Indexing in Large Scale Image Collections: Scaling Properties and Benchmark

Mohamed Aly
Computational Vision Lab
Electrical Engineering, Caltech
Pasadena, CA 91125, USA
vision.caltech.edu/malaa
malaa@vision.caltech.edu

Mario Munich
Evolution Robotics
Pasadena, CA 91106, USA
www.evolution.com
mario@evolution.com

Pietro Perona
Computational Vision Lab
Electrical Engineering, Caltech
Pasadena, CA 91125, USA
vision.caltech.edu
perona@caltech.edu

Abstract

Indexing quickly and accurately in a large collection of images has become an important problem with many applications. Given a query image, the goal is to retrieve matching images in the collection. We compare the structure and properties of seven different methods based on the two leading approaches: voting from matching of local descriptors vs. matching histograms of visual words, including some new methods. We derive theoretical estimates of how the memory and computational cost scale with the number of images in the database. We evaluate these properties empirically on four real-world datasets with different statistics. We discuss the pros and cons of the different methods and suggest promising directions for future research.

1. Introduction

Indexing in a large scale collection of images has become an important problem of widespread applications. There are currently several smart phone apps that allow the user to take a photo and search a database of stored images e.g. Google Goggles[1], Snaptell[2], and Barnes and Noble[3] app. These image collections typically include images of book covers, CD/DVD covers, retail products, buildings and landmarks. Databases vary in size from 10^5 to 10^7 images, and they can conceivably reach billions of images. The ultimate goal is to identify the database image containing the object depicted in a probe image, e.g. an image of a book cover from a different view point and scale. The correct image can then be presented to the user, together with some revenue generating information, e.g. sponsor ads or referral links.

This application poses three main challenges: storage, computational cost, and recognition performance. For example, if we consider 1 billion images, and store 100KB per image, we need to store 100 TB of data just for the feature descriptors, and naively searching for the nearest feature in a database of 10^{12} features would take 4 minutes on a supercomputer with 1 TFLOPS. Given that most applications involve human-machine interaction, real time indexing is highly desirable.

In this work we explore how the memory usage, computational cost, and recognition performance of the two leading approaches scale with respect to with the number of images in the database. These two approaches are based on extracting local features from the images, e.g. SIFT [16] features, and then indexing these features or some information extracted therefrom. This information is then used to find the best match of a probe image in the database images. The first approach, the *full representation* [16] approach, stores the exact features and uses an efficient data structure to search the huge number of database features and identify candidate images; these are further verified for geometric consistency of the features' locations. We consider five variants of this approach, each using a different data structure to perform the fast approximate search in the features database. The second approach, the *bag-of-words* *[21, 10, 9, 20, 11, 13, 14]* approach, stores only occurrence counts of vector quantized features, and uses efficient search techniques to get candidate image matches. We consider two variants of this approach, the exact inverted file method [23, 20], and the approximate Min-Hash method [9].

We make the following contributions:

1. We provide a benchmark of seven different methods based on the two leading approaches: local feature matching (full representation) and visual words histogram matching (bag of words). We compare recognition rate and matching time as the number of images increases. We also include in the benchmark three new

[1] http://www.google.com/mobile/goggles/
[2] http://www.snaptell.com
[3] http://www.barnesandnoble.com/iphone

Algorithm 1 Basic Algorithm

```
1. Extract local features ${\mathbf{f}_{ij}}_j$ from every image $i$
2. Store some function of these features $g(\mathbf{f}_{ij})$ and build some data structure $d(g)$ based on $g()$
3. Given the probe image, extract its local features and compute $g(\mathbf{f}_{qj})$
4. Search through the data structure $d(g)$ for "nearest neighbors"
5. Every nearest neighbor votes for the database image $i$ it comes from, accumulating to its score $s_i$
6. Sort the database images based on their score $s_i$
7. Post-process the sorted list of images to enforce some geometric consistency and obtain a final list of sorted images $s'_i$. The geometric consistency check is done using a RANSAC algorithm to fit an affine transformation between the query image $q$ and the database image $i$.
```

methods: using spherical LSH hash functions [22] and using Hierarchical K-Means [19] for matching local features.

2. We provide theoretical estimates of the storage requirement and computational cost of these methods as a function of the database size. We also explore how these methods are amenable to parallelization.
3. We empirically evaluate these properties on four real world datasets with different statistics.
4. Based on the benchmark, we suggest promising directions for future research on efficient image matching in large databases.

2. Methods Overview

In this work we consider the two leading approaches for image matching: **Full Representation** (**FR**) and **Bag-of-Words** representations (**BoW**). They are based on extracting local features from images e.g. SIFT features [16]. They can all be seen as representing the same approach with varying degrees of approximation to improve speed and/or storage requirements. The basic idea is described in Alg. 1.

2.1. Full Representation (FR)

This follows the same general approach outlined in Alg. 1. The specifics are:

1. The function $g(\mathbf{f}_{ij}) = \mathbf{f}_{ij}$ is the identity function i.e. the full features are stored
2. The data structure $d(\{\mathbf{f}_{ij}\}_{ij})$ tries to facilitate faster nearest neighbor look up at the expense of some additional storage. Four general algorithms are considered here:

   (a) **Exhaustive**: just use brute force search to get the nearest neighbor for each query image feature $\mathbf{f}_{qj}$.

   (b) **Kd-Tree**: a number of randomized Kd-trees [4, 16] are built for the database features $\{\mathbf{f}_{ij}\}_{ij}$ to allow for logarithmic retrieval time

   (c) **LSH**: a number of locality sensitive hash functions [3, 15] are extracted from the database features $\{\mathbf{f}_{ij}\}_{ij}$, and are arranged in a set of tables. Each table has a set of hash functions, which are then concatenated to get the index of the bucket within the table where the feature should go. All features with the same hash value go to the same bucket. Three different hash functions are considered:

      i. **L2**: this approximates the Euclidean distance [3], where the hash function is $h(x) = \left\lfloor \frac{\langle x,r \rangle + b}{w} \right\rfloor$ where $\langle ., . \rangle$ is the dot product, r is a random unit vector, b is a random offset, and w is the bin width. It projects the feature onto a random direction and then returns the bin number where the projection lies.

      ii. **Spherical-Simplex** & **Orthoplex**: this approximates distances on the hyper sphere [22], where the hash function is $h(x) = \text{argmin}_i \langle x, y_i \rangle$ where y_i are the vertices of a random simplex/orthoplex inscribed in the unit hyper sphere. The hash value is the index of the nearest vertex of the simplex.

   (d) **Hierarchical K-Means**: where a hierarchical decomposition [19] is built from the database features $\{\mathbf{f}_{ij}\}_{ij}$. At each level of the tree, K-means algorithm is performed to obtain a clustering of the data in the current node, and the process is repeated recursively until the maximum depth allowed is reached [18].

2.2. Bag-of-Words Representation (BoW)

The differences from Alg. 1 are:

1. The function $g(\mathbf{f}_{ij})$ represents a clustering of the input features. For pre-processing, a "dictionary" is built from the database images by clustering features into representative "visual words". Then, each image is represented by a histogram of occurrences of these visual words $\{\mathbf{h}_i\}_i$, and the original local feature vectors are thrown away. This is inspired from text search applications [23].
2. We consider two data structures that try to perform faster search through the database histograms:

   (a) **Inverted File**: where an inverted file [5, 23] structure is built from the database images.

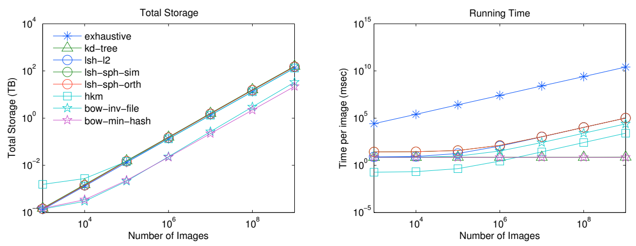
   (b) **Min-Hash**: a number of locality sensitive hash functions [7, 8, 6, 9] are extracted from the database histograms $\{\mathbf{h}_i\}_i$, and are arranged in a set of tables. The histograms are binarized (counts are ignored), and each image is represented as a "set" of visual words $\{\mathbf{b}_i\}_i$. The hash function is defined as $h(\mathbf{b}) = \min \pi(\mathbf{b})$ where π is a random permutation of the numbers $\{1, \ldots, W\}$ where W is the number of words in the dictionary.

3. Theoretical Scaling Properties

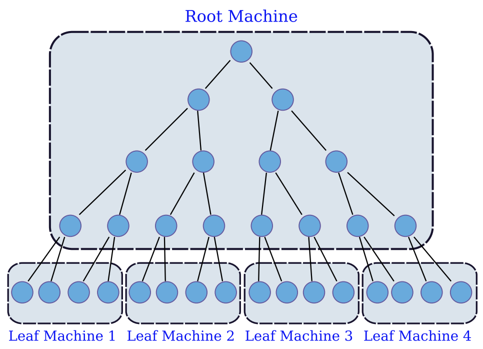
We provide theoretical estimates of how the storage and computational cost of these methods scale with the number of images in the database. We present the definitions of the parameters in Table 1 and summary of the results in Tables 2a-2b, with details in [1]. We note that these calculations are based on minimum theoretical storage and average case scenarios. We also note that we compute distance between vectors using the dot product, which is equivalent to euclidean distance since we assume feature vectors are normalized. We do not consider any compression technique that might decrease storage (e.g. run-length encoding). Moreover, actual run times of these algorithms will differ from the average case presented here, for example for inverted file method or spherical LSH methods, see Fig. 4.

The parallelization considered here is the simplest: for every method, we determine the minimum number of machines, M, that can fit the storage required in their main memory, assuming machines with 50GB of memory. Then we split the images evenly across these machines and each will take a copy of the probe image and search its own share of images. Finally, all the machines will communicate their ranked list of images (of length L) and produce a final list of candidate matches that is further geometrically checked.

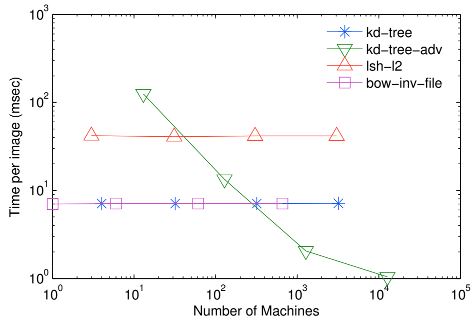
More sophisticated parallelization techniques are possible, that can take advantage of the properties of the method at hand. For example, in the case of Kd-trees, one such advanced approach is to store the top part of the Kd-tree on one machine, and divide the bottom subtrees across other machines, see Fig. 1b. For 10^{12} features, we have 40 levels in the Kd-tree, and so we can store up to 30 levels in one *root* machine, and split the bottom 10 levels evenly across the *leaf* machines. Given a probe image, we first query the root machine and get the list of branches (and hence subtrees) that we need to search with the backtracking. Then the appropriate leaf machines will process these query features and update their ranked list of images. This approach has the advantage of significant speed up and better utilization of the machines at hand, since not all the machines will be working on the same query feature at the same time, rather they will be processing different features from the probe image concurrently, see Fig. 1c. However, it also has some drawbacks: 1) the geometric verification step is more complicated as the node machines will not, in general, have all the features of any particular image; 2) the root machine will become a bottleneck for processing input images, but that can be alleviated by adding more replicas of the root machine to process more features concurrently; and 3) building the Kd-tree is tricky, since the building process needs access to all the features to build a well balanced tree.



(a) Theoretical storage vs. size *(left)*, and run time vs. size *(right)*, assuming a single machine with infinite memory.



(b) Advanced parallelization scheme for Kd-trees.The *root* machine stores the top of the tree, while *leaf* machines store the leaves of the tree.



(c) Time per image vs min. no. of machines required M for different dataset sizes. Each point represents a dataset size from 10^6 to 10^9 images, going left to right. *kd-tree-adv* is the advanced parallelization scheme, see Sec. 3.

Figure 1: Theoretical Scaling Properties. Refer to Tables 1-2

Fig. 1a shows how storage requirements and run-time scale with the number of images in the database, assuming one machine with infinite storage and 10 GFLOPS processor. Fig. 1b shows a schematic of the advanced parallelization scheme for Kd-Trees, while Fig. 1c shows the time per image vs the minimum number of machines M for different dataset sizes from 10^6 to 10^9. We note the following:

- BoW methods take one order of magnitude less storage

Parameter	Description	Typical Value	Parameter	Description	Typical Value
I	no. of images	10^9	H_{l2}	# hash fun LSH-L2	50
s	bytes/feature dim	1	B_{lsh}	# buckets	10^6
d	feature dimension	128	H_{sph}	# funcs for LSH-Spherical	5
F	#features/image	1,000	D	depth of HKM tree	7
M	# machines	varies	k	branching factor of HKM	10
C	main memory/machine	50GB	W	# words for BoW	10^6
T_{kdt}	# kd-trees	4	T_{mh}	# tables for Min-Hash	50
L	length of ranked lists	100	H_{mh}	# hash funs for Min-Hash	1
B_{kdt}	# backtracking	250	B_{mh}	# buckets in Min-Hash	10^6
T_{lsh}	# lsh tables	4			

Table 1: Parameter definitions and typical values, see Sec. 3.

Method	Storage (bytes)	Ex. (TB)	Comp. (FLOP/im)	Ex. (GFLOP/im)
Exhaustive	$(sd+4)IF$	132	$F^2I(2d+1)$	256×10^6
Kd-Tree	$IF(sd+4+2T_{kdt}+T_{kdt}\frac{\log_2 IF}{8})$	160	$B_{kdt}F(2d+1+\log_2 FI)$	0.074
LSH-L2	$IF(sd+4+T_{lsh}\frac{\log_2 IF}{8})$	152	$FT_{lsh}(H_{l2}(2d+2)+\frac{FI}{B_{lsh}}(2d+1))$	1028
LSH-Sim	$IF(sd+4+T_{lsh}\frac{\log_2 IF}{8})$	152	$FT_{lsh}(H_{sph}(2d^2+3d)+\frac{FI}{B_{lsh}}(2d+1))$	1028
LSH-Orth	$IF(sd+4+T_{lsh}\frac{\log_2 IF}{8})$	152	$FT_{lsh}(H_{sph}(2d^2+3d)+\frac{FI}{B_{lsh}}(2d+1))$	1028
HKM	$IF(sd+4)+\frac{k^D-1}{k-1}ksd$	132	$FD(2d+k)+\frac{F^2I}{k^D}\times(2d+1))$	25.7
Inverted File	$Wsd+FI(5+\frac{\log_2 I}{8})$	9	$FB_{kdt}(2d+1+\log_2 W)+F(2+I)$	1
Min-Hash	$Wsd+FI(4+\frac{\log_2 W}{8})+T_{mh}\frac{\log_2 I}{8}$	7	$FB_{kdt}(2d+1+\log_2 W)+4FT_{mh}H_{mh}+\frac{T_{mh}I}{MB_{mh}}$	0.07

(a) Theoretical storage requirement *(Storage)*, computational cost on a single machine with infinite memory *(Comp.)*

Method	Parallel (FLOP/im)	Ex. (GFLOP/im)
Exhaustive	$\frac{F^2I}{M}(2d+1)+L(M-1)$	128×10^3
Kd-Tree	$FB(2d+1+\log_2\frac{FI}{M})+L(M-1)$	0.071
Kd-Tree-adv	$FB\log_2\frac{C}{4T_{kdt}}+\frac{FB}{M}(2dB_{kdt}+B_{kdt})+F\log_2\frac{4FIT}{C}+L(\min(FB,M)-1)$	0.012
LSH-L2	$FT_{lsh}(H_{l2}(2d+2)+\frac{FI}{MB_{lsh}}(2d+1))+L(M-1)$	85
LSH-Sim	$FT_{lsh}(H_{sph}(2d^2+3d)+\frac{FI}{MB_{lsh}}(2d+1))+L(M-1)$	85
LSH-Orth	$FT_{lsh}(H_{sph}(2d^2+3d)+\frac{FI}{MB_{lsh}}(2d+1))+L(M-1)$	85
HKM	$FD(2d+k)+\frac{F^2I}{Mk^D}\times(2d+1)+L(M-1)$	0.021
Inverted File	$FB_{kdt}(2d+1+\log_2 W)+F(2+\frac{IF}{MW})$	0.075
Min-Hash	$FB_{kdt}(2d+1+\log_2 W)+4FT_{mh}H_{mh}+\frac{T_{mh}I}{B_{mh}}$	0.07

(b) Theoretical parallel computational cost with minimum required number of machines M. Kd-Tree-adv is the advanced parallelization scheme, see Fig. 1b.

Table 2: Theoretical Scaling Properties. Refer to Sec. 3, Table 1, and Fig. 1.

than FR methods, due to the fact that we don't need to store the feature vectors

- Run-time for Kd-trees and Min-hash grows very slowly with the database size.
- Inverted file and LSH methods have asymptotically similar run-time. The theoretical run time increases linearly with the number of images. However, in practice, inverted files have much smaller run time than the average case depicted above.
- The advanced parallelization method provides significant speed ups starting at 10^8 images. It might seem paradoxical that increasing the dataset size decreases the run time, however it makes sense when we notice that adding more machines allows us to interleave processing of more features concurrently, and thus allows faster processing. This however comes at a cost of more storage, see Fig. 1c.
- We also note that many of the FR methods have parameters that affect the trade off between run time and accuracy, e.g. number of hash functions or bin size

for LSH. These parameters have to be tuned for every database size under consideration, and we should not use the same settings when enlarging the database. This poses a problem for these methods, which have to be continuously updated, as opposed to Kd-trees which adapt to the current database size and need very little tuning.

4. Evaluation Details

4.1. Datasets

Our benchmarks are based on distractor and probe sets [4]:

1. **Distractors**: constitute the bulk of the database to be searched. In the actual setting, this would include all the objects of interest e.g. book covers, CD covers, ... etc.
2. **Probe**: A set of labeled images used for benchmarking purposes. This has two types of images per object: **(a) Model Image**: represents the ground truth image to be retrieved for that object, one per object; **(b) Probe Images**: 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 [2].
- **D2: Flickr-Buildings** A set of ~1M images of buildings collected from flickr.com
- **D3: Image-net** A set of ~400K images of "objects" collected from image-net.org, specifically images under synsets: instrument, furniture, and tools.
- **D4: Flickr-Geo** A set of ~1M geo-tagged images collected from flickr.com

Probe Sets

- **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 [19]. This was also used in [2].
- **P2: Pasadena Buildings** A set of 6×125=750 images of buildings around Pasadena, CA from [2]. 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: ALOI** A set of 9×80=720 3D objects images from the ALOI collection [12] with different illuminations and view points. We use the first 80 objects, with the frontal view of each object as the model image, and four orientations and four illuminations as the probe images.
- **P4: INRIA Holidays** a set of 957 images, which forms a subset of images from [13], 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]at http://vision.caltech.edu/malaa/research/image-search-bench

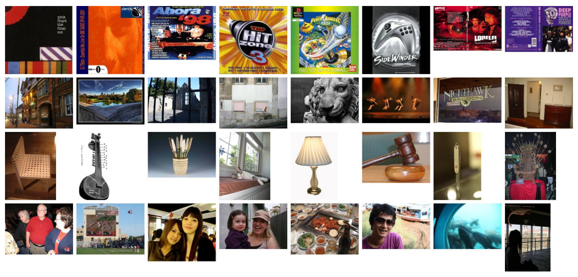


Figure 2: Example distractor images. Each row depicts a different set: D1, D2, D3, and D4, respectively, see Sec. 4.1.

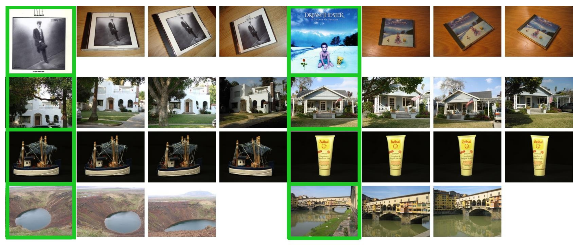


Figure 3: Example probe images. Each row depicts a different set: P1, P2, P3, and P4, respectively, see Sec. 4.1. Each row shows two model images (in *green* border) and 2 or 3 of its probe images (on the *right*).

Fig. 2 shows some examples of images from the distractor sets. Fig. 3 shows some examples of images from the probe sets. Table 3 summarizes the properties of the probe sets.

4.2. Setup

We used four different testing scenarios, where in each we use a specific distractor/probe set pair. Table 3 displays a list of scenarios used. Evaluation was done by increasing the size of the dataset from 100, 1k, 10k, 50k, 100k, and 400k. For each such size, we include all the model images to the specified number of distractor images e.g. for 1k images, we have 1,000 images from the distractor set in addition to all the model images in the probe set.

Performance is measured as the percentage of probe images correctly matched to their ground truth model image

Probe Sets	total	#model	#probe
P1	485	97	388
P2	750	125	525
P3	720	80	640
P4	957	233	724

Evaluation Scenarios: Scenario	Distractor	Probe
1	D1	D1
2	D2	P2
3	D3	P3
4	D4	P4

Table 3: Probe Sets Properties and Evaluation Scenarios. See Sec. 4.1 & 4.2.

i.e. whether the correct model image is the first ranked image returned. In addition, we measure performance before and after the geometric consistency check, which is done using RANSAC algorithm to fit an affine transformation between the probe and the model images. Ranking before the geometric check is based on the number of nearest features in the image, while ranking after is based on the number of inliers of the affine transform.

We want to emphasize the difference between the setup used here and the setup used in other "image retrieval"-like papers [13, 11, 20]. In our setup, we have only ONE correct ground truth image to be retrieved and several probe images, while in the other setting there are a number of images that are considered correct retrievals. Our setting is motivated by the application under consideration. The typical application is searching for a book cover from a database of millions of book covers. The database contains ONE image per book, while there are many probe images that strive to retrieve the same book cover.

We use SIFT [16] feature descriptors with hessian affine [17] 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. We implemented all the algorithms using Matlab and Mex/C++ scripts [5].

5. Results

5.1. Parameter Tuning

All of the methods we compare have different parameters that affect their run time, storage, and recognition performance. We performed parameter tuning in two steps: first quick tuning to get a set of promising parameters using a subset of of probe images from scenario 1, and then we ran experiments using this set of parameters for the four scenarios with sizes from 100 up to 10K images. Based on these results, we chose the settings that made most sense in terms of their recognition performance and run time. More details are in the technical report [1]. Table 4 summarizes the parameters chosen for the full benchmark. For the inverted file we built the dictionaries from a random 100K images of each distractor set using the Approximate K-Means method (AKM) [20] with 1 million visual words.

[5]at http://vision.caltech.edu/malaa/software/research/image-search

	Parameters
Kd-Trees	T= 1 tree, B =100 backtracking steps
LSH-L2	T =4 tables, H =25 hash functions, bin size w =0.25
LSH-Sim	T =4 tables, H =5 hash functions
LSH-Orth	T =4 tables, H =5 hash functions
HKM	tree depth $D = 5$, branching factor $k = 10$
Inverted File	tf-idf weighting, l_2 normalization, cos distance
	raw histograms, $l_1$normalization, l_1 distance
	binary histograms, l_2 normalization, cos distance
Min-Hash	$T = 100$ tables, $H = 1$ hash function

Table 4: The chosen methods parameters for the full experiments, see Sec. 5.1.

5.2. Results and Conclusion

Fig. 4 shows the recognition performance for different dataset sizes, before and after the geometric consistency check. We note the following:

- Full representation (FR) methods based on (approximately) matching the local features provide superior recognition performance to bag-of-words (BoW) methods.
- BoW methods provide nearly constant search time compared to linear increase in case of most FR methods, with the exception of Kd-trees (before exhausting all the main memory, see the big jump in Fig. 4 for 50K images) which provide logarithmic increase in search time.
- FR methods take an order of magnitude more memory than BoW methods e.g. we can easily fit up to 400K images for BoW while for some scenarios we can only fit up to 50K images for some FR methods.
- It is important to use diverse datasets for measuring and quantifying performance of different methods. While the trend is similar for all the four scenarios, we notice that scenarios 1 and 3 are the easiest, while scenarios 2 and 4 provide much more difficulty. We believe that is the case because images in scenarios 2 and 4 have much more clutter than the other two scenarios, and that makes the recognition task much harder.
- We notice that BoW techniques can provide acceptable accuracy in some situations e.g. scenarios 1 & 3. This suggests that for some applications we can use BoW methods, which have the advantage of near constant search time and less storage requirements.
- FR methods pose a trade off between recognition rate and search time. In particular, for scenarios 2 & 4, we

notice that LSH methods are generally better than Kd-tree in terms of recognition rate but inferior in terms of search time which rises sharply with database size. This suggests that we can trade off extra search time for better recognition rate.

- Spherical LSH methods provide comparable recognition rate to LSH-L2 method, but they provide better search time.
- Using the combination of l_1 normalization with l_1 distance in the inverted file method provides better performance than the standard way of using tf-idf weighting with l_2 normalization and dot product. Using binary histograms also outperforms the standard tf-idf scheme, as reported in [14].
- The geometric verification step is in general not necessary, and gives performance comparable to that beforehand, except in the case of Min-Hash method. This is especially true for hard scenarios, e.g. 2 & 4, where there is a lot of *noise* features i.e. features belonging to clutter around the object, like trees around the house. These features get matched to noise features in the probe image and thus make the geometric step harder.
- Kd-trees provide the best overall trade off between recognition performance and computational cost. We notice that its run time grows very slowly with the number of images while giving excellent performance. Moreover, with more sophisticated parallelization schemes (see Sec. 3 & Fig. 1b), we can have significant speedup when running on multiple machines, and the rate of processed images actually increases with more data, in contrast to BoW. The only drawback, which is shared with all FR methods, is the larger storage requirements.
- We believe the two main promising directions for further research in large scale image indexing are:

  1. Devising ways to reduce the run time and storage requirements for FR methods. This includes developing better features that take less storage, finding ways to store less features in the database, and compressing information of features using projection methods (e.g. PCA, etc.).
  2. Devising ways to improve the recognition performance of BoW methods. This includes finding better ways to generate the visual words dictionaries and encoding geometric information in the BoW representation.

Acknowledgements

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

References

[1] Mohamed Aly, Mario Munich, and Pietro Perona. Indexing in large scale image collections: Scaling properties, parameter tuning, and benchmark. Technical report, Caltech, USA, 2010.

[2] 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.

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

[4] 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.

[5] R. Baeza-Yates and B. Ribeiro-Neto. *Modern Information Retrieval*. ACM Press, 1999.

[6] Andrei Broder, Moses Charikar, and Michael Mizenmacher. Min-wise independent permutations. *Journal of Computer and System Sciences*, 60:630–659, 2000.

[7] Andrei Z. Broder, Steven C. Glassman, Mark S. Manasse, and Geoffrey Zweig. Syntactic clustering of the web. *Computer Networks and ISDN Systems*, 29:8–13, 1997.

[8] A.Z. Broder. On the resemblance and containment of documents. In *Proc. Compression and Complexity of Sequences 1997*, pages 21–29, 1997.

[9] O. Chum, J. Philbin, M. Isard, and A. Zisserman. Scalable near identical image and shot detection. In *CIVR*, pages 549–556, 2007.

[10] O. Chum, J. Philbin, J. Sivic, M. Isard, and A. Zisserman. Total recall: Automatic query expansion with a generative feature model for object retrieval. In *ICCV*, 2007.

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

[12] J. M. Geusebroek, G. J. Burghouts, and A. W. M. Smeulders. The amsterdam library of object images. *IJCV*, 61:103–112, 2005.

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

[14] H. Jégou, M. Douze, and C. Schmid. Packing bag-of-features. In *ICCV*, sep 2009.

[15] Y. Ke, R. Sukthankar, and L. Huston. An efficient parts-based near-duplicate and sub-image retrieval system. In *MULTIMEDIA*, pages 869–876, 2004.

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

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

[18] Marius Muja and David G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *VISAPP*, 2009.

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

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

[21] J. Sivic, B. C. Russell, A.. A. Efros, A. Zisserman, and W.T. Freeman. Discovering object categories in image collections. In *ICCV*, 2005.

[22] 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.

[23] J. Zobel and A. Moffat. Inverted files for text search engines. *ACM Comput. Surv.*, (2), 2006.

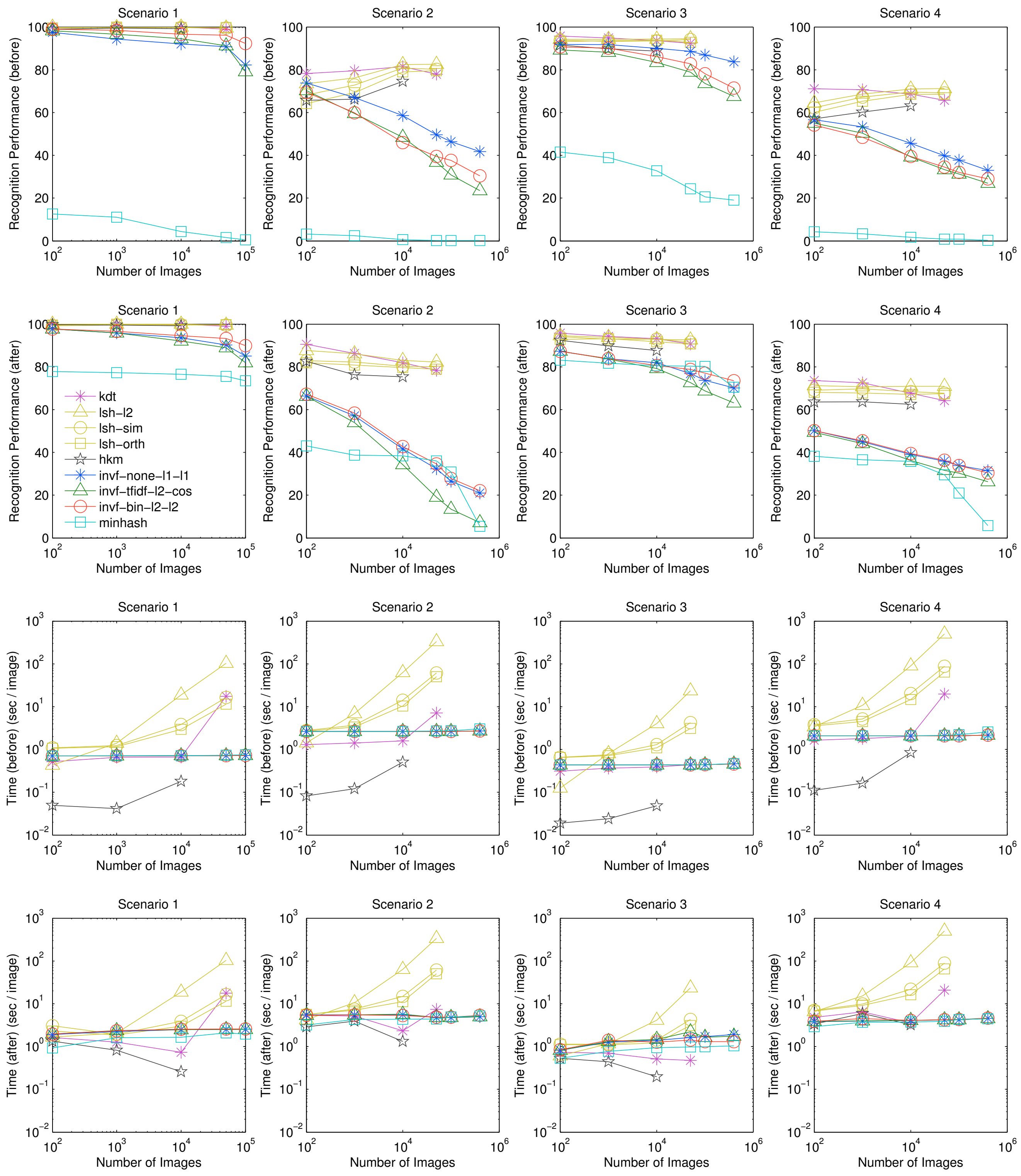


Figure 4: **Recognition Performance and Time Vs Dataset Size**. First two rows show recognition performance before and after the geometric step. Lower two rows show total processing time per image before and after the geometric step. Every column represents a different experimental scenario, see Tables 3 and 4.