CompactKdt: Compact Signatures for Accurate Large Scale Object Recognition

Mohamed Aly *
Computational Vision Lab
Caltech
vision.caltech.edu/malaa
malaa@vision.caltech.edu

Mario Munich
Evolution Robotics, Inc.
www.evolution.com
mario@evolution.com

Pietro Perona
Computational Vision Lab
Caltech
vision.caltech.edu
perona@caltech.edu

Abstract

We present a novel algorithm, Compact Kd-Trees (CompactKdt), that achieves state-of-the-art performance in searching large scale object image collections. The algorithm uses an order of magnitude less storage and computations by making use of both the full local features (e.g. SIFT) and their compact binary signatures to build and search the K-Tree. We compare classical PCA dimensionality reduction to three methods for generating compact binary representations for the features: Spectral Hashing, Locality Sensitive Hashing, and Locality Sensitive Binary Codes. CompactKdt achieves significant performance gain over using the binary signatures alone, and comparable performance to using the full features alone. Finally, our experiments show significantly better performance than the state-of-the-art Bag of Words (BoW) methods with equivalent or less storage and computational cost.

1. Introduction

Large scale object recognition is an important problem with many applications. A typical scenario is having a large database of book covers, where users can take a photo of a book with a cell phone and search the database which retrieves information about that book[1]. The database should be able to handle millions or even billions of images, imagine for example indexing all books and DVD covers (movies, music, games, ... etc.) in the world.

There are two major approaches for building such databases: Bag of Words (BoW) approaches [17, 14, 15, 9, 3] and Feature Matching (FM) approaches [10, 3]. In the first, each image is represented by a histogram of occurrences of quantized features, and search is usually done efficiently using an Inverted File (IF) structure [21]. The second works by getting the approximate nearest neighbors for the features in the query image by searching all the features of the database images using an efficient search method (e.g. Kd-Trees [10]). The first has the advantage of using an order of magnitude less storage, however their performance is far worse [3] (see also Fig. 5).

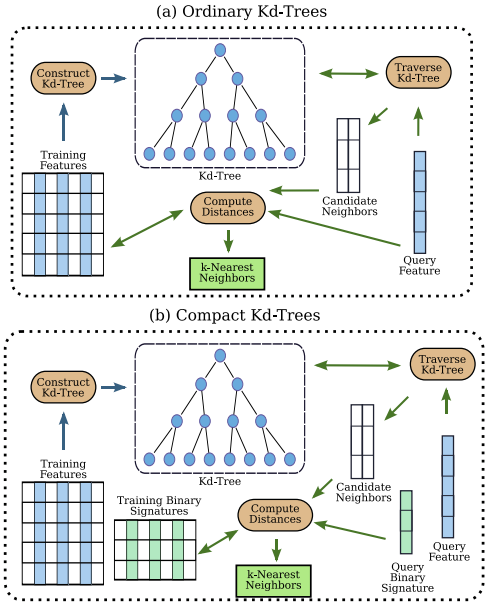


Figure 1: **Schematic of Ordinary and Compact Kd-Tree**. (a) In ordinary Kd-Trees, the training features (*left*) are used to construct the tree, see Alg. 2. Given a query feature (*right*), the tree is traversed for candidate nearest neighbors, which are then filtered by computing distances using the training features to obtain the k-nearest neighbors (*bottom*), see Alg. 3. (b) In Compact Kd-Trees, the full training features are used to construct the tree as usual, and can be discarded after that. Given a query feature, the tree is traversed for candidate nearest neighbors using the full query feature descriptor. However, distances are computed using the query binary signature and the training binary signatures to obtain the k-nearest neighbors, see Alg. 4.

*Works now at Google, Inc.

[1] See for example www.google.com/mobile/goggles

Most of the storage required by FM approaches is for storing the individual features. Therefore, reducing the storage required by the feature descriptors is highly desirable. This can be done in at least two ways: (a) reducing the number of features to be stored by discarding unstable useless features [19], and (b) reducing the memory footprint of the individual feature descriptors.

In this work, we focus on representing the features in as few bits as possible while still retaining good performance. We compare standard PCA dimensionality reduction to three methods for obtaining compact binary signatures from vectors: Spectral Hashing (SH) [20], Locality Sensitive Hashing (LSH) [6, 4], and Locality Sensitive Binary Codes (LSBC) [16] . We present a novel method to achieve an order of magnitude less storage by making use of both the full features and the binary codes while retaining comparable performance. Finally, we compare our method to the state-of-the-art Hamming Embedding BoW method [9] and show report better performance with equivalent or less storage.

We make the following contributions:

1. We provide a thorough comparison of different techniques for compressing SIFT features using binary signatures in the context of large scale object recognition. This is the first such work up to our knowledge.
2. We present a novel algorithm, CompactKdt, that provides two advantages:
   (a) An order of magnitude less storage with recognition performance close to using the full descriptors. Specifically, we can achieve performance within 5-16% of using the full features with 8-14 fold saving in storage.
   (b) An order of magnitude less computations for comparing features since the Euclidean distance is replaced by extremely fast XOR and bit count operations.
3. We report better recognition performance than the state-of-the-art Hamming Embedding Bag of Words method [9] with equivalent or less storage.

2. Algorithm Overview

The basic Feature Matching (FM) image search method is outlined in Alg. 1. Several data structures have been used for fast approximate search, including Kd-Trees (Kdt) [10], Locality Sensitive Hashing (LSH) with different hash functions [6, 18, 4], and Hierarchical K-Means (HKM) [13, 12].

In this work, we focus on using randomized Kd-Trees, which have been shown to provide excellent recognition performance and run time [3, 12]. We use the Best-Bin-First variant of Kd-Trees [5, 10] which utilizes a priority queue

Algorithm 1 Feature Matching Image Search

1. For every database image i, extract local features $\{f_{ij}\}_j$.
2. Build a data structure $D(f_{ij})$ to allow fast approximate nearest-neighbor search.
3. For every query image q, extract its local features $\{f_{qj}\}_j$.
4. For every such feature f_{qj}, search the structure D for its nearest neighbor n_{qj}.
5. Accumulate scores s_i for every database image such that $s_i \propto \#\{\mathrm{id}(n_{qj}) = i\}$ i.e. s_i is proportional to the number of neighbors in image i.
6. (Optional) Process the top ranked k images for geometric consistency of features' locations and scales.

Parameter	Description	Typical Value
I	# images	100K
b	# bytes/feature dim	1
d	feature dimension	128
F	# features/image	1,000
T	# kd-trees	1
t	# backtracking steps	150
L	depth of the tree	24

Table 1: Kd-Trees FM Parameter definitions. See Sec. 2.

[8] to search through all the trees simultaneously. Alg. 2 outlines the process of constructing a set of randomized Kd-Trees and Alg. 3 outlines the search process.

We can express the storage required for Kd-Trees FM algorithm as:

$$S_{Kdt} \approx IF(bd + T\left\lceil \frac{\log_2 IF}{8} \right\rceil) + 2^{L+1}T$$

where the parameters are defined in Table 1. The first term is for storing the feature descriptors. A Kdt with L levels has $2^L - 1$ internal nodes. For every internal node per tree, we need to store 2 bytes, the dimension to split on and split value (last term). For every leaf, we need to store the indices of the features that belong to that leaf (second term), the total of which is IF. The tree can be stored in memory as an array, and so we do not need any pointers [8]. Given the typical values in Table 1, the storage would take $S_{Kdt} \approx 13.2$ GB, out of which it takes 12.8 GB just to store the feature descriptors i.e. ~96% of the total storage is taken by the features!

The number of operations per image can be expressed as

$$T_{Kdt} \approx FLt + Ft\frac{FI}{2^L} \times (2d + 1)$$

where t is the number of backtracking steps. The first term is for traversing the tree to the leaves, the second is for

Algorithm 2 Randomized Kd-Trees Construction

Input: A set of vectors $\{x_i\} \in \mathbb{R}^N$
Output: A set of binary Kd-Trees $\{T_t\}$. Each internal node has a split (dim, val) pair where dim is the dimension to split on and val is the threshold such that all points with $x_i[dim] \leq val$ belong to the left child and the rest belong to the right child. The leaf nodes have a list of indices to the features that ended up in that node.
Operation: For each tree T_t:

1. Assign all the points $\{x_i\}$ to the root node
2. For very *node* in the tree visited in Breadth-First order, compute the split as follows:
   (a) For each dimension $d = 1 \ldots N$, compute its mean $mean(d)$ and variance $var(d)$ from the points in that node
   (b) Choose a dimension d_r at random from the variances within 80% of the maximum variance
   (c) Choose the split value as the mean of that dimension $mean(d_r)$
   (d) For all points that belong to this node: if $x[d_r] \leq mean(d_r)$ assign x to $left[node]$, otherwise assign x to $right[node]$

Algorithm 3 Randomized Kd-Trees Search

Input: A set of Kd-Trees $\{T_t\}$, a set of vectors $\{x_i\} \in \mathbb{R}^N$ used to build the trees, a query vector $q \in \mathbb{R}^N$, maximum number of backtracking steps s
Output: A set of k nearest neighbors $\{n_k\}$ with their distances $\{d_k\}$ to the query vector q.
Operation:

1. Initialize a priority queue Q with the root nodes of the t trees by adding $branch = (t, node, val)$ with $val = 0$. The queue is indexed by $val[branch]$ i.e it returns the branch with smallest val.
2. $count = 0$. $list = []$
3. While $count \leq s$
   (a) Retrieve the top $branch$ from Q.
   (b) Descend the tree defined by $branch$ till $leaf$, adding unexplored branches on the way to Q.
   (c) Add the points in $leaf$ to $list$.
4. Find the k nearest neighbors to q in $list$ and return the sorted list $\{n_k\}$ and their distances $\{d_k\}$.

computing the distance to the candidate nearest neighbors, and the third is for finding the minimum among the candidates. For the typical values in Table 1, it takes $T_{Kdt} \approx 232$ MFLOP (FLoating Point Operation) per image.

# bits / feature	Total Storage (GB)	Features Compression	Total Compression
1024 (Full)	13.2	1	1
512	6.8	2	1.9
256	3.6	4	3.6
128	2	8	6.5
64	1.2	16	10.7
32	0.8	32	15.8

Table 2: **Storage Savings for using Binary Signatures**. The second column depicts the storage used when compressing the SIFT features with binary signatures of different lengths, using FM Kd-Trees with typical parameter values in Table 1. The third column shows the compression factor for storing the features relative to the first row (full features), while the fourth column shows the total compression factor (features + Kd-Tree) achieved. See Sec. 3.

3. Compact Binary Signatures

In this work we focus on methods for producing compact binary signatures for the features. These methods take in N-dimensional vectors $x \in \mathbb{R}^N$ and produce B-dimensional binary signatures $b^x \in \{0,1\}^B$. The basic requirement is such that the hamming distance $d_H(b^x, b^y) = \sum_i \{b_i^x \neq b_i^y\}$ between the binary signatures b^x and b^y of two features x and y provides a good approximation to the Euclidean distance $d_E(x,y) = ||x-y||_2$.

Using binary signatures has two advantages:

1. Storage reduction: using for example 64-bit signatures instead of 128 bytes features, we can reduce the storage for Kd-Trees FM to 1.2 GB instead of 13.2 GB, an order of magnitude less, see Table 2.
2. Run time Speedup: in the search phase of the Kd-Tree (Alg. 3), we need to search the list of candidate nearest neighbors to get the k closest ones. Instead of computing the Euclidean distance on 128 dimensions (which includes 128 additions + 128 multiplications), for 64-bit signatures we do an XOR and bit count (one XOR + 8 table lookups and additions), an order of magnitude less operations. The search time is modified as follows:

$$T_{Kdt-Bin} \approx FtL + Ft\frac{FI}{2^L}(2\left\lceil\frac{d}{8}\right\rceil + 1)$$

where the middle term is for computing the hamming distance of two d-bit vectors with lookup tables for every byte. For 64-bit signatures, this reduces the operations required from 232 MFLOP down to ~18 MFLOP per image.

We consider three methods:

- **Spectral Hashing** (SH): It aims at producing balanced binary codes that minimize the mean hamming distance between similar codes [20]. This is shown to be equivalent to a graph partitioning problem which is *NP*-hard. However, by relaxing the constraints it can be solved efficiently by finding the principal components of a training set, and approximating and thresholding the eigenvectors of the graph Laplacian along the principal components [20].
- **Locality Sensitive Hashing** (LSH): It uses projections on random hyperplanes such that the output binary signature approximates the dot product [6]. Specifically, the i^{th} bit is defined as: $b_i = \text{sign}\langle x, a^i \rangle$ where $a^i \in \mathbb{R}^N$ is a random unit vector $a^i \sim N(\mathbf{0}, I)$. Unlike SH, LSH is data independent and does not require any training.
- **Locality Sensitive Binary Codes** (LSBC): This was derived as a data independent variant of SH while having strong theoretical guarantees [16]. It generates binary codes using random projections such that the expected hamming between the signatures equals the value of a shift-invariant kernel between the two features. For exponential kernels $K(x,y) = \exp(||x-y||^2/\gamma)$, the i^{th} bit is defined as $b_i = 0.5\,[1 + Q_t\,(\cos(\langle \omega, x \rangle + b))]$ where $t \sim Unif[-1,1]$, $Q_t(u) = \text{sign}(u+t)$, $\omega \sim N(0, \gamma I) \in \mathbb{R}^N$, and $b \sim Unif[0, 2\pi]$.

4. Experimental Setup

4.1. Datasets

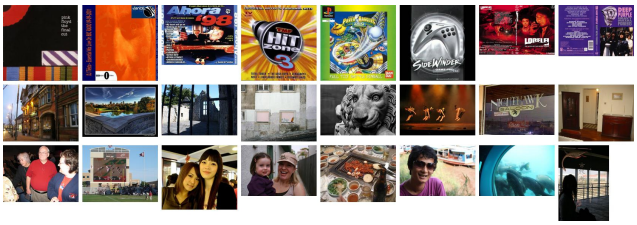
We use three of the datasets used in [2]. Specifically, we have two kinds of datasets:

1. **Distractors**: images that constitute the bulk of the database to be searched. They are chosen to have similar statistics to the images to be searched.
2. **Probe**: labeled images of objects, two types per object: **(a) Model Image**: the ground truth image that is added to the distractor set, and is the target to be retrieved for that object, **(b) Probe Images**: used for querying the database, representing the object in the model image from different view points, lighting conditions, scales, ... etc.

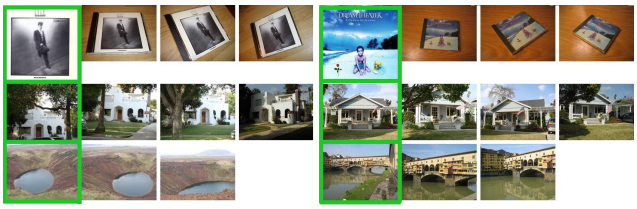
Distractor Datasets

- **D1: Caltech-Covers** A set of ~ 100K images of CD/DVD covers used in [3].
- **D2: Flickr-Buildings** A set of ~1M images of buildings collected from flickr.com.
- **D3: Flickr-Geo** A set of ~1M geo-tagged images collected from flickr.com.

Probe Sets



(a) **Example distractor images**. Each row depicts a different distractor set D1, D2, and D3 respectively.



(b) **Example probe images**. Each row depicts a different probe set P1, P2, and P3 respectively. Each row shows three examples of a model image (in *green* border) with its probe images *(right)*

Figure 2: **Example Dataset Images**. See Sec. 4.1 and Table 3.

Probe Sets			
	total	#model	#probe
P1	485	97	388
P2	750	125	525
P3	957	233	724

Evaluation Sets		
Set	Distractor	Probe
1	D1	P1
2	D2	P2
3	D3	P3

Table 3: Probe Sets Properties and Evaluation Sets. See Sec. 4.1.

- **P1: CD Covers:** A set of 5×97=485 images of CD/DVD covers. The model images come from *freecovers.net* while the probe images come from the dataset used in [13]. This was also used in [3].
- **P2: Pasadena Buildings** A set of 6×125=750 images of buildings around Pasadena, CA from [3]. The model image is image2 (frontal view in the afternoon), and the probe images are images taken at two different times of day from different viewpoints.
- **P3: INRIA Holidays** a set of 957 images, which forms a subset of images from [9], with groups of at least 3 images. There are 233 model images and 724 probe images. The first image in each group is the model image, and the rest are the probe images.

4.2. Setup

We used three different evaluation sets, where in each we use a specific distractor/probe set pair with similar statis-

tics, see Table 3. Evaluation was done by choosing 100K images from the distractor set in addition to all the model images from the probe set i.e. we have only one ground truth image per object. For every probe image (multiple per object), we get a ranked list of the images in the distractor + model sets, where highest ranked images are more likely to be the corresponding ground truth model image. Performance is measured as *precision@1* i.e. the percentage of probe images whose ground truth image is the top ranked image. We did not notice different trends when considering precision@k for different values of k, the curves were simply shifted upwards. All results reported in the paper are without any geometric consistency post-processing.

We want to emphasize the difference between the setup adopted here and used in [3, 1] and the setup used in other "image retrieval"-like papers [9, 7, 14]. In our setup, we have only ONE correct ground truth image to be retrieved and several probe or query images, while in the other setting there are a number of images that are considered correct retrievals. This setup is motivated by the application we consider: identifying the correct object in the database, where the database has only ONE image per object (e.g. searching for a book in a database of all book covers, where the database contains one image per book). This also motivated our choice of using precision@1 as the performance metric.

We use SIFT [10] feature descriptors (128 dimensions) with hessian affine [11] feature detectors. We used the binary available from tinyurl.com/vgg123. All experiments were performed on machines with Intel dual Quad-Core Xeon E5420 2.5GHz processor and 32GB of RAM. For SH, we used the code available from the authors of [20]. We implemented the rest of the algorithms using Matlab and Mex/C++ scripts. For SH, we used a random set of 2.5M for training. For BoW, the dictionary is built from a random set of 10M features.

5. Binary Signature Comparison

By reducing the dimensions of the input features, we can significantly reduce the amount of storage required. Instead of using the full SIFT features, we can instead compute binary signatures (sec. 3) or PCA and use that instead. The binary signatures are then used to build the Kd-Trees, and to compute the K-nearest neighbors (see Alg. 1-3) . Fig. 3 shows the results of using the binary signatures compared to using Principal Component Analysis (PCA) and to using the full features with Kd-Trees. For PCA, all the dimensions were quantized to 8-bits e.g. with 128 bits we only kept the top 16 dimensions. We note the following:

- The Eval Sets have different difficulty, with Set 1 the easiest (full feature precision of about 99%), Set 2 harder (~79% precision), and Set 3 the hardest (~66% precision).
- PCA is quite competitive with the binary signatures for sizes starting at 128 bits per feature (16 PCA dimensions). With using 256 bits (32 PCA dimensions), we can reach performance within 90% of the full features while achieving ~3.6 compression i.e. using almost fourth of the storage.
- Below 128 bits, the binary signatures are significantly better than PCA. SH provides the best performance, followed by LSBC then LSH. However, beyond 128 bits, the performances of SH & LSBC deteriorate while that of LSH becomes better.
- At 128 bits (~6 fold saving in total storage) we lose ~2% of the precision using full features for Eval Set 1, ~28% for Eval Set 2, and ~21% for Eval Set 3.
- At 64 bits (~10 fold saving in total storage) we lose ~4% of the precision using full features for Eval Set 1, ~36% for Eval Set 2, and ~31% for Eval Set 3.

Though we can achieve ~3.6 compression factor with only minimal loss to the performance which is promising, we are seeking even higher compression rates with similar performance. We will show next how to achieve this.

Algorithm 4 Compact Kd-Trees (CompactKdt)

1. For every feature in the training set $\{f_{ij}\} \in \mathbb{R}^N$, compute its binary signature $\{b_{ij}\} \in \{0,1\}^B$.
2. Build the set of Kd-Trees $\{T_t(f_{ij})\}$ using the full feature set $\{f_{ij}\}$, then only store the set of binary signatures $\{b_{ij}\}$
3. For every query feature f_{qj} compute its binary signature b_{qj}.
4. Use the full query feature f_{qj} to retrieve a short list of candidate nearest neighbor features $\{n^l_{qj}\}$.
5. Search the short list $\{n^l_{qj}\}$ of features using their binary signatures to get the closest feature n_{qj}.
6. Accumulate the scores for the image that contains n_{qj}, as in Alg. 1.

6. Compact Kd-Trees (CompactKdt)

Using the binary signatures provides an order of magnitude saving in storage over using the full features, but it still incurs a loss anywhere between 2% and 36% in precision. We can get the best of both worlds by using all the information available i.e. use the full features to get a *good* list of candidate nearest neighbor features, and then use the binary signatures to select the actual nearest features. This idea is easily applicable to Kd-Trees. We build the Kd-Trees using the full training features, then these features can be discarded and we only store their binary signatures. At query time, we traverse the Kd-Trees using the full query features to get the list of candidate close features, but only

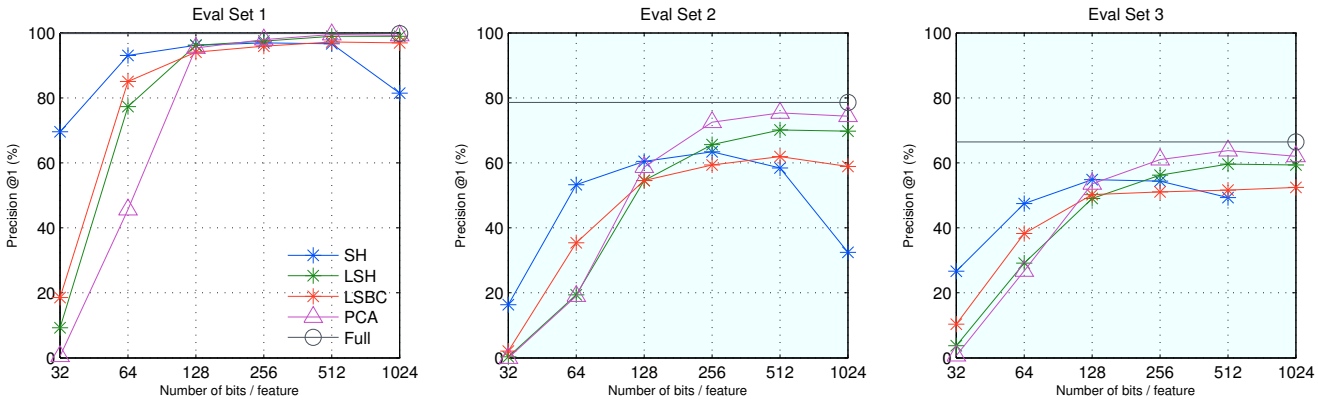


Figure 3: **Recognition Performance for Binary Signatures and PCA**. The X-axis shows the number of bits / feature, while the Y-axis shows the Precision@1. The *black* circles show the performance using the full features (128 bytes). PCA outperforms the binary signatures beyond 128 bits, while the binary signatures (especially SH) perform very well for 32 & 64 bits. See Sec. 3 & 5. PCA achieves performance within 90% of using the full features wit ~3.6 compression factor. To achieve even higher savings, see Fig. 4.

verify the correct nearest neighbor using the binary signatures, see Alg. 4 and Fig. 1. This significantly improves the performance while using the exact same storage and computational cost as using binary signatures with ordinary Kd-Trees, see Table 2 and Sec. 3.

Fig. 4 shows the results of running CompactKdt using the binary signatures and PCA from Sec. 3 compared to using SH with ordinary Kd-Trees which provides the best performance from Fig. 3, all using 1 Kd-Tree. We note the following:

- CompactKdt with SH gives the best results, while using LSH, LSBC, and PCA are worse.
- CompactKdt achieves significant performance improvements over using the binary signatures with ordinary Kd-Trees, while using the same storage. For example, using 32-bits, we achieve 35-200% improvements.
- For Eval Set 1, we reach within ~5% of the full feature precision using only 32 bits with SH-CompactKdt compared to SH alone, a 14 fold saving in storage.
- For Eval Sets 2 & 3, we reach within ~10% (16%) of the full precision using only 96 bits (64 bits) for ~8 fold (10 fold, respectively) of saving in storage. Therefore, if the application is willing to lose 10-16% of the precision, we can achieve a compression factor of 8-10 times.
- Overall, we can save anywhere between 8-14 times in the total storage and stay within 5-16% of the best performance. This is a significant storage saving, since we can store an order of magnitude more images on the same machine with losing minimal performance.

Parameter	Description	Typical Value
I	# images	100K
F	# features/image	1,000
b	# bytes/feature dim	1
d	feature dimension	128
W	# visual words	10^6
B	# bytes / signature	8

Table 4: BoW Parameter definitions. See Sec. 7.

Algorithm	Storage (GB)	Comp. (MFLOP/im)	Query Time (msec/im)
Kd-Tree	13.2	232	232
CompactKdt-64	1.23	18	18
CompactKdt-48	1.03	15	15
BoW-HE	1.1	57	57

Table 5: CompactKdt & BoW Storage and Computational Cost Comparison using the values in Tables 1 & 4. See Sec. 2, 3, 6, and 7 and Fig. 5. Storage is in GB, computations are in MFLOP/image, and query time is in msec/image (using a 1GFLOPS processor).

7. Comparison with Bag of Words

Bag of Words (BoW) techniques have been shown to take an order of magnitude less storage than FM methods, however, they suffer from poor performance, see Fig. 5. The storage taken by BoW methods can be expressed as

$$S_{BoW} = Wbd + IF\left\lceil \frac{\log_2 I}{8} \right\rceil$$

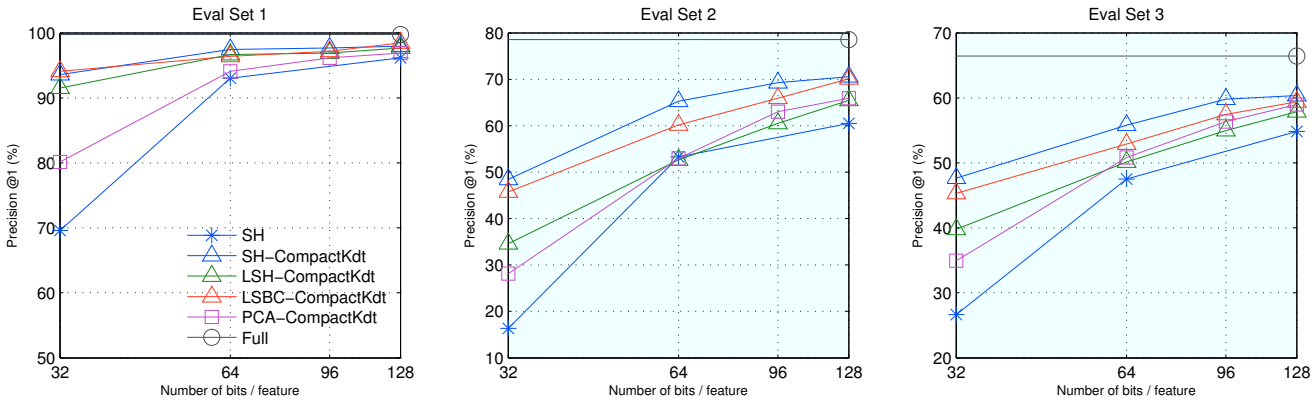


Figure 4: **Recognition Performance for Compact Kd-Trees (CompactKdt)**. The X-axis shows the number of bits / feature, while the Y-axis shows the Precision@1. The *black* circles show the performance using the full features (128 bytes), while the *blue* stars show the SH with ordinary Kd-Trees which provided the best performance from Fig. 3. We note that CompactKdt provides significant performance gains while using the same storage as using ordinary Kd-Trees with binary signatures. CompactKdt with SH gives the best performance, followed by LSBC, LSH, and PCA. See Sec. 6.

where the parameters are defined in Table 4. The first term is for storing the visual words, and is negligible. The second term is for storing the inverted file [21], where for every entry in the word list, we need to store the index of the image the feature points to. We do not need to store word counts for large dictionaries as the histogram usually becomes sparse and binary [9]. For Hamming Embedding (HE) [9], the storage needed is as follows

$$S_{HE} = Wbd + IF\left(\left\lceil \frac{\log_2 I}{8} \right\rceil + B\right)$$

where we need additional storage for the binary signatures of the features. For the typical values in Table 4, $S_{BoW} =$ 0.3 GB and $S_{HE} = 1.1$ GB compared to $S_{Kdt} = 13.2$ GB, which is an order of magnitude more. Using CompactKdt, however, the storage shrinks to $S_{CompactKdt-64} = 1.23$ GB with 64-bit signatures and to $S_{CompactKdt-48} = 1.03$ GB with 48-bit signatures (see Sec. 6 and Table 2 & 5) which is equivalent to or less than the storage taken by HE.

The number of operations per image for HE is as follows:

$$T_{HE} \approx Ft(\log_2 W + 2d + 1) + 2F\frac{FI}{W}B$$

where the first term is for computing the visual words for the image features using a Kd-Tree with W leaves and $\log_2 W$ levels, and the second term is for computing the Hamming distance between the binary signatures (assuming the features are evenly distributed among the visual words, such that each of the W visual words has $\frac{FI}{W}$ features given FI total features). With typical values in Table 4, this gives $T_{HE} \approx 57$ MFLOP compared to $T_{CompactKdt-64} \approx 18$ MFLOP (see Table 5).

Fig. 5 shows results for: (a) Baseline BoW: the standard BoW method with l_1 histogram normalization and l_1 distance using 1M visual words (*cyan*); (b) HE BoW: with tf-idf weighting, l_2 normalization, l_2 distance, and hamming distance threshold of 25 using 100K visual words (*magenta*); (c) Full SIFT features without compression using 1 Kd-Tree (*black*); and (d) CompactKdt with SH 64-bit (*green*) and 48-bit (*red*) signatures using 1 Kd-Tree. We note the following:

- FM method with full features (*black*) is significantly superior to both Baseline (*cyan*) and HE BoW (*magenta*) methods, at the cost of using an order of magnitude more storage.
- CompactKdt, with either 48 or 64-bits (*green & red*), is clearly superior to both Baseline BoW and HE BoW. It provides significantly superior recognition performance with equivalent or less storage and computational cost.

8. Summary and Conclusions

We presented a novel algorithm, CompactKdt, for reducing the storage of SIFT features using compact binary signatures together with Kd-Tree constructed with the full features. We find that CompactKdt can reduce the computational cost and the storage to levels comparable to BoW methods while retaining the superior performance of FM methods. Specifically, we showed an order of magnitude less storage (~8 fold saving) with performance within 10% of the performance using the full features using 12 bytes per feature. In addition, CompactKdt achieves significantly better performance than the state-of-the-art Hamming Embedding method with equivalent or less storage and compu-

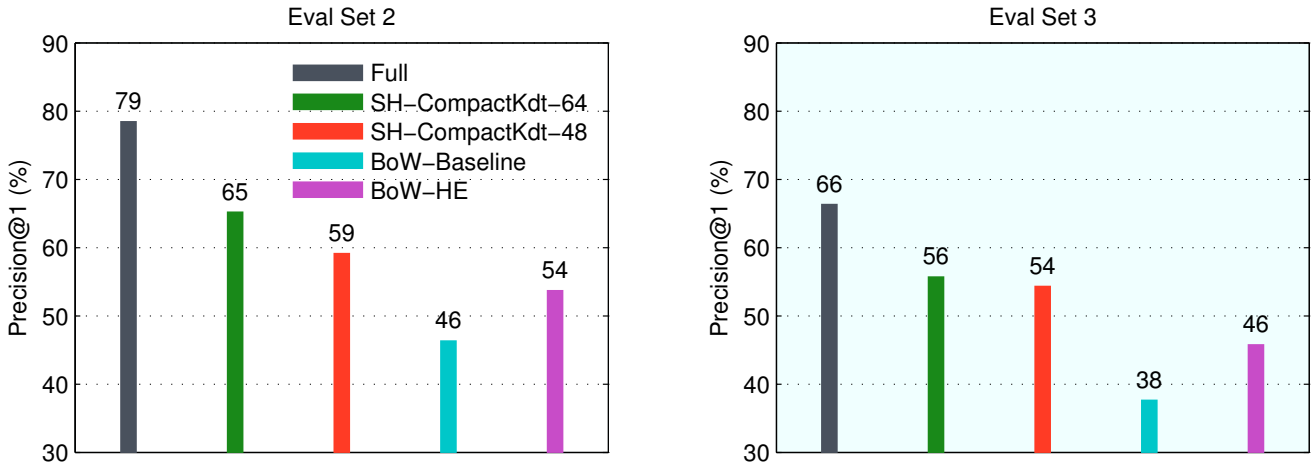


Figure 5: **Comparison of CompactKdt with BoW**. The X-axis shows different algorithms, while the Y-axis shows their Precision@1. The full features with 1Kd-Tree is in *black*. Two bars for CompactKdt with 1 tree using 64-bit signatures (*green*) and 48-bits (*red*). Baseline Bag of Words with 1M visual words is in *cyan*, while Hamming Embedding BoW with 100K visual words is in *magenta*. See Sec. 7.

tational cost.

Acknowledgements

This research was supported by ONR grant N00173-09-C-4005.

References

[1] Mohamed Aly. Online Learning for Parameter Selection in Large Scale Image Search. In *CVPR Workshop OLCV*, June 2010.

[2] Mohamed Aly, Mario Munich, and Pietro Perona. Indexing in large scale image collections: Scaling properties and benchmark. In *WACV*, 2011.

[3] Mohamed Aly, Peter Welinder, Mario Munich, and Pietro Perona. Scaling object recognition: Benchmark of current state of the art techniques. In *ICCV Workshop WS-LAVD*, 2009.

[4] A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1):117–122, 2008.

[5] S. Arya, D.M. Mount, N.S. Netanyahu, R. Silverman, and A.Y. Wu. An optimal algorithm for approximate nearest neighbor searching. *Journal of the ACM*, 45:891–923, 1998.

[6] Moses Charikar. Similarity estimation techniques from rounding algorithms. In *Proc. of 34th STOC*. ACM, 2002.

[7] O. Chum, J. Philbin, and A. Zisserman. Near duplicate image detection: min-hash and tf-idf weighting. In *British Machine Vision Conference*, 2008.

[8] T. Cormen, C. Leiserson, R. Rivest, and C. Stein. *Introduction to Algorithms*. McGraw-Hill, 2001.

[9] H. Jégou, M. Douze, and C. Schmid. Hamming embedding and weak geometric consistency for large scale image search. In *ECCV*, 2008.

[10] David Lowe. Distinctive image features from scale-invariant keypoints. *IJCV*, 2004.

[11] K. Mikolajczyk and C. Schmid. Scale and affine invariant interest point detectors. *IJCV*, 2004.

[12] M. Muja and D. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *VISAPP*, 2009.

[13] D. Nister and H. Stewenius. Scalable recognition with a vocabulary tree. *CVPR*, 2006.

[14] J. Philbin, O. Chum, M. Isard, J. Sivic, and A. Zisserman. Object retrieval with large vocabularies and fast spatial matching. *CVPR*, 2007.

[15] J. Philbin, O. Chum, M. Isard, J. Sivic, and A. Zisserman. Lost in quantization: Improving particular object retrieval in large scale image databases. In *CVPR*, 2008.

[16] M. Raginsky and S. Lazebnik. Locality sensitive binary codes from shift-invariant kernels. In *NIPS*, 2009.

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

[18] T. Terasawa and Y Tanaka. Spherical lsh for approximate nearest neighbor search on unit hypersphere. *Proceedings of the Workshop on Algorithms and Data Structures*, 2007.

[19] P. Turcot and D. Lowe. Better matching with fewer features: The selection of useful features in large database recognition problems. In *ICCV Workshop WS-LAVD*, 2009.

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

[21] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, 2006.