To appear, Proceedings of the IEEE International Conference on Computer Vision (ICCV), Kyoto, Japan, October 2009.

Kernelized Locality-Sensitive Hashing for Scalable Image Search

Brian Kulis
UC Berkeley EECS and ICSI
Berkeley, CA 94720
kulis@eecs.berkeley.edu

Kristen Grauman
UT Austin Computer Science Dept.
Austin, TX 78712
grauman@cs.utexas.edu

Abstract

Fast retrieval methods are critical for large-scale and data-driven vision applications. Recent work has explored ways to embed high-dimensional features or complex distance functions into a low-dimensional Hamming space where items can be efficiently searched. However, existing methods do not apply for high-dimensional kernelized data when the underlying feature embedding for the kernel is unknown. We show how to generalize locality-sensitive hashing to accommodate arbitrary kernel functions, making it possible to preserve the algorithm's sub-linear time similarity search guarantees for a wide class of useful similarity functions. Since a number of successful image-based kernels have unknown or incomputable embeddings, this is especially valuable for image retrieval tasks. We validate our technique on several large-scale datasets, and show that it enables accurate and fast performance for example-based object classification, feature matching, and content-based retrieval.

1. Introduction

Fast indexing and search for large databases is critical to content-based image and video retrieval—particularly given the ever-increasing availability of visual data in a variety of interesting domains, such as scientific image data, community photo collections on the Web, news photo collections, or surveillance archives. The most basic but essential task in image search is the "nearest neighbor" problem: to take a query image and accurately find the examples that are most similar to it within a large database. A naive solution to finding neighbors entails searching over all n database items and sorting them according to their similarity to the query, but this becomes prohibitively expensive when n is large or when the individual similarity function evaluations are expensive to compute. For vision applications, this complexity is amplified by the fact that often the most effective representations are high-dimensional or structured, and best known distance functions can require considerable computation to compare a single pair of objects.

To make large-scale search practical, vision researchers have recently explored *approximate* similarity search techniques, where a predictable loss in accuracy is sacrificed in order to allow fast queries even for high-dimensional inputs [25, 24, 12, 16, 4]. Methods for this problem, most notably *locality-sensitive hashing* (LSH) [10, 3], offer probabilistic guarantees of retrieving items within $(1+\epsilon)$ times the optimal similarity, with query times that are sub-linear with respect to n. The basic idea is to compute randomized hash functions that guarantee a high probability of collision for similar examples. In a similar spirit, a number of methods show how to form low-dimensional binary embeddings that can capture more expensive distance functions [1, 28, 22, 31]. This line of work has shown considerable promise for a variety of image search tasks such as near-duplicate retrieval, example-based object recognition, pose estimation, and feature matching.

In spite of hashing's success for visual similarity search tasks, existing techniques have some important restrictions. Current methods generally assume that the data to be hashed comes from a multidimensional vector space, and require that the underlying embedding of the data be explicitly known and computable. For example, LSH relies on random projections with input vectors; spectral hashing [31] assumes vectors with a known probability distribution.

This is a problematic limitation, given that many recent successful vision results employ *kernel functions* for which the underlying embedding is known only *implicitly* (i.e., only the kernel function is computable). It is thus far impossible to apply LSH and its variants to search data with a number of powerful kernels—including many kernels designed specifically for image comparisons [33, 34, 30], as well as some basic well-used functions like a Gaussian RBF. Further, since visual representations are often most naturally encoded with structured inputs (e.g., sets, graphs, trees), the lack of fast search methods with performance guarantees for flexible kernels is inconvenient.

In this paper, we present an LSH-based technique for performing fast similarity searches over *arbitrary* kernel

functions. The problem is as follows: given a kernel function $\kappa(\boldsymbol{x}_i, \boldsymbol{x}_j) = \phi(\boldsymbol{x}_i)^T \phi(\boldsymbol{x}_j)$ and a database of n objects, how can we quickly find the most similar item to a query object $\boldsymbol{q}$ in terms of the kernel function, that is, $\text{argmax}_i \kappa(\boldsymbol{q}, \boldsymbol{x}_i)$? Like standard LSH, our hash functions involve computing random projections; however, unlike standard LSH, these random projections are constructed using only the kernel function and a sparse set of examples from the database itself. Our main technical contribution is to formulate the random projections necessary for LSH in kernel space. Our construction relies on an appropriate use of the central limit theorem [21], which allows us to approximate a random vector using items from our database. The resulting scheme, which we call kernelized LSH (KLSH), generalizes LSH to scenarios when the feature space embeddings ($\phi(\boldsymbol{x})$, $\phi(\boldsymbol{y})$) are either unknown or incomputable.

We empirically validate our scheme with several visual search tasks. For object recognition, we present results on the Caltech-101 [8], and show that our hashing scheme outperforms existing hashing methods on this data set since it can compute hash functions over arbitrary kernels. For feature indexing with a larger database, we provide results on the Photo Tourism data set of local patches [26, 14]. Finally, we experiment with the Tiny Image data set of 80 million images [27], in order to show our technique's ability to scale to very large databases. Because our algorithm enables fast approximate search for arbitrary kernels, we can now access a much wider class of similarity functions needed for many content-based retrieval applications.

2. Related Work

In this section we review related work in fast search algorithms and their application for visual search problems.

Data structures using spatial partitions and recursive hyperplane decomposition (e.g., $k-d$ trees [9]) provide an efficient means to search low-dimensional vector data exactly, however they are known to break down in practice for high-dimensional data, and cannot provide better than a worst-case linear query time guarantee. Since high-dimensional image descriptors are commonly used in object recognition, methods to mitigate these factors have been explored, such as hierarchical feature quantization [18], decision trees [19], and priority queues [2].

Tree-based search structures that can operate with arbitrary metrics [29, 5] remove the assumption of a vector space by exploiting the triangle inequality. However, in practice selecting useful partitioning strategies requires good heuristics, and, in spite of logarithmic query times in the expectation, metric-tree methods can also degenerate to a linear time scan of all items depending on the distribution of distances for the data set.

Randomized approximate similarity search algorithms have been designed to preserve query time guarantees, even for high-dimensional inputs. Locality-sensitive hashing [10, 3] offers sub-linear time search by hashing highly similar examples together in a hash table; LSH functions that accommodate Hamming distance [15], inner products [3], ℓ_p norms [6], normalized partial matching [12], and learned Mahalanobis metrics [16] have all been developed in prior work. Vision researchers have shown the effectiveness of this class of methods for various image search applications, including shape matching, pose inference, and bag-of-words indexing [25, 24, 12, 16, 4]. However, thus far, arbitrary kernel functions remain off limits for LSH.

Embedding functions offer another useful way to map expensive distance functions into something more manageable computationally. Recent work has considered how to construct or learn an embedding that will preserve the desired distance function, typically with the intention of mapping to a very low-dimensional space that is more easily searchable with known techniques [1, 20, 28, 22, 31]. These methods are related to LSH in the sense that both seek small "keys" that can be used to encode similar inputs, and often these keys exist in Hamming space. While most work with vector inputs, the technique in [1] accepts generic distance functions, though its boosting-based training process is fairly expensive, and search is done with a linear scan. The recent "spectral hashing" algorithm [31] requires that data be from a Euclidean space and uniformly distributed.

3. Background: Locality-Sensitive Hashing

We begin by briefly reviewing Locality-Sensitive Hashing (LSH). Assume that our database is a set of vectors $\boldsymbol{x}_1, ..., \boldsymbol{x}_n$. Given a query vector $\boldsymbol{q}$, we are interested in finding the most similar items in the database to the query.

The basic idea behind LSH is to project the data into a low-dimensional binary (Hamming) space; that is, each data point is mapped to a b-bit vector, called the *hash key*. If this projection is performed appropriately, we can find approximate nearest neighbors in time sub-linear in n. The hash keys are constructed by applying b binary-valued hash functions $h_1, ..., h_b$ to each of the database objects. In order to be valid, each hash function h must satisfy the locality-sensitive hashing property:

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

where $\text{sim}(\boldsymbol{x}_i, \boldsymbol{x}_j) \in [0, 1]$ is the similarity function of interest.[1]The intuition is as follows: if we can rely on only highly similar examples colliding together in the hash table (i.e., being 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. Given

[1]LSH has been formulated in two related contexts—one in which the likelihood of collision is guaranteed relative to a threshold on the radius surrounding a query point [15], and another where collision probabilities are equated with a similarity score [3]. We use the latter definition here.

valid LSH functions, the query time for retrieving $(1+\epsilon)$-near neighbors is bounded by $O(n^{1/(1+\epsilon)})$ for the Hamming distance [10]. One can therefore trade off the accuracy of the search with the query time required.

As an example, consider the well-known inner product similarity: $\text{sim}(\boldsymbol{x}_i, \boldsymbol{x}_j) = \boldsymbol{x}_i^T \boldsymbol{x}_j$. In [3], Charikar showed a hash function for this similarity function based on rounding the output of a product with a random hyperplane:

$$h_{\boldsymbol{r}}(\boldsymbol{x}) = \begin{cases} 1, & \text{if } \boldsymbol{r}^T \boldsymbol{x} \geq 0 \\ 0, & \text{otherwise} \end{cases}, \tag{2}$$

where $\boldsymbol{r}$ is a random hyperplane 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 obeys the locality-sensitive hash property follows from a result from Goemans and Williamson [11], who showed for such a random $\boldsymbol{r}$ that

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

Procedurally, one chooses a random vector $\boldsymbol{r}$ from $\mathcal{N}(0, I)$, then computes the sign of $\boldsymbol{r}^T \boldsymbol{x}$ for each $\boldsymbol{x}$ in the database, then repeats this over the b random vectors for a total of b hash functions. The hash table then consists of the hash keys and their pointers to data items. Given a query vector $\boldsymbol{q}$, one computes its hash key by applying the same b hash functions. A query hashes to certain buckets in the hash table, where it collides with some small portion of the stored examples. Only these examples are searched.

To perform the approximate similarity searches, we use the method in [3], which requires searching $O(n^{1/(1+\epsilon)})$ examples for the $k = 1$ approximate-NN. Given the list of database hash keys, $M = 2n^{1/(1+\epsilon)}$ random permutations of the bits are formed, and each list of permuted hash keys is sorted lexicographically to form M sorted orders. A query hash key indexes into each sorted order with a binary search, and the $2M$ nearest examples found are the approximate nearest neighbors. Additionally, we introduce a parameter B that is the number of neighboring bins to consider as potential nearest neighbors when searching through the sorted permutations. See [3] for more details.

Previously, hash functions have been designed for cases where the similarity function "sim" refers to an ℓ_p norm, Mahalanobis metric, or inner product [6, 16, 3]. In this work, the similarity function of interest is an arbitrary kernel function κ: $\text{sim}(\boldsymbol{x}_i, \boldsymbol{x}_j) = \kappa(\boldsymbol{x}_i, \boldsymbol{x}_j) = \phi(\boldsymbol{x}_i)^T \phi(\boldsymbol{x}_j)$, for some (possibly unknown) embedding function $\phi(\cdot)$. In the next section, we present our algorithm for drawing hash functions that will satisfy Eqn. 1 for any kernel.

4. Kernelized Locality-Sensitive Hashing

The random hyperplane hashing method proposed by Charikar assumes that the vectors are represented explicitly, so that the sign of $\boldsymbol{r}^T \boldsymbol{x}$ can easily be computed.[2] We now consider the case when the data is kernelized; we denote the inputs as $\phi(\boldsymbol{x})$ and assume that the underlying embeddings may be unknown or very expensive to compute. Our access to the data is only through the kernel function $\kappa(\boldsymbol{x}_i, \boldsymbol{x}_j) = \phi(\boldsymbol{x}_i)^T \phi(\boldsymbol{x}_j)$, so it is not clear how to compute the hash functions—for example, the RBF kernel has an infinite-dimensional embedding, making it seemingly impossible to even construct $\boldsymbol{r}$. The 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 our approach is to construct $\boldsymbol{r}$ as a weighted sum of a subset of the database items. An appropriate construction will allow the random hyperplane hash function to be computed purely via kernel function evaluations, but will also ensure that $\boldsymbol{r}$ is approximately Gaussian. Consider each data point $\phi(\boldsymbol{x}_i)$ from the database as a vector from some underlying distribution $\mathcal{D}$ with mean μ and covariance Σ, which are generally unknown. Given a natural number t, define $\boldsymbol{z}_t = \frac{1}{t} \sum_{i \in S} \phi(\boldsymbol{x}_i)$, where S is a set of t database objects chosen i.i.d. from $\mathcal{D}$. The central limit theorem [21] tells us that, for sufficiently large t, the random vector $\tilde{\boldsymbol{z}}_t = \sqrt{t}(\boldsymbol{z}_t - \mu)$ is distributed according to the multi-variate Gaussian $\mathcal{N}(0, \Sigma)$. By applying a whitening transform, the vector $\Sigma^{1/2} \tilde{\boldsymbol{z}}_t$ will be distributed according to $\mathcal{N}(0, I)$, precisely the distribution required in Eqn. 2.

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

$$h(\phi(\boldsymbol{x})) = \begin{cases} 1, & \text{if } \phi(\boldsymbol{x})^T \Sigma^{1/2} \tilde{\boldsymbol{z}}_t \geq 0 \\ 0, & \text{otherwise} \end{cases}. \tag{3}$$

Both the covariance matrix Σ and the mean μ of the data are unknown, so they 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 $\phi(\boldsymbol{x}_1), ..., \phi(\boldsymbol{x}_p)$ of the database (we will discuss the choice of the value of p later). Now we may (implicitly) estimate the mean as $\mu = \frac{1}{p} \sum_{i=1}^{p} \phi(\boldsymbol{x}_i)$. Conceptually, we can also form the covariance matrix Σ over the p samples, though we cannot store this matrix explicitly.

In order to compute $h(\phi(\boldsymbol{x}))$, we will use a technique similar to that used in kernel PCA [23] to project onto the eigenvectors of the covariance matrix. In our case, if the eigendecomposition of Σ is $V \Lambda V^T$, then $\Sigma^{1/2} = V \Lambda^{1/2} V^T$. Therefore,

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

Define a kernel matrix over the p sampled points, denote it as K, and let its eigendecomposition be $K = U \Theta U^T$. Note

[2] Note that other LSH functions exist, but all involve an explicit representation of the input, and are not amenable to the kernelized case.

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, we can compute the projection

$$\boldsymbol{v}_k^T \phi(\boldsymbol{x}) = \sum_{i=1}^{p} \frac{1}{\sqrt{\theta_k}} \boldsymbol{u}_k(i) \phi(\boldsymbol{x}_i)^T \phi(\boldsymbol{x}), \tag{4}$$

where the $\phi(\boldsymbol{x}_i)$ are the sampled p data points. We complete the computation of $h(\phi(\boldsymbol{x}))$ by performing this computation over all k eigenvectors, resulting in the following expression:

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

We substitute Eqn. 4 for each of the eigenvector inner products and expand the resulting expression:

$$= \sum_{k=1}^{p} \sqrt{\theta_k} \Big(\sum_{i=1}^{p} \frac{1}{\sqrt{\theta_k}} \boldsymbol{u}_k(i) \phi(\boldsymbol{x}_i)^T \phi(\boldsymbol{x}) \Big) \cdot \Big(\sum_{i=1}^{p} \frac{1}{\sqrt{\theta_k}} \boldsymbol{u}_k(i) \phi(\boldsymbol{x}_i)^T \tilde{\boldsymbol{z}}_t \Big).$$

Now we reorder the summations and reorganize terms:

$$\begin{aligned} &= \sum_{k=1}^{p} \sum_{i=1}^{p} \sum_{j=1}^{p} \frac{1}{\sqrt{\theta_k}} \boldsymbol{u}_k(i) \boldsymbol{u}_k(j) \Big(\phi(\boldsymbol{x}_i)^T \phi(\boldsymbol{x}) \Big) \cdot \Big(\phi(\boldsymbol{x}_j)^T \tilde{\boldsymbol{z}}_t \Big) \\ &= \sum_{i=1}^{p} \sum_{j=1}^{p} \Big(\phi(\boldsymbol{x}_i)^T \phi(\boldsymbol{x}) \Big) \cdot \Big(\phi(\boldsymbol{x}_j)^T \tilde{\boldsymbol{z}}_t \Big) \Big(\sum_{k=1}^{p} \frac{1}{\sqrt{\theta_k}} \boldsymbol{u}_k(i) \boldsymbol{u}_k(j) \Big). \end{aligned}$$

Finally, we use the fact that $K_{ij}^{-1/2} = \sum_{k=1}^{p} \frac{1}{\sqrt{\theta_k}} \boldsymbol{u}_k(i) \boldsymbol{u}_k(j)$ and simplify further to obtain:

$$\begin{aligned} &= \sum_{i=1}^{p} \sum_{j=1}^{p} K_{ij}^{-1/2} \Big(\phi(\boldsymbol{x}_i)^T \phi(\boldsymbol{x}) \Big) \Big(\phi(\boldsymbol{x}_j)^T \tilde{\boldsymbol{z}}_t \Big) \\ &= \sum_{i=1}^{p} \boldsymbol{w}(i) \Big(\phi(\boldsymbol{x}_i)^T \phi(\boldsymbol{x}) \Big), \end{aligned}$$

where $\boldsymbol{w}(i) = \sum_{j=1}^{p} K_{ij}^{-1/2} \phi(\boldsymbol{x}_j)^T \tilde{\boldsymbol{z}}_t$.[3]

[3]We can alternatively reach this conclusion by noting that $\phi(\boldsymbol{x})^T \Sigma^{1/2} \tilde{\boldsymbol{z}}_t = \phi(\boldsymbol{x})^T \Phi K^{-1/2} \Phi^T \tilde{\boldsymbol{z}}_t$, where $K = \Phi^T \Phi$. This follows by computing the singular value decomposition of Φ and simplifying.

This means that the Gaussian random vector can be expressed as $\boldsymbol{r} = \sum_{i=1}^{p} \boldsymbol{w}(i) \phi(\boldsymbol{x}_i)$, so it is a weighted sum over the feature vectors chosen from the set of p sampled database items. We now expand $\tilde{\boldsymbol{z}}_t$; recall that $\tilde{\boldsymbol{z}}_t = \sqrt{t}(\frac{1}{t}\sum_{i \in S} \phi(\boldsymbol{x}_i) - \mu) = \sqrt{t}(\frac{1}{t}\sum_{i \in S} \phi(\boldsymbol{x}_i) - \frac{1}{p}\sum_{i=1}^{p} \phi(\boldsymbol{x}_i))$. Substituting this into $\boldsymbol{w}(i)$ yields

$$\boldsymbol{w}(i) = \frac{1}{t} \sum_{j=1}^{p} \sum_{\ell \in S} K_{ij}^{-1/2} K_{j\ell} - \frac{1}{p} \sum_{j=1}^{p} \sum_{k=1}^{p} K_{ij}^{-1/2} K_{jk}.$$

Note that we are ignoring the $\sqrt{t}$ term as it does not affect the sign of the hash function, and further we assume that the t points selected for S are a subset of the p sampled points (i.e., the t points are sampled from the set of points used to sample the mean and covariance). In that case, the expression for $\boldsymbol{w}(i)$ simplifies as follows: if $\boldsymbol{e}$ is a vector of all ones, and $\boldsymbol{e}_S$ is a vector with ones in the entries corresponding to the indices of S, then the expression simplifies:

$$\boldsymbol{w} = K^{1/2} \Big(\frac{1}{t} \boldsymbol{e}_S - \frac{1}{p} \boldsymbol{e} \Big). \tag{5}$$

Putting everything together, the resulting method is surprisingly simple. To summarize our kernelized locality-sensitive hashing algorithm (KLSH):

- Select p data points and form a kernel matrix K over this data.
- Form the hash table over database items: for each hash function $h(\phi(\boldsymbol{x}))$, form $\boldsymbol{e}_S$ by selecting t indices at random from $[1, ..., p]$, then form $\boldsymbol{w} = K^{1/2}(\frac{1}{t}\boldsymbol{e}_S - \frac{1}{p}\boldsymbol{e})$, and assign bits according to $h(\phi(\boldsymbol{x})) = \text{sign}(\sum_i \boldsymbol{w}(i) \kappa(\boldsymbol{x}, \boldsymbol{x}_i))$.
- For each query, form its hash key using these hash functions and employ existing LSH methods to find the approximate nearest neighbors.

Computationally, the most expensive step is in the single offline computation of $K^{1/2}$, 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 function can be computed with p evaluations of the kernel function. In order to maintain efficiency, and to maintain sub-linear time searches, we want p to be much smaller than n—for example, $p = O(\sqrt{n})$ would guarantee that the algorithm maintains sub-linear search times.

5. Discussion

Some additional care must be taken to verify that the analysis for KLSH holds when the underlying embeddings are infinite-dimensional (for example, with the Gaussian

kernel), but in fact the general case does hold. To summarize the main details associated with arbitrary reproducing kernel Hilbert spaces: first, the central limit theorem holds in general Banach spaces (for which RKHSs are a special case) under certain conditions—see [13] for a discussion. Second, in the infinite-dimensional case, we whiten the data via the covariance operator; this has been studied for the kernel ICA problem [32]. Finally, projecting onto eigenvectors of the covariance is performed as in kernel PCA, which holds for infinite-dimensional embeddings. We stress that the KLSH algorithm is unchanged for such embeddings.

Additionally, 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 we employ an approximation using a sample of p points. Furthermore, since general bounds for the central limit theorem are not known, it is possible that the resulting $\boldsymbol{r}$ vectors do not approximate a Gaussian well unless p and t are extremely large. This may be the case if the underlying embeddings of the kernel function are very high-dimensional. The good news is that empirical evidence suggests that we do not need very many samples to compute a satisfactory random vector for kernelized hashing; as we will see in the experimental results, with $p = 300$ and $t = 30$ we obtain good hashing results even over very large data sets.

We would also like to stress the general applicability and simplicity of our approach. Even if the underlying embedding for a particular kernel function is known, our technique still may be desirable due to its relative simplicity. Kernels such as the pyramid match kernel [12] or the proximity distribution kernel [17] have known sparse, high-dimensional embeddings for which standard LSH methods can be applied; however, in such scenarios the computation of the hash functions is dependent on the kernel embeddings, requiring separate (and sometimes intricate) hashing implementations for each particular kernel function. In contrast, our approach is general and only requires knowledge of the kernel function. As a result, the KLSH scheme may be preferable even in these cases.

6. Experimental Results

To empirically validate the effectiveness of the proposed hashing scheme, we provide results on three data sets. Our primary goal is to verify that the proposed KLSH method can take kernels with unknown feature embeddings, and use them to perform searches that are fast but still reliable relative to a linear scan.

Throughout, we present results showing the percentage of database items searched with hashing as opposed to timing results, which are dependent on the particular optimizations of the code. In terms of additional overhead, finding the approximate nearest neighbors given the query hash key is very fast, particularly if the computation can be distributed across several machines (since the random permutations of the hash bits are independent of one another). The main computational cost is in taking the list of examples that collided and sorting them by their similarity to the query; this running time is primarily controlled by ϵ.

Example-Based Object Recognition. Our first experiment uses the Caltech 101 data set, a standard benchmark for object recognition. We use our technique to perform nearest neighbor classification, categorizing each novel image according to which of the 101 categories it belongs. This data set is fairly small ($\sim$9K total images); we use it because recent impressive results for this data have applied specialized image kernels, including some with no known embedding function. The goal is therefore to show that our hashing scheme is useful in a domain where such kernel functions are typically employed, and that nearest-neighbor accuracy does not significantly degrade with the use of hashing. We also use this data set to examine how changes in the parameters affect accuracy.

We employ the correspondence-based local feature kernel (CORR) designed in [33], and learn a kernel on top of it using the metric learning algorithm given in [7]. The learned kernel basically specializes the original kernel to be more accurate for the classification task, and was shown in [16] to provide the best reported single kernel results on the Caltech-101 (when using linear scan search). We train the metric with 15 images per class. To compute accuracy, we use a simple k-nearest-neighbor classifier ($k = 1$) using both a linear scan baseline and KLSH.

Figure 1 compares our accuracy using hashing versus that of an exhaustive linear scan, for varying parameter settings. The parameters of interest are the number of bits used for the hash keys b, the value of ϵ (from standard LSH), and the values of t and p (from our KLSH algorithm). When varying one of the parameters, the others remain fixed at the following: $b = 300$, $\epsilon = 0.5$, $t = 30$, and $p = 300$. We ran KLSH 10 times and averaged the results over the 10 runs. The results of changing ϵ and the number of hash bits are consistent with the behavior seen for standard LSH (see for example [12, 16]). From the plots it appears that the KLSH parameters (t,p), are reasonably robust; the accuracy improves modestly as p is increased, but there is little change in the accuracy as t is increased.

The parameter ϵ trades off accuracy for speed, and thus has a more significant impact on final classifier performance. Our best result of 58.5% hashing accuracy, with $\epsilon = 0.2$, is significantly better than the best previous hashing-based result on this dataset: 48% with the pyramid match, as reported in [16]. Note that hashing with the pyramid match was possible in the past only because that kernel has a known explicit embedding function [12]; with

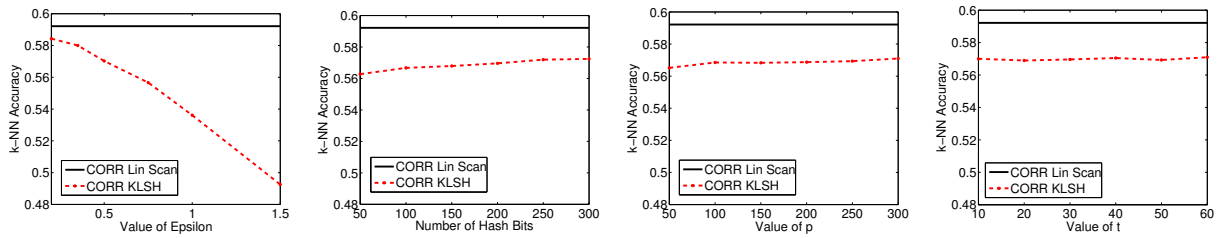

Figure 1. Results on the Caltech-101 data set. The plots illustrate the effect of parameter settings on classification accuracy when using KLSH with a learned CORR kernel. When varying a parameter, the others are fixed at $\epsilon = .5$, $t = 30$, $p = 300$, and $b = 300$. The value of ϵ controls the accuracy vs. speed tradeoff, and most influences results. When searching only 6.7% of the data ($\epsilon = 0.5$), accuracy is 57%, versus 59% with a linear scan. Overall, accuracy is quite stable with respect to the number of bits and our algorithm's p and t parameters.

our method, we can hash with matching kernels for which the embedding function is unknown (e.g., CORR). This 10-point accuracy improvement illustrates the importance of being able to choose the best-suited kernel function for the task—which KLSH now makes possible.

In terms of the speed, the percentage of database items searched was uniform as t and p changed. On average, KLSH searched 17.4% of the database for $\epsilon = 0.2$; 6.7% for $\epsilon = 0.5$, and 1.2% for $\epsilon = 1.5$. As we will see in subsequent sections, even lower percentages of the database are searched once we move on to much larger data sets.

One possible baseline is to run kernel PCA followed by standard LSH. The disadvantage of such an approach is that it would introduce an extra level of approximation—information is lost when performing kernel PCA, and further information loss occurs when hashing on top of the PCA embedding. KLSH, on the other hand, is a means to perform LSH on top of the original embeddings to directly compute locality-sensitive hash functions. We found this to be borne out in practice—on the Caltech-101 we observed higher average accuracy with our approach than with a kernelPCA+LSH baseline.

As another baseline, we also ran this experiment using the metric-tree (M-tree) approach developed in [5], using the implementation provided online by the authors. This is a well-known exact search algorithm that accepts arbitrary metrics as input. To map the CORR kernel values $\kappa(x, y)$ to distance values, we compute: $D(\boldsymbol{x}, \boldsymbol{y}) = (\kappa(\boldsymbol{x}, \boldsymbol{x}) + \kappa(\boldsymbol{y}, \boldsymbol{y}) - 2\kappa(\boldsymbol{x}, \boldsymbol{y}))^{\frac{1}{2}}$.

While accuracy with this method is consistent with a linear scan, its search time was quite poor; of the n database items, the M-tree required searching $n \pm 30$ examples to return the first NN. The speed was unchanged when we varied the splitting function (between the generalized hyperplane and balanced strategies), the promotion method used to promote objects in the parent role (random, maximum upper bound on distances, or minimum maximum radius policies), or the minimum node utilization parameter (which we tested for values from 0 to 0.5 in increments of 0.1). The likely problem is that the distribution of distances between the indexed objects has relatively low variance, making the tree-based measure less effective. Thus, even though the M-tree can operate with arbitrary metrics, KLSH has a clear performance advantage for this data.

Local Patch Indexing. Our second experiment uses the patch data set [14] associated with the Photo Tourism project [26]. It consists of local image patches of Flickr photos of various landmarks. The goal is to compute correspondences between local features across multiple images, which can then be provided to a structure-from-motion algorithm to generate 3D reconstructions of the photographed landmark [26]. Thus, one critical sub-task is to take an input patch and retrieve its corresponding patches within any other images in the database—another large-scale similarity search problem.

We use the $n = 100,000$ image patches provided for the Notre Dame Cathedral. Since the goal is to extract ideally *all* relevant patches, we measure accuracy in terms of the recall rate. We consider two measures of similarity between the patches' SIFT descriptors: Euclidean distance (L_2), and a Gaussian RBF kernel computed with a metric learned on top of L_2 (again, using [7] to learn the parameters). For the first measure, standard LSH can be applied. For the second, only KLSH is applicable, since the underlying embedding of the data is not easily computable (in fact, its dimension is technically infinite).

Figure 2 shows the results. The two lower curves (nearly overlapping) show the recall results when using either a linear scan or LSH with the SIFT vectors and Euclidean distance. The two higher curves show recall using a Gaussian RBF kernel on top of the learned SIFT features. In both cases, the recall rate relative to a linear scan is hardly affected by the hashing approximation. However, with KLSH, a stronger kernel can be used with the hashing (the RBF), which improves the accuracy of the results. For this data set, KLSH requires searching only 0.26% of the database on average, a significant improvement.

Again, the M-tree baseline offered no better performance than a linear scan on this data set; as earlier, we suspect this is because of the high-dimensionality of the features, and the low variance in the inter-feature distances.

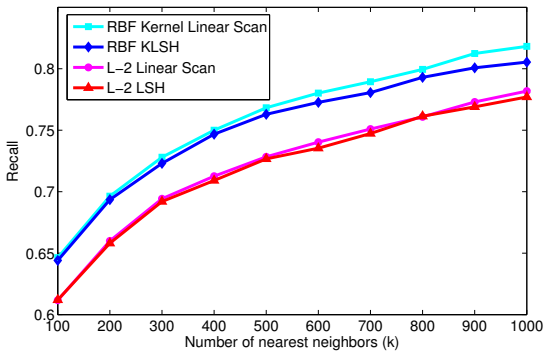

Figure 2. Results on the Photo Tourism data set. The lower two curves show LSH or linear scan applied for L_2 search on SIFT descriptors; the upper two curves show KLSH or linear scan applied for a Gaussian RBF kernel after metric learning is run on the SIFT vectors. For both, recall rates using hashing are nearly equivalent to those using linear scan search. However, now that our KLSH enables hashing with the RBF, more accurate retrieval is possible in sub-linear time.

Large-Scale Image Search with Tiny Images. Finally, we provide results using a very large-scale data set of **80 million** images, provided by the authors of [27]. Here, the task is content-based image retrieval. The images are "tiny": 32 $\times$ 32 pixels each. Following [31], we use a global Gist descriptor for each image, which is a 384-d vector describing the texture within localized grid cells. Given that the images were collected with keyword-based Web crawlers, the data set does not have definitive ground truth to categorize the images. However, we can use the data to qualitatively show the kinds of images that are retrieved, to quantitatively show how well KLSH approximates a linear scan, and to confirm that our algorithm is amenable to rapidly searching very large image collections.

For this experiment we use only 130 hash key permutations (which corresponds to $\epsilon \approx 2.74$) in order to restrict the memory overhead when storing the sorted orders. To counter the accuracy loss that a higher value of ϵ may entail, we increase the B parameter to 100 so as to search nearby hash buckets and thereby explore more hashing matches. We use $b = 300$ bits per image and randomly select 100 images as queries. The KLSH parameters are fixed at $t = 30$ and $p = 300$, as before. Here we apply KLSH to a Gaussian RBF kernel over the Gist features.

On average, KLSH searched only 0.98% of the entire database in order to produce the top 10 approximate nearest neighbors per query. (We can easily decrease this percentage even further by using a smaller value of B; our choice may have been too liberal.) Figure 4 shows example retrieval results. The leftmost image in each set is the query, the top row shows the linear scan results, and the row below it shows our KLSH results. Ideally, linear scan and KLSH would return the same images, and indeed they are often

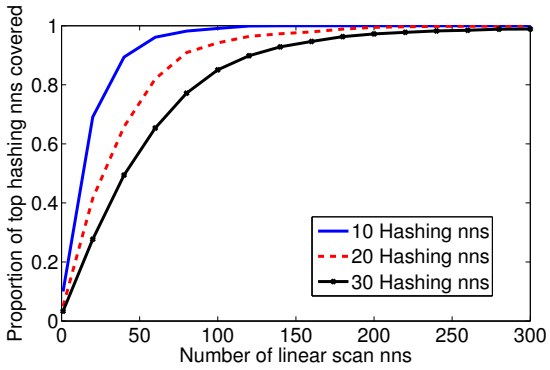

Figure 3. Results on the Tiny Images data set. The plot shows how many linear scan neighbors are needed to cover the first 10, 20, or 30 hashing neighbors.

close and overlap. The Gist descriptors appear to perform best on images with clearly discernable shape structure—sometimes the descriptors are insufficient, and so both the linear scan and hashing results do not appear to match the query. This experiment demonstrates the practical scalability of our hashing approach for very large image databases.

We can quantify the results on this data set by determining how well the hashing results approximate the ideal linear scan. To do this, we looked at how many of the top ranked linear scan neighbors must be included before the top 10, 20, or 30 hashing results are all accounted for. The results appear in Figure 3, where we see for example that for the top ten hashing neighbors, 90% will be within the top 50 linear scan neighbors on average.

Finally, the average running times for a NN query on Tiny Images with our Matlab code fully optimized are: 0.001s for hash key construction and permutation (overhead), 0.13s for binary search to find approximate NN's (overhead), and 0.44s for searching the approximate NN's. In contrast, a linear scan takes 45 seconds per query on average, assuming that all images are stored in memory. However, due to their size, the images must typically be stored on disk or across several machines; in contrast, the binary data can easily be stored on a single machine.

Conclusions. We presented a general algorithm to draw hash functions that are locality-sensitive for arbitrary kernel functions, thereby permitting sub-linear time approximate similarity search. This significantly widens the accessibility of LSH to generic kernel functions, whether or not their underlying feature space is known. Since our method does not require assumptions about the data distribution or input, it is directly applicable to many existing useful measures that have been studied for image search and other domains.

Acknowledgements. We thank the anonymous reviewers for helpful comments, and Rob Fergus, Antonio Torralba, and Bill Freeman for the Tiny Image data. This research was supported in part by NSF CAREER award 0747356, a

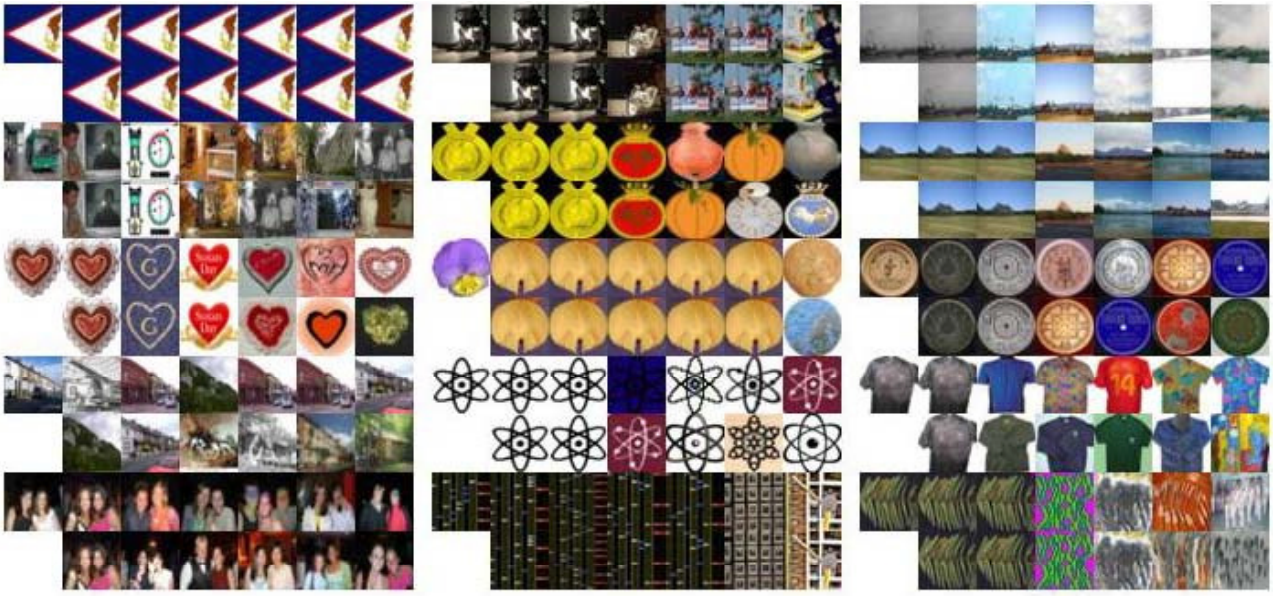

Figure 4. Example results for the Tiny Image data set. For each example, the query appears as the leftmost image. The top row corresponds to results of a linear scan with the Gist vectors; the second row corresponds to the KLSH results. Our method often retrieves neighbors very similar to those of the linear scan, but does so by searching only 0.98% of the 80 Million database images.

Microsoft Research New Faculty Fellowship, DARPA VIRAT, NSF EIA-0303609, and the Henry Luce Foundation.

References

[1] V. Athitsos, J. Alon, S. Sclaroff, and G. Kollios. BoostMap: A Method for Efficient Approximate Similarity Rankings. In *CVPR*, 2004.

[2] J. Beis and D. Lowe. Shape Indexing Using Approximate Nearest-Neighbour Search in High Dimensional Spaces. In *CVPR*, 1997.

[3] M. Charikar. Similarity Estimation Techniques from Rounding Algorithms. In *ACM Symposium on Theory of Computing*, 2002.

[4] O. Chum, J. Philbin, and A. Zisserman. Near Duplicate Image Detection: min-hash and tf-idf Weighting. In *BMVC*, 2008.

[5] P. Ciaccia, M. Patella, and P. Zezula. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. In *Proc Int'l Conf on Very Large Data Bases*, Aug. 1997.

[6] 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.

[7] J. Davis, B. Kulis, P. Jain, S. Sra, and I. Dhillon. Information-Theoretic Metric Learning. In *ICML*, 2007.

[8] L. Fei-Fei, R. Fergus, and P. Perona. Learning Generative Visual Models from Few Training Examples: an Incremental Bayesian Approach Tested on 101 Object Cateories. In *Workshop on Generative Model Based Vision*, Washington, D.C., June 2004.

[9] 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.

[10] A. Gionis, P. Indyk, and R. Motwani. Similarity Search in High Dimensions via Hashing. In *Proceedings of the 25th International Conference on Very Large Data Bases*, 1999.

[11] M. Goemans and D. Williamson. Improved Approximation Algorithms for Maximum Cut and Satisfiability Problems Using Semidefinite Programming. *JACM*, 42(6):1115–1145, 1995.

[12] K. Grauman and T. Darrell. Pyramid Match Hashing: Sub-Linear Time Indexing Over Partial Correspondences. In *CVPR*, 2007.

[13] J. Hoffmann-Jorgensen and G. Pisier. The Law of Large Numbers and the Central Limit Theorem in Banach Spaces. *Ann. Probability*, 4:587–599, 1976.

[14] G. Hua, M. Brown, and S. Winder. Discriminant Embedding for Local Image Descriptors. In *ICCV*, 2007.

[15] P. Indyk and R. Motwani. Approximate Nearest Neighbors: Towards Removing the Curse of Dimensionality. In *30th Symposium on Theory of Computing (STOC)*, 1998.

[16] P. Jain, B. Kulis, and K. Grauman. Fast Image Search for Learned Metrics. In *CVPR*, 2008.

[17] H. Ling and S. Soatto. Proximity Distribution Kernels for Geometric Context in Category Recognition. In *ICCV*, 2007.

[18] D. Nister and H. Stewenius. Scalable Recognition with a Vocabulary Tree. In *CVPR*, 2006.

[19] S. Obdrzalek and J. Matas. Sub-linear Indexing for Large Scale Object Recognition. In *BMVC*, 2005.

[20] A. Rahimi and B. Recht. Random Features for Large-Scale Kernel Machines. In *NIPS*, 2007.

[21] J. Rice. *Mathematical Statistics and Data Aanalysis*. Duxbury Press, 2001.

[22] R. Salakhutdinov and G. Hinton. Semantic Hashing. In *ACM SIGIR*, 2007.

[23] B. Schölkopf, A. Smola, and K.-R. Müller. Nonlinear Component Analysis as a Kernel Eigenvalue Problem. *Neural Computation*, 10:1299–1319, 1998.

[24] G. Shakhnarovich, T. Darrell, and P. Indyk, editors. *Nearest-Neighbor Methods in Learning and Vision: Theory and Practice*. The MIT Press, 2006.

[25] G. Shakhnarovich, P. Viola, and T. Darrell. Fast Pose Estimation with Parameter-Sensitive Hashing. In *ICCV*, 2003.

[26] N. Snavely, S. Seitz, and R. Szeliski. Photo Tourism: Exploring Photo Collections in 3D. In *SIGGRAPH*, 2006.

[27] A. Torralba, R. Fergus, and W. T. Freeman. 80 Million Tiny Images: A Large Dataset for Non-Parametric Object and Scene Recognition. *TPAMI*, 30(11):1958–1970, 2008.

[28] A. Torralba, R. Fergus, and Y. Weiss. Small Codes and Large Image Databases for Recognition. In *CVPR*, 2008.

[29] J. Uhlmann. Satisfying General Proximity / Similarity Queries with Metric Trees. *Information Processing Letters*, 40:175–179, 1991.

[30] M. Varma and D. Ray. Learning the Discriminative Power-Invariance Trade-Off. In *ICCV*, 2007.

[31] Y. Weiss, A. Torralba, and R. Fergus. Spectral Hashing. In *NIPS*, 2009.

[32] J. Yang, X. Gao, D. Zhang, and J. Yang. Kernel ICA: An Alternative Formulation and its Application to Face Recognition. *Pattern Recognition*, 38(10):1784–1787, October 2005.

[33] H. Zhang, A. Berg, M. Maire, and J. Malik. SVM-KNN: Discriminative Nearest Neighbor Classification for Visual Category Recognition. In *CVPR*, 2006.

[34] J. Zhang, M. Marszalek, S. Lazebnik, and C. Schmid. Local Features and Kernels for Classification of Texture and Object Categories: A Comprehensive Study. *IJCV*, 73(2):213–238, 2007.