High-Dimensional Indexing by Sparse Approximation

Pedro Borges, André Mourão and João Magalhães
Dept. Computer Science
Universidade Nova Lisboa
Campus de Caparica
{p.borges,a.mourao}@campus.fct.unl.pt, jm.magalhaes@fct.unl.pt

ABSTRACT

In this paper we propose a high-dimensional indexing technique, based on sparse approximation techniques to speed up the search and retrieval of similar images given a query image feature vector. Feature vectors are stored on an inverted indexed based on a sparsifying dictionary for l_0 regression, optimized to reduce the data dimensionality. It concentrates the energy of the original vector on a few coefficients of a higher dimensional representation. The index explores the coefficient locality of the sparse representations, to guide the search through the inverted index. Evaluation on three large-scale datasets showed that our method compares favorably to the state-of-the-art. On a 1 million dataset of SIFT vectors, our method achieved 60.8% precision at 50 by inspecting only 5% of the full dataset, and by using only 1/4 of the time a linear search takes.

Categories and Subject Descriptors

H.3.3 [**Information Search and Retrieval**]

Keywords

High-dimensional indexing; image indexing; Approximate nearest neighbor search; Sparse coding; l_0 penalty

1. INTRODUCTION

New approaches to the problem of nearest neighbor search shows that it is possible to achieve good scalability as the number of dimensions increase by trading precision over speed. That is, by giving only a probabilistic guarantee that the top k most similar documents are retrieved.

In a metric space S, multimedia information is represented as feature vectors $x = (x_1, \ldots, x_n)$, where $x_i \in \mathbb{R}$ and n is the dimensionality of the space. Let $S \subseteq Fv$ be a set of document feature vectors, were Fv is the set of all possible feature vectors, and $d : Fv \times Fv \rightarrow \mathbb{R}_0^+$ a metric. Formally, the k-Nearest Neighbors Query is defined as follows: given a pair (q, k) with $q \in Fv$ and $k \in \mathbb{N}^+$, retrieve from S the k most similar vectors to q. Formally, we wish to retrieve a set of vectors $Y = \{y_1, y_2, \ldots, y_k\}$ such that, $\forall y_i \in Y$, $\forall x_i \in S - Y$, $d(q, y_i) \leq d(q, x_i)$.

A linear scan on the metric space S, is not feasible when the cardinality of S grows rapidly. Recent approaches have relaxed the search process by either (i) hashing the vectors into a Hamming embedding [2,5,6,19] where the metric function as a very low computational cost, or (ii) leveraging the probabilistic density of data to reduce the search space complexity [11,12,14,23,25]. As we shall see in the related work section, there are several recent data structures that can answer approximate nearest-neighbor searches more efficiently by avoiding scanning the entire set of feature vectors.

The hypothesis we address in this paper is that retrieval efficiency can be improved by designing a dictionary for mapping the feature vectors onto sparse feature vectors embedded in a higher dimensional space. This space preserves locality structure on the non-zero coefficients of the sparse representation. That is, similar feature vectors in the original feature space should yield similar feature vectors in the target vector space. The left side of Figure 1 depicts an ideal case where two similar feature vectors, represented by similar blue colors are transformed into sparse feature vectors matching many of their non-zero coefficients. Conversely, dissimilar feature vectors in the original space would be mapped to vectors with different non-zero coefficients, right side of Figure 1.

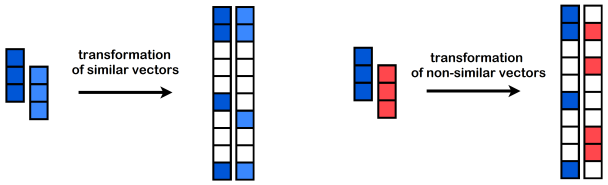

Figure 1: Insertion in the index. The colored positions on each vector are the non-null coefficients.

Our contributions are two-fold: we describe a sparsifying dictionary design process based on l_0 regression and an adapted K-SVD algorithm that explores feature vector locality and an indexing and retrieval system, optimized for the high-dimensional sparse space.

This paper is organized as follows: first we review approximate nearest neighbor techniques; in section 3 we formalize our indexing by sparse approximation method; section 4 describes the sparse regression algorithms; section 5 details the sparsifying dictionary and section 6 details the evaluation.

ICMR '15 June 23 - 26, 2015, Shanghai, China

DOI: http://dx.doi.org/10.1145/2671188.2749371.

2. RELATED WORK

Indexes that use space partitioning to improve the performance of exact similarity search were refined over the years so they could index bigger multimedia repositories and generate better space partitions. However, this improvements become more and more marginal as the dimensionality of the indexed features increases, [24]. The effectiveness of the index can degrade so much that a linear scan over all features might be faster than using the index. This fundamental difficulty lead researchers to investigate new kinds of similarity queries that drop the exactness requirement and do not use the dissimilarity function to partition the search space in order to improve the search performance.

Algorithms for approximate similarity search tackle the complexity of searching high-dimensional spaces by trading precision over performance. Some approximate methods rely on the trees, combined with a dimensionality reduction approach (e.g. PCA). A second family of techniques which have received much attention recently, are hash-based techniques. These techniques transform the original feature space into a Hamming embedding, where each sample is represented as a binary code. Similar feature vectors are converted to the same binary code (bucket), causing a collision. Thus, nearby buckets are within a short Hamming distance.

The Locality Sensitive Hashing (LSH), [2], is an indexing algorithm that can answer both randomized c-approximate R-near neighbor queries and randomized R-near neighbor queries. LSH hashes similar features to the same bins with more probability than dissimilar features. The algorithm is applicable when the features are real or boolean vectors and an appropriate family of hash functions exists for the employed similarity metric. These approaches, rely in hash functions that are either random [5, 8, 19], or deterministic because they follow a given structure, e.g., a grid [24] or a distribution [6].

More recent approaches to multimedia indexing [10, 12, 14, 23, 25] try to leverage the latent relationships in the data to guide the construction of representations and data-structures to accelerate similarity queries. Using machine learning techniques to learn these relationships it is possible to beat the search speed and accuracy of the LSH algorithm. Spherical hashing [9] is a data depended variant of LSH that partitions the high-dimensional query space in hyperspheres instead of hyperplanes. Torralba et al. [23] propose a method to learn small binary feature vectors from real valued vector features using Boosting. Hinton et al. [10] generate compact hash representations using an autoencoder based on Restricted Boltzmann machines. Spectral Hashing [25] binary codes are calculated by thresholding a subset of eigenvectors of the Laplacian of the similarity graph.

Approaches based in binary codes and Hamming distances make it possible to search millions of multimedia documents in less than a second. However, it still requires a linear search to be performed over the entire dataset. Other approaches, explore a divide-and-conquer strategy to tackle the large-scale scalability problem. Li et al. [14] proposed learning a data structure that balances between precision and computational cost. Based in boosted search forests, it can naturally be integrated in a search index. Jegou et al. [12, 13] proposed a series of methods that rely on an inverted index to organize the data into manageable parts. The inverted index [12] is constructed with a balanced K-means that quantizes the feature space into codewords with similar priors [20]. The sparse hashing method proposed by Zhu et al. [26], departs from the dense codes and designs large but sparse binary codes. MSIDX [21] is a novel technique that explores the divide-and-conquer strategy. It is based on the theory that descriptors' dimensions with higher cardinality have higher discriminative power. The algorithm multi-sorts the indexed descriptors according to their value cardinalities. Indexed descriptors are re-ordered with descriptors higher values on the most discriminative dimensions being most important. The querying algorithm follows the same principle: the closest match to the query is found using the sort algorithm.

3. EFFICIENT SIMILARITY MATCHING WITH SPARSE REPRESENTATIONS

The careful transformation of data into a high-dimensional, but sparse approximations can achieve a high degree of compression. This new space preserves locality structure on the non-zero coefficients of the sparse representation. That is, similar feature vectors in the original feature space should yield similar feature vectors in the the target vector space. Conversely, dissimilar feature vectors in the original space would be mapped to a vectors with different non-zero coefficients. With such locality preserving transformation, finding similar feature vectors in higher dimensional space would be as simple as checking which other sparse representations had non-zero coefficients in the same coefficients as the query, and rank those candidates with the original metric. A setup like this has three major advantages:

- traversing the data structure does not depend on the metric function, which allow us to avoid the previously discussed limitations of the tree indexes and other indexes based on space partitioning;
- sparse representations can be encoded efficiently in terms of space which should make it possible to build an index for a dataset with low storage overhead;
- it is more efficient than searching the entire search space, since the representations are sparse only a small number of coefficients would have to be searched for similar vectors. Hence, using this mechanism enables search space pruning.

Mathematically, these goals can be expressed as an optimization problem:

$$\begin{gathered} \text{argmin}_{A,X} \|B - AX\|_F^2 \\ \text{subject to} \\ \|X_i\|_0 \leq k \\ \text{for } i = 0, 1, \cdots, n \end{gathered} \tag{1}$$

where A is the dictionary, B is a matrix where each column is a measurement vector, X is the matrix where each column X_i is the sparse signal recovered for the measurement vector B_i, k is the sparsity threshold for each signal and n is the number of columns of X. Solving this problem for A and X simultaneously is computational hard due to the presence of the l_0 norm and the non-convexity of the problem in both A and X. In this process there are two main components: the sparse approximation procedure to solve Eq. (1), section

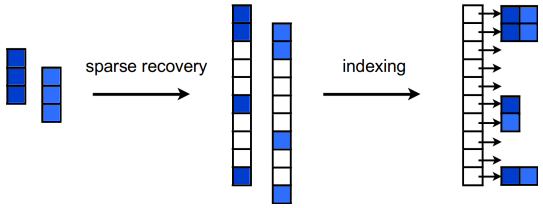

Figure 2: Collision of similar descriptors.

4 and the sparsifying dictionary A that preserves locality information in high-dimensional spaces, section 5.

3.1 Inverted indexes

The construction of the inverted index for sparse representations is illustrated by Figure 2. It shows the indexing of two descriptors using their sparse representations. Algorithm 1 describes the indexing construction process. Given a set of descriptors B, we can recover their sparse solutions X with a dictionary A (line 3) (section 5). This sparse representation can be interpreted as determining the atoms of the dictionary whose coefficient are non-zero. The goal is to build an index to efficiently retrieve feature vectors whose sparse representation contains the same atoms as the query feature vectors, line 6. The final step is to sort each bucket, by placing the descriptors with higher coefficients at the beginning, line 8. This final sort is necessary so that, at query time, the most representative descriptors of each bucket are examined first.

If the sparse recovery of the descriptors preserves similarity in the sparse domain, the inverted index can be used to retrieve similar descriptors for a given query, by inspecting the lists corresponding to its non-zero coefficients in the sparse domain. The ideal situation is depicted in Figure 2 were two similar descriptors are indexed in many common positions.

Algorithm 1 Indexing algorithm

Input: a sparsifying dictionary A, a set of vectors to index B.
Output: Inverted index I with $cols(A)$ buckets

1: $I \leftarrow []$
2: **for** b_i in B **do**
3: $\quad x_i \leftarrow sparseApproximation(A, b_i)$
4: $\quad$ **for** j in $1, ..., dim(x_i)$ **do**
5: $\quad\quad$ **if** $x_{ij} \neq 0$ **then**
6: $\quad\quad\quad I_j \leftarrow I_j \cup \{x_{ij}, b_i\}$
7: **for** I_j in I **do**
8: $\quad I_j \leftarrow sortByCoef(I_j)$ ▷ sort vectors by $coef_j$
9: **return** I

The per document memory overhead of the indexing structure is $O(2L)$, where L is the number of non-zero coefficients (set to be at most 10), and 2 represents bucket metadata: document id and bucket coefficient value.

3.2 Retrieval

The task of the retrieval algorithm is to search for descriptors that have a collision in the buckets where the non-zero coefficients of the sparse representation of the query are located. The search procedure gives priority to the buckets where the coefficients are larger. Intuitively, this tries to exploit the fact that sparse regression greedy algorithms for sparse regression, select the most correlated atom of the dictionary first.

Algorithm 2 describes the retrieval process, given the previously computed index I. When the user submits a query q, its sparse representation is recovered, line 2. Then, the coefficients $x_{0,...,i}$ (that correspond to the buckets) of the sparse representation are extracted and sorted in descending order according to their absolute values, line 3. Buckets with higher coefficient values are visited first. The search algorithm visits each bucket adding each feature vector found in a temporary candidates ($cand$) list, line 10. This process continues until the procedure visits every entry of the index or the limit for the visited feature vectors is reached, lines 6 and 9. After the search stops, the descriptors in the temporary list are sorted by their distance to the query, line 16 and the k closest descriptors returned, line 17.

Algorithm 2 Retrieval procedure

Input: a sparsifying dictionary A, an inverted index I, a set of indexed vectors B, a query vector q, the number of neighbors to retrieve k and a percentage of the index to examine w.
Output: a list of k (nearest) neighbors R.

1: $limit \leftarrow w \times len(b)$
2: $X \leftarrow sparseApproximation(A, q)$
3: $O \leftarrow sortCoefs(X)$ ▷ get sorted coefficients
4: $cand \leftarrow \emptyset$
5: $i \leftarrow 0$
6: **while** $len(cand) < limit \cap i < len(O)$ **do**
7: $\quad bucket \leftarrow O_i$
8: $\quad j \leftarrow 0$
9: $\quad$ **while** $len(cand) < limit \cap j < len(I_{bucket})$ **do**
10: $\quad\quad cand \leftarrow cand \cup I_{bucket,j}$
11: $\quad\quad j \leftarrow j + 1$
12: $\quad i \leftarrow i + 1$
13: $R \leftarrow []$
14: **for** c in $cands$ **do**
15: $\quad R \leftarrow R \cup \{\|q, c\|_2, c\}$
16: $R \leftarrow sortByNorm(R)$ ▷ Perform final l_2 re-ranking
17: **return** $R_{1,...,k}$

4. SPARSE APPROXIMATIONS

The proposed inverted indexed is based on the l_0 or l_1 regression for obtaining the high-dimensional sparse representations. This algorithm is fundamental in reducing the data dimensionality, by concentrating the energy of the original vector on a few coefficients of a higher dimensional space. The sparse reconstruction algorithm should answer two requirements: (1) be sufficiently fast to be run in query time, (2) compute highly sparse representations while preserving maximum locality. In this section we compare four candidate algorithms for our sparse reconstruction step.

4.1 l_0 approximations

The sparse approximation can be viewed as the recovery of a k-sparse signal from its measurement vector given a dictionary. A possible way is to solve the constrained l_0

minimization problem,

$$\min_x \|Ax - b\|_2$$
$$\text{subject to} \quad (2)$$
$$\|x\|_0 \leq k,$$

where $A \in \mathbb{R}^{m \times n}$ is the dictionary, $b \in \mathbb{R}^m$ the measurement vector, and k is the sparsity factor that we wish to obtain. By solving the relaxed problem we allow the recovery of sparser solutions even though these solutions violate the equality constrains by a small margin. Next, we describe two greedy algorithms that selects the atoms at each iteration that have the best fit according to a rule to integrate the solution.

Orthogonal Matching Pursuit. The basic idea behind the OMP algorithm, [18], is to, at each iteration, choose the atom of the dictionary that best correlates with the residual, a vector holding the recovery error using the currently selected atoms. Initially, the residual vector, r, is set to the measurement vector and the solution vector x is set to a zero vector, since no atom of the dictionary was selected. In the iterative procedure, each iteration computes the correlation between each atom of the dictionary and the residual. With the correlations computed the atom with the maximum correlation is considered to integrate the solution.

Thresholding. The thresholding algorithm can be seen as a simplification of the OMP algorithm described before. Instead of taking multiple steps and making a greedy decision, thresholding uses just one step where it picks the most correlated atoms with the measurements to take part in the solution.

4.2 l_1 approximations

The sparse recovery problem has an inherently combinatorial nature due to the usage of the l_0 norm. An alternative research direction to the greedy algorithms presented in the previous subsection has been the reformulation of the sparse recovery into a problem that exhibits properties that allow the usage of efficient optimization algorithms and, under certain conditions, can still recover the original sparse solution.

One of the possible modifications to the optimization problem is to replace the l_0 norm with another function f. Results by Tao and Candés [4] and Donoho and Elad [7] demonstrate conditions under which the minimization of the l_1 norm leads to the same solution as the l_0 norm. Thus, in this section we describe algorithms to solve the sparse recovery problem using the l_1 norm:

$$\min_{x \in \mathbb{R}^n} \|x\|_1$$
$$\text{subject to} \quad (3)$$
$$Ax = b$$

Fast Iterative Shrinkage Thresholding Algorithm. FISTA, [3], is an algorithm that solves problem 3 by taking an unconstrained approximation of the original problem. FISTA has its roots in the iterative soft thresholding algorithm (ISTA). ISTA iterates by solving the unconstrained problem at a given point using a thresholding function and then taking a gradient step from the local optimal point. FISTA improves the ISTA algorithm by using an improved method to determine the next point at which the minimization step is evaluated that guarantees faster convergence.

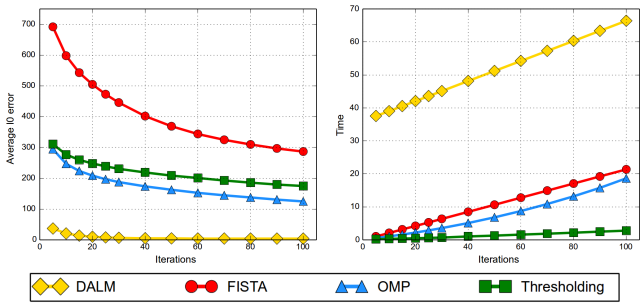

Figure 3: Average recovery error (a) and time (b) over 100 iteration on a 504 dimensional space; (c) DALM recovery after 100 iterations using 30 dimensions zoomed.

Augmented Lagrangian Methods. Augmented Lagrangian Methods extend the unconstrained problem with quadratic penalization, by adding a Lagrangian term to the objective function. The addition of this term improves the conditioning of the problem and guarantees the convergence under more general assumptions, [16]. A lower bound for the solution of the problem can be obtained by solving the dual problem of the augmented objective function which is more efficient than solving the original problem.

4.3 Experimental comparison of reconstruction time, sparsity and error

Figure 3 illustrates the recovery time and error for the implemented algorithms. DALM exhibits an interesting behavior with an almost constant and low recovery error. A more careful look into the coefficients computed by DALM showed that the solutions have a very large number of near-zero coefficients, which defeats our sparse index goal. In terms of recovery time the OMP and the Thresholding methods were the best. We chose OMP, as it delivers the lowest recovery error with a sparse solution and its the second fastest method.

5. SPARSIFYING DICTIONARY

Our inverted index structure is designed to use a dictionary created to yield sparse representations that approximate the original feature vectors. In this section we describe the dictionary construction algorithm, a modified K-SVD [1] algorithm, to explicitly take into account the proximity between the descriptors used in its training. The idea is to add a penalty term to the dictionary learning problem that promotes similar descriptors to have similar sparse representations. Ideally, this penalty would take into account all pair-wise distances between the descriptors of the training dataset. In practice that approach is infeasible for learning dictionaries from large datasets of examples because the size of the problem would scale at least quadratically with the number of training examples. To simplify the optimization problem we group the training descriptors in clusters according to their similarity and penalize the differences in the sparse representations within the cluster at once.

Instead of computing a new dictionary all at once, K-SVD iteratively optimizes each atom and their corresponding coefficients in the sparse representations. For the dictionary update step the alternate optimization scheme is achieved by

splitting the contributions of each atom and the respective coefficients for the value of the objective function, f:

$$\begin{aligned} f &= \|AX - B\|_F^2 \\ f &= \|A_i X_{\text{row}(i)} + \sum_{j \neq i} A_j X_{\text{row}(j)} - B\|_F^2 \\ f &= \|A_i c_i + \sum_{j \neq i} A_j c_j - B\|_F^2, \end{aligned} \quad (4)$$

here c_i is the row of the matrix X holding the coefficients for the atom A_i of the dictionary. The last expression in 4 immediately suggests the alternative optimization used by K-SVD. For each atom of the dictionary, the algorithm fixes the other atoms and the respective coefficients and iteratively solves problems of the form:

$$\underset{A_i, c_i}{\text{argmin}} \|A_i c_i + E_i\|_F^2 \quad (5)$$

where $E_i = (\sum_{j \neq i} A_j c_j - B)$ and A_i and c_i are the new atom and its new coefficients respectively. Since A_i is a column vector and c_i is a row vector their product results in a matrix with rank-1, so the problem 5 is equivalent to finding the rank-1 matrix approximation of E_i and then factoring this matrix into the vector A_i and c_i. The SVD factorization that gives name to the algorithm provides a convenient way to determine the approximation matrix and a suitable factorization at once. Given the matrix $E_i \in \mathbb{R}^{m \times n}$ its SVD factorization is given by:

$$E_i = G \Sigma V^T.$$

Assuming that the singular vectors in the matrix Σ are sorted in decreasing order through its diagonal, the best rank-1 approximation matrix for the matrix E_i is:

$$approx(E_i) = G \Sigma^1 V^T$$

where Σ^1 is the matrix Σ where every entry but $\Sigma_{0,0}$, the entry corresponding to the largest singular value, are null. The factorization into the new A_i and c_i can be done by taking $A_i = G_0$ and $c_i = \Sigma^1_{0,0} V_{\text{row}(0)}$, this makes the product equivalent to the factorization 5 since the null entries of Σ_1 cancel every other column of G and row of V^T.

The optimization procedure described previously reduces the recovery error of the system but has the undesirable effect of producing coefficient vectors c_i less sparse than the originals since the sparsity is not enforced anywhere during the decomposition. To overcome this problem, only the subset of signals with non zero coefficients in the dictionary atom under consideration take part in the optimization. Assuming I is the set including the index of all signals with non-zero coefficients in A_i atom the update step is:

$$\underset{A_i, (c_i)_I}{\text{argmin}} \|A_i (c_i)_I + (E_i)_I\|_F^2.$$

6. EXPERIMENTS

In this section we present the initial experiments to determine the number of iterations of the dictionary training algorithm and other parameters. A thorough set of experiments were then run over 3 large-scale datasets where we compared our method to other methods (which include state-of-the-art solutions). All experiments were run on a machine with an i7-3930K processor and 64GB of RAM. All descriptors, indexes and other structures were placed in the main memory. The code for the Sparse Approximation (SA) indexer is available online [1].

6.1 Datasets

We tested 3 datasets for the large-scale indexing experiments: the Tiny Images dataset, [22] and the Billion vectors dataset (2 feature types) [12]. These datasets were used to train the indexing dictionaries, to build the indexes and provide query examples. These datasets were also previously used in the literature in the same settings [25].

The Tiny Images dataset [22], contains over 80 million images and the corresponding GIST descriptors [17] with 384 dimensions. This makes the dataset particularly adequate to simulate the conditions of a large-scale high-dimensional search. All descriptors are stored in a single contiguous binary file of floating point numbers.

The Billion vectors dataset [12], contains over 1 billion feature vectors. It was designed to evaluate the quality of nearest neighbors search algorithm with different descriptors and database sizes. A subset of 1 million images [12] and their GIST and SIFT descriptors (960 and 128 dimensions, respectively), are divided into training, indexing and testing subsets.

6.2 Experiment Design

6.2.1 Data Setup

The datasets were split into a training, validation and test subsets (see Table 6.2.1). For BV1 and BV2, we used the splits provided by the authors[2] for comparison. As the provided splits don't provide validation vectors, we selected the validation set randomly from the training set. For the TI, we used 3 disjoint data setups extracted from the original dataset. The results are the average of all data setup results.

6.2.2 Methods

We tested our Sparse Approximation (*SA*) method against a set of widely known indexing algorithms:

- **Trees:** [15] provides a k-means tree (*kMeansTree*), a randomized kd-tree (*kdTree*) and an exhaustive linear search (*Linear*) implementations[3]; We varied the number of trees in the kd-tree algorithm and the branching factor in the k-means algorithm;
- **LSH:** we tested the implementation[4] described in [9], with multiple hash code sizes;
- **Spherical Hashing:** The tested implementation is described in [9], and based on exhaustive search in the index of training hashes with the Hamming (*SH*) and Spherical Hamming (*SH_shd*) distance. We tested multiple hash code sizes;
- **MSIDX:** The tested implementation[5] is from the original authors [21]. We varied the percentage of database to examine {0.5%, 1%, 2.5%, 5%, 10%, 15%, 20%, 25%, 30%, 40%, 50%};

[1] `https://novasearch.org/software/`
[2] `http://corpus-texmex.irisa.fr/`
[3] `http://www.cs.ubc.ca/research/flann/`
[4] `http://sglab.kaist.ac.kr/Spherical_Hashing/`
[5] `http://vcl.iti.gr/msidx/`

Table 1: Data types and setup. The last four columns contain the number of vectors used for training, validation index, validation queries, test index and test queries. Sets marked with an asterisk represent vectors extracted from the provided training set.

Name	Dataset	Features	Dims	Train vectors	Val. index	Val. queries	Test index	Test queries
TI	Tiny Image	GIST	384	100,000	100,000	1,000	1,000,000	1,000
BV1	Billion vectors	GIST	960	100,000*	100,000*	1,000*	1,000,000	1,000
BV2	Billion vectors	SIFT	128	49,000*	50,000*	1,000*	1,000,000	1,000

6.2.3 *Protocol*

We wanted to assess the efficiency and quality of the retrieval results of the proposed method, and compare it to other state-of-the-art methods. We opted for following metrics, widely used in the multimedia retrieval literature [9,21, 23]: **average precision @50 and @1000** vs. **query time** and vs. **inspected percentage of the dataset**.

For each query, we measure the retrieval efficiency of each method and compare the retrieval precision to the linear search output. Hence, the top k nearest neighbors correspond to the relevant items.

6.3 Parameters learning

The validation set was used to assess the combination of relevant parameters that yield the best results. The two main parameters that have a direct influence in the retrieval performance are the:

- **Bucket size:** as the number of buckets of the index increases the number of descriptor indexed per entry will decrease. In practice, this should lead to more discrimination in how close the other descriptors must be to be indexed in the same dimensions;
- **Number of indexed positions:** using more iterations of the OMP algorithm will result in the recovery of denser representations which means each vector is indexed in more buckets. In this case, the probability of collision between two descriptors increases. The goal is to find the best collision parameter for the index.

Figure 4 (a) shows the average percentage of the 50 nearest neighbors retrieved from indexes with 1024, 2048, 4096 and 8192 buckets, constrained to inspect just a fraction of the index descriptors. Due to space constrains, we will only display the results for the TI dataset with multiple bucket counts. The same behavior was observed on the other dataset. As a baseline, the blue dashed line represents a random limited linear search on an uniformly distributed dataset (examining 10% of the dataset returns 10% of relevant query results). This figure indicates a rapid increase in the number of neighbors found on average as the percentage of allowed descriptor inspections increases from 0 to 10% of the database.

Figure 4 (b) details the results in terms of query time versus retrieval precision. When examining this figure, we observe that a higher bucket size decreased the retrieval performance and increased query time. We reckon that this is due to the greedy nature of the indexing and querying algorithms.

Following this analysis, we set the bucket size to 1024, as we were able to achieve higher precision while reducing the query time when examining more than 3% of the dataset.

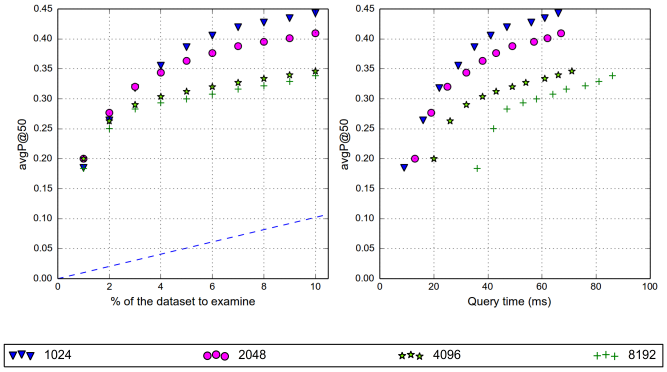

Figure 4: Average precision of the 50 nearest neighbors retrieved with limited number of index inspections for multiple bucket count. The dashed line represents a random linear search on a uniformly distributed dataset.

6.4 Results and discussion

In this section, we discuss the comparative performance (percentage of relevant documents retrieved vs. query time) of our method versus other state-of-the-art methods. The hashing based methods (LSH, SH and SH-shd) were tested with {64, 128, 256, 512} bit codes. The number of trees in the k-d-tree was set to {4, 8, 16, 32, 64, 128}. The kMeansTree branching factor was set to {16, 32, 64}. Both our method and MSIDX were set to inspect a fixed percentage of the index {1% ,2% ,3%, 4%, ... ,10% ,20% ,30% ,40% ... ,100%}. For each one of these methods/settings we measured retrieval time and precision.

6.4.1 *Robustness analysis*

To examine the global behavior in terms of robustness and stability of the tested methods, we compared the retrieval results for a long rank of documents ($k = 1000$) in all three datasets. Observing the first row of Figure 5, two major trends stand-out. The first trend occurs in the initial part of the curve where precision grows very rapidly as query time increases. In the second trend precision grows linearly towards the maximum possible precision. These two trends are more noticeable in the MSIDX method and the Sparse Approximation method. The different slopes indicate that both methods are optimized for efficiently retrieving the top nearest neighbors. As a consequence, after a given point, the curve slope is not as steep showing a loss in performance, as a side-effect of the top-results optimization. The same conclusions can be drawn from all datasets. Other hash-based methods do not exhibit these two trends.

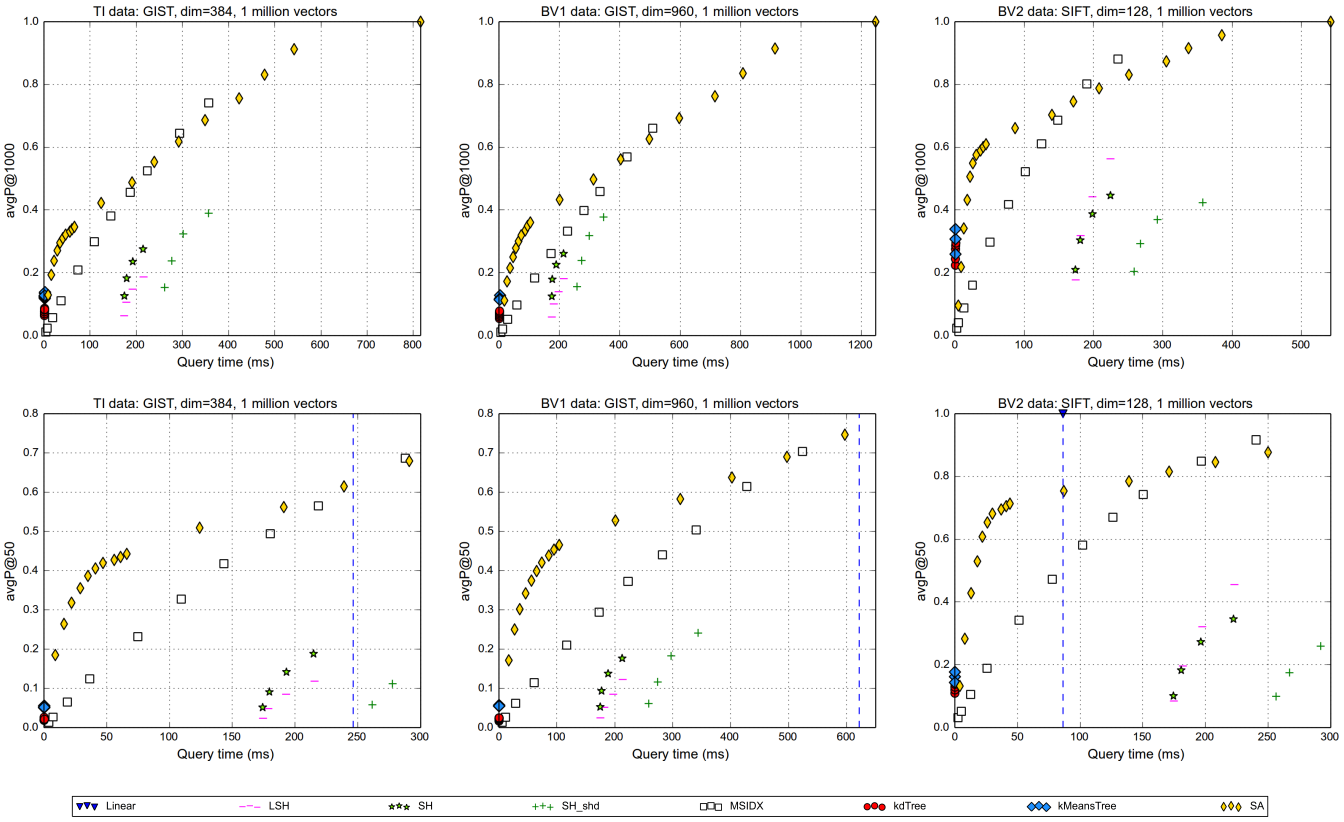

Figure 5: Average % of 1000 and 50 nearest neighbors retrieved vs. time per retrieved neighbor Comparative results for all datasets and algorithms.

6.4.2 *Top-precision analysis*

To assess the proposed algorithm on the top-retrieved results, we evaluated its performance on the top $k = 50$ results. Figure 5 shows the comparative results for $k = 50$. The vertical dashed line marks the linear search time (i.e., 100% precision).

All methods show a behavior similar to the 1000 nearest neighbor experiment. Both MSIDX and SA methods show the two trends but in a more pronounced way: there are two clearly different trends. In this experiment, the first trend is steeper which evidences how effective is the proposed sparse optimization framework at achieving a high precision on the top results.

On the second trend, which the SA optimization did not consider during the training phase, our method examines a larger percentage of the dataset (therefore examining more buckets), causing the performance to degrade, becoming sub-optimal.

In general, the SA results compare favorably to the other tested methods. The MSIDX algorithm scales linearly with the percentage of the dataset examined on all examined datasets. When examining a larger set of the dataset, MSIDX is able to be faster than our approach, but with a query time similar or slower than a linear search. LSH and variants (SH) achieve slower results: the algorithms' performance improve with the size of the binary codes, but they are still slower than the proposed method. The tested LSH-based algorithms also suffer a linear search in the Hamming space.

In the bottom row of Figure 5, we can carefully analyse the performance of the faster algorithms (faster than 100 ms). When looking closer at these results, we observe the full advantage of the proposed method: in the Tiny Images dataset, we achieved 38% precision by inspecting 5% of data and in less than 15% of the linear search time (5% of inspected data corresponds to the fifth point in the SA curve). When using SIFT features (BV2 dataset) we obtained 60.8% precision at 50, after inspecting only 5% of data in 25% of time a linear search takes.

In our experiments, tree-based algorithms were very fast, but at the cost of very low precisions. These methods were not able to retrieve more than a few of the nearest neighbors. The hashing based approaches were slower and can even be slower than linear searches.

7. CONCLUSION

In this paper we introduced a high-dimensional indexing structure and a matching retrieval algorithm by sparse approximation. We also refined the optimization of the dictionary used to index and retrieve descriptors to explicitly take into account the locality of their sparse representations. The described algorithms were analyzed in terms of retrieval quality, computational time and compared against existing state-of-the-art methods and other well-known indexing trees for a comprehensive evaluation.

The proposed indexing data structure compared favorably with the other state-of-the-art methods, achieving a preci-

sion of 60.8% by inspecting only 5% of the full dataset, and by using only 1/4 of the time a linear search takes in the same conditions.

We are currently researching how to improve the learned dictionary for indexing. More specifically we are researching the use of label information to adjust the dictionaries to non-linearities in the feature space.

Acknowledgements. This work has been partially funded by the projects PTDC/EIA-EIA/111518/2009 and UTA-Est/MAI/0010/2009 funded by the Portuguese National Foundation for Science and Technology (FCT).

8. REFERENCES

[1] M. Aharon, M. Elad, and A. Bruckstein. K-SVD: An Algorithm for Designing Overcomplete Dictionaries for Sparse Representation. *IEEE Trans. on Sig. Proc.*, 54(11):4311–4322, Nov. 2006.

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

[3] A. Beck and M. Teboulle. A fast iterative shrinkage-thresholding algorithm for linear inverse problems. *SIAM J. Img. Sci.*, 2(1):183–202, Mar. 2009.

[4] E. J. Candes and T. Tao. Decoding by Linear Programming. *IEEE Transactions on Information Theory*, 51(12):4203–4215, Dec. 2005.

[5] O. Chum, J. Philbin, and A. Zisserman. Near duplicate image detection: min-hash and tf-idf weighting. In *BMVC'08*, 2008.

[6] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *SCG'04*, pages 253–262, New York, NY, USA, 2004. ACM.

[7] D. L. Donoho and M. Elad. Optimally sparse representation in general (non-orthogonal) dictionaries via l1 minimization. In *Proc. Natl Acad. Sci.*, 2002.

[8] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *VLDB'99*, VLDB '99, pages 518–529, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[9] J.-P. Heo, Y. Lee, J. He, S.-F. Chang, and S.-E. Yoon. Spherical hashing. In *CVPR'12*, pages 2957–2964, June 2012.

[10] G. E. Hinton and R. R. Salakhutdinov. Reducing the dimensionality of data with neural networks. *Science*, 313(5786):504–507, 2006.

[11] H. Jégou, M. Douze, and C. Schmid. Hamming embedding and weak geometric consistency for large scale image search. In *ECCV'08*, ECCV '08, pages 304–317, Berlin, Heidelberg, 2008.

[12] H. Jégou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 33(1):117–128, 2011.

[13] H. Jégou, T. Furon, and J.-J. Fuchs. Anti-sparse coding for approximate nearest neighbor search. In *ICASSP'12*, pages 2029–2032, 2012.

[14] Z. Li, H. Ning, L. Cao, T. Zhan, Y. Gong, and T. S. Huang. Learning to search efficiently in high dimensions. In *NIPS'11*, 2011.

[15] M. Muja and D. G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *International Conference on Computer Vision Theory and Application VISSAPP'09)*, pages 331–340. INSTICC Press, 2009.

[16] J. Nocedal and S. J. Wright. *Numerical Optimization*. Springer, Aug. 2000.

[17] A. Oliva and A. Torralba. Modeling the shape of the scene: A holistic representation of the spatial envelope. *Int. J. Comput. Vision*, 42:145–175, May 2001.

[18] Y. Pati, R. Rezaiifar, and P. Krishnaprasad. Orthogonal Matching Pursuit : recursive function approximation with application to wavelet decomposition. In *Asilomar Conf. on Signals, Systems and Computer*, 1993.

[19] M. Raginsky and S. Lazebnik. Locality-sensitive binary codes from shift-invariant kernels. In *NIPS*, pages 1509–1517, 2009.

[20] R. Tavenard, H. Jégou, and L. Amsaleg. Balancing clusters to reduce response time variability in large scale image search. In *CBMI'11*, Madrid, Spain, June 2011.

[21] E. Tiakas, D. Rafailidis, A. Dimou, and P. Daras. Msidx: Multi-sort indexing for efficient content-based image search and retrieval. *Multimedia, IEEE Transactions on*, 15(6):1415–1430, Oct 2013.

[22] A. Torralba, R. Fergus, and W. Freeman. 80 million tiny images: A large data set for nonparametric object and scene recognition. *IEEE Trans on PAMI'08*, 30(11):1958–1970, Nov 2008.

[23] A. Torralba, R. Fergus, and Y. Weiss. Small codes and large image databases for recognition. In *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*, pages 1–8, June 2008.

[24] R. Weber, H.-J. Schek, and S. Blott. A quantitative analysis and performance study for similarity-search methods in high-dimensional spaces. In *VLDB'98*, pages 194–205, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[25] Y. Weiss, A. Torralba, and R. Fergus. Spectral hashing. *NIPS*, 9(1):6, 2008.

[26] X. Zhu, Z. Huang, H. Cheng, J. Cui, and H. T. Shen. Sparse hashing for fast multimedia search. *ACM Trans. Inf. Syst.*, 31(2):9:1–9:24, May 2013.