Selective K-means Tree Search

Tuan Anh Nguyen, Yusuke Matsui, Toshihiko Yamasaki, Kiyoharu Aizawa
The University of Tokyo, Japan
{t_nguyen, matsui, yamasaki, aizawa}@hal.t.u-tokyo.ac.jp

ABSTRACT

In object recognition and image retrieval, an inverted indexing method is used to solve the approximate nearest neighbor search problem. In these tasks, inverted indexing provides a nonexhaustive solution to large-scale search. However, a problem of previous inverted indexing methods is that a large-scale inverted index is required to achieve a high search recall rate. In this study, we address the problem of reducing the time required to build an inverted index without degrading the search accuracy and speed. Thus, we propose a selective k-means tree search method that combines the power of both hierarchical k-means tree and selective nonexhaustive search. Experiments based on approximate nearest neighbor search using a large dataset comprising one billion SIFT features showed that the hierarchical inverted file based on the selective k-means tree method could be built six times faster, while obtaining almost the same recall and search speed as the state-of-the-art inverted indexing methods.

Categories and Subject Descriptors

H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval—*Clustering, Retrieval Models, Search Process, Selection Process*

General Terms

Theory, Algorithms, Experimentation

Keywords

Approximate Nearest Neighbor Search, Hierarchical K-means Tree, Inverted Indexing, Product Quantization

1. INTRODUCTION

In most computer vision and image retrieval applications, approximate nearest neighbor (ANN) search is an important problem [7]. An inverted file is the most popular data structure employed during nonexhaustive search to improve the search speed in large-scale databases. In particular, Jégou et al. [4] proposed an inverted file with asymmetric distance computation (IVFADC) index by combining inverted files and product quantization. IVFADC outperforms previous methods [2, 4, 7] during ANN search in terms of the trade-off between memory usage and accuracy. Recently, Babenko et al. [1] improved IVFADC by proposing the inverted multi-index (Multi-D-ADC). Both IVFADC and Multi-D-ADC are built using clustering processes. IVFADC performs clustering in the search space to create K codewords, whereas Multi-D-ADC performs clustering in multiple subspaces before combining multiple independent codewords based on the Cartesian product to yield K^m codewords, where m is the number of subspaces. Therefore, Multi-D-ADC has a dense space-partitioning form that accelerates the ANN search, while still achieving the same recall as IVFADC. The problem addressed in the present study is how to create a dense space partitioning for fast search to allow Multi-D-ADC to perform a large-scale indexing process. In previous studies [4, 1], the time required to build an inverted file was shown to be linear up to K.

The main contributions of this study are as follows. (1) To reduce the time required to build an inverted file, we propose the selective k-means tree method based on the hierarchical k-means tree [8]. (2) We obtain an inverted file structure (SKT-ADC), which combines the strength of the selective k-means tree and product quantization for use in ANN search. (3) The selective approach limits the search to highly likely candidates, thereby accelerating the search process in the selective k-means tree method.

Our experiments showed that Multi-D-ADC required a dense space partitioning (codebook size $K = 2^{14}$) to achieve fast search, whereas SKT-ADC achieved the same recall and runtime with a 16-times smaller codebook size, i.e., $K = 2^{10}$, where the inverted file-building process was six times faster (4 hours compared with 24 hours by Multi-D-ADC using SIFT1B data on the same machine). Our search method is fast (2 milliseconds for a single query). It is also simpler compared with the state-of-the-art search methods used with Multi-D-ADC.

2. PREVIOUS WORK

Inverted indexing: In [4], Jégou et al. proposed IVFADC, which comprises two components: (i) a *dictionary* (Figure 1) created by k-means [6]; and (ii) a *posting lists* containing the compressed versions of database vectors. The database vectors are compressed by product quantization to facilitate better memory usage. To create a dictionary, k-means is applied to a learning dataset $\mathcal{L}$, which creates K clusters in a d-dimensional space $\mathbb{R}^d$ with K codewords (clusters' centroids) $\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_K$. Each vector $\mathbf{y}$ in the set of database of vectors $\mathcal{Y}$ is assigned to a cluster with its nearest codeword in $\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_K$.

Data compression with product quantization: To avoid accessing the disk to obtain the vector data $\mathbf{y}$, a resid-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.
MM'15, October 26–30, 2015, Brisbane, Australia.
© 2015 ACM. ISBN 978-1-4503-3459-4/15/10 ...$15.00.
DOI: http://dx.doi.org/10.1145/2733373.2806353.

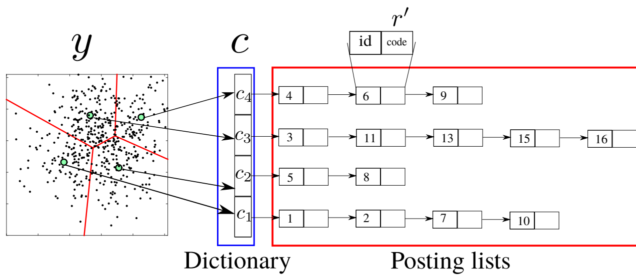


Figure 1: An inverted file with four clusters: k-means clustering is used to divide the search space into four parts, where each part represents a cluster (codeword vector c) in the inverted file. The database vectors y in each cluster are compressed and stored in the inverted file. Each compressed version (code $\mathbf{r}'$) is associated with an identifier (id) to retrieve the original vector data y.

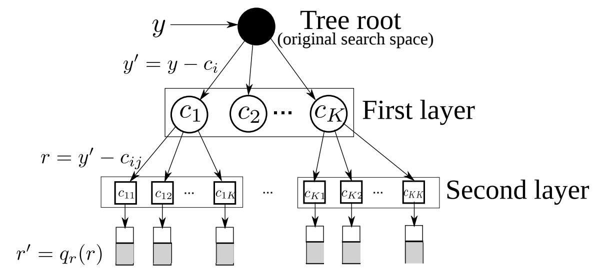


Figure 2: Our inverted file, SKT-ADC: the first layer contains K codewords that correspond to K subsearch spaces, where each codeword forms a new subsearch space by residual vector computation. Each subsearch space is then divided again into K clusters to form K child nodes for the codeword in the first layer. K^2 codewords are created to form the second layer. Finally, the database vectors are compressed by product quantization.

ual vector $\mathbf{r}$ between $\mathbf{y}$ and the nearest codeword in $\mathbf{c} \in \{\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_K\}$ to $\mathbf{y}$ is computed and quantized by a product quantizer $q_r(.)$: $\mathbf{r} = \mathbf{y} - \mathbf{c} \xrightarrow{q_r(.)} \mathbf{r}' = q_r(\mathbf{r})$. A database vector $\mathbf{y}$ is approximated by the sum $\mathbf{c} + \mathbf{r}'$.

Multi-D-ADC: Multi-D-ADC [1] is an improved version of IVFADC. To build a dictionary, product quantization is used instead of single k-means clustering [6]. The total training time required by a product quantizer to build a codebook of K^2 codewords is almost the same as that needed to build a codebook of K codewords in IVFADC, i.e., $O(MKd)$, where M is the size of the learning dataset and d is the dimensionality.

Search methods: In both IVFADC and Multi-D-ADC, the search methods comprise two processes. (i) *Dictionary scanning* : a k nearest neighbor search is performed using the set of codewords to retrieve a list of candidate clusters. For Multi-D-ADC, a multisequence algorithm [1] was proposed to reduce the scanning time in two parallel sorted sequences. (ii) *Posting lists scanning*: the ANN is found by computing the distances between the query $\mathbf{q}$ and the compressed vectors $\mathbf{c} + \mathbf{r}'$. In the search method employed for Multi-D-ADC [1], all of the dot products and norms are precomputed to facilitate a faster search process. In this case, the approximated distance is computed in $O(m)$ time, where m is the number of divisions during product quantization.

3. SELECTIVE K-MEANS TREE SEARCH

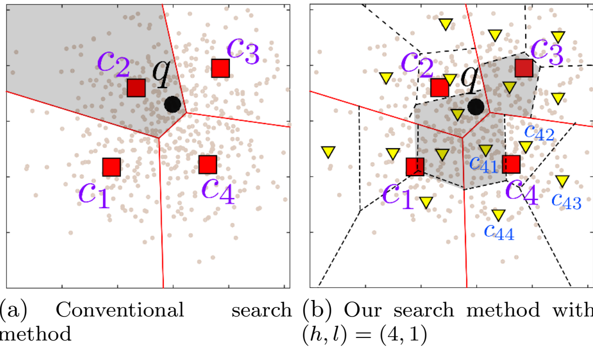
In this section, we introduce our inverted file (SKT-ADC) based on a two-layer selective k-means tree (Figure 2). A general version with more than two layers is straightforward.

Building a dictionary: First, a learning dataset $\mathcal{L} = \{\mathbf{y} \in \mathbb{R}^d\}$ is divided into K clusters with K codewords $\mathbf{c}_1, \mathbf{c}_2, \ldots, \mathbf{c}_K$, where d is the dimensionality. Then, each cluster is again divided into K clusters with K codewords $\mathbf{c}_{i1}, \mathbf{c}_{i2}, \ldots, \mathbf{c}_{iK}$, $1 \leq i \leq K$. A product quantizer $q_r(.)$ is learned from the residual vectors of $\mathbf{y} - \mathbf{c}_i - \mathbf{c}_{ij}$, where $\mathbf{y}$ is a training vector, $\mathbf{c}_i$ is the nearest codeword of $\mathbf{y}$ in the first layer, and $\mathbf{c}_{ij}$ is the nearest codeword of $\mathbf{y} - \mathbf{c}_i$ in the second layer. Let M be the size of the learning dataset $\mathcal{L}$. The complexity of k-means [6] is linear with respect to the size of the training set, dimensionality, and the number of clusters, so we need $O(MKd)$ operations to construct the first layer. Let $M_1, M_2, \ldots, M_K$ be the number of learning vectors in each of the clusters in the first layer and the sum of all M_is equals M. Therefore, we need $O(\sum_{i=1}^{K} M_i K d) = O(MKd)$ operations to construct the second layer. The time complexity of the dictionary-building process is $O\left(MKd\right)$.

Data indexing: After the codewords $\mathbf{c}_i, \mathbf{c}_{ij}$, $1 \leq i \leq K$, $1 \leq j \leq K$ have been computed, a database vector $\mathbf{y}$ is compressed according to the following steps (Figure 2). (i) *Step 1:* The nearest neighbor c_i of $\mathbf{y}$ in $\{\mathbf{c}_i\}_{i=1}^{K}$ is retrieved and the residual vector $\mathbf{y}' = \mathbf{y} - \mathbf{c}_i$ is computed. (ii) *Step 2:* The nearest neighbor $\mathbf{c}_{ij}$ of $\mathbf{y}'$ in $\{\mathbf{c}_{ij}\}_{j=1}^{K}$ is retrieved and the residual vector $\mathbf{r} = \mathbf{y}' - \mathbf{c}_{ij}$ is computed. (iii) *Step 3:* The residual vector $\mathbf{r}$ is compressed into an m-byte code by product quantization. The time complexity of the data-indexing process is $O\left(NKd\right)$ for a dataset of N vectors. Note that the time complexities required to build the dictionary and to perform data indexing for Multi-D-ADC are also $O(MKd)$ and $O(NKd)$, respectively.

Search method: We illustrate the search process in Figure 3. Let $\mathbf{q}$ be the query vector. Our search process comprises the following three steps. (i) *Step 1:* Compute all of the distances between $\mathbf{q}$ and the codewords in the first layer to find h nearest neighbor (h-NN) of $\mathbf{q}$ in the first layer. (ii) *Step 2:* In each cluster $\mathbf{c}_i$ corresponding to h-NN for $\mathbf{q}$ in the first layer, l-NN for $\mathbf{q} - \mathbf{c}_i$ in the set of codewords $\{\mathbf{c}_{ij}\}_{j=1}^{K}$ are computed. (iii) *Step 3:* In this step, we retrieve $h \times l$ selected clusters and from all of the candidate vectors in the $h \times l$ clusters after Step 2, a final list of candidates is obtained. We compute all of the approximated distances between $\mathbf{q}$ and all of the candidate vectors. The R smallest values and their corresponding identifiers are exported as the search results. When the length of the candidate list exceeds a predefined candidate list length T, the process in Step 2 is terminated. In our experiments, for SIFT1B when $K \leq 2^{11}$, the setting values of $h = 32 \sim 128$ and $l = 8 \sim 16$ yielded an acceptable recall rate.

The main operations in Steps 1 and 2 are distance-computing operations. In Step 1, the distances between the query and all K of the codewords in layer 1 are computed in $O(Kd)$ operations. In Step 2, $O(Kd)$ operations are required for



(a) Conventional search method (b) Our search method with $(h, l) = (4, 1)$

Figure 3: Our search method: the gray area is the search area corresponding to a query, q. (a) After assigning the query q to cluster c_2, all of the database vectors in this cluster are listed as candidates. The candidates in cluster c_3 and c_4 are missed. (b) In SKT-ADC, the tree structure allows the retrieval process to focus only on the nearby subclusters. The most likely possible candidates are scanned in all four clusters: c_1, c_2, c_3, c_4 (gray area).

each cluster retrieved after Step 1. In Steps 1 and 2, to improve the recall rate, h-NN (l-NN) are reranked by sorting the top h (l) smallest distances in ascending order. The number of reranking operations is low when $h, l \ll K \times d$ and h clusters are retrieved after Step 1; therefore, the total required for Steps 1 and 2 is $O(hKd)$ operations. In Step 3, as described in Section 2, with the precomputing processes, a single approximated distance-computing process requires $O(m)$ operations; therefore, the distance-computing tasks in Step 3 require $O(mT)$ operations in total. In summary, the search process using our inverted file requires $O(hKd + mT)$ operations for a single query.

4. EXPERIMENTS

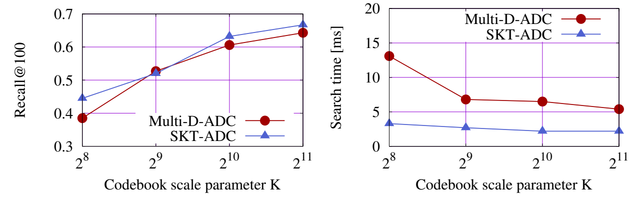
4.1 Experimental setup

We conducted experiments using several publicly available datasets: SIFT1M [4] (1 million 128-dimensional SIFT features + 10000 queries), GIST1M [4] (1 million 960-dimensional GIST features + 1000 queries), and SIFT1B [5] (1 billion 128-dimensional SIFT features + 10000 queries). The search quality was measured based on the recall@R measure, i.e., the proportion of queries with nearest neighbor ranked in the first R positions in the final candidate list. We considered that recall rates where $R \leq 100$ were sufficient for a large scale ANN search [1]. All of the experiments using SIFT1B are conducted on a computer with 6-core Intel Xeon 3.50 GHz CPUs and 512 GB RAM. The search experiments were performed in the single-thread mode. All of the algorithms were implemented in C++ with BLAS instructions. For clustering tasks, we employed the k-means algorithm in [3].

4.2 Results

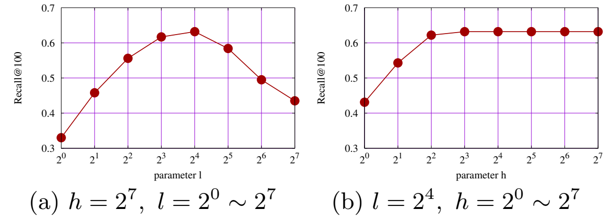
In this section, we describe the results obtained using SIFT1B. The results of the experiments using SIFT1M and GIST1M are described in the supplementary material.

Inverted file-building time: Table 1 shows the relationship between time required to build the inverted file and the search performances of IVFADC, Multi-D-ADC, and SKT-ADC with different codebook scales K. SKT-ADC with $K = 2^{10}$ achieved the same recall@100 as Multi-D-ADC with $K = 2^{14}$. The training times for both SKT-ADC and Multi-D-ADC were linear with respect to the codebook scale K, so SKT-ADC could achieve a good recall rate with only a short training time. In our experiments, to build a codebook with $K = 2^{10}$ for SKT-ADC using a 6-core computer, the training process required about 4 hours, whereas building Multi-D-ADC with $K = 2^{14}$ required about 24 hours and IVFADC with $K = 2^{13}$ required about 12 hours on the same computer.



(a) Recall@100 for SKT-ADC and Multi-D-ADC (b) Search time for $T = 100K$

Figure 4: Search performance obtained by SKT-ADC and Multi-D-ADC using SIFT1B ($K = 2^8 \sim 2^{11}$, $T = 100K$), and by SKT-ADC with $(h, l) = (32, 16)$: (a) recall@100; (b) search time.



(a) $h = 2^7,\ l = 2^0 \sim 2^7$ (b) $l = 2^4,\ h = 2^0 \sim 2^7$

Figure 5: The recall@100 rate obtained by SKT-ADC varied with (h, l) using SIFT1B with $K = 2^{10}$: a) $h = 2^7,\ l = 2^0 \sim 2^7$; (b) $l = 2^4,\ h = 2^0 \sim 2^7$.

Search performance: With the same candidate list length $T = 100K$ and codebook scale $K = 2^{10}$, SKT-ADC also achieved the same recall rate as Multi-D-ADC, but the search was three times faster. Table 1 shows that SKT-ADC required 20 GB of memory while searching. These overheads are required because SKT-ADC must precompute more norms and dot products than Multi-D-ADC to perform an efficient search [1]. In our implementations, the memory overheads incurred were linear with respect to K for Multi-D-ADC and with respect to K^2 for SKT-ADC (please refer to the supplementary material for mathematical descriptions). Thus, the storage volume required for these computations was also larger than that for Multi-D-ADC.

Figure 4 shows the performance measures for SKT-ADC and Multi-D-ADC when scanning 100 K candidate vectors. Figure 4(a) shows that with the same K, SKT-ADC achieved a better recall than Multi-D-ADC, and Figure 4(b) shows that with the same codebook scale K, SKT-ADC performed a faster search than Multi-D-ADC. Our experiments showed that SKT-ADC was two times faster than Multi-D-ADC.

Table 1: Performance of the proposed method for SIFT1B (recall for top 1–100 + time in milliseconds). A vector was compressed to an 8-byte code. The search times are shown per query. For SKT-ADC, we set the codebook scale parameter $K = 2^{10}$ and $(h, l) = (32, 16)$ for SKT-ADC, $K = 2^{10},\ 2^{14}$ for Multi-D-ADC, and $K = 2^{13}$ for IVFADC. The candidate list length T ranged from 10 K to 1 M. The inverted files were built using a 6-core computer.

Method	K	T	R@1	R@10	R@100	Time [ms]	Memory [GB]	IVF building time [h]
IVFADC	2^{13}	1M	0.095	0.338	**0.652**	20.0	**12**	12
Multi-D-ADC	2^{14}	10K	**0.136**	**0.419**	0.631	2.3	**12**	24
Multi-D-ADC	2^{10}	100K	0.104	0.317	0.606	6.5	**12**	**2**
SKT-ADC	2^{10}	100K	0.099	0.315	0.632	**2.2**	20	4

Table 2: Number of nonempty clusters with each method using SIFT1B.

K	IVFADC	Multi-D-ADC	SKT-ADC
2^8	2^8	65,386	65,535
2^9	2^9	257,248	261,991
2^{10}	2^{10}	975,912	1,044,686
2^{11}	2^{11}	3,406,222	4,141,985

How dense is SKT-ADC? With the same codebook scale parameter K, IVFADC [4, 5] could create K clusters, whereas Multi-D-ADC [1] and SKT-ADC each created K^2 clusters. Table 2 shows the number of nonempty clusters, which varied with the codebook scale parameter K. SKT-ADC created more nonempty clusters than IVFADC and Multi-D-ADC in almost the same time $O(MKd)$, where M is the size of the learning dataset and d is the dimensionality. Multi-D-ADC creates the codewords based on the Cartesian product between two independent spaces, whereas SKT-ADC creates the codewords by performing k-means clustering recursively in the same space.

Selecting the parameters (h, l). Figure 5 shows the recall rates with different values for h, l. Figure 5(a) shows that when $h = 2^7$, the recall rate reached its maximal value at $l = 2^4$. Single k-means was performed to construct K codewords in the first layer of SKT-ADC. When K was small, k-means yielded an ANN search result with many noisy vectors. In our search method, the parameter l was set to filter these noisy vectors. To scan the candidate clusters in the first layer, SKT-ADC only searched the candidates in l nearest clusters. The distance between a query and a cluster was treated as the distance between the query and the codeword (the centroid of the cluster). In our experiments, SKT-ADC with $l = 2^4$ obtained the best filtering results with the highest recall. The candidate list length T was limited, so more noisy vectors were scanned and the recall was degraded when $l > 2^4$. In addition, as shown in Figure 5(b), SKT-ADC with $h \geq 2^3$ resulted in a high recall search. After increasing h while $l = 2^4$ remained fixed, the candidate list was unchanged after retrieving all T candidates; therefore, the recall rate was stable when $h \geq 2^3$.

5. CONCLUSION

In this study, we proposed a selective k-means tree search method, which is based on the hierarchical k-means tree [8] with a selective search approach. Our proposed inverted index SKT-ADC obtained almost the same recall and search speed, and the inverted file was built six times faster compared with the best conventional inverted index, i.e., Multi-D-ADC. With the same candidate list length T and codebook scale K, our method also achieved better recall rates than Multi-D-ADC. We also performed our experimental validations using small and large datasets. Our implementation is publicly available[1].

In our future research, we will investigate the following. (1) We will try to reduce the memory overheads incurred by SKT-ADC. The number of codewords in the selective k-means tree is linear with respect to K^2, so the storage required for precomputing is also linear with respect to K^2, thereby incurring large memory overheads. (2) Both Multi-D-ADC [1] and SKT-ADC produce good candidate lists, thereby facilitating high recall search, so combining these data structures will be an objective in our future works.

Acknowledgments: We would like to thank Krishna Onkar for his helpful comments. This work was partially supported by JSPS Grant-in-Aid for challenging Exploratory Research Grant Number 26540078 and the Strategic Information and Communications R&D Promotion Programme (141203018).

6. REFERENCES

[1] A. Babenko and V. Lempitsky. The inverted multi-index. *IEEE Trans. Pattern Anal. Mach. Intell.*, PP(99):1–1, 2014.

[2] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proc. SCG*, pages 253–262. ACM, 2004.

[3] G. Hamerly. Making k-means even faster. In *SDM*, pages 130–140, 2010.

[4] H. Jégou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *IEEE Trans. Pattern Anal. Mach. Intell.*, 33(1):117–128, 2011.

[5] H. Jégou, R. Tavenard, M. Douze, and L. Amsaleg. Searching in one billion vectors: Re-rank with source coding. In *Proc. ICASSP*, pages 861–864. IEEE, 2011.

[6] S. Lloyd. Least squares quantization in pcm. *IEEE Trans. Inf. Theory*, 28(2):129–137, 1982.

[7] M. Muja and D. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *Proc. VISAPP*, pages 331–340. IEEE, 2009.

[8] D. Nistér and H. Stewénius. Scalable recognition with a vocabulary tree. In *Proc. CVPR*, volume 2, pages 2161–2168. IEEE, 2006.

[1] https://github.com/marker68/skt