This CVPR paper is the Open Access version, provided by the Computer Vision Foundation.
Except for this watermark, it is identical to the version available on IEEE Xplore.

Product Split Trees

Artem Babenko
Yandex,
National Research University
Higher School of Economics
artem.babenko@phystech.edu

Victor Lempitsky
Skolkovo Institute of Science and Technology
(Skoltech)
lempitsky@skoltech.ru

Abstract

In this work, we introduce a new kind of spatial partition trees for efficient nearest-neighbor search. Our approach first identifies a set of useful data splitting directions, and then learns a codebook that can be used to encode such directions. We use the product-quantization idea in order to make the effective codebook large, the evaluation of scalar products between the query and the encoded splitting direction very fast, and the encoding itself compact. As a result, the proposed data srtucture (Product Split tree) achieves compact clustering of data points, while keeping the traversal very efficient. In the nearest-neighbor search experiments on high-dimensional data, product split trees achieved state-of-the-art performance, demonstrating better speed-accuracy tradeoff than other spatial partition trees.

1. Introduction

Spatial partition trees are popular data structures for approximate nearest neighbor search in high-dimensional spaces. Currently partition trees provide the state-of-the-art performance for medium-scale settings when the database points can be held within the main memory but the linear scan over them is slow. Partition trees proceed by hierarchically splitting the search space into a large number of regions corresponding to tree leaves, whereas each such region contains one or few database points. At the search stage, a query is propagated down the tree, a limited number of leaves are visited in the depth-first order, and only distances to points from these leaves are evaluated. In practice, this approach is much faster that the exhaustive linear scan while providing high accuracy given constrained time budget.

State-of-the-art partition trees typically use binary space splitting by a hyperplane $\langle w, x \rangle = b$ in each internal node. Here, w is the hyperplane normal, which we refer to as *splitting direction* and b is the threshold. When searching a given query q at each internal node the sign of the expression

$$\langle w, q \rangle - b \tag{1}$$

determines the subtree that should be visited first.

The practical performance of a partition tree mostly depends on two factors. The first factor is the *quality of splitting directions* that define the space partition. Typically, the tree construction algorithms seek to use directions w with high variance of data projections $\underset{x}{\mathrm{Var}}\langle w, x \rangle$. The partition regions produced with high-variance directions are more compact, hence the probability that the depth-first search will manage to move from the query' leaf to its nearest neighbor leaf in a small amount of steps is higher. The second factor is *propagation efficiency*, which depends on the complexity of evaluation of (1) in each internal node. The existing state-of-the-art approaches propose different tradeoffs between these two factors.

Probably, the most well-known partition trees are *KD-trees*[7], which use axis-aligned hyperplanes i.e. $||w||_0 = 1$. Typically, the hyperplane orthogonal to the axis with the highest data variance is used. The propagation down a KD-tree is very efficient as evaluating (1) requires only one comparison operation in each internal node. However, the quality of splitting directions within a KD-tree is inferior as only d possible directions w are considered in each node during tree construction. Being dependent on the original data coordinate basis these directions might be suboptimal in a sense that data variance along all of them in some nodes can be small.

In contrast, another approach, called *PCA-tree*[20], produces superior space partition by splitting along the main principal direction learnt from the subset of points belonging to a particular node. Trinary projection tree (*TP-tree*)[21] considers a restricted set of splitting directions that is more reach than within KD-tree and less reach than within PCA-tree. Overall, PCA-tree and TP-tree produce separating hyperplanes more adapted to the particular dataset compared to KD-trees, which results in more compact partition regions. On the other hand, the evaluation

Tree	KD-tree	RP-tree	PCA-tree	TP-tree	PS-tree
Splitting directions quality	lowest	low	highest	high	high
Propagation cost per node	$O(1)$	$O(d)$	$O(d)$	$O(\bar{d})$	$O(M)$
Memory per node	$O(1)$	$O(d)$	$O(d)$	$O(\bar{d})$	$O(M)$
Number/cardinality of the potential split set	d	$\lvert\mathbf{R}^d\rvert$	$\lvert\mathbf{R}^d\rvert$	$\frac{3^d-1}{2}$	K^M
Data-adapted potential split set	No	No	Yes	No	Yes
Query preprocessing complexity	0	0	0	0	$O(Kd)$

Table 1. Comparison of main design features for the state-of-the-art partition trees. d denotes the dimensionality of the search database. $\bar{d}$ is the maximum number of active coordinates within the TP-tree splitting directions. K is the size of subdirection codebooks within the PS-tree and M is the number of codebooks. Overall, PS-tree provides both high quality splitting directions and runtime/memory efficiency. The only additional cost of PS-tree is query preprocessing, but in most practical circumstances this cost can be made negligible by picking small enough K, which can still provide diverse enough split set.

of (1) in these trees is more expensive and the propagation is slower. Moreover, PCA-trees and TP-trees require more additional memory to hold the splits.

In this work we introduce *Product Split Trees (PS-trees)*, a new spatial partition trees that use codebook learning in order to combine splitting directions of high quality and efficient propagation down the tree within the same approach. The main idea of PS-trees is to preliminary learn a large set of splitting directions that are promising to provide high data variance on all levels of the tree. These directions are learnt in the offline stage and then the values of dataset points projections in these directions are used to build a tree in the usual KD-tree manner. Then, at the querying stage, projections of a query in all learnt directions are evaluated and then the typical KD-tree propagation is performed based on these values.

PS-trees achieve both efficiency and good data adaptation via learning splitting directions in the Product Quantization (PQ) form [10]. PQ-based approaches based on inverted indices already provide state-of-the-art for nearest neighbor search in very large "billion-scale" datasets [10, 4, 11]. In this work we extend PQ success to a smaller "million-scale" regime, where space-partition trees perform better than inverted indices. We thus show how to incorporate PQ ideas into partition trees and provide new state-of-the-art that outperforms existing approaches.

The performance of PS-trees is evaluated on four datasets of image descriptors typical for computer vision tasks (SIFT vectors, "deep" descriptors, GIST descriptors, MNIST images). We show that for a given time budget PS-trees provide higher recall compared to the existing top-performing partition trees. Furthermore, we also show that for a given additional memory budget PS-trees also outperform the existing approaches. This encourages the usage of PS-trees in the scenarios with limited resources, for example in mobile applications.

2. Related work

In this section we describe several methods and ideas from the previous works that are essential for description of PS-trees. We also introduce notation for the following sections.

Approximate nearest neighbor search. The task of nearest neighbor search is an important component of many machine learning systems that involve large-scale clustering, classification, kernel regression etc. Given a dataset $X = \{x_1, \dots, x_N\} \subset \mathbf{R}^d$ and a query q the goal is to find the closest point to this query $\arg\min_i ||x_i - q||^2$. Exhaustive linear scan requires $O(dN)$ operations, which could be too expensive. In such cases, approximate nearest neighbor (ANN) search methods are useful. In this paper we focus on the time-constrained ANN search which means that the search method is terminated after certain time budget has elapsed. Typically the methods are compared by *recall* measure which is the rate of queries for which the nearest neighbor was successfully found in a given time.

Partition trees. Paritition trees are a common method for ANN search. They recursively split the search database X into subsets up to a few close points (or even a single point) in each tree leaf. At the search stage, a query q is propagated down the tree in depth-first manner and only distances to points from a small number of visited leaves are evaluated. In this paper we focus on trees that use binary hyperplane partitions for splitting. In this case each internal node has parameters $w \in \mathbf{R}^d$ and $b \in \mathbf{R}$, and corresponds to the splitting of the space into two half-spaces $\{x | \langle w, x\rangle < b\}$ and $\{x | \langle w, x\rangle \geq b\}$. Below, we assume that w has a unit norm and we refer to it as *splitting direction*, while calling b the spliting threshold.

Typical methods tend to use splitting directions w which provide high variance of data projections in it $\underset{x}{\mathrm{Var}}\langle w, x\rangle$ as the resulting trees tend to group more closely located data points together. The threshold b is usually set to the mean or the median of projection values. During the propagation

of a query q, at each internal node the sign of the expression $(\langle w, q\rangle - b)$ determines the subtree that should be visited first. The other subtree is inserted into a priority queue. The propagation algorithm maintains this queue to extract subtrees that are promising to contain query neighbors.

KD-trees. Various partition trees differ by the sets from which they choose the splitting direction w. KD-trees[7] use only axis-aligned separating hyperplanes, i.e. $||w||_0 = 1$. Thus, each internal node of a KD-tree is assigned a particular coordinate i and a threshold b. The propagation through a KD-tree is efficient as it requires just one comparison operation in each internal node. The performance of KD-trees can be substantially boosted by combining them in ensembles[18, 14]. In this case several KD-trees are built for the same dataset and a shared priority queue is used during simulteneous traversal of all trees. Diversity of trees is reached by using randomness when choosing splitting axis for each internal node. Usually a random axis is picked from several axes with the highest variance.

PCA-trees. KD-trees are constrained to use axis-aligned separating hyperplanes, hence only d possible splitting directions are available during tree construction for each internal node. In contrast, PCA-tree[20] uses the main principal direction as a splitting direction at each node which is guaranteed to be the direction with maximum variance. Computing exact principal directions in each node is very expensive and the approximate version, using power method, was proposed in [13]. Apart from the slow construction time, the high quality of splitting directions in PCA-tree comes at a cost of expensive tree traversal as it requires $O(d)$ operations in each internal node. Moreover, PCA-tree requires much more additional memory as it keeps d-dimensional splitting direction in each node. Assuming that the deepest possible tree is constructed (as is usually beneficial for search accuracy) and each leaf therefore contains a single data point, the stored split directions take approximately same amount of memory as the data points themselves. While such doubling of the amount of memory in the case of a single tree might be acceptable, this additional memory requirement often becomes problematic when a forest of partition trees needs to be constructed for better performance.

RP-trees. Random-projection trees (RP-trees)[8] use randomly generated splitting directions for separating hyperplanes. Since such splitting directions are not adapted to a particular dataset, the RP-tree performance is typically lower compared to PCA-trees and the only advantage of RP-trees is faster tree construction process. However, [13] demonstrated that the usage of approximate principal directions allows to provide both efficient construction and high-quality space partitioning. The propagation cost in RP-trees is also expensive as it requires $O(d)$ operations.

TP-trees. Another approach, trinary-projection tree (TP-tree)[21] constructs splitting directions as sums of several coordinate axes with weights $+1$ or -1, which allows to avoid multiplications and use only additions and subtraction during tree traversal. The number of active coordinates in each splitting direction is limited by a small number $\bar{d}$ to make tree traveral even faster. Thus, TP-trees provide better splitting directions compared to KD-trees, and require $O(\bar{d})$ operations in each node during tree traversal, which is faster than in a PCA-tree.

Non-binary trees. Several state-of-the-art methods construct non-binary partition trees. [16] proposed to perform Hierarchical K-Means (HKM) to split the dataset recursive into K subsets in every non-leaf node. [15] demonstrated that HKM and KD-trees ensembles are complementary in a sense that for different datasets different methods are optimal. [15] also proposed FLANN framework that automatically chooses the optimal method for a particular dataset and tunes the parameters. Currently, FLANN is probably the most popular framework for ANN search problems.

Locality Sensitive Hashing. Another research direction in the field of nearest neighbor search are Locality Sensitive Hashing (LSH) methods[9, 3]. The original LSH method maps data points into a number of buckets using several hash functions such that the probability of collision is much higher for close points than for points which are further apart. At the search stage a query is also hashed and distances to all the points from the corresponding buckets are evaluated. Different extensions for the LSH idea were proposed such as multi-probe LSH[12] and the LSH forest[6]. While LSH have nice theoretical background, several works[14, 21, 19, 2] reported that they are outperformed by partition trees.

Product Quantization. Product quantization (PQ) is a lossy compression scheme for high-dimensional vectors [10]. The existing state-of-the-art methods for billion-scale ANN search, such as inverted multi-index [4] and its modifications [11], utilize PQ to perform very fine partition of search space into a large number of Voronoi cells. The centroids of these cells are constructed in a special form, so that each centroid is a concatenation of M codewords from M $\frac{d}{M}$-dimensional codebooks $C_1, \dots, C_M$, each containing K codewords. In other words, set of cell centroids is a Cartesian product $C = C_1 \times \dots C_M$. Such form allows to compute scalar products between a given query q and a large number of centroids via an efficient *ADC procedure* [10] based on lookup tables:

$$\langle q, c\rangle = \langle q, [c_1, \dots, c_M]\rangle = \sum_{m=1}^{M} \langle q_m, c_m\rangle \quad (2)$$

where q_m is the mth subvector of a query q. This sum can be calculated in M additions and lookups given that scalar products of query subvectors and codewords are precomputed. Thus, the ADC process requires $O(M \cdot K \cdot \frac{d}{M}) =$

$O(Kd)$ operations for lookup tables precomputations and $O(MN)$ operations to evaluate scalar products between query and N encoded points. K is usually taken small to make precomputation time negligible and the total computation time smaller than the $O(dN)$ computation time required for linear scan (exhaustive search). In our paper we adapt PQ idea to calculate query projection onto a large set of splitting directions efficiently.

3. Product Split Trees

In this section we provide a detailed description of the PS-tree data structure. Compared to existing partition trees, the important new element of the PS-tree is a special set of splitting directions $W \subset \mathbf{R}^d$ that is learned from data. Similarly to PQ idea, PS-tree forms W as a Cartesian product of M codebooks $W^1, \dots, W^M$:

$$W = W^1 \times \cdots \times W^M. \quad (3)$$

Each codebook W^m contains K d/M-dimensional vectors, $W^m = \{w_1^m, \dots, w_K^m\}, \quad m = 1, \dots, M$. In the following, we refer to these vectors as *subdirections*. Each splitting direction $w \in W$ is formed as a concatenation of M subdirections $[w^1, \dots, w^M]$, hence the total number of splitting directions $|W|$ equals K^M, which is a large number even for small K and M. For simplicity, in the following description we assume that $M = 2$, which means that PS-tree uses two subdirections codebooks W^1 and W^2. We discuss the case of larger M at the end of this section.

PS-tree construction. Let us assume that a search database $X = \{x_1, \dots, x_N\}$ of d-dimensional points is given and subdirections codebooks W_1 and W_2 have already been learned. Before the tree construction process starts, scalar products of database points and the subdirections from both codebooks are precomputed. We organize the values of these products into two tables $T^1, T^2 \in \mathbf{R}^{N \times K}$:

$$T^1[i,k] = \langle x_i^1, w_k^1 \rangle, \;\; i = 1, \dots, N \;\; k = 1, \dots, K$$
$$T^2[i,k] = \langle x_i^2, w_k^2 \rangle, \;\; i = 1, \dots, N \;\; k = 1, \dots, K, \quad (4)$$

where x_i^1 and x_i^2 are $d/2$-dimensional vectors corresponding to the first and the second halves of a point x_i respectively. After these preliminary precomputations the construction process starts.

As well as existing partition trees, PS-tree is constructed recursively by splitting the points in the particular node by a hyperplane $\langle w, x \rangle = b$, where w is a splitting direction with the highest data variance. Now we describe a procedure of choosing the splitting direction for a node that contains subset of points with indices $I = \{i_1, \dots, i_l\}$. Our goal is to solve the maximization problem:

$$\max_{w} \operatorname*{Var}_{i \in I} \langle w, x_i \rangle \quad (5)$$

As each $w \in W$ is a concatenation of some $w_m^1 \in W^1$ and $w_n^2 \in W^2$, the problem (5) can be transformed into the following:

$$\max_{m,n} \left[\operatorname*{Var}_{i \in I} \left(\langle w_m^1, x_i^1 \rangle + \langle w_n^2, x_i^2 \rangle \right) \right] =$$
$$\max_{m,n} \left[\operatorname*{Var}_{i \in I} \langle w_m^1, x_i^1 \rangle + \operatorname*{Var}_{i \in I} \langle w_n^2, x_i^2 \rangle + 2 \operatorname*{Cov}_{i \in I} \left(\langle w_m^1, x_i^1 \rangle, \langle w_n^2, x_i^2 \rangle \right) \right], \quad (6)$$

which can be rewritten in the form that uses precomputed look-up tables T^1 and T^2:

$$\max_{m,n} [\operatorname*{Var}_{i \in I} \left(T^1[i,m] \right) + \operatorname*{Var}_{i \in I} \left(T^2[i,n] \right) + 2 \operatorname*{Cov}_{i \in I} \left(T^1[i,m], T^2[i,n] \right)]. \quad (7)$$

The exact maximization of (7) would require evaluation of K^2 possible pairs (m, n) that is inefficient, especially for large K. Instead, a simple heuristic is used to boost tree construction efficiency. We compute variances of $T^1[i,m]$ and $T^2[i,n]$ for all m, n and choose a small number of candidates $\{m_1, \dots, m_t\}, \{n_1, \dots, n_t\}$ that provide the largest variances:

$$\operatorname*{Var}_{i \in I, m \in \{m_1, \dots, m_t\}} T^1[i,m] \geq \operatorname*{Var}_{i \in I, m' \notin \{m_1, \dots, m_t\}} T^1[i,m']$$
$$\operatorname*{Var}_{i \in I, n \in \{n_1, \dots, n_t\}} T^2[i,n] \geq \operatorname*{Var}_{i \in I, n' \notin \{n_1, \dots, n_t\}} T^2[i,n']. \quad (8)$$

Then the expression from (7) is evaluated only for the pairs $\{(m,n) | m \in \{m_1, \dots, m_t\}, n \in \{n_1, \dots, n_t\}\}$, which reduces the number of evaluations from K^2 to t^2. In the experiments, we observed that this heuristic has almost no influence on the PS-tree performance while making the tree construction process much faster. In all experiments we used $t = 10$.

After finding the optimal pair (m, n) the threshold b is adjusted. In previous works, b is typically set to the mean or to the median of projection coordinates onto the splitting direction. While providing the same quality in practice, the mean value is faster to compute and we use it in our experiments with PS-trees. Futhermore, in the case of PS-trees the calculation of the median would require explicit array of $(T^1[i,m] + T^2[i,n])$ values, while the mean can be found as a simple sum of expectations $\underset{i \in I}{E} T^1[i,m]$ and $\underset{i \in I}{E} T^2[i,n]$ which by the time of construction are already pre-computed in the process of variance evaluations.

After the split is constructed, the node points are distributed between two children nodes according to the usual hyperplane splitting: points $\{x_i | T^1[i,m] + T^2[i,n] < b\}$ go to the first child node and points $\{x_i | T^1[i,m] + T^2[i,n] \geq b\}$ go to the second child node, and the recursion proceeds.

Querying. The process of PS-tree querying is almost the same as for existing partition trees. The only difference is that query projections in all splitting subdirections are precomputed and reused during propagation. In more details,

when a PS-tree receives a query q, the lookup-tables S^1 and S^2 are filled in using the following rule:

$$S^1[k] = \langle w_k^1, q^1 \rangle \quad k = 1, \dots, K$$
$$S^2[k] = \langle w_k^2, q^2 \rangle \quad k = 1, \dots, K \tag{9}$$

where q^1 and q^2 are query subvectors corresponding to the first and the second halves of q respectively. The runtime cost of these calculations is negligible comparing to the total search time as typical values of K are small.

When tables S^1 and S^2 are precomputed, q is propagated down the tree in the usual partition tree manner. At each internal node with splitting direction $[w_m^1 w_n^2]$ a projection of q in this direction is efficiently calculated using two lookups and one sum operation:

$$\langle q, [w_m^1 w_n^2] \rangle = S^1[m] + S^2[n] \,, \tag{10}$$

after which the value (10) is compared with the splitting threshold b. The query is then propagated into the corresponding child node. The other child node is inserted into a priority queue with the priority $\left(S^1[m] + S^2[n] - b\right)^2$ (in the analysis, we assume that a min-heap is used for the priority queue). When q reaches a leaf, the exact distances from q to points from this leaf are evaluated. Then the next subtree is extracted from the priority queue. After given search time budget, the visited point with minimum distance to the query is returned.

Learning splitting directions. Now we describe how subdirection codebooks W^1 and W^2 are learned for a dataset $X = \{x_1, \dots, x_N\}$. Our learning approach is motivated by the assumption that for a particular dataset very good set of splitting directions can be obtained from the nodes of a PCA-tree constructed for this dataset. This is because in each node the splitting direction of the PCA-tree is guaranteed to provide the highest data variance. Another observation is that with a constrained number of splitting directions, the directions in the top nodes are more important. Achieving lower variance fast (i.e. in the top levels of the tree) is important as this e.g. would prevent "tall" subtrees from being added to the priority queue too often during traversal.

With these considerations in mind we propose the following learning method for the codebooks W^1 and W^2, assuming each of them has size K. We form two datasets $X^1 = \{x_1^1, \dots, x_N^1\}$ and $X^2 = \{x_2^1, \dots, x_N^2\}$ where x_i^1 and x_i^2 are the first and the second halves of x_i respectively. Then, for both X^1 and X^2 we construct the "tops" of PCA-trees progressing upto the depth $\lfloor \log_2 K \rfloor + 1$. By gathering the splitting directions from these shallow trees, we obtain the codebooks W^1 and W^2.

3.1. Analysis

We now analyse PS-tree design, and compare it to existing partition trees. The comparison is also summaraized in Table 1.

Splitting directions quality. Due to better adaptation to a particular dataset and a wide choise of possible splitting directions the PS-tree constructs much better separating hyperplanes compared to the KD-tree and provides comparable quality of space partition with PCA-tree and TP-tree.

Query propagation cost. In each internal node query proragation requires only three operations which makes the PS-tree almost as efficient as the KD-tree and more efficient than the PCA-tree and the TP-tree.

Storage cost. Each internal node of the PS-tree keeps two subdirection indices and a threshold value. Thus, its storage cost is almost the same as for the KD-tree. Additional memory requirements for the PCA-tree and the TP-tree are considerably higher. Thus, PCA-tree keeps d-dimensional vector in each internal node, while TP-tree node keeps $O(\bar{d})$ of information where $\bar{d}$ is a number of active coordinates.

Query preprocessing cost. Unlike existing trees, the PS-tree performs query preprocessing before propagation. This preprocessing includes $2 \times K$ calculations of dot-products of $d/2$-dimensional vectors, which has $O(Kd)$ complexity. As typical value of K is much smaller than the typical number of visited points (and the number of evaluated distances) the cost is negligible compared to the total search time.

Ensembles of randomized PS-trees. As other partition trees, PS-trees can be generalized to be used in ensembles with shared priority queues during search. Typically for this purpose, randomization is included into the tree construction process. In PS-tree we incorporate randomization in the following (standard) way. When evaluating t^2 possible (m, n) pairs we randomly choose one of the top five pairs with the highest variances. In the experiments below we show that the usage of randomized PS-tree ensembles boosts performance significantly (in the same way as it does for other partition trees). As we use the same codebook for all PS-trees in the ensemble, using multiple trees does not increase the query pre-processing time.

PS-trees with $M = 1$. In principle, one can omit the usage of Product Quantization in PS-trees and use $M = 1$. In this case the splitting directions can be also obtained from the top levels of the PCA-tree constructed for the particular dataset. However, the total number of possible splitting directions would then be equal to K. As K should be small for the efficient query preprocessing, PS-trees with $M = 1$ provide smaller numbers of splitting directions. We show below that this results in an inferior performance, especially for the case of tree ensembles.

PS-trees with $M > 2$. PS-trees also can be generalized for more than two subdirection codebooks. This however results in several drawbacks. First, finding an optimal splitting direction would require to search over K^M combinations, which becomes more difficult for the case $M > 2$

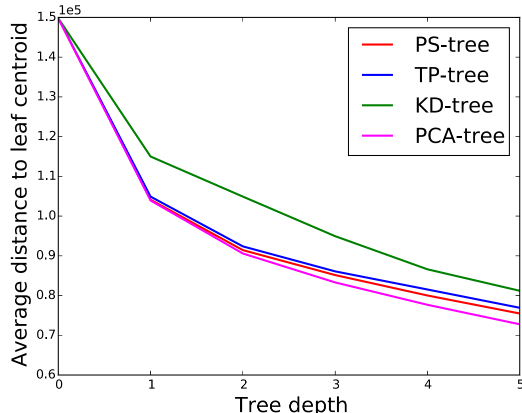

Figure 1. The mean distances from data points to centroids of partition regions that they belong to (for the SIFT1M dataset). The partition regions evaluated are those produced by the PS-tree, the TP-tree, the PCA-tree and the KD-tree of small depths. The PS-tree and the TP-tree/PCA-tree produce more compact partition regions compared to the KD-tree, which translates into more eficient leaf visitation order and therefore in a more efficient NN search.

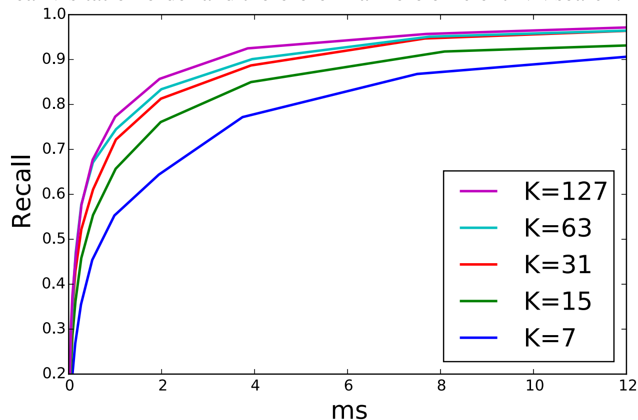

Figure 2. The comparison of PS-trees with different values of codebook sizes K on the SIFT1M dataset (recall vs. time) for $M = 2$. The performance saturates after $K = 127$. Even for small K the PS-tree provides reasonable performance that indicates the importance of splitting directions adaptation.

because the number of pairwise covariances becomes C_M^2. Moreover, the propagation cost is $O(M)$ operations and each node has to keep M indices instead of just two. In practice, the PS-tree with $M = 2$ already provides sufficient number of splitting directions and outperforms existing approaches. Similarly to inverted multi-indices (and largely for the same reasons), $M = 2$ seems to be the "sweet spot".

4. Experiments

In this section we provide the results of the experiments that compare the PS-tree with other existing partition trees and some non-tree ANN-search methods. The experiments were performed on four datasets:

1. **SIFT1M** dataset[10] that contains one million of 128-dimensional SIFT descriptors along with precomputed groundtruth for $10,000$ queries.
2. **DEEP1M** dataset [5], containing one million of 256-dimensional deep descriptors along with precomputed groundtruth for $1,000$ queries. Deep descriptors are formed as normalized and PCA-compressed outputs of fully-connected layer of a CNN pretrained on the Imagenet dataset[1]. Currently such deep descriptors are an important test case for nearest neighbor search algorithms.
3. **GIST1M**[10] that contains one million of 960-dimensional GIST descriptors[17] along with the precomputed groundtruth for $1,000$ queries.
4. **MNIST**[22], containing $70,000$ 784-dimensional vectors, corresponding to 28×28 images of handwritten digits. We used a random subset of $1,000$ vectors as queries and the remaining $69,000$ vectors as the base set.

In most of our experiments, different methods are compared based on the *recall* measure which is a rate of queries for which true nearest neighbor was successfully found. We measure the recall only with respect to the first nearest neigbor, as the ability to find several nearest neighbors is also important but in practice it is highly correlated with an ability to find the closest one. For efficiency, during subdirection codebook learning, we used the approximate version of PCA-tree construction from [13]. For KD-trees, we use the well-known implementation from the FLANN library [15]. The TP-tree implementation was not avaliable online or on request, and we carefully reimplemented it in C++ within the FLANN framework with the same level of optimization as the PS-tree. Thus all compared methods are implemented as different classes inherited from the same base class and differ only by their splitting and propagation functions. All other implementation details are kept the same to ensure fair comparisons. We intend to publish our code as additional FLANN classes with the publication. All the timings are obtained in the single-thread mode on the Intel(R) Xeon(R) CPU E5-2650 v2 @ 2.60GHz machine as the averaged results over ten runs.

Splitting quality. In the first experiment we demonstrate that PS-tree produces splitting directions which are of comparable quality with the directions of TP-tree or PCA-tree and are much better than KD-tree splitting directions. For that purpose, we construct a single tree of each type with the depths in the range $[1, 5]$ for SIFT1M dataset. For each depth value we measure the average distances from the data points to the centroids of the leaves they belong to, which is a quantitative measure of the partition compactness. Figure 1 shows that splitting directions of PS-tree and TP-tree/PCA-tree are of the same quality and are much better than KD-tree directions.

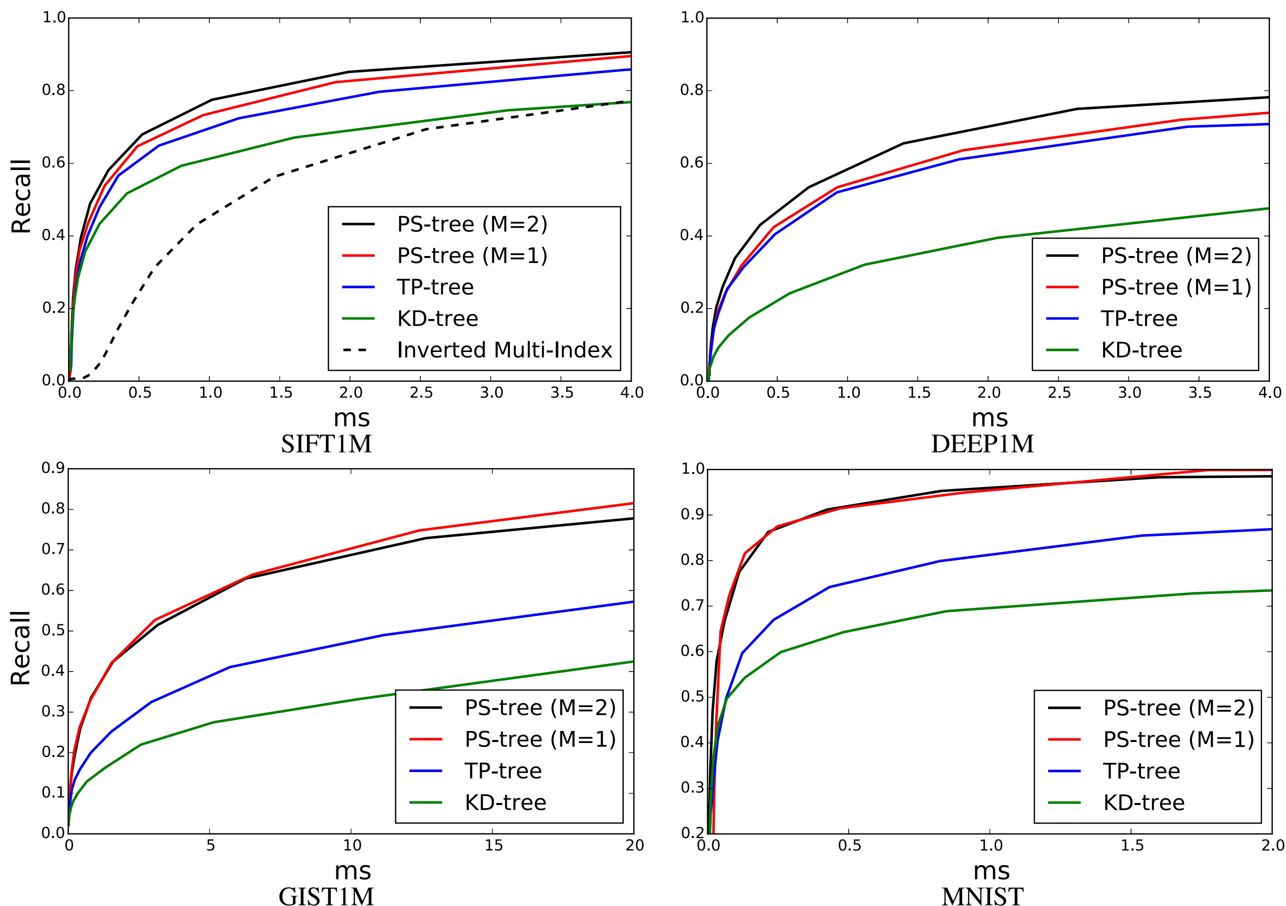

Figure 3. The comparison of a single PS-tree, a single TP-tree and a single KD-tree on the four datasets. For the whole range of possible time budget values PS-trees provide the highest recall. For SIFT1M the performance of the Inverted Multi-Index is also provided. For typical search times useful at the "million-scale", the inverted multi-index is inferior to all kinds of partition trees.

Choice of K. Then we investigate a question of an optimal K for a given dataset. Figure 2 demonstrates the performance of PS-trees for different values of K. The performance saturates with the values of K larger than 127. Note, that even with very small $K = 7$ (and 49 available splitting directions) the PS-tree provides quite reasonable performance that proves the importance of splitting directions adaptation to the particular dataset. In the experiments below we chose K to be close to the dataset dimensionality, namely $K = 127$ for SIFT1M, $K = 255$ for DEEP1M, 511 for MNIST and 1023 for GIST1M.

Searching with a single tree. Now we compare search performance of a single PS-tree, a single TP-tree and a single KD-tree. We compare the PS-tree variants with $M = 1$ and $M = 2$ to demonstrate the benefit of the PQ usage. PCA-tree is not included in the comparison as its performance was shown to be inferior [21]. For all benchmark datasets we measure recall achieved with different time budgets. The results are presented in Figure 3. For SIFT1M we also add the performance of the Inverted Multi-Index, which is the state-of-the-art method for the billion-scale ANN search. The parameters of all methods were accurately tuned on the hold-out datasets. Figure 3 shows that a single PS-tree outperforms existing methods on both datasets. The performance of the Inverted Multi-Index is lower than for all partition trees, which reveals its inefficiency at this scale (perhaps due to very unbalanced splitting of the data).

Performance of tree ensembles. We then compare the ensembles of PS-trees with the ensembles of TP-trees and FLANN. The size of ensembles is chosen so that the amount of required additional memory is approximately the same as the size of the indexed database. For a dataset of N points each partition tree contains $N-1$ internal nodes, assuming that each leaf contains only one point. Each leaf node keeps only an integer index (four bytes). Each internal nodes keeps two pointers to child nodes (which gives eight bytes), float-valued threshold (four bytes) and some information about splitting direction, depending on the tree kind. The KD-tree requires only one byte for the coordinate axis and the PS-tree requires two bytes for the two indices m, n. The TP-tree node requires $\bar{d}$ bytes which is a number of active coordinates. In our experiments the average value of $\bar{d}$ was eleven for SIFT1M, twelve for DEEP1M and fourteen for MNIST and GIST1M. Overall, internal nodes of the TP-

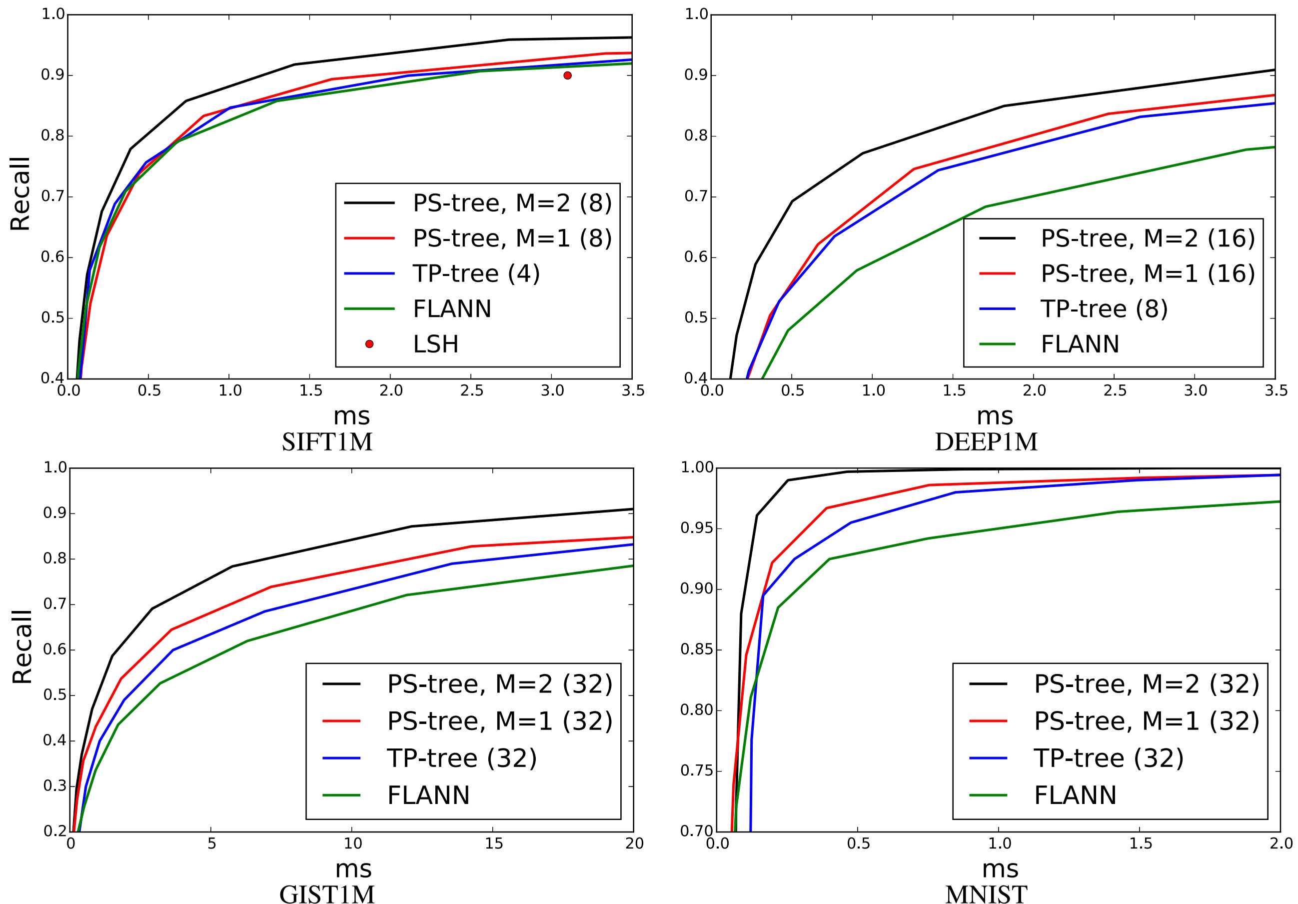

Figure 4. The comparison of different partition tree ensembles in the regime with the constrained additional memory. In this experiment, we cap the amount of the additional memory to be the same as the size of the search database. Therefore, for 128-dimensional SIFT1M eight KD-trees, eight PS-trees and four TP-trees are compared. For 256-dimensional deep descriptors, 16 KD-trees, 16 PS-trees and eight TP-trees are compared. In this regime, memory-effective PS-trees with $M = 2$ outperform existing partition tree ensembles considerably. PS-trees also outperform the state-of-the-art LSH method[3] under the same memory constraint.

tree require twice as many memory as required by the PS-tree. Hense, for SIFT1M we compare ensembles of eight PS-trees and four TP-trees, while for DEEP1M ensembles of 16 PS-trees and eight TP-trees are compared. For MNIST and GIST1M we compare ensembles of 32 PS-trees and 32 TP-trees, as the performance of PS-trees ensemble saturates and the addition of more than 32 trees gives no performance benefit. For comparison, we also provide the performance of the state-of-the-art LSH method [3] on SIFT1M with the same memory constraints. [3] reported that for LSH 0.9 accuracy level is achieved in 3.1 ms (on a faster CPU) with about 13 thousand candidates checked. PS-tree ensemble achieves 0.9 recall level in 1.5 ms on our CPU which corresponds to four thousand candidates on average. The results are presented in Figure 4. It demonstrates that for the regime with limited additional memory an ensemble of PS-trees outperforms other multiple partition trees on all four datasets. Note that for all datasets, PS-trees with $M = 2$ provides much higher recall compared to $M = 1$. This verifies the benefit of using product-quantized codebooks.

5. Summary

In this paper we have introduced Product Split (PS)-trees, a new data structure for the approximate nearest neighbor search. PS-trees provide a new state-of-the-art among partition trees for high-dimensional search, due to the combination of compact space partition and the computational efficiency of splitting functions. The high-level idea is to find a large set of promising splitting directions adapted to a particular dataset and then to learn a product codebook for the representation of this set. Using the learned codebooks, we then construct partition trees that can be traversed efficiently. Our approach thus builds on the product quantization idea, and extends it to the “million-scale” nearest neighbor search territory. Whereas previously product quantization was successfully combined with inverted indices, we show the merit of combining it with partition trees.

References

[1] A Berg and J Deng and and L Fei-Fei. Large scale visual recognition challenge (ILSVRC). http://www.image-net.org/challenges/LSVRC/2010/, 2010. 6

[2] M. Aly, M. E. Munich, and P. Perona. Indexing in large scale image collections: Scaling properties and benchmark. In *IEEE Workshop on Applications of Computer Vision (WACV 2011), 5-7 January 2011, Kona, HI, USA*, pages 418–425, 2011. 3

[3] A. Andoni, P. Indyk, T. Laarhoven, I. P. Razenshteyn, and L. Schmidt. Practical and optimal LSH for angular distance. In *NIPS*, 2015. 3, 8

[4] A. Babenko and V. Lempitsky. The inverted multi-index. In *CVPR*, 2012. 2, 3

[5] A. Babenko and V. S. Lempitsky. Tree quantization for large-scale similarity search and classification. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2015, Boston, MA, USA, June 7-12, 2015*, pages 4240–4248, 2015. 6

[6] M. Bawa, T. Condie, and P. Ganesan. LSH forest: self-tuning indexes for similarity search. In *Proceedings of the 14th international conference on World Wide Web, WWW 2005, Chiba, Japan, May 10-14, 2005*, pages 651–660, 2005. 3

[7] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18, 1975. 1, 3

[8] S. Dasgupta and Y. Freund. Random projection trees and low dimensional manifolds. In *Proceedings of the 40th Annual ACM Symposium on Theory of Computing, Victoria, British Columbia, Canada, May 17-20, 2008*, pages 537–546, 2008. 3

[9] M. Datar, N. Immorlica, P. Indyk, and V. S. Mirrokni. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the 20th ACM Symposium on Computational Geometry, Brooklyn, New York, USA, June 8-11, 2004*, pages 253–262, 2004. 3

[10] H. Jégou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *IEEE Trans. Pattern Anal. Mach. Intell.*, 33, 2011. 2, 3, 6

[11] Y. Kalantidis and Y. Avrithis. Locally optimized product quantization for approximate nearest neighbor search. In *in Proceedings of International Conference on Computer Vision and Pattern Recognition (CVPR 2014)*. IEEE, 2014. 2, 3

[12] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li. Multi-probe LSH: efficient indexing for high-dimensional similarity search. In *Proceedings of the 33rd International Conference on Very Large Data Bases, University of Vienna, Austria, September 23-27, 2007*, pages 950–961, 2007. 3

[13] M. McCartin-Lim, A. McGregor, and R. Wang. Approximate principal direction trees. In *Proceedings of the 29th International Conference on Machine Learning, ICML 2012, Edinburgh, Scotland, UK, June 26 - July 1, 2012*, 2012. 3, 6

[14] M. Muja and D. G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *VISAPP 2009 - Proceedings of the Fourth International Conference on Computer Vision Theory and Applications, Lisboa, Portugal, February 5-8, 2009 - Volume 1*, 2009. 3

[15] M. Muja and D. G. Lowe. Scalable nearest neighbor algorithms for high dimensional data. *IEEE Trans. Pattern Anal. Mach. Intell.*, 36, 2014. 3, 6

[16] D. Nistér and H. Stewénius. Scalable recognition with a vocabulary tree. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2006), 17-22 June 2006, New York, NY, USA*, 2006. 3

[17] A. Oliva and A. Torralba. Modeling the shape of the scene: A holistic representation of the spatial envelope. *International Journal of Computer Vision*, 42(3):145–175, 2001. 6

[18] C. Silpa-Anan and R. Hartley. Optimised kd-trees for fast image descriptor matching. In *2008 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2008), 24-26 June 2008, Anchorage, Alaska, USA*, 2008. 3

[19] K. Sinha. LSH vs randomized partition trees: Which one to use for nearest neighbor search? In *13th International Conference on Machine Learning and Applications, ICMLA 2014, Detroit, MI, USA, December 3-6, 2014*, pages 41–46, 2014. 3

[20] R. F. Sproull. Refinements to nearest-neighbor searching in k-dimensional trees. *Algorithmica*, 6, 1991. 1, 3

[21] J. Wang, N. Wang, Y. Jia, J. Li, G. Zeng, H. Zha, and X. Hua. Trinary-projection trees for approximate nearest neighbor search. *IEEE Trans. Pattern Anal. Mach. Intell.*, 36, 2014. 1, 3, 7

[22] Yann LeCun and Corinna Cortes and Christopher J.C. Burges. The MNIST database of handwritten digits. http://yann.lecun.com/exdb/mnist/. 6