorem assumes that the mean and covariance of the data are known exactly, whereas KLSH employs an approximation using a sample of p points.

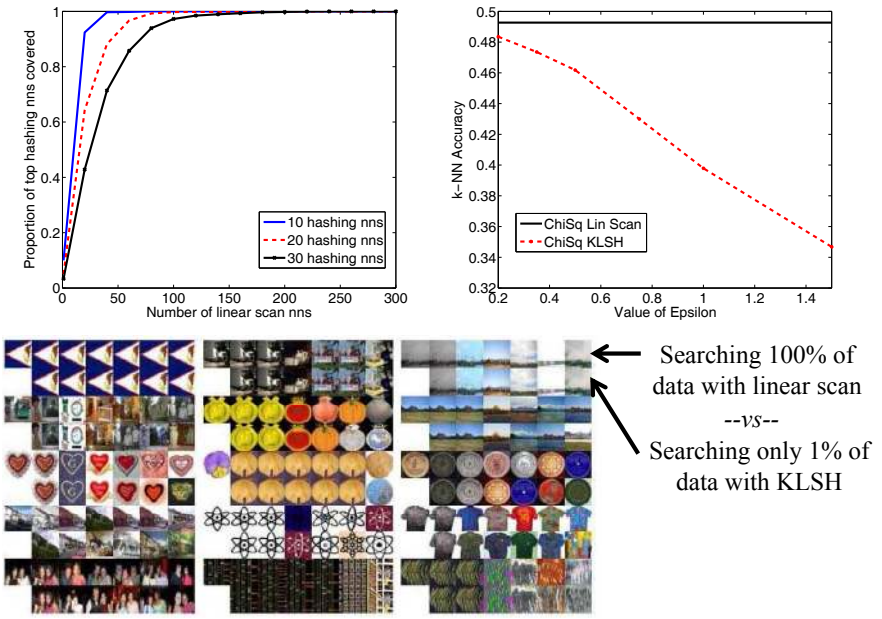

Fig. 16 Results using KLSH [39] to search the 80 Million Tiny Images data set (top left) and Flickr scenes dataset (top right) with useful image kernels—a Gaussian RBF learned kernel on Gist, and the χ^2-kernel on local features, respectively. Top left: Plot shows how many linear scan neighbors are needed to cover the first 10, 20, or 30 KLSH hashing neighbors. The ideal curve would reach the top left corner of the plot. Top right: Plot shows k-nearest neighbor accuracy of a linear scan and the KLSH algorithm as a function of LSH's ε parameter, revealing how hashing accuracy approaches that of a linear scan for smaller values of ε. Bottom: Example Tiny Image queries and the retrieved result using either a linear scan or KLSH.

function can be computed with p evaluations of the kernel function. In order to maintain efficiency, we want p to be much smaller than n—for example, $p = \sqrt{n}$ would guarantee that the algorithm maintains sub-linear search times. Empirical results for various large-scale image search tasks done in [39] suggest relatively few samples are sufficient to compute a satisfactory random vector (e.g., $p = 300$ and $t = 30$, for n up to 80 million).

Illustrative Results Figure 16 shows some example results using KLSH for image search. In both cases, kernels are employed that would not be supported by any previous LSH algorithm. The example image retrievals show qualitatively that KLSH often retrieves neighbors very similar to those of a linear scan, but does so by searching less than 1% of the 80 Million images. At the same time, the quantitative results show exactly how much accuracy is traded off. The 10-hashing NN's curve on the Tiny Images data (top left) shows, for example, that 100% of the neighbors in KLSH's top ten are within the top 50 returned with an exhaustive linear scan.

4.4 Other Unsupervised Methods

A few other methods in the vision and learning literature tackle the problem of unsupervised binary embeddings for different metrics. Most related to some of the techniques here, Athitsos et al. [2, 3] propose a boosting-based approach which gives a parametric function for mapping points to binary vectors, and can accommodate metric and non-metric target similarity functions. Salakhutdinov and Hinton [56] use a neural network trained with an NCA objective [26] to build codes for text-documents. Both these approaches are explored in Torralba et al. [65], as detailed in Section 3.3.1 and Section 3.3.2, but with the similarity function being defined by Euclidean distance rather than label overlap. Most recently, Kulis and Darrell [38] use a kernel-based approach that jointly learns a set of projections that minimize reconstruction error. This objective can be directly and efficiently minimized using coordinate-descent.

5 Conclusions

We have reviewed a variety of methods for learning compact and informative binary projections for image data. Some are purely unsupervised (e. g. Spectral Hashing), but most can be applied in both supervised and unsupervised settings. As illustrated by the results displayed in this chapter, they offer crucial scalability for useful image search problems.

Despite their common goal, the approaches draw on a wide range techniques, including random projections, spectral methods, neural networks, boosting, and kernel methods. This diversity reflects the open nature of the problem and the extensive attention it has received lately. We anticipate that advances in machine learning and algorithms will continue to be relevant to this problem of great practical interest.

Acknowledgments

KG and RF would like to thank their co-authors Prateek Jain, Brian Kulis, Antonio Torralba and Yair Weiss for their contributions to the work covered in the chapter. They also thank the IEEE for permitting the reproduction of figures from the authors' CVPR/ICCV/PAMI papers. Finally, thanks to the Flickr users kevgibbo, bridgepix, RickC, ell brown, pdbreen, and watchsmart for sharing their photos appearing in the location recognition example of Figure 1 under the Creative Commons license.

References

1. A. Andoni and P. Indyk. Near-Optimal Hashing Algorithms for Near Neighbor Problem in High Dimensions. In *IEEE Symposium on Foundations of Computer Science (FOCS)*, 2006.
2. V. Athitsos, J. Alon, S. Sclaroff, and G. Kollios. BoostMap: A Method for Efficient Approximate Similarity Rankings. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2004.
3. V. Athitsos, J. Alon, S. Sclaroff, and G. Kollios. BoostMap: An Embedding Method for Efficient Nearest Neighbor Retrieval. *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, 30(1), 2008.
4. B. Babenko, S. Branson, and S. Belongie. Similarity Metrics for Categorization: from Monolithic to Category Specific. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2009.
5. A. Bar-Hillel, T. Hertz, N. Shental, and D. Weinshall. Learning a Mahalanobis Metric from Equivalence Constraints. *Journal of Machine Learning Research*, 6:937–965, June 2005.
6. M. Belkin and P. Niyogi. Laplacian Eigenmaps and Spectral Techniques for Embedding and Clustering. In *Neural Information Processing Systems (NIPS)*, pages 585–591, 2001.
7. M. Belkin and P. Niyogi. Towards a theoretical foundation for laplacian based manifold methods. *J. of Computer System Sciences*, 2007.
8. Y. Bengio, J.-F. Paiement, P. Vincent, O. Delalleau, N. Le Roux, and M. Ouimet. Out-of-Sample Extensions for LLE, Isomap, MDS, Eigenmaps, and Spectral Clustering. In *Neural Information Processing Systems (NIPS)*, 2004.
9. J. Bentley. Multidimensional Divide and Conquer. *Communications of the ACM*, 23(4):214–229, 1980.
10. A. Broder. On the Resemblance and Containment of Documents. In *Proceedings of the Compression and Complexity of Sequences*, 1997.
11. M. Bronstein, A. Bronstein, F. Michel, and N. Paragios. Data Fusion through Cross-modality Metric Learning using Similarity-Sensitive Hashing. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2010.
12. M. Charikar. Similarity Estimation Techniques from Rounding Algorithms. In *ACM Symp. on Theory of Computing*, 2002.
13. O. Chum, M. Perdoch, and J. Matas. Geometric min-Hashing: Finding a (Thick) Needle in a Haystack. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2009.
14. O. Chum, J. Philbin, and A. Zisserman. Near Duplicate Image Detection: min-Hash and tf-idf Weighting. In *British Machine Vision Conference*, 2008.
15. R. Coifman, S. Lafon, A. B. Lee, M. Maggioni, B. Nadler, F. Warner, and S. W. Zucker. Geometric Diffusions as a Tool for Harmonic Analysis and Struture Definition of Data: Diffiusion Maps. *Proc. Natl. Academy of Sciences*, 102(21):7426–7431, May 2005.
16. K. Crammer, J. Keshet, and Y. Singer. Kernel Design Using Boosting. In *Neural Information Processing Systems (NIPS)*, 2002.
17. M. Datar, N. Immorlica, P. Indyk, and V. Mirrokni. Locality-Sensitive Hashing Scheme Based on p-Stable Distributions. In *Symposium on Computational Geometry (SOCG)*, 2004.
18. R. Datta, D. Joshi, J. Li, and J. Z. Wang. Image Retrieval: Ideas, Influences, and Trends of the New Age. *ACM Computing Surveys*, 2008.
19. J. Davis, B. Kulis, P. Jain, S. Sra, and I. Dhillon. Information-Theoretic Metric Learning. In *Proceedings of International Conference on Machine Learning (ICML)*, 2007.
20. R. Fergus, H. Bernal, Y. Weiss, and A. Torralba. Semantic Label Sharing for Learning with Many Categories. In *Proceedings of European Conference on Computer Vision (ECCV)*, September 2010.
21. C. Fowlkes, S. Belongie, F. Chung, and J. Malik. Spectral Grouping Using the Nystrom Method. *PAMI*, 26(2):214–225, 2004.
22. J. Freidman, J. Bentley, and A. Finkel. An Algorithm for Finding Best Matches in Logarithmic Expected Time. *ACM Transactions on Mathematical Software*, 3(3):209–226, September 1977.

23. A. Gionis, P. Indyk, and R. Motwani. Similarity Search in High Dimensions via Hashing. In *Proc. Intl Conf. on Very Large Data Bases*, 1999.
24. A. Globerson and S. Roweis. Metric Learning by Collapsing Classes. In *Neural Information Processing Systems (NIPS)*, 2005.
25. M. Goemans and D. Williamson. Improved Approximation Algorithms for Maximum Cut and Satisfiability Problems Using Semidefinite Programming. *JACM*, 42(6):1115–1145, 1995.
26. J. Goldberger, S. T. Roweis, R. R. Salakhutdinov, and G. E. Hinton. Neighborhood Components Analysis. In *Neural Information Processing Systems (NIPS)*, 2004.
27. K. Grauman and T. Darrell. The Pyramid Match Kernel: Discriminative Classification with Sets of Image Features. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2005.
28. K. Grauman and T. Darrell. Pyramid Match Hashing: Sub-Linear Time Indexing Over Partial Correspondences. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2007.
29. R. Hadsell, S. Chopra, and Y. LeCun. Dimensionality Reduction by Learning an Invariant Mapping. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2006.
30. T. Hertz, A. Bar-Hillel, and D. Weinshall. Learning Distance Functions for Image Retrieval. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2004.
31. T. Hertz, A. Bar-Hillel, and D. Weinshall. Learning a Kernel Function for Classification with Small Training Samples. In *Proceedings of International Conference on Machine Learning (ICML)*, 2006.
32. G. E. Hinton and R. R. Salakhutdinov. Reducing the Dimensionality of Data with Neural Networks. *Nature*, 313(5786):504–507, July 2006.
33. P. Indyk and R. Motwani. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *30th Symposium on Theory of Computing*, 1998.
34. P. Indyk and N. Thaper. Fast Image Retrieval via Embeddings. In *Intl Workshop on Statistical and Computational Theories of Vision*, 2003.
35. Q. Iqbal and J. K. Aggarwal. CIRES: A System for Content-Based Retrieval in Digital Image Libraries. In *International Conference on Control, Automation, Robotics and Vision*, 2002.
36. P. Jain, B. Kulis, I. Dhillon, and K. Grauman. Online Metric Learning and Fast Similarity Search. In *Neural Information Processing Systems (NIPS)*, 2008.
37. P. Jain, B. Kulis, and K. Grauman. Fast Image Search for Learned Metrics. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2008.
38. B. Kulis and T. Darrell. Learning to Hash with Binary Reconstructive Embeddings. In *Neural Information Processing Systems (NIPS)*, 2009.
39. B. Kulis and K. Grauman. Kernelized Locality-Sensitive Hashing for Scalable Image Search. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2009.
40. B. Kulis, P. Jain, and K. Grauman. Fast Similarity Search for Learned Metrics. *IEEE Transactions on Pattern Analysis and Machine Intelligence (PAMI)*, 31, 2009.
41. S. Lazebnik, C. Schmid, and J. Ponce. Beyond Bags of Features: Spatial Pyramid Matching for Recognizing Natural Scene Categories. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2006.
42. R.-S. Lin, D. Ross, and J. Yagnik. SPEC Hashing: Similarity Preserving Algorithm for Entropy-based Coding. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2010.
43. H. Ling and S. Soatto. Proximity Distribution Kernels for Geometric Context in Category Recognition. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2007.
44. T. Liu, A. Moore, A. Gray, and K. Yang. An Investigation of Practical Approximate Nearest Neighbor Algorithms. In *Neural Information Processing Systems (NIPS)*, 2005.
45. D. Lowe. Distinctive Image Features from Scale-Invariant Keypoints. *International Journal of Computer Vision (IJCV)*, 60(2), 2004.

46. Y. Mu, J. Shen, and S. Yan. Weakly-supervised hashing in kernel space. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2010.
47. M. Muja and D. Lowe. Fast Approximate Nearest Neighbors with Automatic Algorithm Configuration. In *International Conference on Computer Vision Theory and Application (VISSAPP)*, 2009.
48. B. Nadler, S. Lafon, R. Coifman, and I. Kevrekidis. Diffusion maps, spectral clustering and reaction coordinates of dynamical systems. http://arxiv.org, 2008.
49. A. Ng, M. I. Jordan, and Y. Weiss. On Spectral Clustering, Analysis and an Algorithm. In *Neural Information Processing Systems (NIPS)*, 2001.
50. A. Oliva and A. Torralba. Modeling the Shape of the Scene: a Holistic Representation of the Spatial Envelope. *International Journal in Computer Vision*, 42:145–175, 2001.
51. J. Philbin, O. Chum, M. Isard, J. Sivic, and A. Zisserman. Object Retrieval with Large Vocabularies and Fast Spatial Matching. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2007.
52. M. Raginsky and S. Lazebnik. Locality-Sensitive Binary Codes from Shift-Invariant Kernels. In *Neural Information Processing Systems (NIPS)*, 2009.
53. A. Rahimi and B. Recht. Random Features for Large-Scale Kernel Machines. In *Neural Information Processing Systems (NIPS)*, 2007.
54. J. Rice. *Mathematical Statistics and Data Aanalysis*. Duxbury Press, 2001.
55. S. Roweis and L. Saul. Nonlinear Dimensionality Reduction by Locally Linear Embedding. *Science*, 290(5500):2323–2326, 2000.
56. R. R. Salakhutdinov and G. E. Hinton. Learning a Nonlinear Embedding by Preserving Class Neighbourhood Structure. In *AISTATS*, 2007.
57. R. R. Salakhutdinov and G. E. Hinton. Semantic Hashing. In *SIGIR workshop on Information Retrieval and applications of Graphical Models*, 2007.
58. B. Schölkopf, A. Smola, and K.-R. Müller. Nonlinear Component Analysis as a Kernel Eigenvalue Problem. *Neural Computation*, 10:1299–1319, 1998.
59. M. Schultz and T. Joachims. Learning a Distance Metric from Relative Comparisons. In *Neural Information Processing Systems (NIPS)*, 2003.
60. G. Shakhnarovich. *Learning Task-Specific Similarity*. PhD thesis, MIT, 2005.
61. G. Shakhnarovich, P. Viola, and T. Darrell. Fast pose estimation with parameter sensitive hashing. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2003.
62. J. Shawe-Taylor and N. Cristianini. *Kernel Methods for Pattern Analysis*. Cambridge University Press, 2004.
63. J. Sivic and A. Zisserman. Video Google: A Text Retrieval Approach to Object Matching in Videos. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2003.
64. J. Tenenbaum, V. de Silva, and J. Langford. A Global Geometric Framework for Nonlinear Dimensionality Reduction. *Science*, 290(5500):2319–2323, December 2000.
65. A. Torralba, R. Fergus, and Y. Weiss. Small Codes and Large Image Databases for Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2008.
66. J. Uhlmann. Satisfying General Proximity / Similarity Queries with Metric Trees. *Information Processing Letters*, 40:175–179, 1991.
67. L. van der Maaten and G. Hinton. Visualizing High-Dimensional Data Using t-SNE. *Journal of Machine Learning Research*, 9:2579–2605, November 2008.
68. M. Varma and D. Ray. Learning the Discriminative Power-Invariance Trade-off. In *Proceedings of the IEEE International Conference on Computer Vision (ICCV)*, 2007.
69. J. Wang, S. Kumar, and S.-F. Chang. Semi-Supervised Hashing for Scalable Image Retrieval. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2010.
70. J. Wang, S. Kumar, and S.-F. Chang. Sequential Projection Learning for Hashing with Compact Codes. In *Proceedings of International Conference on Machine Learning (ICML)*, 2010.

71. K. Weinberger, J. Blitzer, and L. Saul. Distance Metric Learning for Large Margin Nearest Neighbor Classification. In *Neural Information Processing Systems (NIPS)*, 2006.
72. Y. Weiss, A. Torralba, and R. Fergus. Spectral Hashing. In *Neural Information Processing Systems (NIPS)*, 2008.
73. E. Xing, A. Ng, M. Jordan, and S. Russell. Distance Metric Learning, with Application to Clustering with Side-Information. In *Neural Information Processing Systems (NIPS)*, 2002.
74. D. Xu, T.J. Cham, S. Yan, and S.-F. Chang. Near Duplicate Image Identification with Spatially Aligned Pyramid Matching. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2008.
75. T. Yeh, K. Grauman, K. Tollmar, and T. Darrell. A Picture is Worth a Thousand Keywords: Image-Based Object Search on a Mobile Platform. In *Proceedings of the ACM Conference on Human Factors in Computing Systems*, 2005.
76. H. Zhang, A. Berg, M. Maire, and J. Malik. SVM-KNN: Discriminative Nearest Neighbor Classification for Visual Category Recognition. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2006.
77. J. Zhang, M. Marszalek, S. Lazebnik, and C. Schmid. Local Features and Kernels for Classification of Texture and Object Categories: A Comprehensive Study. *International Journal of Computer Vision (IJCV)*, 73(2):213–238, 2007.