Revisiting kd-tree for Nearest Neighbor Search

Parikshit Ram*
p.ram@gatech.edu
IBM Research AI
Georgia, USA

Kaushik Sinha*
kaushik.sinha@wichita.edu
Wichita State University
Kansas, USA

ABSTRACT

kd-tree [16] has long been deemed unsuitable for exact nearest-neighbor search in high dimensional data. The theoretical guarantees and the empirical performance of kd-tree do not show significant improvements over brute-force nearest-neighbor search in moderate to high dimensions. kd-tree has been used relatively more successfully for approximate search [36] but lack theoretical guarantees. In the article, we build upon randomized-partition trees [14] to propose kd-tree based approximate search schemes with $O(d \log d + \log n)$ query time for data sets with n points in d dimensions and rigorous theoretical guarantees on the search accuracy. We empirically validate the search accuracy and the query time guarantees of our proposed schemes, demonstrating the significantly improved scaling for same level of accuracy.

KEYWORDS

nearest-neighbor search; similarity search; randomized algorithms; space-partitioning trees

ACM Reference Format:
Parikshit Ram and Kaushik Sinha. 2019. Revisiting kd-tree for Nearest Neighbor Search. In *The 25th ACM SIGKDD Conference on Knowledge Discovery and Data Mining (KDD '19), August 4–8, 2019, Anchorage, AK, USA.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3292500.3330875

1 MAKING kd-TREE COMPETITIVE

We focus on the ubiquitous and well-studied problem of Euclidean nearest-neighbor search – for any set of points $S \subset \mathbb{R}^d$ and any query $q \in \mathbb{R}^d$, find the point in S closest to q with respect to the ℓ_2 metric. The brute-force solution of scanning the complete set S for a single q becomes infeasible for sets with large cardinality (that is, $|S| = n$). kd-tree [16] was one of the first space-partitioning tree index proposed to solve this problem exactly in logarithmic time (that is, in $O(\log n)$ time) using a depth-first tree traversal algorithm followed by backtracking. The kd-tree is an axis-aligned partition of the space containing the set S, resulting in a hierarchical index of hyper-rectangles. However, the kd-tree based nearest-neighbor search suffered from the "curse of dimensionality" resulting in performance (empirical and theoretical) equivalent to or worse than the brute-force solution in many cases. Subsequently, various other space-partitioning trees (with related tree-traversal algorithms) such as metric trees [35], cover trees [10], PCA-trees [34] and more were explored for exact and approximate nearest-neighbor search.

Recognizing the hardness of exact nearest-neighbor search, especially in high dimensions, the focus shifted to finding the approximate nearest-neighbors – neighbors which are approximately as close as exact nearest-neighbors. Locality sensitive hashing (LSH) is one of the most popular techniques for approximate nearest-neighbor search [24], with various variants and improvements. The methods take advantage of randomization to provide worst-case guarantees for search accuracy and/or query time. An alternate to LSH are data-dependent schemes such as product quantization [25] and similarity graphs [22], which have been show to be extremely efficient empirically. However, these data dependent schemes lack rigorous theoretical guarantees, making it hard to understand how well these schemes work and when these schemes might fail. kd-tree has also been shown to be empirically competitive for approximate nearest-neighbor search with the introduction of randomization and ensembling, resulting in a widely popular FLANN nearest-neighbor search tool [36]. However, this incarnation of kd-tree still lacks favorable theoretical guarantees for high dimensional data.

Relatively recently, randomized-partition tree (`RPTree`) [14] was proposed for nearest-neighbor search, with theoretical guarantees on the search accuracy for the $O(d \log n)$ defeatist-tree search algorithm (depth-first tree traversal *without* any backtracking), and an ensemble of `RPTrees` were shown to empirically outperform LSH [40] – it is hard to compare `RPTree` and LSH theoretically since their guarantees have different forms (detailed in Section 2).

In this paper, we demonstrate how the theoretical search accuracy guarantees of a `RPTree` can be transferred to a kd-tree with improved search time scaling. Specifically, we show that:

- A kd-tree built on a randomly rotated version of the data set S has the same search accuracy guarantees as a `RPTree`, leading to a $O(d^2 + \log n)$ search algorithm based on a kd-tree.
- A $O(d \log d)$ approximate random rotation of the data allows kd-tree to retain the above search accuracy guarantees, resulting in a $O(d \log d + \log n)$ kd-tree based search algorithm – an improvement over the $O(d \log n)$ search time of a `RPTree` while possessing the same search accuracy guarantees.

These results allow a kd-tree based nearest-neighbor search scheme with rigorous theoretical guarantees on the search accuracy and a time complexity almost linear in d and logarithmic in n.

We discuss the existing literature on nearest-neighbor search and the relevant background for our proposed schemes in Section 2. We present the new kd-tree based search schemes with their theoretical guarantees in Section 3 and present relevant empirical evaluation

*Both authors contributed equally to this research.

KDD '19, August 4–8, 2019, Anchorage, AK, USA

ACM ISBN 978-1-4503-6201-6/19/08…$15.00
https://doi.org/10.1145/3292500.3330875

of the proposed schemes in Section 4. Finally, we conclude with the limitations and the potential future improvements in Section 5.

2 NEAREST-NEIGHBOR SEARCH

Nearest-neighbor search or similarity search is commonly encountered in computer science (for example, in machine learning, data mining) as well as in other physical sciences. Assuming that the items in the set S being searched over have some numeric representation, the set S of n items is encoded as a subset of $\mathbb{R}^d$. The problem of *exact* nearest-neighbor search for any query $q \in \mathbb{R}^d$ with respect to a distance (or dissimilarity) function $d\colon \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}_+$ is to find the point $p^* \in S$ such that

$$p^* = \arg\min_{p \in S} d(q, p). \tag{1}$$

The most widely studied version of this problem focuses on the ℓ_2 metric as the distance function ($d(q,p) = \|q-p\|_2$). Nearest-neighbor search with general normed metrics are often reduced to the ℓ_2 metric [3, 4] for efficient (approximate) solution. Space-partitioning tree indices (such as kd-tree [16], metric tree [35]) provided fast exact solutions for low to moderately high dimensions. However, their performances degrade in higher dimensions (higher values of d). One critical reason for the unfavourable performance of these tree-based exact search schemes is that even though the algorithm is able to find the nearest-neighbor quite fast, the process of empirically certifying that the candidate neighbor is *in fact the nearest* is computationally very expensive [39].

With the focus on approximate nearest-neighbor search, the usual procedure involves indexing the set S. For any query q, the index (in conjunction with an index traversal algorithm) is used to return a subset $S_q \subset S$, and the brute-force algorithm is used over S_q to find the approximate nearest-neighbor. The query time has two major parts: (i) the time to process q through the index to obtain the candidate neighbors, and (ii) the time to search over the candidates (possibly brute-force) to find the best candidate for q. The quality of the search is usually quantified as the recall[1] of the query's true nearest-neighborhood. The precision[2] is directly related to the search efficiency – high precision implies a small (and high-quality) subset S_q, which means the brute-force algorithm processes S_q faster. The overall success of any index-based search scheme is judged by its precision-recall trade-off (if available). Some common and representative search schemes are:

(i) The construction of binary-space-partitioning trees [5, 12, 16, 30, 36] on the set S in conjunction with a tree-traversal algorithm (such as a depth-first defeatist search algorithm) for any query q to obtain candidate neighbors.
(ii) Locality preserving hashing of S and q into a table [11, 15, 19, 24], and utilizing multiple hash tables to boost recall.
(iii) The generation of a codebook which quantizes the points in S in a data-dependent manner such that nearby points are quantized to the same code [18, 25, 48], and similar codes indicate points in relatively close proximity.
(iv) The generation of a (sparse) similarity graph between the points in S [22, 32, 33, 49] followed by a greedy graph traversal algorithm; these bear similarities to the traditional skip-lists (see Ken Clarkson's survey [12]) used in conjunction with Orchard's algorithm [38] or AESA [45].

The hashing based schemes are extremely popular. They have favorable theoretical guarantees and empirical performance. The quantization based methods and the similarity graph based methods are currently considered the state-of-the-art in terms of empirical performance. However, they lack any theoretical guarantees. FLANN [36] demonstrated that kd-tree based approximate search can also be extremely competitive even in high-dimensional data (FLANN actually uses both kd-tree and k-means tree [17, 37]). However, FLANN lacks any theoretical guarantees. The results we present here can be considered as a step towards filling that gap.

2.1 Randomized partition trees for search

`RPTree` is a binary space-partitioning tree which comes with favorable theoretical guarantees [14] and is empirically shown to outperform locality-sensitive hashing based schemes [40]. Whether `RPTree` avoids the curse of dimensionality depends on the underlying structure of the problem. We cannot directly compare the theoretical guarantees of `RPTree` and LSH: With LSH, the guarantees are of the following form for the *planted nearest-neighbor* problem by solving the point-location in Euclidean ball problem – for a given $\epsilon > 0$, the search returns all points within a distance of $(1+\epsilon)$ times the true nearest-neighbor distance with high probability in time $O(n^\rho)$ where $\rho < 1$, while being polynomial in the ambient dimensionality d. There are a couple of follow-up comments: (i) The guarantee requires knowledge of the true nearest-neighbor distance. In practice, this is handled by guess-and-halve. (ii) This requirement imposes practical issues where some queries have small or empty candidate sets[3], while others get large candidate sets, making it hard to control the precision-recall trade-off.

There has been numerous improvements to LSH based search over the last two decades. A lot of focus has been on learning data-dependent hash functions [20, 46, 47] such that the search returns better quality results compared to vanilla LSH. The schemes focus on learning data-structures/hashes (implicitly learning representations) on the data that preserve local neighborhoods while separating non-neighbors with lowest possible number of hash bits.

On the other hand, a `RPTree` [14] provides guarantees of the following form – for a query q and a set S of n points in $\mathbb{R}^d$, with a search time $O(d \log n)$, the search does not return the *exact nearest-neighbor* with probability at most λ where $\lambda \propto \Phi(q, S)$ where

$$\Phi(q,S) = \frac{1}{n-1} \sum_{i=2}^{n} \frac{\|q - p_{(1)}\|_2^2}{\|q - p_{(i)}\|_2^2}, \tag{2}$$

and $p_{(i)}$ is the i^{th} nearest-neighbor of q in S. Φ is called the potential function[4]. This definition is also extended to the k-nearest-neighbor problem for $k > 1$ (Theorem 9(b) in [14]) where the probability that the recall (of the k neighbors) is not equal to 1 is upper-bounded. There are a couple of things to note here:

- A small potential function value indicates that the nearest-neighbor is significantly closer to q than the rest of the points

[1] If S_q^* is the true neighbor set of q, recall is defined as $|S_q^* \cap S_q|/|S_q^*|$.
[2] The precision term is defined as $|S_q \cap S_q^*|/|S_q|$.
[3] Although this is practically handled via multi-probe LSH [31].
[4] $\Phi(q)$ can be explicitly computed, albeit, at $O(dn)$ cost per q. However, this quantity does not need to be computed for the purposes of the search.

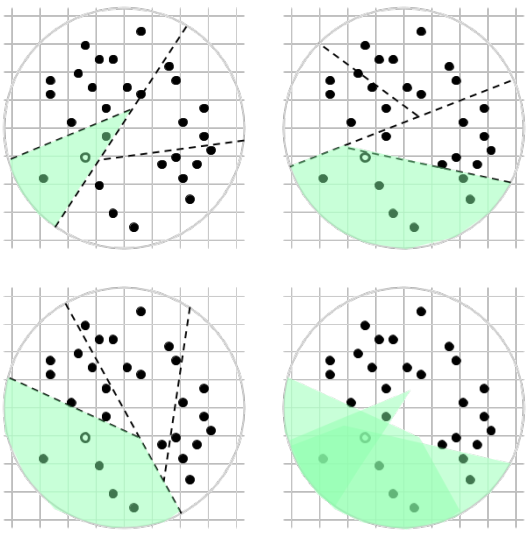


Figure 1: Illustration of the search with RPTree. The top row and the bottom left depict different RPTrees on the same data set S (filled points •) and query (single unfilled point ○). The bottom right image shows the compiled set of candidate neighbors found by the RPTree ensemble for the query.

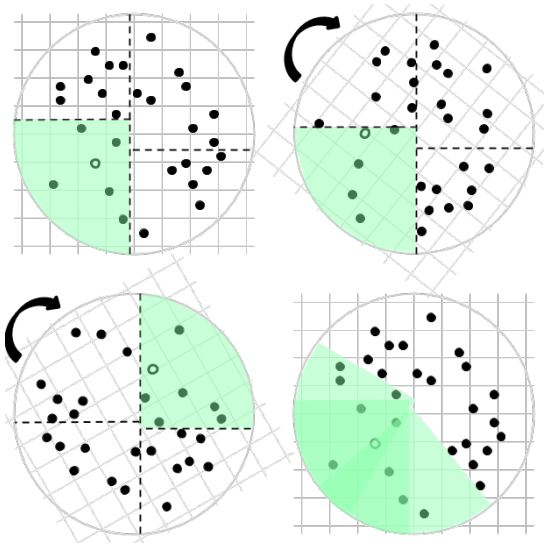


Figure 2: Illustration of the search with kd-tree on randomly rotated datasets (RR:KDTree and variants; see Section 3). The top row and the bottom left depict different kd-trees (with its axis aligned splits) on different rotations of the same data set S (filled points •) and query (single unfilled point ○). The bottom right image shows the compiled set of candidate neighbors found by the kd-tree ensemble for the query. This procedure is different from the one illustrated in Figure 1 for RPTree, but the results are similar.

in S. A large value indicates that all items (including the nearest-neighbor) are almost equally far away from q.

- The probability of missing the exact nearest-neighbor can be made arbitrarily small by utilizing multiple RPTrees. Figure 1 illustrates this search procedure. Using L trees boosts the success probability to $1 - \lambda^L$ with a $O(Ld \log n)$ search time.
- A forest of RPTrees allows for fine-grained control over the precision-recall trade off – L (approximately) balanced trees with maximum leaf-size n_0 always returns at most Ln_0 candidates, and users can control the trade-off by varying L.
- A crucial distinction is that RPTree provides a high probability guarantee on the **exact** nearest-neighbor while guarantees for LSH are only on the **approximate** nearest-neighbor.

However, the original RPTree has $O(nd)$ space complexity which was subsequently reduced to $O(nd^\rho)$ for some $\rho \in (0, 1)$ [41] and $O(d \log n)$ [28] using different techniques. A variant of RPTree [23] was also shown to be empirically very efficient although the precise search scheme lacks any theoretical guarantees. For L trees with leaf size n_0, the overall search time for RPTree based scheme is naively $O(Ld \log(n/n_0) + Ldn_0)$. For a small constant n_0, the search scales with $O(Ld \log n)$ (See appendix Section A for a detailed discussion on the choice of n_0). In practice, the Ldn_0 term can be significantly reduced by using techniques like voting [23], partial optimized distance computation and even quantization, while still retaining the theoretical accuracy guarantees of RPTree. Another important distinction is that the widely used SimHash [11] and the more recent near-optimal LSH [2] solve (and provide guarantees) for the angular nearest-neighbor search, which is equivalent to ℓ_2 nearest-neighbor search if all the points in S have the same norm. In practice, these are useful for ℓ_2 nearest-neighbor search since points close in ℓ_2 are close in angle; the only issue is that the search might return a lot of false positive candidates, hurting the precision without affecting the recall. In contrast, the guarantees for RPTree are on the ℓ_2 nearest-neighbor problem.

A connection was made between random-projection tree [13] and kd-tree by Vempala, 2012 [43] – it was shown that a kd-tree built on a data set after a randomized rotation adapts to the intrinsic dimensionality (specifically, the Assouad dimension [6]). A binary space partitioning tree is said to *adapt to the intrinsic dimensionality* $d' < d$ if the diameter of any node in the tree halves after $O(d')$ splits. The result was in a flavor similar to the intrinsic dimension adaptivity of the random-projection trees proposed by Dasgupta & Freund, 2008 [13] while preserving the axis-aligned splits of the kd-tree. The adaptivity guarantees of the random-projection tree was in terms of a notion of intrinsic dimensionality known as the *local covariance dimension*. Neither Vempala, 2012 [43] nor Dasgupta & Freund, 2008 [13] explicitly focused on nearest-neighbor search. The general understanding was that better adaptivity to the intrinsic dimension leads to better search performance, but an explicit (theoretical) connection was never made [44].

The RPTree (randomized partition tree) proposed by Dasgupta & Sinha, 2013 [14] for search is very similar to the aforementioned random-projection tree proposed by Dasgupta & Freund, 2008 [13] – there are subtle differences, mostly because they are geared towards different explicit goals. In the following section, we present a kd-tree based search scheme that retains the search accuracy guarantees of RPTree while achieving an improved search time scaling.

Algorithm 1: Tree indexing of set S with preconditioner $P(\cdot)$ and the node splitting rule `ChooseRule` ($P(\cdot)$ and `ChooseRule` for different trees are presented in Table 1 and Algorithm 2).

1 Tree $T \leftarrow$ `CreateIndex` $(S, n_0, P(\cdot))$
2 **Function `CreateIndex`:** $(S, n_0, P(\cdot)) \rightarrow T$
3 $S' \leftarrow \{P(x)\colon x \in S\}$ `// precondition data`
4 $l \leftarrow 0$ `// start building at level 0`
5 $T \leftarrow$ `MakeTree` $(S', n_0, 0)$ `// build tree`
6 **return** T
7 **end**
8 **Function `MakeTree`:** $(S, n_0, l) \rightarrow T$
9 **if** $|S| \leq n_0$ **then**
10 Make leaf node T such that $S_T \leftarrow S$
11 **else**
12 Rule $\rho(\cdot) \leftarrow$ `ChooseRule` (S, l) `// node split rule`
13 $S_l \leftarrow \{x \in S\colon \rho(x) = 0\}$
14 $T_l \leftarrow$ `MakeTree` $(S_l, n_0, l+1)$ `// left subtree`
15 $S_r \leftarrow S \setminus S_l$
16 $T_r \leftarrow$ `MakeTree` $(S_r, n_0, l+1)$ `// right subtree`
17 $T \leftarrow (\rho(\cdot), T_l, T_r)$
18 **end**
19 **return** T
20 **end**

Table 1: Preconditioners and node splitting rules for different trees (the variables used here are defined in the text; different choices for `ChooseRule` are detailed in Algorithm 2).

Method	Preconditioner $P(x)$	`ChooseRule`
`RPTree` [14]	x	`GaussianRP`
`SpGa:RPT` [41]	$H_d D x$	`SparseGaussianRP`
`RR:KDTree`	Γx	`KDTreeSplit`
`RC:KDTree`	$Dx \circledast \gamma$	`KDTreeSplit`
`FF:KDTree`	$H_d G \Pi H_d D x$	`KDTreeSplit`

2.2 `RPTree` construction and search algorithms

In this subsection, we present the precise algorithms for `RPTree` and our proposed schemes. The generic tree construction is presented in Algorithm 1 and the defeatist tree search is presented in Algorithm 3. The tree construction is preceded by a preconditioning step (Algorithm 1, line 3) where the data is modified in a pairwise-distance preserving way. Different space partitioning policies (tree node splitting rules `ChooseRule`) are presented in Algorithm 2 – `GaussianRP` function (line 2) corresponds to the original `RPTree` [14], `SparseGaussianRP` (p) routine (line 10) corresponds to a sparse `RPTree` (`SpGa:RPT`) with a sparsity parameter $p \in (0, 1]$ [41], and a simplified splitting policy of a kd-tree is presented in `KDTreeSplit` (line 19). The different choices for preconditioning are presented in Table 1.

The notation used in Table 1, Algorithms 1 - 3, and subsequently in the text are as follows: $\mathcal{N}$, $\mathcal{U}$, $\mathcal{B}$ and $\mathcal{R}$ denote the Gaussian, Uniform, Bernoulli and Radamacher distributions respectively. $P\colon \mathbb{R}^d \rightarrow \mathbb{R}^d$ is the preconditioning function. $H_d \in \{-1, +1\}^{d \times d}$ is the Walsh-Hadamard matrix (see Equation (5)). D is a diagonal matrix with $D_{ii} \sim \mathcal{R}$. $\Gamma \in \mathbb{R}^{d \times d}$ is a random rotation matrix with $\Gamma_{ij} \sim \mathcal{N}(0, 1)$. $\circledast$ is the circular convolution operator (see Equation (3)). $\gamma \in \mathbb{R}^d$ is a random vector with $\gamma_i \sim \mathcal{N}(0, 1)$. G is a diagonal matrix with $G_{ii} \sim \mathcal{N}(0, 1)$. $\Pi \in \{0, 1\}^{d \times d}$ is a random permutation matrix.

Algorithm 2: Definitions of different node splitting rules `ChooseRule` used in the tree construction algorithm described in Algorithm 1 for a set S at any tree level l (for `SparseGaussianRP`, $p \in (0, 1]$ controls the sparsity level).

1 Node splitting rule $\rho(\cdot) \leftarrow$ `ChooseRule` (S, l)
2 **Function `GaussianRP`:** $(S, l) \rightarrow \rho(\cdot)$**[14]**
3 Sample random vector $w \in \mathbb{R}^d\colon w_i \sim \mathcal{N}(0, 1)$
4 $V \leftarrow \{w^\top x \; \forall x \in S\}$
5 Sample $\beta \sim \mathcal{U}(1/4, 3/4)$
6 $v \leftarrow \beta$-th fractile of V
7 $\rho(x) \leftarrow \mathbb{I}(w^\top x > v)$
8 **return** $\rho(\cdot)$
9 **end**
10 **Function `SparseGaussianRP`** (p)**:** $(S, l) \rightarrow \rho(\cdot)$**[41]**
11 Sample vector $w \in \mathbb{R}^d\colon w_i \sim \mathcal{N}(0, 1)$
12 Sample diagonal matrix B such $B_{ii} \sim \mathcal{B}(p)$
13 $V \leftarrow \{(Bw)^\top x \; \forall x \in S\}$
14 Sample $\beta \sim \mathcal{U}(1/4, 3/4)$
15 $v \leftarrow \beta$-th fractile of V
16 $\rho(x) \leftarrow \mathbb{I}((Bw)^\top x > v)$
17 **return** $\rho(\cdot)$
18 **end**
19 **Function `KDTreeSplit`:** $(S, l) \rightarrow \rho(\cdot)$**[Section 3]**
20 $i \leftarrow l \mod d$
21 $V \leftarrow \{x_i \; \forall x \in S\}$ `//` x_i `i-th coordinate of` x
22 Sample $\beta \sim \mathcal{U}(1/4, 3/4)$
23 $v \leftarrow \beta$-th fractile of V
24 $\rho(x) \leftarrow \mathbb{I}(x_i > v)$
25 **end**

3 RANDOM ROTATION + kd-TREE

Building upon Vempala, 2012 [43], we use the same scheme of randomly rotating the data set S before building the kd-tree. Once S is rotated via a multiplication to a random matrix Γ, the kd-tree is built using `MakeTree` (Algorithm 1), where `ChooseRule` is set as `KDTreeSplit` (Algorithm 2, line 19), giving us `RR:KDTree` (Table 1). Query processing requires a randomized rotation of q followed by `DefeatistTreeSearch` (Algorithm 3) on the kd-tree with Γq. This search procedure is illustrated in Figure 2. The randomized rotation takes $O(d^2)$ time, and the defeatist search with kd-tree takes $O(\log n)$ time, leading to a $O(d^2 + \log n)$ query time compared to the $O(d \log n)$ search time of the original `RPTree`.

Although we will subsequently show that `RR:KDTree` possesses the same theoretical guarantees on search accuracy as `RPTree`, the query time bound of $O(d^2 + \log n)$ quickly becomes prohibitive for moderately high dimensions because of the d^2 term. To circumvent

Algorithm 3: Defeatist search algorithm for query q with tree T and preconditioner $P(\cdot)$. This search algorithm is employed for all the binary space-partitioning trees considered here.

```
Candidate neighbor set S_q ← SearchIndex (q, P(·), T)
Function SearchIndex: (q, P(·), T) → S_q
    S_q ← DefeatistTreeSearch (P(q), T)
    return S_q
end
Function DefeatistTreeSearch: (q, T) → S_q
    if T is a leaf node then
        S_q ← S_T                    // all points in leaf node
    else
        ρ(·), T_l, T_r ← T // Node split rule & children
        if ρ(q) = 0 then // go left
            S_q ← DefeatistTreeSearch (q, T_l)
        else // go right
            S_q ← DefeatistTreeSearch (q, T_r)
        end
    end
    return S_q
end
```

this issue, we present two efficient ways of approximating the randomized rotation while retaining the guarantees of RPTree.

3.1 Randomized circular convolution + kd-tree

Circular convolution [21] between vectors $x, \gamma \in \mathbb{R}^d$ is defined as

$$x \circledast \gamma = x^\top \begin{bmatrix} \gamma_1 & \gamma_d & \gamma_{d-1} & \cdots & \gamma_2 \\ \gamma_2 & \gamma_1 & \gamma_d & \cdots & \gamma_3 \\ \cdots & \cdots & \cdots & \cdots & \cdots \\ \gamma_d & \gamma_{d-1} & \gamma_{d-2} & \cdots & \gamma_1 \end{bmatrix}. \tag{3}$$

For a random Gaussian vector γ, $x \circledast \gamma$ emulates a randomized rotation with the restriction that the rows of the rotation matrix are *not independent*. The main motivation behind using circular convolution is that Equation (3) can be efficiently computed as

$$x \circledast \gamma = \mathcal{F}^{-1}\left(\mathcal{F}(x) \circ \mathcal{F}(\gamma)\right), \tag{4}$$

where $\circ$ is the Hadamard product (elementwise multiplication of vectors), and $\mathcal{F}(\cdot)$ is the Discrete Fourier Transform operator which can be computed in $O(d \log d)$ time with Fast Fourier Transform (FFT) compared to the $O(d^2)$ Γx operation in RR:KDTree. Circular convolution has previously been used in the context of nearest-neighbor search for learning binary embeddings [50].

For our purpose, we consider $P(x) = Dx \circledast \gamma$ as the approximate randomized rotation operation of any point x (D and γ are as defined in Section 2.2). We call this scheme RC:KDTree. For any query q, the $P(q) = Dq \circledast \gamma$ can be performed in $O(d \log d)$ time, with kd-tree DefeatistTreeSearch requiring $O(\log n)$ time, giving us a final query time complexity of $O(d \log d + \log n)$ for RC:KDTree.

3.2 FastFood[29] + kd-tree

Le et al., 2013 [29] build upon the fast Johnson Lindenstrauss transform [1] in the context of approximating expansions in kernel methods. Specifically, for diagonal matrices D and G, random permutation matrix Π and Walsh-Hadamard matrix H_d (as defined in Section 2.2), they introduce the "FastFood" matrix $H_d G \Pi H_d D$ as a surrogate for a dense Gaussian matrix and show that "any given row of the matrix $HG\Pi HD$ is i.i.d. Gaussian" (Section 3.2 in [29]). The matrix H_d is defined recursively with $H_1 := 1$ as:

$$H_m = \frac{1}{\sqrt{2}} \begin{bmatrix} H_{m/2} & H_{m/2} \\ H_{m/2} & -H_{m/2} \end{bmatrix} \forall m = 2^l, l \in \mathbb{Z}_+. \tag{5}$$

This is assuming $d = 2^l$ for some integer $l > 0$. This can be achieved in general by padding the points with additional columns of zeros.

For our purpose, we approximate the randomized rotation operation, a multiplication with a dense random matrix Γ, with the multiplication to $H_d G \Pi H_d D$. This gives us FF:KDTree– approximate randomized rotation via FastFood followed by DefeatistTreeSearch (Algorithm 3) on the kd-tree. For any query q, it can be shown that the $H_d G \Pi H_d D q$ operation can be performed in $O(d \log d)$ time, and the tree search takes $O(\log n)$, giving us a query time of $O(d \log d + \log n)$ similar to RC:KDTree.

3.3 Properties of randomized kd-tree

In this subsection, we formally present the time complexity and search accuracy guarantees for the proposed schemes. For search accuracy, we show that the proposed schemes possess the same guarantees as the original RPTree. Theorem 1 demonstrates how the search accuracy guarantee of RR:KDTree is identical to that of RPTree, while Theorems 2 and 3 extend this result to provide similar guarantees for RC:KDTree and FF:KDTree.

Theorem 1. *Given any query $q \in \mathbb{R}^d$, RR:KDTree requires $O(d^2)$ preconditioning time, $O(\log(n/n_0))$ tree traversal time, $O(dn_0)$ point processing time per tree and ensures that the failure probability of finding the exact nearest neighbor of q is identical to that of RPTree.*

Proof. For the random rotation matrix Γ in any RR:KDTree (Γ_{ij} i.i.d. $\mathcal{N}(0,1)$), let $\Gamma_1, \ldots, \Gamma_d \in \mathbb{R}^d$ be the d rows of Γ. For any $x \in S$, the i^{th} coordinate of $\Gamma x \in \mathbb{R}^d$ can be interpreted as projection of x onto a random projection direction Γ_i, whose entries are i.i.d. $\mathcal{N}(0,1)$. Therefore, at any internal node of RR:KDTree, if the KDTreeSplit function chooses the i^{th} coordinate of the randomly rotated S (via Γ), then during RR:KDTree construction, this internal node is split into left and right child nodes based the β fractile of the projected points corresponding to this internal node onto Γ_i. From RPTree perspective, this can be thought of as storing a random projection direction Γ_i at this internal node and splitting of this node into left and right child nodes by projecting the points corresponding to this node onto Γ_i and based on β fractile v of the projected points. Similarly, while answering a query with DefeatistTreeSearch, RR:KDTree routs the query q at this internal node (towards left or right child node) based on the i^{th} coordinate of Γq by comparing it with the β fractile v of this node. Again, from RPTree perspective, this can be thought of as projecting q onto the stored random projection direction Γ_i at this node and comparing this projected value with v. Therefore, the probability that q is separated from its nearest neighbor at this internal node of the RR:KDTree is exactly same as the probability that q is separated from its nearest neighbor at any internal node of an

Table 2: Indexing and search costs of the different randomized partition trees (Precond. → Preconditioning, trav. → traversal, proc. → processing). The leaf size for each tree is at most n_0. For sparse `RPTree`, $p \in (0, 1]$ controls the projection vector sparsity.

Method	Tree construction time	Index size	Precond. time	Tree trav. time	Leaf proc. time
`RPTree`	$(nd + n\log n)\log(n/n_0)$	$d\log(n/n_0) + (n/n_0) + n$	–	$d\log(n/n_0)$	dn_0
`SpGa:RPT`	$nd\log d + (npd + n\log n)\log(n/n_0)$	$d + pd\log(n/n_0) + (n/n_0) + n$	$d\log d$	$pd\log(n/n_0)$	dn_0
`SpRa:RPT`	$nd\log d + (npd + n\log n)\log(n/n_0)$	$d + pd\log(n/n_0) + (n/n_0) + n$	$d\log d$	$pd\log(n/n_0)$	dn_0
`RR:KDTree`	$nd^2 + (n\log n)\log(n/n_0)$	$d^2 + (n/n_0) + n$	d^2	$\log(n/n_0)$	dn_0
`RC:KDTree`	$nd\log d + (n\log n)\log(n/n_0)$	$d + (n/n_0) + n$	$d\log d$	$\log(n/n_0)$	dn_0
`FF:KDTree`	$nd\log d + (n\log n)\log(n/n_0)$	$d + (n/n_0) + n$	$d\log d$	$\log(n/n_0)$	dn_0

`RPTree` which represents the same set of points from S and utilizes Γ_i as random projection direction. The only difference between `RPTree` and `RR:KDTree` is that while the random projection direction at each internal node of an `RPTree` is different (thus requiring $O(n)$ distinct projection directions), in case of `RR:KDTree` only d random projection directions are implicitly used. However, since the failure probability of finding the exact nearest-neighbor of q by either `RPTree` or `RR:KDTree` is bounded from above by a union bound of the probabilities that q is not separated from its nearest neighbor at any internal node along the query's path from root to leaf, this probability bound is identical in both cases.

Finally, the preconditioning time is $O(d^2)$ for the matrix-vector multiplication Γq; the tree traversal time is $O(\log(n/n_0))$ since the tree depth is $O(\log(n/n_0))$ and constant computation is required during query processing at any internal node along q's path from root to leaf; $O(dn_0)$ time is required for the explicit distance computation between q and the n_0 points in the leaf q is routed to. □

Theorem 2. *Given any $q \in \mathbb{R}^d$, `RC:KDTree` requires $O(d\log d)$ preconditioning time, $O(\log(n/n_0))$ tree traversal time, $O(dn_0)$ point processing time per tree and ensures that the failure probability of finding the exact nearest neighbor of q is identical to that of `RPTree`.*

Proof. Let D and γ be as defined in Section 2.2. For any $x \in S$, the i^{th} coordinate of $Dx \circledast \gamma \in \mathbb{R}^d$ is $(Dx)^\top \sigma_i(\gamma)$, where $\sigma_i(\gamma)$ represents a circular permutation of γ, in particular the i^{th} column of the matrix in equation (3). Now $(Dx)^\top \sigma_i(\gamma) = x^\top(diag(D) \circ \sigma_i(\gamma))$, where $\circ$ represents Hadamard product and $diag(D) \in \mathbb{R}^d$ is the vector of the diagonal elements of D. Denoting $W_i = diag(D) \circ \sigma_i(\gamma)$, the i^{th} coordinate of $(Dx \circledast \gamma)$ is obtained by projecting x onto W_i. Using Lemma 1, we show that coordinates of W_i are i.i.d. $\mathcal{N}(0, 1)$. Subsequently, we use the same argument presented in the proof of Theorem 1 to demonstrate that the probability that `RC:KDTree` fails to find exact nearest neighbor of q is identical to that of `RPTree`.

The preconditioning $Dq \circledast \gamma$ via circular convolution operation can be computed in $O(d\log d)$ time with FFT during query processing. The tree traversal time and point processing times are same as in Theorem 1, giving us the statement of the theorem. □

Theorem 3. *Given any $q \in \mathbb{R}^d$, `FF:KDTree` requires $O(d\log d)$ preconditioning time, $O(\log(n/n_0))$ tree traversal time, $O(dn_0)$ point processing time per tree and ensures that the failure probability of finding the exact nearest neighbor of q is identical to that of `RPTree`.*

Proof. Let H_d, G, Π, D be as defined in Section 2.2 and let $A \in \mathbb{R}^{d\times d}$ be such that $A = H_d G \Pi H_d D$. Let $A_i \in \mathbb{R}^d$ be the i^{th} row of A. Le et al., 2013 (Section 3.2, [29]) show that coordinates of A_i are i.i.d. $\mathcal{N}(0, 1)$. Therefore, for any $x \in S$ (respectively, $q \in \mathbb{R}^d$), the i^{th} coordinate of Ax (respectively, Aq) can be obtained by projecting x (respectively, q) onto random projection direction A_i. With this interpretation and using the argument presented in the proof of Theorem 1, we can say that the probability that `FF:KDTree` fails to find exact nearest neighbor of q is identical to that of `RPTree`.

As shown in [29], the preconditioning computation $H_d G \Pi H_d D q$ can be computed in $O(d\log d)$ time – the multiplication with the diagonal matrices D and G can be done in $O(d)$ time; the permutation of a vector of length d (via implicit multiplication with the permutation matrix Π) takes $O(d)$ time; the multiplication to the Walsh-Hadamard matrix H_d can be done in $O(d\log d)$ time via the Fast Walsh-Hadamard Transform. The tree traversal time and point processing times are same as in Theorem 1. □

Lemma 1. *Let $u \in \mathbb{R}^d$ be a random vector whose entries are i.i.d. Radamacher variables and let $v \in \mathbb{R}^d$ be a random vector whose entries are i.i.d. $\mathcal{N}(0, 1)$. Then $y = u \circ v$ is a random vector in $\mathbb{R}^d$ whose entries are i.i.d. $\mathcal{N}(0, 1)$, where $\circ$ represents Hadamard product.*

Proof. First we show that each coordinate of y follows $\mathcal{N}(0, 1)$. Let y_j be the j^{th} coordinate of y, then $y_j = u_j v_j$. For any scalar t:

$$\begin{aligned}\Pr(y_j \le t) &= \mathbb{E}\left(\Pr\left(v_j \le t \mid u_i\right)\right)\\ &= \Pr(v_j \le t)\Pr(u_j = 1) + \Pr(-v_j \le t)\Pr(-u_j = 1)\\ &= \frac{1}{2}\Pr(v_j \le t) + \frac{1}{2}Pr(-v_j \le t)\\ &= \Pr(v_j \le t)\end{aligned}$$

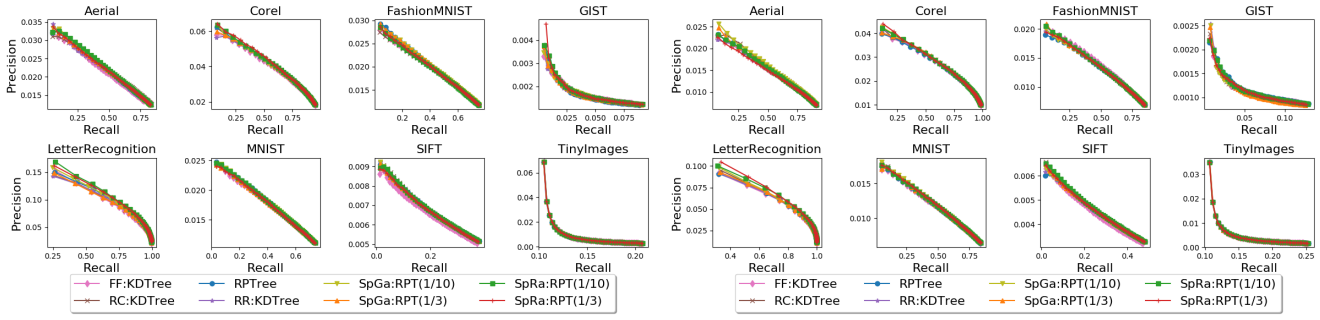
where the last equality follows from the fact that v_j and $-v_j$ both follow $\mathcal{N}(0, 1)$. Therefore, y_j and v_j have the same distribution.

Next, for any $i \neq j$ note that $y_i = u_i v_i$ and $y_j = u_j v_j$. Since u_i, u_j, v_i, v_j are independent of each other, y_i and y_j are independent. Therefore, all coordinates of y are pairwise independent. □

The query time complexities of `RR:KDTree` and its variants are summarized in Table 2 alongside the guarantees of existing `RPTree` and variants. We also present the tree construction time complexity and tree space complexity for completeness but skip the formal proof since these can be obtained through standard analyses. The complexities indicate improvements in both time and space for the proposed `RC:KDTree` and `FF:KDTree` relative to `RPTree`.

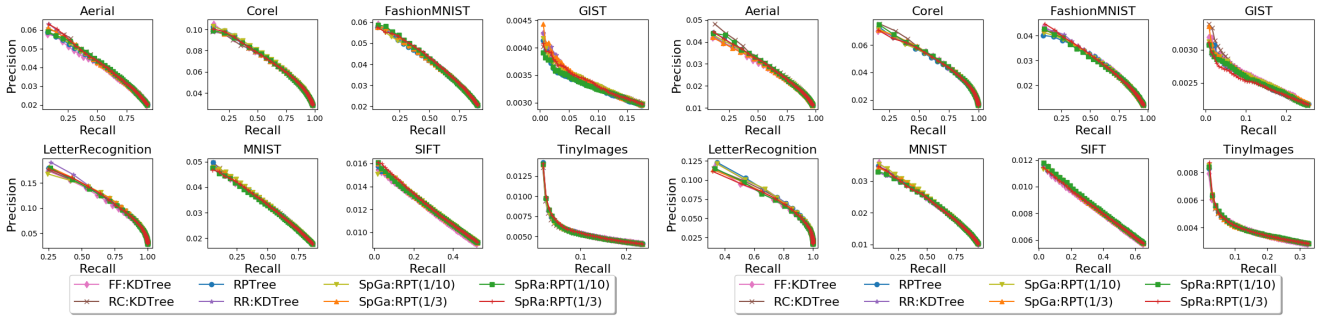
4 EMPIRICAL EVALUATIONS

In this empirical section, we focus on demonstrating that the proposed *kd*-tree based tree index is as effective as the theoretically



(a) $k = 10$, $n_0 = 10$

(b) $k = 10$, $n_0 = 20$



(c) $k = 100$, $n_0 = 100$

(d) $k = 100$, $n_0 = 200$

Figure 3: Recall vs. Precision curves for `RPTree` based schemes and the 3 proposed schemes for k-nearest-neighbor search with different values of k and leaf size n_0 for each tree for all the methods. The curves are generated with $L = 50$ trees.

guaranteed `RPTree` with respect to the nearest-neighbor search accuracy with improved scaling. We have limited the scope of the evaluation to methods having rigorous theoretical guarantees. We skip comparison to LSH since it has been demonstrated that the original `RPTree` significantly outperforms LSH while providing a much more fine-grained control over the accuracy-efficiency trade-off [40]. A more thorough empirical evaluation would involve comparisons to FLANN [36], product-quantization and similarity graph based methods. These methods have demonstrated great empirical performance but are limited in theoretical guarantees. Hence, a fair comparison to these schemes need to be in the form of accuracy (recall) vs. efficiency (number of queries per second) trade off [8]. However, the actual run times are very dependent on the implementation – a highly optimized implementation in C/C++ making use of SSE/SSE2/AVX intrinsics[5] can be *significantly* faster than a Python implementation. For this evaluation, we implement all the considered schemes in Python (for the ease of prototyping). We consider the original `RPTree` as well as the `SpGa:RPT` [41] and a variant of it where the Gaussian random projections (Algorithm 2, line 11) are replaced by Radamacher random variables to give us `SpRa:RPT` (sparse Radamacher `RPTree`). This is similar to the randomized partition trees considered by Hyvönen et al., 2016 [23] (barring the H_dDx preconditioning in `SpGa:RPT` (Table 1, [41])). For the purposes of evaluation, we consider the column sampling probability (the sparsity paramter) (i) $p = 1/3$ to give us `SpGa:RPT` (1/3) & `SpRa:RPT` (1/3), and (ii) $p = 1/10$ for `SpGa:RPT` (1/10) & `SpRa:RPT` (1/10)) for the above sparse `RPTree` variants based on the empirical evaluation by Sinha & Keivani, 2017 [41].

Implementation details. All the search schemes considered in this empirical evaluation are implemented in the high-level Python language (version 2.7)[6]. We use the `numpy` library for basic linear algebra. For the $\mathcal{F}$ and $\mathcal{F}^{-1}$ operations needed in `RC:KDTree`, we use `numpy.fft`. For the Fast Walsh-Hadamard transform, we use the extremely optimized Fast-Fast Hadamard Transform (`FFHT`[7]) implementation. All the evaluations were run single-threaded on a 24 core machine with 120GB memory running Ubuntu 18.04.

Datasets. We use various open datasets [7]. The dataset sizes are listed in Table 3 (see appendix Section B). The MNIST, Fashion MNIST, SIFT and GIST datasets are obtained from the ANN benchmark for Euclidean nearest-neighbor search [8]. We use a subset of the 80 million tiny images [42] to create the Tiny Images set.

4.1 Recall vs. Precision

In this comparison, we consider the same values of (L, n_0) for all methods and only perform the defeatist search *without* sorting the candidates we get from defeatist search. So, for l trees, the search time of the defeatist search with `RPTree` would be $O(ld \log n)$ instead of $O(ld \log n + ldn_0)$. The reason for this is because the

[5] https://software.intel.com/sites/landingpage/IntrinsicsGuide/

[6] https://github.com/rithram/rrkdt

[7] https://github.com/FALCONN-LIB/FFHT

constant term dn_0 is same for all the methods we are considering in our empirical evaluation. Then we compute the precision and recall for each $l = 1, \ldots, L$ to generate the recall vs. precision curves for all methods. Figure 3 presents the results on the 8 datasets for k-nearest-neighbor search, with trees of leaf size at most n_0 and $L = 50$ for different values of k, n_0. These results indicate that there is no apparent difference in the search accuracy-efficiency trade-offs between the proposed schemes and the baselines, the theoretical guarantees for `RR:KDTree`, `RC:KDTree` and `FF:KDTree` are validated in practice. The area under the curves for $k = 100, n_0 = 100$ are presented in the appendix C in Table 4 to demonstrate how similar the recall vs. precision trade off performances are.

4.2 Tree traversal time scaling with n and d

Combining the SIFT, GIST and Tiny Images data sets, we create data sets with $n = 1024, 2048, 4096, \ldots, 1000000$ and $d = 2^l, l = 7, \ldots, 11$ to evaluate the scaling of the proposed schemes and the baseline(s) (see dataset details in appendix Section B). We used a subset of size 25000 as the set of queries. The purpose of this evaluation is to show the combined scaling of the search algorithms (or rather implementation of the algorithms) with changing d **and** n. For this purpose, we consider the query time of all algorithms at $d = 128, n = 1024$ as their respective baselines and study the relative increase in run time with increasing d as well as n. For this evaluation, we only focus on the `RPTree` baseline (with $O(d \log n)$ scaling) and the proposed `RC:KDTree` and `FF:KDTree` (with $O(d \log d + \log n)$ scaling). The scaling is presented in Figure 4. The values reported in the results are the ratios of the actual query run time with that of the method at $n = 1024, d = 128$ with increasing n (horizontal axis) and increasing d (different markers). This allows us to see the effect of increasing n and d collectively.

The results demonstrate the favorable scaling of the proposed `RC:KDTree` and `FF:KDTree` with respect to n for a fixed d relative to that of `RPTree`. This is because of the fact, that for `RC:KDTree` and `FF:KDTree`, the $O(d \log d)$ preconditioning is independent of n, and the subsequent tree traversal cost for the kd-tree scales only at $O(\log n)$ independent of d. In contrast, `RPTree` query time increases (relatively) more significantly because of the $O(d \log n)$ scaling where the cost of increasing n gets multiplied by d.

The scaling with respect to d for the proposed `RC:KDTree` and `FF:KDTree` is not favorable relative to `RPTree`. However, one of the major reasons for this is that the actual runtimes are very dependent on the implementations, and we made no attempt to optimize our implementation beyond the available reference tools. The vector-vector dot-product operation, which is the main computational step is `RPTree` is extremely optimized in the standard linear algebra packages like `numpy`– the vector-vector or the vector-matrix multiplication is a super cache friendly operation because it is possible to stream all the operations straight through all the required memory without any cache misses, and potentially few waits on cold data. This is demonstrated by the d-scaling of *the implementation* of the $O(d \log n)$ `RPTree` for a fixed n – increasing d 16× (from 128 to 2048) only increases the actual run time by less than 2× even though the run time complexity of the `RPTree` defeatist search is pretty tight. In contrast, the FFT reference implementation is `numpy` is not optimized. For example, the sin/cos tables involved in the

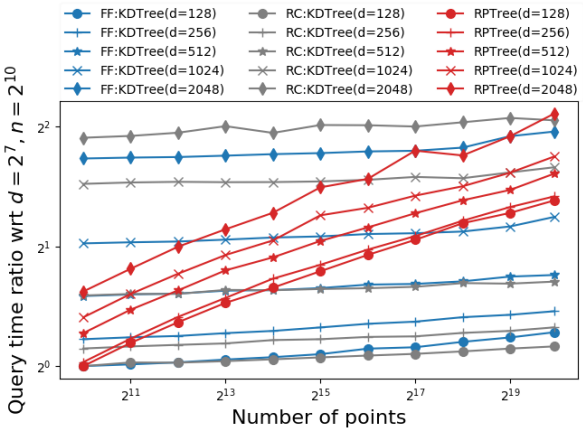


Figure 4: Scaling of `RPTree`, `RC:KDTree` and `FF:KDTree` with respect to d and n. The query run times are re-scaled so that the ratio for $n = 1024, d = 128$ is equal to 1. For this scaling experiment, we considered trees with leaf size $n_0 = 5$.

FFT operations are not cached between calls to FFT/inverse-FFT for vectors of same length. We can have a more efficient implementation of `RC:KDTree` by utilizing the highly optimized FFTW library. We discuss and demonstrate the implication of different implementations in detail in the Appendix D.

4.3 Index size scaling with n and d

The tree index size scaling is presented in Figure 6 in appendix Section E. The results indicate that for smaller values of n, `RC:KDTree` and `FF:KDTree` have much smaller tree index size in memory compared to `RPTree`. But as n grows, the index size is dominated by the terms linear in n, making the index sizes of all three schemes very close to each other. Note that the proposed schemes `RC:KDTree` and `FF:KDTree` do not ever have larger index size than `RPTree`.

5 CONCLUDING REMARKS

We proposed a way to make kd-tree based nearest-neighbor search theoretically (and empirically) competitive in high dimensions, resulting in a scheme with $O(d \log d + \log n)$ query time. This scheme can be used for improved theoretical and empirical scaling of `RPTree` for the problem of maximum inner-product search [27, 28]. However, this is still some ways off from the $O(\log n)$ query time of kd-tree in FLANN [36] not withstanding the lack of any theoretical guarantees for FLANN. We believe that our proposed scheme can be improved upon to get a kd-tree based search scheme with close to $O(\log n)$ scaling while still retaining the search accuracy guarantees. In addition to improved theoretical guarantees, we also plan to create and open-source an optimized implementation of the proposed schemes that is implemented in a low-level language (such as C/C++) and makes use of most efficient available libraries (such as FFTW for FFT and inverse FFT). Moreover, we plan to explore (i) auxilliary information and priority functions [26] to improve the overall space complexity, and (ii) inverted multi-index [9] with kd-tree for improved search efficiency.

REFERENCES

[1] Nir Ailon and Bernard Chazelle. 2009. The fast Johnson–Lindenstrauss transform and approximate nearest neighbors. *SIAM Journal on computing* 39, 1 (2009), 302–322.

[2] Alexandr Andoni, Piotr Indyk, Thijs Laarhoven, Ilya Razenshteyn, and Ludwig Schmidt. 2015. Practical and optimal LSH for angular distance. In *Advances in Neural Information Processing Systems*. 1225–1233.

[3] Alexandr Andoni, Assaf Naor, Aleksandar Nikolov, Ilya Razenshteyn, and Erik Waingarten. 2018. Hölder homeomorphisms and approximate nearest neighbors. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*. IEEE, 159–169.

[4] Alexandr Andoni, Huy L Nguyen, Aleksandar Nikolov, Ilya Razenshteyn, and Erik Waingarten. 2017. Approximate near neighbors for general symmetric norms. In *Proceedings of the 49th Annual ACM SIGACT Symposium on Theory of Computing*. ACM, 902–913.

[5] Sunil Arya, David M Mount, Nathan Netanyahu, Ruth Silverman, and Angela Y Wu. 1994. An optimal algorithm for approximate nearest neighbor searching in fixed dimensions. In *Proc. 5th ACM-SIAM Sympos. Discrete Algorithms*. 573–582.

[6] Patrice Assouad. 1979. Etude d'une dimension metrique liee a la possibilite de plongements dans Rˆ n. *CR Acad. Sci. Paris Sér. AB* 288 (1979), A731–A734.

[7] Arthur Asuncion and David Newman. 2007. UCI machine learning repository. https://archive.ics.uci.edu/ml/datasets.html

[8] Martin Aumüller, Erik Bernhardsson, and Alexander Faithfull. 2017. ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. In *International Conference on Similarity Search and Applications*. Springer, 34–49. https://github.com/erikbern/ann-benchmarks

[9] Artem Babenko and Victor Lempitsky. 2012. The inverted multi-index. In *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*. IEEE, 3069–3076.

[10] Alina Beygelzimer, Sham Kakade, and John Langford. 2006. Cover trees for nearest neighbor. In *Proceedings of the 23rd international conference on Machine learning*. ACM, 97–104.

[11] Moses S Charikar. 2002. Similarity estimation techniques from rounding algorithms. In *Proceedings of the thiry-fourth annual ACM symposium on Theory of computing*. ACM, 380–388.

[12] Kenneth L Clarkson. 2006. Nearest-neighbor searching and metric space dimensions. *Nearest-neighbor methods for learning and vision: theory and practice* (2006), 15–59.

[13] Sanjoy Dasgupta and Yoav Freund. 2008. Random projection trees and low dimensional manifolds. In *Proceedings of the fortieth annual ACM symposium on Theory of computing*. ACM, 537–546.

[14] Sanjoy Dasgupta and Kaushik Sinha. 2013. Randomized partition trees for exact nearest neighbor search. In *Conference on Learning Theory*. 317–337.

[15] Mayur Datar, Nicole Immorlica, Piotr Indyk, and Vahab S Mirrokni. 2004. Locality-sensitive hashing scheme based on p-stable distributions. In *Proceedings of the twentieth annual symposium on Computational geometry*. ACM, 253–262.

[16] Jerome H Friedman, Jon Louis Bentley, and Raphael Ari Finkel. 1976. An algorithm for finding best matches in logarithmic time. *ACM Trans. Math. Software* 3, SLAC-PUB-1549-REV. 2 (1976), 209–226.

[17] Keinosuke Fukunaga and Patrenahalli M. Narendra. 1975. A branch and bound algorithm for computing k-nearest neighbors. *IEEE transactions on computers* 100, 7 (1975), 750–753.

[18] Tiezheng Ge, Kaiming He, Qifa Ke, and Jian Sun. 2014. Optimized product quantization. *IEEE transactions on pattern analysis and machine intelligence* 36, 4 (2014), 744–755.

[19] Aristides Gionis, Piotr Indyk, Rajeev Motwani, et al. 1999. Similarity search in high dimensions via hashing. In *Vldb*, Vol. 99. 518–529.

[20] Yunchao Gong, Svetlana Lazebnik, Albert Gordo, and Florent Perronnin. 2013. Iterative quantization: A procrustean approach to learning binary codes for large-scale image retrieval. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 35, 12 (2013), 2916–2929.

[21] Robert M Gray et al. 2006. Toeplitz and circulant matrices: A review. *Foundations and Trends® in Communications and Information Theory* 2, 3 (2006), 155–239.

[22] Kiana Hajebi, Yasin Abbasi-Yadkori, Hossein Shahbazi, and Hong Zhang. 2011. Fast approximate nearest-neighbor search with k-nearest neighbor graph. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, Vol. 22. 1312.

[23] Ville Hyvönen, Teemu Pitkänen, Sotiris Tasoulis, Elias Jääsaari, Risto Tuomainen, Liang Wang, Jukka Corander, and Teemu Roos. 2016. Fast nearest neighbor search through sparse random projections and voting. In *Big Data (Big Data), 2016 IEEE International Conference on*. IEEE, 881–888.

[24] Piotr Indyk and Rajeev Motwani. 1998. Approximate nearest neighbors: towards removing the curse of dimensionality. In *Proceedings of the thirtieth annual ACM symposium on Theory of computing*. ACM, 604–613.

[25] Herve Jegou, Matthijs Douze, and Cordelia Schmid. 2011. Product quantization for nearest neighbor search. *IEEE transactions on pattern analysis and machine intelligence* 33, 1 (2011), 117–128.

[26] Omid Keivani and Kaushik Sinha. 2018. Improved nearest neighbor search using auxiliary information and priority functions. In *International Conference on Machine Learning*. 2578–2586.

[27] Omid Keivani, Kaushik Sinha, and Parikshit Ram. 2017. Improved maximum inner product search with better theoretical guarantees. In *2017 International Joint Conference on Neural Networks (IJCNN)*. IEEE, 2927–2934.

[28] Omid Keivani, Kaushik Sinha, and Parikshit Ram. 2018. Improved maximum inner product search with better theoretical guarantee using randomized partition trees. *Machine Learning* (2018), 1–26.

[29] Quoc Le, Tamás Sarlós, and Alex Smola. 2013. Fastfood-approximating kernel expansions in loglinear time. In *Proceedings of the international conference on machine learning*, Vol. 85.

[30] Ting Liu, Andrew W Moore, Ke Yang, and Alexander G Gray. 2005. An investigation of practical approximate nearest neighbor algorithms. In *Advances in neural information processing systems*. 825–832.

[31] Qin Lv, William Josephson, Zhe Wang, Moses Charikar, and Kai Li. 2007. Multi-probe LSH: efficient indexing for high-dimensional similarity search. In *Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment, 950–961.

[32] Yury Malkov, Alexander Ponomarenko, Andrey Logvinov, and Vladimir Krylov. 2014. Approximate nearest neighbor algorithm based on navigable small world graphs. *Information Systems* 45 (2014), 61–68.

[33] Yu A Malkov and Dmitry A Yashunin. 2016. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *arXiv preprint arXiv:1603.09320* (2016).

[34] James McNames. 2001. A fast nearest-neighbor algorithm based on a principal axis search tree. *IEEE Transactions on Pattern Analysis & Machine Intelligence* 9 (2001), 964–976.

[35] Andrew W Moore. 2000. The anchors hierarchy: Using the triangle inequality to survive high dimensional data. In *Proceedings of the Sixteenth conference on Uncertainty in artificial intelligence*. Morgan Kaufmann Publishers Inc., 397–405.

[36] Marius Muja and David G Lowe. 2009. Flann, fast library for approximate nearest neighbors. In *International Conference on Computer Vision Theory and Applications (VISAPP'09)*, Vol. 3. INSTICC Press.

[37] David Nister and Henrik Stewenius. 2006. Scalable recognition with a vocabulary tree. In *Computer vision and pattern recognition, 2006 IEEE computer society conference on*, Vol. 2. Ieee, 2161–2168.

[38] Michael T Orchard. 1991. A fast nearest-neighbor search algorithm. In *Acoustics, Speech, and Signal Processing, 1991. ICASSP-91., 1991 International Conference on*. IEEE, 2297–2300.

[39] Parikshit Ram, Dongryeol Lee, and Alexander G Gray. 2012. Nearest-neighbor search on a time budget via max-margin trees. In *Proceedings of the 2012 SIAM International Conference on Data Mining*. SIAM, 1011–1022.

[40] Kaushik Sinha. 2014. LSH vs Randomized Partition Trees: Which One to Use for Nearest Neighbor Search?. In *Machine Learning and Applications (ICMLA), 2014 13th International Conference on*. IEEE, 41–46.

[41] Kaushik Sinha and Omid Keivani. 2017. Sparse Randomized Partition Trees for Nearest Neighbor Search. In *Artificial Intelligence and Statistics*. 681–689.

[42] Antonio Torralba, Rob Fergus, and William T Freeman. 2008. 80 million tiny images: A large data set for nonparametric object and scene recognition. *IEEE transactions on pattern analysis and machine intelligence* 30, 11 (2008), 1958–1970.

[43] Santosh S Vempala. 2012. Randomly-oriented kd trees adapt to intrinsic dimension. In *LIPIcs-Leibniz International Proceedings in Informatics*, Vol. 18. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik.

[44] Nakul Verma, Samory Kpotufe, and Sanjoy Dasgupta. 2009. Which spatial partition trees are adaptive to intrinsic dimension?. In *Proceedings of the twenty-fifth conference on uncertainty in artificial intelligence*. AUAI Press, 565–574.

[45] Enrique Vidal. 1994. New formulation and improvements of the nearest-neighbour approximating and eliminating search algorithm (AESA). *Pattern Recognition Letters* 15, 1 (1994), 1–7.

[46] Jun Wang, Wei Liu, Sanjiv Kumar, and Shih-Fu Chang. 2016. Learning to hash for indexing big dataàĂŤa survey. *Proc. IEEE* 104, 1 (2016), 34–57.

[47] Jingdong Wang, Ting Zhang, Nicu Sebe, Heng Tao Shen, et al. 2018. A survey on learning to hash. *IEEE Transactions on Pattern Analysis and Machine Intelligence* 40, 4 (2018), 769–790.

[48] Xiang Wu, Ruiqi Guo, Ananda Theertha Suresh, Sanjiv Kumar, Daniel N Holtmann-Rice, David Simcha, and Felix Yu. 2017. Multiscale quantization for fast similarity search. In *Advances in Neural Information Processing Systems*. 5745–5755.

[49] Yubao Wu, Ruoming Jin, and Xiang Zhang. 2014. Fast and unified local search for random walk based k-nearest-neighbor query in large graphs. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of Data*. ACM, 1139–1150.

[50] Felix Yu, Sanjiv Kumar, Yunchao Gong, and Shih-Fu Chang. 2014. Circulant binary embedding. In *International conference on machine learning*. 946–954.

A CHOICE OF LEAF SIZE n_0

The leaf size n_0, in some sense, balances between a global search and a local search. The whole idea is to quickly identify a local region (which most likely will contain nearest neighbors) and perform exhaustive search there, as opposed to performing a linear scan (global search). Of course making n_0 too small will make the local region uninformative (may be too small and may not contain true nearest neighbors) while making n_0 too big would result in unnecessary exhaustive search within this local region increasing query time. While this problem sounds kind of similar to the same issue that LSH faces (in terms of requiring knowledge of true nearest-neighbor distance or else ending up with disparate candidate set sizes for different queries), typically effect of n_0 on search accuracy is quite benign. Assuming data is generated from an underlying probability distribution which is a "doubling measure" with doubling constant d_0 (that is, probability mass of any ball can be covered by 2^{d_0} balls of half the radius, where d_0 is much much smaller than ambient data dimension d), a constant failure probability of finding all true k nearest neighbors requires $n_0 \sim O\left((d_0 k)^{d_0}\right)$ (see Theorem 9 in [14] for exact dependence).

B DATASET DETAILS

The sizes of the datasets used for the empirical evaluation are presented in Table 3.

Table 3: Data sets for evaluation. The last 4 datasets are from the ANN-Benchmarks.

Data set	$\|R\|$	$\|Q\|$	d
Letter recognition	18000	2000	16
Corel	56615	10000	89
Aerial	265465	10000	60
Tiny images	1000000	10000	384
MNIST	60000	10000	784
Fashion MNIST	60000	10000	784
SIFT	1000000	10000	128
GIST	1000000	1000	960

Dataset details for the scaling experiment. For $d = 128$, we just used SIFT. For $d = 256$, we randomly sampled 256/384 columns in the Tiny Images set. For $d = 512$, we column-subsampled the GIST dataset. For $d = 1024$, we combined SIFT and GIST and then subsampled 1024 out of the 1088 columns. For $d = 2048$, we used 3 million rows from the 80 million tiny images set, and concatenated them to get 1 million rows with 1152 columns, and then concatenated that to the 1024 column set of GIST + SIFT and subsampled 2048 out of the 2176 columns. The query set was created in a similar consistent manner to ensure that there was no overlap in the data between the query and reference sets.

C FURTHER RESULTS ON RECALL VS. PRECISION

The area under the recall-precision curves in Figure 3 is presented in Table 4.

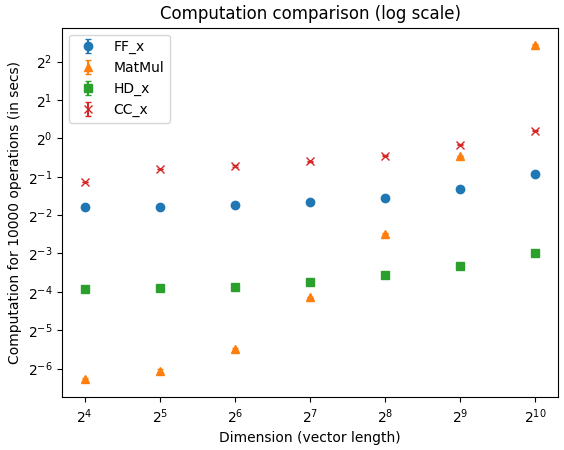


Figure 5: Comparison of implementations of different preconditioning functions listed in Table 5.

D IMPLEMENTATION DETAILS

Here we present the actual runtimes of the different preconditioning operations we have considered in the different methods. Table 5 details the operations we will be focusing on. The goal of this section is to show that the actual run time of the implementations can be significantly different than their theoretical runtimes because of optimizations in the implementation. The results in Figure 5 indicate that the $O(d^2)$ MatMul operation (used by `RR:KDTree`) is the fastest for up to $d \approx 2^7$ (the $O(d \log(n/n_0))$ query time of `RPTree` will only be faster than $O(d^2)$) and only passes the $O(d \log d)$ CC_x operation (used in `RC:KDTree`) for $d > 2^9$. As mentioned in Section 4.2, the vector-vector/vector-matrix multiplication can be really fast in practice. For example, increasing the dimension (horizontal axis) by $4\times$ ($2^4 - 2^6$) increases the run time (vertical axis) by less than $2\times$ (instead of increasing run time by $16\times$). This is one of the reasons for the (relatively) favorable d scaling of `RPTree` in Section 4.2. The $O(d^2)$ scaling of MatMul starts showing up for $d > 2^9$.

In contrast, our un-optimized implementation of our proposed schemes (CC_x for `RC:KDTree` and FF_x for `FF:KDTree`) suffers from various inefficiencies:

- As mentioned in Section 4.2, we use a reference implementation of FFT instead of the optimized FFTW library[8].
- We use the standard `numpy` random permutation (with fixed seed) for the permutation operation in FF_x. This can be significantly improved with a custom C++ implementation.
- Moreover, our operations (CC_x and FF_x in Table 5) switches between low-level (C/C++) and high level (Python) significantly more than the standard vector-vector/vector-matrix operations, which comes with significant overheads as well:
  - Something like Γx (in `RR:KDTree`) and $w^\top x$ (in `RPTree`) can be obtained by a single round trip between the high-level Python layer and the low-level `numpy` implementations.

[8] http://fftw.org

Method	Aerial	Corel	FashMNIST	GIST	LetterReg.	MNIST	SIFT	TinyImages
RPTree	0.040 ± 0.001	0.076 ± 0.004	0.038 ± 0.001	5.9e-4 ± 3e-5	0.139 ± 0.003	0.030 ± 0.000	6.53e-3 ± 12e-5	1.45e-3 ± 5e-5
SpGa:RPT(1/10)	0.040 ± 0.001	0.077 ± 0.001	0.039 ± 0.001	6.0e-4 ± 2e-5	0.135 ± 0.011	0.031 ± 0.001	6.55e-3 ± 11e-5	1.41e-3 ± 3e-5
SpGa:RPT(1/3)	0.041 ± 0.002	0.074 ± 0.002	0.039 ± 0.001	6.0e-4 ± 3e-5	0.141 ± 0.005	0.030 ± 0.000	6.50e-3 ± 8e-5	1.46e-3 ± 4e-5
SpRa:RPT(1/10)	0.041 ± 0.000	0.074 ± 0.001	0.039 ± 0.001	5.9e-4 ± 1e-5	0.138 ± 0.007	0.030 ± 0.001	6.53e-3 ± 9e-5	1.43e-3 ± 5e-5
SpRa:RPT(1/3)	0.042 ± 0.001	0.075 ± 0.003	0.039 ± 0.001	6.0e-4 ± 2e-5	0.139 ± 0.012	0.030 ± 0.001	6.62e-3 ± 10e-5	1.46e-3 ± 4e-5
RR:KDTree	0.041 ± 0.001	0.074 ± 0.002	0.039 ± 0.001	6.1e-4 ± 1e-5	0.144 ± 0.005	0.030 ± 0.001	6.54e-3 ± 8e-5	1.43e-3 ± 1e-5
RC:KDTree	0.042 ± 0.002	0.073 ± 0.003	0.039 ± 0.001	6.0e-4 ± 2e-5	0.137 ± 0.003	0.031 ± 0.001	6.51e-3 ± 10e-5	1.40e-3 ± 4e-5
FF:KDTree	0.039 ± 0.001	0.076 ± 0.003	0.037 ± 0.001	5.4e-4 ± 4e-5	0.134 ± 0.007	0.030 ± 0.000	6.36e-3 ± 15e-5	1.44e-3 ± 3e-5

Table 4: Area under the recall vs. precision curves presented in Figure 3 averaged over 5 runs for k-nearest-neighbor search with $k = 100$ and the leaf size $n_0 = 100$ for each tree for all the methods.

Label	Operation	Complexity	Implementation (in Python)	High-low round trips
HD_x	H_dDx	$O(d \log d)$	`ffht.fht(D * x)`	2
MatMul	Γx	$O(d^2)$	`np.dot(Γ, x)`	1
CC_x	$Dx \circledast \gamma$	$O(d \log d)$	`np.fft.ifft(np.fft.fft(D * x) * np.fft.fft(γ))`	4
FF_x	$H_dG\Pi H_dDx$	$O(d \log d)$	`ffht.fht(G * np.random.permutation(ffht.fht(D * x)))`	5

Table 5: The details of the different preconditioning operations we have considered in Table 1 and implemented for the empirical evaluation. Here $x \in \mathbb{R}^d$ with $d = 2^l$ for some integer $l > 0$. `np` stands for the `numpy` linear algebra package in Python. `ffht` is the FFHT package in the FALCONN-LIB library. The actual computation times of the implementations considered are presented in Figure 5. Note that we cache the value of `np.fft.fft(`γ`)` instead of computing it for every circular convolution.

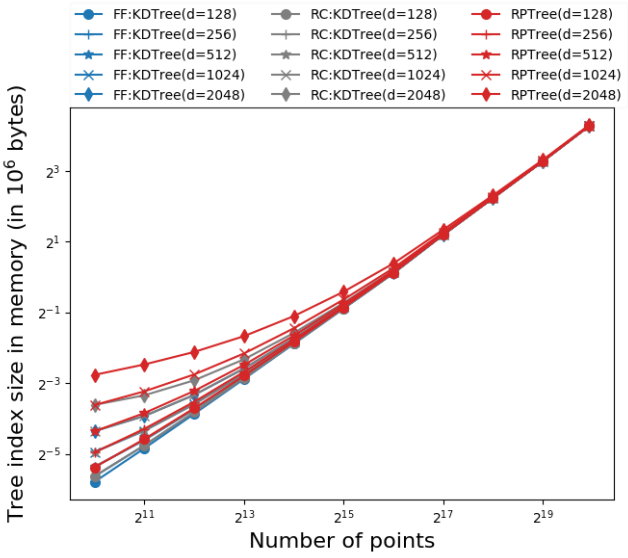


Figure 6: Tree index size scaling of `RPTree`, `RC:KDTree` and `FF:KDTree` with respect to the number of points n and dimensions d.

– The $Dx \circledast \gamma$ operation (CC_x, Equation 4) requires 4 round trips – assuming we precompute $\mathcal{F}(\gamma)$, we need 1 for Dx, 1 for $\mathcal{F}(Dx)$, 1 for $\mathcal{F}(Dx) \circ \mathcal{F}(y)$ and 1 final one for the $\mathcal{F}^{-1}(\cdot)$ operation.
– The $H_dG\Pi H_dDx$ operation (FF_x) requires 5 rounds trips. However, FF_x is more efficient than the CC_x operation mostly because of the AVX optimized FFHT library[9] for the FWHT as opposed to the un-optimized implementation of FFT in `numpy`.

To create a level playing field between all these operations, we believe that all the preconditioning operations need to be completely implemented in a low-level language (such as C/C++) and the preconditioning operation can be computed in a single high-low level round trip as in the case of the $O(d^2)$ matrix multiplication or the $O(d)$ dot product.

E TREE INDEX SIZE SCALING

The tree index size scaling is presented in Figure 6.

[9] https://github.com/FALCONN-LIB/FFHT