Efficient Autotuning of Hyperparameters in Approximate Nearest Neighbor Search

Anonymous authors

No Institute Given

Abstract. Approximate nearest neighbor algorithms are used to speed up nearest neighbor search in a wide array of applications. However, current indexing methods feature several hyperparameters that need to be tuned to reach an acceptable accuracy–speed trade-off. A grid search in the parameter space is often impractically slow due to a time-consuming index-building procedure. Therefore, we propose an algorithm for automatically tuning the hyperparameters of indexing methods based on randomized space-partitioning trees. In particular, we present results using randomized k-d trees, random projection trees and randomized PCA trees. The tuning algorithm adds minimal overhead to the index-building process but is able to find the optimal hyperparameters accurately. We demonstrate that the algorithm is significantly faster than existing approaches, and that the indexing methods used are competitive with the state-of-the-art methods in query time while being faster to build.

Keywords: Nearest neighbor search · Approximate nearest neighbors · Randomized space-partitioning trees · Indexing methods · Autotuning

1 Introduction

Nearest neighbor search is a common component of algorithms and pipelines in areas such as machine learning [5, 18], computer vision [1, 14] and robotics [11]. In modern applications the search is typically performed in high-dimensional spaces (100–10000 dimensions) over large data sets.

An exhaustive k-nearest neigbor (k-NN) search is often prohibitively slow in applications which either require real-time responses (see e.g. [18]) or run on a resource-constrained device (see e.g. [11]). Hence, *approximate* nearest neighbor (ANN) search is often used instead. ANN algorithms first build an index in an offline phase, after which the index can be used to perform k-NN queries in sublinear time in an online phase. Most of the efficient algorithms fall into one of four categories: product quantization (PQ) [8], locality-sensitive hashing (LSH) [7, 4], graph-based methods [10], and tree-based methods [15, 13].

Because ANN algorithms are typically used as an auxiliary component of a pipeline, it can be important for a user that an algorithm requires minimal hand-tuning, especially if the type or size of the data can vary significantly. However, ANN algorithms typically have several hyperparameters which need to be tuned by a time-consuming grid search to achieve a given accuracy level or search time.

This problem is solved by an autotuning algorithm where the user specifies an accuracy level, and the tuning algorithm finds the optimal hyperparameter values. Previously, autotuning methods have been proposed for VP-trees [19], multi-probe LSH [4], k-means trees and RKD trees [13]. In this paper, we propose an autotuning method that is significantly faster than these methods.

Our approach is based on exploiting the structure of randomized space-partitioning trees [15, 13, 3, 6]. ANN algorithms based on randomized space-partitioning trees have been used recently for example in machine translation [5], object detection [1], recommendation engines [18] and head pose estimation [14].

Trees have several advantages: they are fast in high-dimensional spaces (see e.g. experiments in [13, 6]); they are simple to implement; they support easy insertion and deletion of points and they are independent, making the parallel implementation trivial. Also of great importance to us is that the structure of a tree-based index can be exploited to speed up the hyperparameter tuning.

Several types of randomized space-partitioning trees have been proposed for ANN search. Randomized k-d (RKD) trees [15] with a priority queue search are used in the popular open-source library FLANN [13]. Random projection (RP) trees [3] with a voting search have a stronger empirical performance than RKD trees with a priority queue search [6]. However, a single principal component (PCA) tree has been found to be more accurate than a single RP tree [17]. The PCA tree has two problems: it is not randomized, and indexing is slow. To solve these problems, we design a randomized variant of the PCA tree.

Typically ANN algorithms are compared in terms of the accuracy–speed trade-off. However, for the algorithm to be useful in practice, the index building procedure must be efficient as well. We test three different types of trees (RKD, RP and randomized PCA) with two search methods (priority queue and voting) considering both the query stage and the index building stage.

More specifically, in this article we:

- Propose an autotuning algorithm to optimize the hyperparameters of tree-based ANN search, and demonstrate that it is faster and more accurate than existing autotuning methods for ANN algorithms.
- Compare experimentally the effect of a) the randomization strategy and b) the search method on the efficiency of randomized trees. In particular, we find RP trees combined with voting search to be the best-performing.
- Demonstrate that the best tree-based method is nearly on par with the state-of-the-art ANN algorithms when measured on the accuracy–speed trade-off, and faster when measured on the index building time.

2 Approximate nearest neighbor search

In *k-nn search*, we have a data set $\mathbf{x} = (x_1, \ldots, x_n)$, where each $x_i \in \mathcal{A}$, from which we want to find the indices $f(q)$ of the k nearest neighbors for an arbitrary query point $q \in A$ measured by a dissimilarity measure $\mathrm{dis}(u, v) : \mathcal{A}^2 \mapsto \mathbb{R}$. We assume the dissimilarity measure to be the Euclidean distance $\|u - v\|_2$.

In *approximate nearest neighbor* (ANN) search, it is sufficient that the k points returned by the approximation algorithm are the true nearest neighbors of the query point only with high probability. We denote the returned points by $\hat{f}(q; \boldsymbol{\alpha}, \mathbf{r})$, where $\boldsymbol{\alpha}$ stands for the hyperparameters of the algorithm, and $\mathbf{r}$ stands for the realization of a set of random vectors used by the algorithm.

The accuracy of the approximation is measured by the *error rate* $\mathrm{Err}(q; \boldsymbol{\alpha}, \mathbf{r}) = \frac{1}{k} \sum_{j=1}^{k} \mathbb{1}(f_j(q) \notin \hat{f}(q; \boldsymbol{\alpha}, \mathbf{r}))$, which is the proportion of missed true nearest neighbors; the indices of the true nearest neighbors are denoted by $f(q) = (f_1(q), \ldots, f_k(q))$. Equivalently, we can use *recall*: $\mathrm{Rec}(q; \boldsymbol{\alpha}, \mathbf{r}) = 1 - \mathrm{Err}(q; \boldsymbol{\alpha}, \mathbf{r})$.

In addition to the error rate, we also consider the query time, denoted $\mathrm{Time}(q; \boldsymbol{\alpha}, \mathbf{r})$, when assessing the performance of an ANN algorithm. The hyperparameter optimization problem can be formulated in two ways:

1. Fix the expected error rate $e \in (0, 1)$ and find the hyperparameters $\boldsymbol{\alpha}$ that minimize $E[\mathrm{Time}(Q; \boldsymbol{\alpha}, \mathbf{R})]$ under the constraint $E[\mathrm{Err}(Q; \boldsymbol{\alpha}, \mathbf{R})] \leq e$.
2. Fix the expected query time $t \in (0, \infty)$ and find the hyperparameters $\boldsymbol{\alpha}$ that minimize $E[\mathrm{Err}(Q; \boldsymbol{\alpha}, \mathbf{R})]$ under the constraint $E[\mathrm{Time}(Q; \boldsymbol{\alpha}, \mathbf{R})] \leq t$.

The expectations $E[\cdot]$ are over both the distribution of a query point Q and the random vectors $\mathbf{R}$. These expectations can be estimated using a validation set of query points and a generated sample of random vectors.

3 Randomized space-partitioning trees

A binary space-partitioning tree recursively divides the data points into different cells with the assumption that nearby points fall into the same cells. Performing a query is subsequently a matter of determining to which cell the query point belongs and performing an exact search only within that cell.

3.1 Index construction

We describe how to grow a single tree recursively (Algorithm 1). At each branch of the recursion, the data set $\mathbf{x}$ is projected onto a chosen direction and assigned into one of the two child nodes by applying a predefined split criterion. In practice we use the median split to ensure balanced trees. This process is continued recursively at the child nodes until the maximum depth ℓ is met.

The type of a space-partitioning tree is determined by its choice of projection direction (see Figure 1 for an illustration on 2D data). In Algorithm 1, each different type of tree implements its own version of the abstract function GENERATE-DIRECTION which chooses this direction. Its argument $\boldsymbol{\psi}$ represents the tree-type dependent tuning parameters.

In randomized space-partitioning trees, the projection direction is chosen in a non-deterministic fashion. Randomized k-d (RKD) trees [15] choose a coordinate direction uniformly at random from m directions of the highest variance as the projection direction (we use $m = 5$ as suggested in [15]). Another popular

Algorithm 1

```
function GROW-TREE(depth, x, ℓ, ψ)
    if depth == ℓ then
        return indices of points in x as a tree node
    direction ← GENERATE-DIRECTION(ψ)
    p ← PROJECT(x, direction)
    cut ← SPLIT(p)
    left ← GROW-TREE(depth + 1, x[p ≤ cut], ℓ)
    right ← GROW-TREE(depth + 1, x[p > cut], ℓ)
    return (left, right, cut, direction) as a tree node
```

randomized variant is a random projection (RP) tree [3] in which the projection direction is chosen uniformly at random from the d-dimensional unit sphere. We use a sparse version [6], in which only a proportion $a = 1/\sqrt{d}$ of the components of the random vectors are non-zero.

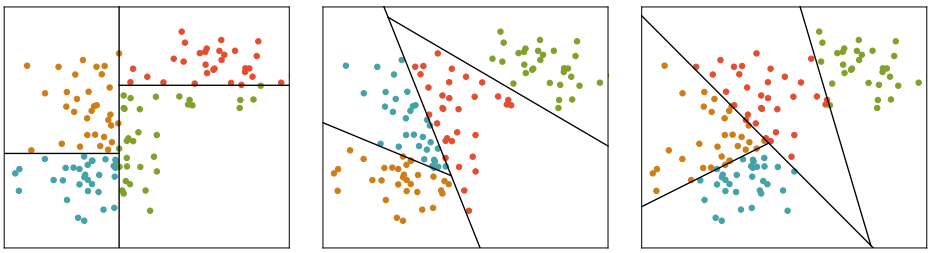

Fig. 1. Different projection directions: k-d (left), RP (middle) and PCA (right).

If the first principal component of the data is used as the projection direction, the resulting data structure is a principal component (PCA) tree [17]. However, the original PCA trees are on the one hand deterministic which makes improving accuracy with multiple trees impossible, and on the other hand slow to compute, as computing exact PCA is costly. To speed up the computation, using gradient descent updates to approximate the first principal component of the data at each node of the tree has been suggested [12]. However, index construction still takes $\mathcal{O}\left(nd^2(i+\ell)\right)$ time, where i is the number of gradient descent updates.

We make PCA trees more practical for ANN search by modifying the gradient descent update[1] to choose uniformly at random only $a = \sqrt{d}$ dimensions of the data at each node of the tree, and compute the estimated covariance matrix using only these dimensions. Growing a randomized PCA tree is an $\mathcal{O}\left(nd(i+\ell)\right)$ operation since now computing the sample covariance matrix takes only $\mathcal{O}\left(nd\right)$ operations. Considering only a sample of dimensions also ensures that the trees are randomized, allowing us to build multiple trees to increase accuracy.

[1] The gradient descent consistently converges with the learning rate $\gamma = 0.01$ in all our experiments; we did not observe further tuning of the learning rate to be necessary.

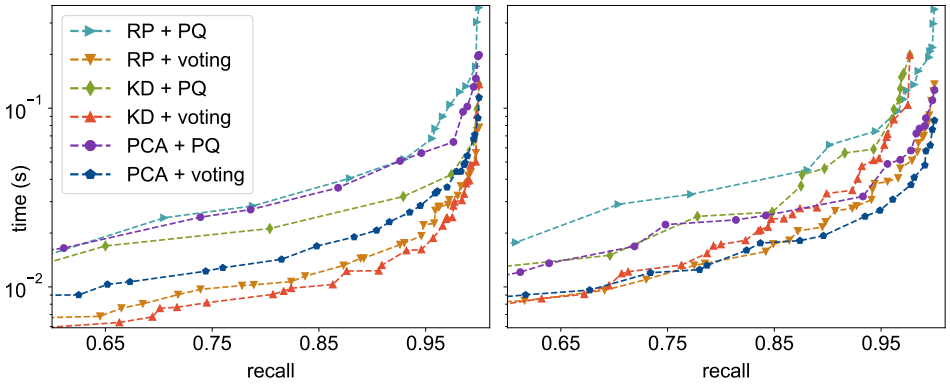

Fig. 2. Recall vs. query time with different trees and search methods for MNIST (left) and Fashion-MNIST (right) for $k = 10$. Towards bottom right is better.

3.2 ANN search using multiple trees

To use an index consisting of T randomized space-partitioning trees to find k approximate nearest neighbors of a query point q, the query point is first routed down to a leaf at each of the trees: at each level the query point is first projected onto the saved projection direction and then routed into the left or the right child node depending on which side of the split point its projection falls. There are two strategies to choose the candidate set of points for which the true distances are evaluated: priority queue search and voting search. Both of these are independent of the randomization strategy used to grow the trees.

Priority queue search In a priority queue search [15], a single priority queue, ordered according to the distance from the query point to the splitting hyperplanes, is maintained for all trees. When distances from the query point to all the points sharing a leaf with the query point are evaluated, b extra branches are explored; the priority queue is used to choose the branches.

Voting search In a voting search [6], distances are computed only to the subset of the points sharing a leaf with the query point. When a data point belongs to the same leaf as a query point in a tree, it gets a vote, and distances are evaluated only to the points that have at least v votes.

3.3 Comparison of randomization and search methods

Figure 2 shows the accuracy–speed trade-off for all combinations of the considered tree types and search methods on two benchmark data sets. For RP trees, the results are in line with previous experiments [6]. For each type of tree, voting outperforms priority queue (for a given recall level, its query time is faster).

For different tree types, the results vary between the data sets. Note that although both data sets for which results are shown have the same sample size

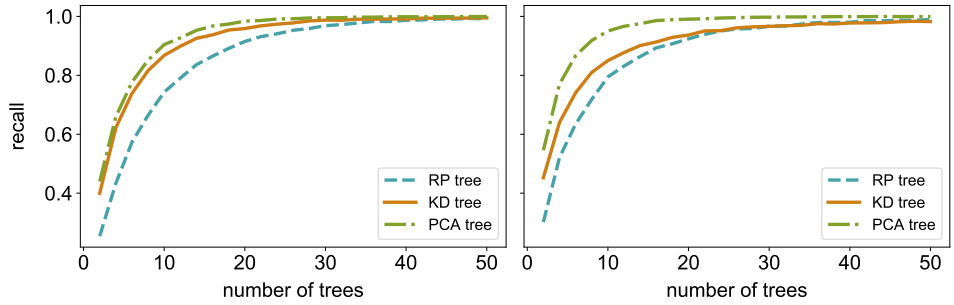

Fig. 3. Recall (for $k = 10$) as a function of the number of trees on MNIST (left) and Fashion-MNIST (right) for RP, RKD and randomized PCA trees. $\ell = 8, v = 1$.

($n = 60000$) and dimensionality ($d = 784$), the relative order of the trees is different: for MNIST, RKD trees are the fastest, and randomized PCA trees are the slowest; whereas for Fashion-MNIST, randomized PCA trees are the fastest, and RKD trees are the slowest. This means that the relative performance of different randomization strategies depends also on the distribution of the data.

Figure **??** further illustrates the differences between the tree types with fixed parameters: on Fashion-MNIST, PCA trees are noticeably more accurate than RKD trees and RP trees, especially for a small amount of trees. This explains the stronger performance of PCA trees with the optimal parameters; observe that the slightly stronger performance of RKD trees on MNIST is due to their faster projection times (1 vs. $\sqrt{d}$ operations per projection).

However, the differences between tree types are less pronounced than the difference between search methods. Since we can use the same projection vector on each node at the same level of an RP tree, they are the fastest to build (see Table 2). Thus, we present some of the experimental results only for them.

4 An autotuning algorithm

Since voting outperforms using a priority queue for all the data sets and the tree types, we present an autotuning algorithm for the voting search. Any of the different tree types can be used. Hence, the tuned hyperparameters $\boldsymbol{\alpha}$ are the number of trees T, the depth of the trees ℓ, and the vote threshold v.

The optimal hyperparameter values are searched from the whole range $\boldsymbol{\alpha}_{\text{lim}} = (1, \ldots, T_{\max}) \times (\ell_1, \ldots, \ell_{\max}) \times (1, \ldots, v_{\max})$; setting a grid interval is not required. Here we use $v_{\max} = T_{\max}$, $\ell_{\max} = \lfloor \log_2 n \rfloor$. Since each individual tree consumes the same amount of memory and takes an equal time to grow, $T_{\max}$ can be chosen as a limit on the building time or the memory consumption.

4.1 Estimating recall and candidate set size

The autotuning algorithm (Algorithm 2) first builds an index consisting of $T_{\max}$ trees of depth $\ell_{\max}$ (function GROW-TREES). The true neighbors of each test query q_i are subsequently found by the function EXACT-KNN.

Algorithm 2

function GENERATE-INDEX-AUTO($\boldsymbol{\alpha}_{\text{lim}}, \mathbf{x}, \mathbf{q}, k, \boldsymbol{\psi}$)
  trees $\leftarrow$ GROW-TREES($\mathbf{x}, \boldsymbol{\alpha}_{\text{lim}}, \boldsymbol{\psi}$)
  for $i = 1, \ldots, m$ **do**
    true-knn $\leftarrow$ EXACT-KNN($q_i, k, \mathbf{x}$)
    $A_i \leftarrow$ COUNT-ELECTED($\boldsymbol{\alpha}_{\text{lim}}$, q_i, true-knn)
    $B_i \leftarrow$ COUNT-ELECTED($\boldsymbol{\alpha}_{\text{lim}}$, q_i, $\{1, \ldots, n\}$)
  recalls $\leftarrow \frac{1}{km} \sum_{i=1}^{m} A_i$
  query-times $\leftarrow$ FIT-TIMES($\frac{1}{m} \sum_{i=1}^{m} B_i$, $\mathbf{x}$.dim)
  return recalls, query-times, trees

For each test query, the elected points are counted by COUNT-ELECTED (Algorithm 3) for two sets: the whole data set and the set of true k nearest neighbors. When using an index consisting of the first T trees, all the points that were elected when using an index consisting of the first $T - 1$ trees are also elected for the fixed vote threshold v. This means that we only have to count the points which get their v:th vote at the T:th tree (line **??** of Algorithm 3). Hence, we can count the numbers of elected points for all $1, \ldots, T_{\text{max}}$ number of trees with minimal overhead compared to counting them only for T_{max} trees.

Algorithm 3

function COUNT-ELECTED($\boldsymbol{\alpha}_{\text{lim}}$, q, I)
  initialize three-dimensional tensor A
  for $\ell = \ell_1, \ldots, \ell_{\text{max}}$ **do**
    initialize votes as zero vector of length n
    initialize c as zero vector of length v_{max}
    for $T = 1, \ldots, T_{\text{max}}$ **do**
      $c \leftarrow c +$ COUNT-VOTES(T, ℓ, q, I, votes)
      write c to A
  return A

Algorithm 4

function COUNT-VOTES(T, ℓ, q, I, votes)
  initialize counts as zero vector of length v_{max}
  leaf $\leftarrow$ node containing q at level ℓ of the T:th tree
  for point in leaf **do**
    if point $\in I$ **then**
      votes[point] $\leftarrow$ votes[point] + 1
      counts[votes[point]] $\leftarrow$ counts[votes[point]] + 1
  return counts

The counting is done by the function COUNT-VOTES (Algorithm 4) which adds a vote for each point of the node, and for each $v = 1, \dots v_{\max}$, counts how many points of this node get their v:th vote.

Finally, the expected recall and candidate set size can be estimated by their sample means for each parameter combination (lines 8 and 9 in Algorithm 2). Since a brute force strategy of performing actual test queries and timing them for each possible hyperparameter combination in the set $\boldsymbol{\alpha}_{\lim}$ is impractically slow, the function FIT-TIMES estimates the expected query time as a function of the candidate set size and data dimension as described in the following section.

4.2 Estimating the query time

We exploit linear scaling of the components of a query to build a model which estimates the query time. The query time can be split into the candidate pruning time and the final search time. Further, the candidate pruning phase is dominated by two operations: projecting the points onto the split directions, and vote counting. This suggests that we can estimate each of the three times separately:

$$\mathrm{Time}(q; \boldsymbol{\alpha}, \mathbf{r}) \approx \mathrm{Time}_{\mathrm{proj}}(q; \boldsymbol{\alpha}, \mathbf{r}) + \mathrm{Time}_{\mathrm{vote}}(q; \boldsymbol{\alpha}, \mathbf{r}) + \mathrm{Time}_{\mathrm{dist}}(q; \boldsymbol{\alpha}, \mathbf{r}).$$

Projection time The projection time depends on the type of randomization used in the trees. In RKD trees, the projection time is insignificant because coordinate axes are used as split directions. For RP trees and randomized PCA trees, the query point is projected onto a sparse vector at each level of each tree. Hence, the projection time is approximately linear w.r.t. the number of random vectors $z := T\ell$ the query point is projected onto. Thus, we can use a linear model to estimate the projection time for known hyperparameters T and ℓ.

To collect the data for the model, we design an experiment by choosing a representative sample $\mathbf{z} = (z_1, \dots, z_w)$ of sparse random matrices with d columns and $z_1, \dots, z_w$ total components, and measuring the elapsed times to multiply a d-component vector by each of these matrices. The sparsity is fixed as $a = 1/\sqrt{d}$.

When measuring running times, we observed that the random variation is typically small, but sometimes outliers appear, for example due to other processes activating on the background. This is why we use the Theil-Sen estimator [16] to model the dependence between the number of random vectors and projection time. It is a non-parametric estimator for a linear trend, and is much more robust against outliers than ordinary least squares regression.

Now the expected projection time for the hyperparameter values $\boldsymbol{\alpha} = (T, \ell, v)$ can be estimated as $\widehat{\mathrm{Time}}_{\mathrm{proj}}(\mathbf{q}; \boldsymbol{\alpha}, \mathbf{r}) = \hat{\beta}_0 + z\hat{\beta}_1$, where $z = T\ell$, and $\hat{\beta}_0$ and $\hat{\beta}_1$ are the intercept and the slope estimated by the Theil-Sen method.

Voting time For one tree, counting the votes means adding a vote for each point of the leaf the query point falls into. For T trees, this means that the whole voting step takes roughly Tn_0 operations, where $n_0 = \lceil n/2^{\ell} \rceil$ is the maximum leaf size. This means that we can model the voting time as a linear function of $y := Tn_0$, and proceed as in estimating the projection times.

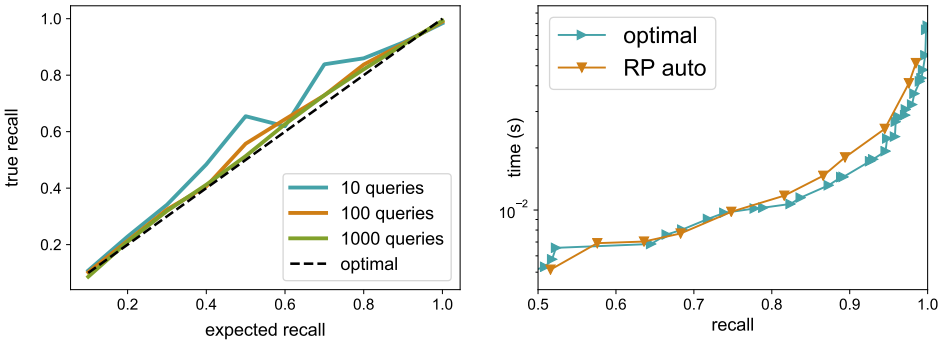

Fig. 4. Left: Recall estimated by autotuning vs. recall on the test set. Right: Recall vs. query time on test set for optimal parameters and auto-tuned parameters. k=10.

Final search time The final search in the candidate set is dominated by computing the distances to all $|S|$ points of the candidate set, which takes $|S|d$ operations; hence it is approximately linear with respect to the candidate set size $|S|$. Thus, we can proceed as before, this time measuring the time it takes to compute the distances from any d-dimensional query point to $|S|$ vectors of dimension d. After fitting the model, the final search time can be estimated as

$$\widehat{\text{Time}}_{\text{dist}}(\mathbf{q};\boldsymbol{\alpha},\mathbf{r}) = \hat{\alpha}_0 + |\bar{S}(\mathbf{q};\boldsymbol{\alpha},\mathbf{r})|\hat{\alpha}_1,$$

where $\hat{\alpha}_0$ and $\hat{\alpha}_1$ are the coefficients of the Theil-Sen estimator, and $|\bar{S}(\mathbf{q};\boldsymbol{\alpha},\mathbf{r})|$ is the observed mean candidate set size for this hyperparameter combination.

4.3 Using the autotuning index

After the expected recall levels and the query times have been computed, finding the optimal parameter combination is a matter of a simple table lookup. Since the index has already been built, growing new trees is not required: if the optimal parameter combination is $\hat{\boldsymbol{\alpha}} = (\hat{T}, \hat{\ell}, \hat{v})$, we can just pick the first $\hat{T}$ trees that have already been built, and prune them to depth $\hat{\ell}$.

5 Experimental results

First, we verify using RP trees that the autotuning algorithm accurately estimates the recall. Figure 3 (a) shows estimated recall on a validation set against recall on an independent test set for the MNIST data set. Larger validation sets yield sharper estimates, indicating the consistency of the estimator. The results are similar for other data sets and tree types. Figure 3 (b) compares on an independent test set hyperparameters optimized by the autotuning algorithm (RP auto) for the validation set to hyperparameters optimized for the test set (optimal). The parameters found by the algorithm are near-optimal.

		Target recall 80%				Target recall 90%			
		RP	LSH	VPtree	FLANN	RP	LSH	VPtree	FLANN
MNIST	tuning	**13.23**	26.84	744.4	102.2	**13.23**	24.61	926.1	113.9
	search	**0.111**	1.164	0.739	0.206	**0.169**	2.513	1.368	0.311
	recall	**0.822**	0.853	0.831	0.654	**0.909**	0.939	0.911	0.790
	stdev	±0.009	–	–	±0.020	±0.004	–	–	±0.017
Fashion	tuning	**13.22**	26.70	396.5	104.8	**13.23**	25.38	427.4	136.4
	search	**0.129**	0.917	0.353	0.310	**0.198**	1.575	0.557	0.216
	recall	**0.798**	0.850	0.813	0.693	0.881	0.927	**0.908**	0.825
	stdev	±0.007	–	–	±0.034	±0.006	–	–	±0.025
Trevi	tuning	**75.89**	156.1	3026	724.9	**76.28**	158.9	*	751.6
	search	**1.730**	14.01	13.58	2.813	**3.371**	25.63	*	4.276
	recall	**0.822**	0.837	0.832	0.566	**0.914**	0.918	*	0.679
	stdev	±0.011	–	–	±0.028	±0.006	–	*	±0.016
Random	tuning	**32.78**	55.48	120.6	134.6	**32.76**	54.56	134.0	149.1
	search	0.074	0.256	0.409	**0.049**	0.095	0.249	0.659	**0.087**
	recall	**0.804**	0.882	0.827	0.602	**0.902**	0.941	0.911	0.728
	stdev	±0.012	–	–	±0.015	±0.007	–	–	±0.015
GIST	tuning	**317.4**	484.1	960.4	*	**318.1**	437.9	1127	*
	search	**9.253**	122.4	41.55	*	**15.51**	205.7	66.54	*
	recall	**0.784**	0.862	0.864	*	**0.881**	0.942	0.940	*
	stdev	±0.011	–	–	*	±0.005	–	–	*

Table 1. Comparison of autotuning algorithms. Autotuning times (seconds), query times for 1000 queries (s) and recall (for $k = 10$) measured on a test set (* = did not complete within one hour). For the randomized algorithms (RP and FLANN), average recalls of 10 runs with the corresponding standard deviations are reported. The best result in each case is typeset in boldface.

Next, we compare the performance of the presented algorithm with RP trees to other autotuning algorithms for ANN: autotuning for VP-trees [19] and multi-probe LSH [4] implemented in NMSLib [2] and autotuning for RKD trees and hierarchical k-means trees in FLANN [13]. To the best of our knowledge, these are the only available ANN libraries that feature an autotuning method. The compared libraries and our own code are all written in C++.

The data sets used in the experiments are MNIST (n=60000, d=784), Fashion-MNIST (n=60000, d=784), Trevi (n=101120, d=4096), Random (n=256000, d=256) and GIST (n=1000000, d=960). Table 1 shows for two target recall rates (0.8 and 0.9) the autotuning time (including the index-building time), and the query time and recall on a test set which was not used to tune the hyper-parameters. The proposed tuning algorithm (with RP trees) is fastest at index building in all cases. Our approach has significantly faster query times than VP trees and LSH in all cases, and faster query times than FLANN for all but one data set. Our approach is also the most accurate at estimating the recall in most cases. The other methods systematically over- or underestimate the recall.

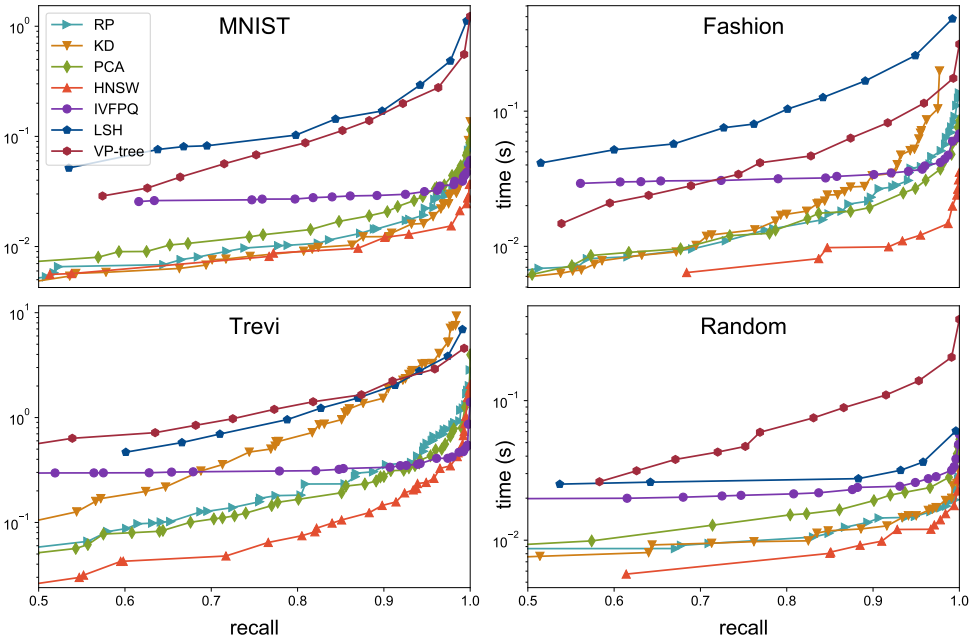

Fig. 5. Recall vs. query time (s) for 100 queries for different ANN algorithms. $k = 10$.

We also compare tree-based ANN against state-of-the-art quantization-, and graph-based algorithms: IVFPQ [8] and HNSW [10] implemented in FAISS [9]. As autotuning for these methods is not available, we perform a grid search on the possible parameter values. Figure 4 shows that tree-based methods are faster than PQ (except on highest recall levels) and close to the performance of HNSW. We emphasize that according to an independent benchmarking project[2], HNSW is the fastest ANN algorithm available. Multi-probe LSH and VP tree are also included in the comparison; they are significantly slower than the other methods.

	RP	RKD	PCA	HNSW	IVFPQ
MNIST	**3.62**	7.40	12.63	25.1	27.31
Fashion	**1.86**	14.3	13.12	20.2	30.71
Trevi	102	**43.2**	185	266	262.5
Random	**2.23**	6.83	27.8	90.3	63.59

Table 2. Index building times (seconds) for optimal parameters at 90% recall for different ANN algorithms. The best result on each data set is typeset in boldface.

[2] https://github.com/erikbern/ann-benchmarks

Finally, we compare the index building time (Table 2). Even though HNSW has faster query times, RP trees are significantly faster to build. The whole autotuning takes less time than building a single HNSW index. We emphasize that these results are on a single thread; the differences become more pronounced with multiple threads as the indexing process is embarrassingly parallel for trees.

References

1. Bilen, H., Pedersoli, M., Tuytelaars, T.: Weakly supervised object detection with convex clustering. In: CVPR. pp. 1081–1089. IEEE (2015)
2. Boytsov, L., Naidan, B.: Engineering efficient and effective non-metric space library. In: International Conference on Similarity Search and Applications. pp. 280–293. Springer (2013)
3. Dasgupta, S., Sinha, K.: Randomized partition trees for nearest neighbor search. Algorithmica **72**(1), 237–263 (2015)
4. Dong, W., Wang, Z., Josephson, W., Charikar, M., Li, K.: Modeling LSH for performance tuning. In: CIKM. pp. 669–678. ACM (2008)
5. Hassan, H., Elaraby, M., Tawfik, A.Y.: Synthetic data for neural machine translation of spoken-dialects. Small **16**, 17–33 (2017)
6. Hyvönen, V., Pitkänen, T., Tasoulis, S., Jääsaari, E., Tuomainen, R., Wang, L., Corander, J., Roos, T.: Fast nearest neighbor search through sparse random projections and voting. In: Big Data, IEEE International Conf. on. pp. 881–888 (2016)
7. Indyk, P., Motwani, R.: Approximate nearest neighbors: towards removing the curse of dimensionality. In: STOC. pp. 604–613. ACM (1998)
8. Jégou, H., Douze, M., Schmid, C.: Product quantization for nearest neighbor search. TPAMI **33**(1), 117–128 (2011)
9. Johnson, J., Douze, M., Jégou, H.: Billion-scale similarity search with GPUs. arXiv preprint arXiv:1702.08734 (2017)
10. Malkov, Y.A., Yashunin, D.A.: Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. arXiv:1603.09320 (2016)
11. McBryde, C.R.: Spacecraft Visual Navigation Using Appearance Matching and Multi-Spectral Sensor Fusion. Ph.D. thesis, Georgia Institute of Technology (2018)
12. McCartin-Lim, M., McGregor, A., Wang, R.: Approximate principal direction trees. In: ICML. pp. 1611–1618 (2012)
13. Muja, M., Lowe, D.G.: Scalable nearest neighbor algorithms for high dimensional data. TPAMI **36**(11), 2227–2240 (2014)
14. Papazov, C., Marks, T.K., Jones, M.: Real-time 3d head pose and facial landmark estimation from depth images using triangular surface patch features. In: CVPR. pp. 4722–4730. IEEE (2015)
15. Silpa-Anan, C., Hartley, R.: Optimised kd-trees for fast image descriptor matching. In: CVPR. pp. 1–8. IEEE (2008)
16. Theil, H.: A rank-invariant method of linear and polynomial regression analysis. In: Henri Theil's contributions to economics and econometrics, pp. 345–381 (1992)
17. Verma, N., Kpotufe, S., Dasgupta, S.: Which spatial partition trees are adaptive to intrinsic dimension? In: UAI. pp. 565–574. AUAI Press (2009)
18. Wang, L., Tasoulis, S., Roos, T., Kangasharju, J.: Kvasir: Scalable provision of semantically relevant web content on big data framework. IEEE Transactions on Big Data **2**(3), 219–233 (2016)
19. Yianilos, P.N.: Data structures and algorithms for nearest neighbor search in general metric spaces. In: SODA. vol. 93, pp. 311–321 (1993)