Improved nearest neighbor search using auxiliary information and priority functions

Omid Keivani [1] **Kaushik Sinha** [1]

Abstract

Nearest neighbor search using random projection trees has recently been shown to achieve superior performance, in terms of better accuracy while retrieving less number of data points, compared to locality sensitive hashing based methods. However, to achieve acceptable nearest neighbor search accuracy for large scale applications, where number of data points and/or number of features can be very large, it requires users to maintain, store and search through large number of such independent random projection trees, which may be undesirable for many practical applications. To address this issue, in this paper we present different search strategies to improve nearest neighbor search performance of a single random projection tree. Our approach exploits properties of single and multiple random projections, which allows us to store meaningful auxiliary information at internal nodes of a random projection tree as well as to design priority functions to guide the search process that results in improved nearest neighbor search performance. Empirical results on multiple real world datasets show that our proposed method improves the search accuracy of a single tree compared to baseline methods.

1. Introduction

Nearest neighbor search is extensively used as a subroutine for k-nn classifier and many complex graph based methods in wide range of domains such as machine learning, computer vision, pattern recognition and robotics. The basic problem of nearest neighbor search is as follows: given a set of n d-dimensional data points $\mathcal{S} = \{x_1, x_2, \ldots, x_n\} \subset \mathbb{R}^d$ and a query point $q \in \mathbb{R}^d$, one needs to build a data structure using $\mathcal{S}$, so that nearest point (when measured using appropriate distance metric) to q from $\mathcal{S}$ can be found quickly. The naive linear time solution, that scans through each data point $x_i \in \mathcal{S}$, often becomes impractical for large n and d. Towards this end, in recent years there has been a conscious effort towards designing sub-linear time algorithms for solving this problem. Most of these efforts can broadly be classified into two groups, namely, (a) tree based methods ((Bentley, 1975; Uhlmann, 1991; Ciaccia et al., 1997; Katayama & Satoh, 1997; Liu et al., 2004; Beygelzimer et al., 2006; Dasgupta & Sinha, 2013; Sinha, 2015; Sinha & Keivani, 2017; Babenko & Lempitsky, 2017)) and (b) methods based on hashing ((Gionis et al., 1999; Andoni & Indyk, 2008; Datar et al., 2004)). Basic principle for both these approaches is to quickly retrieve a smaller subset $\mathcal{S}' \subset \mathcal{S}$ and perform linear scan within $\mathcal{S}'$ to answer a nearest neighbor search query. Locality sensitive hashing (LSH) (Gionis et al., 1999; Andoni & Indyk, 2008; Datar et al., 2004) is a representative of these hashing based methods that provides a sub-linear time solution for *approximate* nearest neighbor search with some theoretical guarantee. Unfortunately, many recent studies (Sinha, 2014; Muja & Lowe, 2009; Liu et al., 2004) have reported that nearest neighbor search using LSH is outperformed by tree based methods.

Nearest neighbor search methods that use space partition trees hierarchically partition the search space into a large number of regions corresponding to tree leaves, where each leaf contains only a small subset of the database points. Most of these partition trees thus constructed are binary in nature. Using such partition tree, a nearest neighbor query is answered by adopting either (a) a *defeatist* search strategy or (b) a guided depth first search (DFS) strategy. In a defeatist search strategy, a query is routed from the root node of the tree to a leaf node, by applying a simple test function at each internal node along the route that dictates whether to follow left or right branch at that node. Once at leaf node, a linear scan is performed within the points lying in the leaf node to answer a nearest neighbor search query. If the partition tree is (approximately) balanced, depth of such a tree is $O(\log n)$ and therefore a query can reach a leaf node fairly quickly and subsequently performing a liner scan within a constant number of data points lying in that leaf node, nearest neighbor query can be answered in sub-linear (in n) time. Unfortunately, defeatist search strategy often fails to find exact nearest neighbor of the query because the leaf

[1]Department of Electrical Engineering & Computer Science, Wichita State University, KS, USA. Correspondence to: Kaushik Sinha <kaushik.sinha@wichita.edu>.

Proceedings of the 35^{th} International Conference on Machine Learning, Stockholm, Sweden, PMLR 80, 2018.

node where the query is routed to may not contain its nearest neighbor. A remedy for this is to construct a forest of such trees and defeatist search strategy is performed to each tree in this forest. This improves search performance but comes at the cost of memory overhead of storing the entire forest, which for many large scale applications become unrealistic. In comparison, a guided DFS search is often applied in metric trees (Uhlmann, 1991; Ciaccia et al., 1997; Omohundro, 1990), where a single space partition tree is used but data points are retrieved from multiple leaf nodes by following a depth first search strategy. Typically, at each internal node, one of the two child nodes is visited first, if it is more likely (based on some heuristic) to contain the nearest neighbor, before visiting the other child node. Moreover, additional heuristics are often applied to prune out nodes (or subtrees) from visiting, if they are unlikely to contain nearest neighbors. While guided DFS search strategy works reasonably well for low dimensional datasets (data dimension is less than 30) (Liu et al., 2004), their performance degrades with increasing data dimension and for high dimensional dataset their performance often becomes no better than linear scan of the entire database. In practice, often a budgeted version of guided DFS search is used, where leaf nodes are still visited in a guided DFS manner, but only until data points from a predefined fixed number of leaves are retrieved. However, problem with such approach is that if a mistake is made at the top level (near the root node), that is, say left subtree at this level is not visited due to budget constraint and this subtree contains the true nearest neighbor, then rest of the search becomes useless. To avoid some of the shortcomings of the above two search strategies, spill tree and virtual spill tree (Liu et al., 2004) have also been proposed. A spill tree still uses a defeatist search strategy, however, while constructing a tree, each time an internal node is split, overlap is allowed between its two children nodes. With this modification, search performance typically improves with amount of allowed overlap. However, such improvement comes at the cost of super-linear (as opposed to linear) space complexity for a single tree, which may be unacceptable for large scale applications. In comparison, in a virtual spill tree no overlap is allowed(linear space complexity), but the effect of spill tree is mimicked by visiting multiple tree leaf nodes. In particular, at each internal nodes, if the query point upon projection lies close to the split point (and thus within a "virtual" overlapped region) both subtrees rooted at this node are visited. While search performance typically improves with allowed virtual overlap, enforcing user defined control on how many leaf nodes to retrieve becomes problematic and often leaf nodes containing useless information are retrieved.

To address these issues, in this paper we propose various strategies to improve nearest neighbor search performance of a single space partition tree by using auxiliary information and priority functions. We use properties of random projection to choose auxiliary information. We use random projection tree (RPT) as our base tree construct due to its proven theoretical guarantees in solving *exact* nearest neighbor search problem (Dasgupta & Sinha, 2013; 2015) and its superior performance over LSH based methods (Sinha, 2014). Our proposed auxiliary information based prioritized guided search improves nearest search performance of a single RPT significantly. We make the following contributions in this paper.

- Using properties of random projection, we show how to store auxiliary information of additional space complexity $\tilde{O}(n+d)$ at the internal nodes of an RPT to improve nearest neighbor search performance using defeatist search strategy.
- We propose two priority functions to retrieve data points from multiple leaf nodes of a single RPT and thus extend the usability of RPT beyond defeatist search. Such priority functions can be used to perform nearest neighbor search under a computational budget, where to goal is to retrieve data points from most informative leaf nodes of a single RPT specified by the computational budget.
- We combine the above two approaches to present an effective methodology to improve nearest neighbor search performance of a single RPT and perform extensive experiments on six real world datasets to demonstrate the effectiveness of our proposed method.

Rest of the paper is organized as follows. We discuss related work in section 2 and present our proposed method in section 3. We present our experimental evaluations in section 4 and conclude in section 5.

2. Related work

In this section we review related work. A space partition tree hierarchically partitions the input data space into non-overlapping subsets in the form of a binary tree, where root node corresponds to the original dataset and the leaf nodes correspond to non-overlapping subsets. At each internal node of such a tree, all points lying in that node are split into two disjoint subsets (left and right child nodes) based on stored information at that node. Once a query comes, it is routed from the root node to a leaf node, following the stored information at internal nodes to guide its routing, and linear scan is performed within the data-points lying in that leaf node. Space partition trees differ from one another in how they perform split at the internal nodes. In metric trees (Ciaccia et al., 1997; Omohundro, 1990; Uhlmann, 1991) two pivot points are chosen at each internal node and all data points corresponding to that internal node are projected on the line joining these two pivot points and are partitioned based on the median of the projected points. In KD tree (Bentley, 1975), axis aligned partitions are allowed where at

each internal node, the space is partitioned based on a particular coordinate i and a threshold b. In PCA tree (Sproull, 1991), projection direction at each internal node is chosen to be the largest principal component direction. In random projection tree, projection directions are chosen randomly from a unit sphere (Dasgupta & Sinha, 2015). In product split tree (Babenko & Lempitsky, 2017) product quantization idea is used to make an effective codebook to choose projection direction. In max-margin tree (Ram et al., 2012), a hyperplane that separates data points of an internal node by a large margin is used to perform the split. In addition, spill tree (Liu et al., 2004; Dasgupta & Sinha, 2013) is a variant of space partition tree where children of a node can "spill over" on to one another and share common data points. A different class of tree based algorithms performs search in a coarse-to-fine manner, including navigating nets (Krauthgamer & Leel, 2004), cover trees (Beygelzimer et al., 2006) and rank cover trees (Houle & Nett, 2015). They maintain sets of sub-sampled data points at different levels of granularity and descend through the hierarchy of neighborhoods of decreasing radii around the query.

Another class of algorithms called Locality Sensitive Hashing (LSH) based methods (Gionis et al., 1999; Datar et al., 2004; Andoni & Indyk, 2008) partitions the space into regular cells, whose shapes are implicitly defined by the choice of the hash function. Roughly speaking, a locality sensitive hashing function has the property that if two points are "close", then they hash to same bucket with "high" probability, whereas if they are "far apart", then they hash to same bucket with "low" probability. Once a query comes, same hash functions are applied to identify a hash bucket and a linear scan is performed among the data points lying in that bucket. While LSH based methods have been quite popular in practical applications for solving approximate nearest neighbor methods, they are often outperformed by tree based methods (Sinha, 2014; Muja & Lowe, 2009; Liu et al., 2004). A different class of algorithms that avoids partitioning the input space altogether has recently been proposed (Li & Malik, 2016; 2017). Instead of partitioning the space into discrete cells, these methods construct continuous indices, each of which imposes an ordering on data points such that closeness in position serves as an approximate indicator of proximity in the vector space.

3. Proposed method

In this section we present various strategies to improve nearest neighr search performance using a single tree. As mentioned earlier, we use RPT a base tree construct for nearest neighbor search. Each internal node of an RPT stores a random projection direction and a scalar split point. Once query reaches an internal node, it is first projected onto the random projection direction stored at this internal node, and depending on whether the resulting 1-d projection lies to the left or right side of the stored split point, the corresponding branch (left or right) is taken. Space complexity of such an RPT is $O(nd)$. It is shown in (Dasgupta & Sinha, 2015) that the probability that an RPT fails to find exact nearest neighbors of a query can be restricted to an arbitrary constant, if the following sufficient conditions are met: (a) the leaf nodes contain number of data points exponential in the intrinsic dimension of the dataset and (b) data points are drawn from a underlying probability distribution that satisfies *doubling measure* condition. While the intrinsic dimension of real world datasets is often much smaller than its ambient dimension, accurately estimating intrinsic data dimension is often an extremely difficult task. In addition, due to its exponential dependence, unless intrinsic data dimension is really low, sufficient leaf node size (to ensure small failure probability) can be very high yielding a large number of retrieve points. Moreover, the doubling measure condition may not be satisfied for real world datasets. As a result, in practical applications, tree based nearest neighbor search using defeatist strategy often results in high failure probability. To compensate for this, often a forest of independent RPTs are used to increase overall success probability, requiring $O(nd)$ space complexity overhead for each additional RPT in the forest. In the following, we present two approaches to boost success probability of nearest neighbor search of an individual RPT and then present a combination of both approaches.

3.1. Defeatist search with auxiliary information

In our first approach, we introduce a modified defeatist search strategy, where, at each internal node we store auxiliary information to compensate for the fact that while routing a query from root node to a leaf node, only one of the two branches is chosen at each internal node. The stored auxiliary information at any internal node aims to compensate for the unvisited subtree rooted at this node by identifying a small set of candidate nearest neighbors that lie in this unvisited subtree. Note that this small set of candidate nearest neighbor points otherwise would not be considered had we adopted the traditional defeatist search strategy. Now, to answer a nearest neighbor query, a linear scan is performed among the points lying in the leaf node (where the query is routed to) and union of the set of identified candidate nearest neighbor points at each internal node along the query routing path as shown in Figure 1. A natural

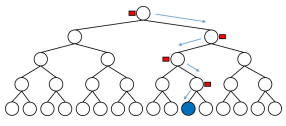

Figure 1. Defeatist query processing using auxiliary information. Blue node is where a query is routed to. Red rectangles indicates auxiliary information stored on the opposite side of the split point from which candidate nearest neighbors for the unvisited subtree are selected.

question that arises is, what kind of auxiliary information can we store to achieve this? On one hand, we would like to ensure that auxiliary information does not increase space complexity of the data structure significantly, while on the other hand, we would like the candidate nearest neighbors identified at each internal node along the query routing path to be query dependent (so that not the same candidate nearest neighbors are used for every query), and therefore, this additional query dependent computation (for each query) needs to be performed quickly without significantly increasing overall query processing time. We argue next that we

Algorithm 1 RPT construction with auxiliary information

Input : data $\mathcal{S} = \{x_1, \ldots, x_n\} \subset \mathbb{R}^d$, maximum number of data points in leaf node n_0, auxiliary index size c, m independent random vectors $\{V_1, \ldots, V_m\}$ sampled uniformly from $\mathcal{S}^{d-1}$.
Output : tree data structure
function `MakeTree`(S, n_0)

if $|S| \leq n_0$ **then**
  return leaf containing S
else
  Pick U uniformly at random from $\mathcal{S}^{d-1}$
  Let v be the median of projection of S onto U
  Set `ail` be the set of indices of c points in S so that upon projection onto U, they are the c closest points to v from the left.
  Set `air` be the set of indices of c points in S so that upon projection onto U, they are the c closest points to v from the right.
  Construct a $c \times m$ matrix $\texttt{L}_{\texttt{cnn}}$ whose i^{th} row is the vector $(V_1^\top x_{\texttt{ail}(i)}, V_2^\top x_{\texttt{ail}(i)}, \ldots, V_m^\top x_{\texttt{ail}(i)})$.
  Construct a $c \times m$ matrix $\texttt{R}_{\texttt{cnn}}$ whose i^{th} row is the vector $(V_1^\top x_{\texttt{air}(i)}, V_2^\top x_{\texttt{air}(i)}, \ldots, V_m^\top x_{\texttt{air}(i)})$.
  `Rule`$(x) = (x^\top U \leq v)$
  `LSTree` = `MakeTree`$(\{x \in S :$ `Rule`(x) = `true`$\}, n_0)$
  `RSTree` = `MakeTree`$(\{x \in S :$ `Rule`(x) = `false`$\}, n_0)$
  return (`Rule`, `LSTree, RSTree`)
end if

can exploit properties of random projection to store auxiliary information that helps us achieves the above goals. Note that if we have an ordering of the distances of points from $\mathcal{S}$ to query q, then any one dimensional random projection has the property that upon projection, this ordering (of projected points) is perturbed locally near projected q but is preserved globally with high probability as shown below.

Theorem 1. *Pick any query $q \in \mathbb{R}^d$ and set of database points $\mathcal{S} = \{x_1, \ldots, x_n\} \subset \mathbb{R}^d$ and let $x_{(1)}, x_{(2)}, \ldots$ denote the re-ordering of the points by increasing distance from q, so that $x_{(1)}$ is nearest neighbor of q in $\mathcal{S}$. Consider any internal node of RPT that contains a subset $S \subset \mathcal{S}$ containing $x_{(1)}$ and q. If q and points from S are projected onto a direction U chosen at random from a unit sphere, then for any $1 \leq k < |S|$, the probability that there exists a subset of k points from S that are all not more than $|U^\top(q - x_{(1)})|$ distance away from $U^\top q$ upon projection is at most $\frac{1}{k}\sum_{i=1}^{|S|} \frac{\|q-x_{(1)}\|_2}{\|q-x_{(i)}\|_2}$.*

Roughly speaking, this means that upon projection, those points that are far away from projected q are unlikely to be close to q in original high dimension (thus unlikely to be its nearest neighbors). On the other hand, the projected points that are close to projected q, may or may not be its true nearest neighbors in original high dimension. In other words, with high probability, true nearest neighbor of any query will remain close to the query even after projection, since distance between two points does not increase upon projection, however points which were far away from query in original high dimension may come closer to the query upon projection. Therefore, among the points which are close to the projected q will definitely contain q's true nearest neighbor but will also contain points which are not q's true nearest neighbors (we call such points *nearest neighbor false positives*). We utilize this important property to guide us choose auxiliary information as follows. At any internal node of an RPT, suppose the projected q lies on left side of the split point (so that left child node falls on the query routing path). From the above discussion, it is clear that if q's nearest neighbor lie on the right subtree rooted at this node, then their 1-d projections will be very close to the split point on the right side. Therefore, to identify q's true nearest neighbor, one possibility is to store actual c high dimensional points (where c is some pre-defined fixed number) which are closest c points (upon projection) to the right side of the split point as auxiliary information for this node. There are two potential problems with this approach. First, additional space complexity due to such auxiliary information is $O(nd)$ which may be prohibitive for large scale applications. Second, because of 1-d random projection property (local ordering perturbation), we may have nearest neighbor false positives within these c points. To prune out these nearest neighbor false positives for each query, if we attempt to compute actual distance from q to these c points in original high dimensions and keep only closest points based on actual distance as candidate nearest neighbors, this extra computation, will increase query time for large d. To alleviate this, we rely on celebrated Johnson Lindenstrauss lemma (Johnson & Lindenstrauss, 1984) which says that if we use $m = O\left(\frac{\log(c+1)}{\epsilon^2}\right)$ random projections then pairwise distance between c points and q are preserved in $\mathbb{R}^m$ within a multiplicative factor of $(1 \pm \epsilon)$ of the original high dimensional distance in $\mathbb{R}^d$. Equipped with this result, at each internal node of an RPT we store two matrices of size $c \times m$ (one for left subtree, one for right) as auxiliary information, that is for each of these c points, we store their m dimensional representation as a proxy for their original high dimensional representation. For all our experiments, we set m to be 20. Algorithm 1 provides details of an RPT construction with this modification and the following theorem shows that additional space complexity due to auxiliary information is is $\tilde{O}(n + d)$, where we hide

the $\log\log(n/n_0)$ factor.

Theorem 2. *Consider a modified version of RPT where, at each internal node of the RPT, auxiliary information is stored in the form of two matrices, each of size $c \times m$ (one for left subtree, one for right). If we choose $c \leq 10n_0$, additional space complexity of this modified version of RPT due to auxiliary information is $\tilde{O}(n+d)$.*

Here n_0 is user defined maximum leaf node size of an RPT and for all practical purpose $\log\log(n/n_0)$ can be treated as a constant. Therefore, additional space complexity is merely $O(n+d)$. While processing a query q using this strategy, to prune out the nearest neighbor false positives at any internal node of an RPT, we project q onto m random projection directions to have q's m dimensional representation $\tilde{q}$. At each internal node along the query routing path, we use $\tilde{q}$ to add c', where $c' < c$ and c' is a pre-defined fixed number, candidate nearest neighbors to the set of retrieved points (by computing distances from $\tilde{q}$ to c data points in m dimensional representation and keeping the c' closest ones) to compensate for the unvisited subtree rooted at this node. We use $c' = 10$ for all our experiments. Details of query processing using this approach is presented in the supplementary material. Note that due to this modification, time to reach leaf node increases from $O(d\log(n/n_0))$ to $O\left((d + cm + c\log(c))\log(n/n_0) + md\right)$ and the number of retrieved points that require a linear scan increases from n_0 to $(n_0 + c'\log(n/n_0))$. We also note that using spill tree for defeatist search yields a super-linear space complexity as shown by the following theorem (a similar result was shown in (Dasgupta & Sinha, 2013)).

Theorem 3. *Space complexity for a spill tree with α percentile overlap, where $\alpha \in (0, 1)$, on each side of the median at each internal node is $O\left(dn^{\left(\frac{1}{1-\log_2(1+2\alpha)}\right)}\right)$.*

3.2. Guided prioritized search

In our second approach, we seek to retrieve data points from multiple leaf nodes, as opposed to a single leaf node, of an RPT as candidate nearest neighbors. We can specify a constant number of leaf nodes (say, l) a-priori from which points will be retrieved and a linear scan will be performed. In order to identify l such appropriate leaf nodes, we present a guided search strategy based on priority functions. The general strategy is as follows. First, the query is routed to the appropriate leaf node as usual. Next, we compute a priority score for all the internal nodes along the query routing path. These priority scores along with their node locations are stored in a priority queue sorted by priority scores in decreasing order. Now we choose the node with highest priority score, remove it from priority queue, and route the query from this node to its child node that is not visited earlier. Once reached at this child node, standard query routing is followed to reach to a different leaf node. Next, priority scores are computed for all internal nodes along this new query routing path and are inserted to the priority queue. This process is repeated until l different leaf nodes are visited (see Figure 2). This search process is guided by the current highest priority score where high priority score of an internal node indicates that there is a high likelihood that nearest neighbors of the query lies in unexplored subtree rooted at this node and must be visited to improve nearest neighbor search performance. In this paper, we use local perturbation property of 1-d random projection to define a priority score. At any internal node (with stored projection direction U and split point v) of an RPT, we define priority score at this node to be

$$f_{\texttt{pr1}}(U, v, q) = \frac{1}{|v - U^\top q|} \tag{1}$$

Here the intuition is that if the projected query lies very close to the split point and since distance ordering is perturbed locally (Theorem 1), there is a very good chance that true nearest neighbor of the query, upon projection is located on the other side of the split point. Therefore, this node should be assigned a high priority score.

Just because a query upon projection lies close to the split point at any internal node of an RPT does not necessarily mean that nearest neighbor of the query lies on the opposite side of the split point (unvisited subtree rooted at this node). However, at any internal node of an RPT, if the minimum distance (original distance in $\mathbb{R}^d$) between the query and the set of points lying on the same side of the split point as the query is larger than the minimum distance between the query and the set of points lying on the opposite side of the split point, then visiting the unvisited child node rooted at this node will make more sense. We use this idea to design our next priority function. Since computing actual distance is in $\mathbb{R}^d$ will increase query time, we use auxiliary information for this purpose. Note that at each internal node, on each side of the split point, we store m dimensional representation of c closest points from the split point upon 1-d projection as auxiliary information. For any query, once at any internal node of an RPT, using m dimensional representation of q, we first compute the closest point (in $\mathbb{R}^m$) among these c points that lie on the same side of the split point as the query upon 1-d projection and call it $d^{\texttt{same}}_{\texttt{min}}$. In a similar manner we compute $d^{\texttt{opp}}_{\texttt{min}}$. Due to JL lemma, $d^{\texttt{same}}_{\texttt{min}}$ and $d^{\texttt{opp}}_{\texttt{min}}$ are good proxy for original minimum distance between and q and those c points in $\mathbb{R}^d$ on both sides of the split point. Ideally, if $d^{\texttt{opp}}_{\texttt{min}} \leq d^{\texttt{same}}_{\texttt{min}}$, the priority score should increase and vice versa because one of the c points on the unexplored side is closer to the query compared to the c points on the same side of the query. To take this into account, we propose a new priority function defined as,

$$f_{\texttt{pr2}}(U, v, q, d^{\texttt{opp}}_{\texttt{min}}, d^{\texttt{same}}_{\texttt{min}}) = \frac{1}{|v - U^\top q|} \cdot \left(\frac{d^{\texttt{same}}_{\texttt{min}}}{d^{\texttt{opp}}_{\texttt{min}}}\right) \tag{2}$$

Note that at each internal node, while the first priority function can be computed in $O(1)$ time, the second priority

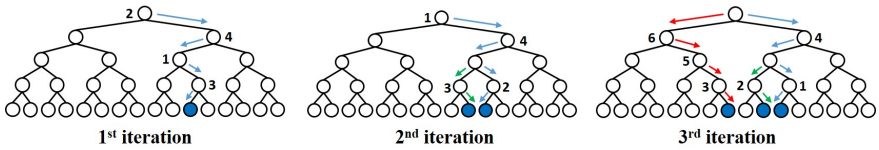

Figure 2. Query processing for three iterations (visiting three different leaf nodes) using priority function. The three retrieved leaf nodes are colored blue. At each internal node an integer represents the priority score ordering. Lower the value, higher the priority. After visiting each new leaf node ordering of priority scores is updated. Note that if a mistake is made at the root level and true nearest neighbor lie on the left subtree rooted at root node, with three iterations, DFS will never visit this left subtree and fails to find true nearest neighbor.

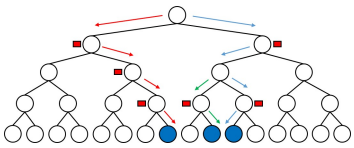

Figure 3. Query processing using combined approach. Blue circles indicate retrieved leaf nodes. Red rectangles indicate auxiliary information stored on the opposite side of the split point at each internal node, from which candidate nearest neighbors are selected, along query processing path for which only one subtree rooted at that node is explored. This figure should be interpreted as a result of applying ideas from section 3.1 to Figure 2 after 3rd iteration.

function takes time $O(mc)$ plus an additional $O(md)$ time to compute m dimensional representation $\tilde{q}$. We note that, while a priority function similar to $f_{\texttt{pr1}}$ has been proposed recently (Babenko & Lempitsky, 2017), $f_{\texttt{pr2}}$ is new. In all our experiments, prioritized search based on $f_{\texttt{pr2}}$ outperforms $f_{\texttt{pr1}}$. Algorithm for query processing using priority function is presented in the supplementary material.

3.3. Combined approach

Integrating ideas from section 3.1 and 3.2, we present a combined strategy for effective nearest neighbor search using RPT, where data points are retrieved from multiple informative leaf nodes based on priority function (as described in section 3.2) and also from from internal nodes along these query processing routes as described in section 3.1. Note that while accessing multiple leaf nodes using priority function, if at any internal node of an RPT both its subtrees are visited then there is no need to use auxiliary information at that node. This combined approach is shown in Figure 3 and algorithm for query processing is presented in the supplementary material. Due to space limitation, proofs of all theorems are also presented in the supplementary material.

4. Empirical Evaluation

In this section we present empirical evaluations of our proposed method and compare them with baseline methods[1]. We use six real world datasets of varied size and dimension as shown in Table 1. Among these, MNIST, SIFT and SVHN are image datasets, JESTER is a recommender systems dataset and 20Newsgroup and SIAM07 are text mining datasets. Both SIAM07 and 20Newsgroup have very large data dimensions (d), while SIFT has very large number of instances (n). For each dataset, we randomly choose instances (as shown in 2nd column of Table 1) to build appropriate data structure and randomly choose queries (as shown in 3rd column of Table 1) to report nearest neighbor search performance of various methods. We design three different experiments. For the first two experiments, we present our results for 1-NN search problem, whereas for the third experiment, we present results for 10-NN search problem. We use accuracy to measure the effectiveness of various methods. For 1-NN search, accuracy is simply calculated as the fraction of the query points for which its true nearest neighbor is within the retrieved set of points returned by respective methods. For 10-NN search problem, accuracy of a query q is defined as $|\mathcal{A}_q^T(k) \cap \mathcal{A}_q^R(k)|/|\mathcal{A}_q^T(k)|$, where $\mathcal{A}_q^T(k)$ is the set of true k nearest neighbors and $\mathcal{A}_q^R(k)$ be the set of k-nearest neighbors reported by a nearest neighbor search algorithm. We report accuracy averaged over number of queries listed in third column of Table 1. NN-search accuracy is plotted against the ratio of the number of retrieved points (for which we need to perform a linear scan) to the total number instances. For all our experiments, we set n_0 to be 100 and use median split while constructing an RPT. We set $m = 20, c = 500$ and $c' = 10$ for all our experiments[2]

Dataset	# instances	#queries	#dimensions
MNIST	65000	5000	768
SIFT	400000	10000	128
SVHN	68257	5000	3072
JESTER	68421	5000	101
20Newsgroup	15846	3000	26214
SIAM07	23596	5000	30438

Table 1. Datasets details

4.1. Experiment 1

In the first experiment, we empirically show how auxiliary information stored at internal nodes of an RPT improves 1-NN search accuracy using defeatist search. We compare our proposed method (which we call RPT1) with vanilla RPT without any stored information (which we call Normal RPT) and spill tree with different percentile of overlap[3] (α).

[1]We do not compare with LSH, since earlier study demonstrated that that RPT performs better than LSH (Sinha, 2014).

[2]We experimentally observed that across datasets the ratio of accuracy to actual query time as a function of c increases first and after on and around c=500 it starts to decrease indicating that beyond c=500 increase in accuracy comes at the cost of slower query time and may not worth it.

[3]Note that, when we split an internal node of a spill tree with overlap α on both sides of the median, left and right child nodes

As can be seen from Table 2, RPT1 outperforms all other methods by a significant margin. With increasing α, search accuracy of spill tree increases but so does space complexity. For example, when $\alpha = 0.1$, space complexity of spill tree is $O\left(dn^{1.36}\right)$, which is super-linear in n (see Theorem 3).

Dataset	RPT1	Normal RPT	ST$_{\alpha=0.025}$	ST$_{\alpha=0.05}$	ST$_{\alpha=0.1}$
MNIST	**44%**	12%	12%	16%	21%
SIFT	**47%**	23%	26%	30%	38%
SVHN	**39%**	8%	8%	12%	15%
JESTER	**48%**	13%	13%	19%	23%
20Newsgroup	**33%**	7%	9%	9%	13%
SIAM07	**12%**	5%	5%	6%	7%

Table 2. Comparison of 1-NN search accuracy using defeatist search strategy with auxiliary information with baseline methods.

4.2. Experiment 2

In the second experiment we empirically evaluate how two priority functions $f_{\texttt{pr1}}$ and $f_{\texttt{pr2}}$ help in improving guided 1-NN search accuracy by retrieving multiple leaf nodes of a single RPT. A natural competitor of our approach is depth first search (DFS) strategy and virtual spill tree. Note that a virtual spill tree is same as vanilla RPT, except at each internal node, in addition to a random projection direction U and a split point v, two points corresponding to $(\frac{1}{2} - \alpha)$ percentile point upon projection onto U (call it l) and $(\frac{1}{2}+\alpha)$ percentile point upon projection onto U (call it r) are stored. While processing a query q at any internal node, if $U^\top q \in [l, r]$ then both left and right child nodes are visited otherwise, similar to defeatist query processing in RPT, a single child node is visited. Empirical comparison of these four methods are provided in Table 3. In case of $f_{\texttt{pr1}}$, $f_{\texttt{pr2}}$ and DFS, $iter$ is used to indicate how many distinct leaf nodes are accessed, while α is used to indicate virtual spill amount in case of a virtual spill tree. As can be seen from Table 3, 1-NN search accuracy of all methods improve with increasing $iter$ and α. Observe that $f_{\texttt{pr1}}$ outperforms both DFS and Virtual spill tree. Moreover, $f_{\texttt{pr2}}$ always performs better than $f_{\texttt{pr1}}$. One observation that we make from Table 3 is that, for $f_{\texttt{pr1}}$ or $f_{\texttt{pr2}}$, as we increase number of iterations, initially accuracy increases at a much faster rate but with increasing iterations, it slows down. This indicates that later iterations are not as useful as initial iterations. One of the possible reason for this could be that in a single tree, amount of randomness is limited. Therefore, to increase accuracy further, we possibly need to use multiple trees and use fewer number of iterations per tree. But note that using multiple trees increases space complexity. A natural question that arises is what is the right trade-off? We explore this in the next experiment.

MNIST					
$(iter, \alpha)$	**(2,0.025)**	**(5,0.05)**	**(10,0.075)**	**(15,0.1)**	**(20,0.15)**
$f_{\texttt{pr2}}$	19%	33%	47%	55%	61%
$f_{\texttt{pr1}}$	19%	32%	44%	51%	56%
DFS	15%	22%	28%	30%	34%
Virtual	15%	20%	25%	30%	42%
SIFT					
$(iter, \alpha)$	**(2,0.025)**	**(5,0.05)**	**(10,0.075)**	**(15,0.1)**	**(20,0.15)**
$f_{\texttt{pr2}}$	34%	47%	57%	61%	65%
$f_{\texttt{pr1}}$	32%	43%	52%	58%	61%
DFS	27%	32%	36%	37%	40%
Virtual	29%	35%	41%	47%	58%
SVHN					
$(iter, \alpha)$	**(2,0.025)**	**(5,0.05)**	**(10,0.075)**	**(15,0.1)**	**(20,0.15)**
$f_{\texttt{pr2}}$	12%	23%	34%	42%	48%
$f_{\texttt{pr1}}$	13%	24%	34%	41%	47%
DFS	10%	16%	22%	25%	29%
Virtual	10%	14%	18%	21%	34%
JESTER					
$(iter, \alpha)$	**(2,0.025)**	**(5,0.05)**	**(10,0.075)**	**(15,0.1)**	**(20,0.15)**
$f_{\texttt{pr2}}$	21%	34%	46%	54%	60%
$f_{\texttt{pr1}}$	20%	32%	43%	50%	55%
DFS	16%	22%	28%	30%	35%
Virtual	17%	22%	26%	31%	41%
20Newsgroup					
$(iter, \alpha)$	**(2,0.025)**	**(5,0.05)**	**(10,0.075)**	**(15,0.1)**	**(20,0.15)**
$f_{\texttt{pr2}}$	11%	20%	29%	34%	40%
$f_{\texttt{pr1}}$	11%	19%	28%	33%	36%
DFS	9%	14%	18%	21%	26%
Virtual	9%	11%	13%	15%	22%
SIAM07					
$(iter, \alpha)$	**(2,0.025)**	**(5,0.05)**	**(10,0.075)**	**(15,0.1)**	**(20,0.15)**
$f_{\texttt{pr2}}$	8%	14%	19%	22%	25%
$f_{\texttt{pr1}}$	7%	12%	17%	21%	24%
DFS	6%	9%	13%	15%	18%
Virtual	6%	7%	9%	11%	14%

Table 3. Comparisons of 1-NN accuracy of prioritized search with baseline methods.

4.3. Experiment 3

In this experiment, we aim to empirically find a trade-off between number of trees and number of iterations per tree. All results presented in this section are for 10-NN search problem. Since the previous two experiments demonstrate that both auxiliary information and guided prioritized search help in improving NN search accuracy, in this section we use a combination of both methods as described in section 3.3, where we use $f_{\texttt{pr2}}$ as our priority function. To address the issue of number of trees vs number of iterations per tree, we design our experiment as follows. We first consider 10-NN search problem with budgeted query, where, we are allowed to access only 20 leaf nodes using combined approach. This can be done in multiple ways, such as, 20 iterations in a single tree, 10 iterations each in two trees, etc. While using three trees, number of iterations per tree will be roughly 7. We call this search strategy 'Multi-Combined', as we are using multiple trees and using combined search strategy in each tree. The results are listed in Table 4.

As can be seen from Table 4, increasing number of trees increases search accuracy (of course, at the cost of additional space complexity). The biggest increasing in accuracy occurs when we use two trees instead of one. This happens not only because we are introducing additional randomness by adding an extra tree, but also because we are reducing later iterations in each tree which we already observed are not very useful in increasing accuracy. We see from Table 4 that beyond three trees, accuracy increases very slowly, and due to additional space complexity overhead, adding more than three trees is probably not worth it. Equipped with this observation, next we compare how does normal RPT fare with Multi-Combined method and combined method, if we are allowed to create multiple normal RPTs. To ensure a fair comparison we set the number of iterations (for combined and multi-combined method) equal to number of trees in Normal RPT, which we select from the set $\{2, 5, 10, 15, 20\}$. In this experiment, we choose to use 3 trees for multi-combined method, although this depends on the user and how much space she wants to use.

represent data points corresponding to 0 to $(\frac{1}{2}+\alpha)*100$ percentile and $(\frac{1}{2} - \alpha) * 100$ to 100 percentile upon projection.

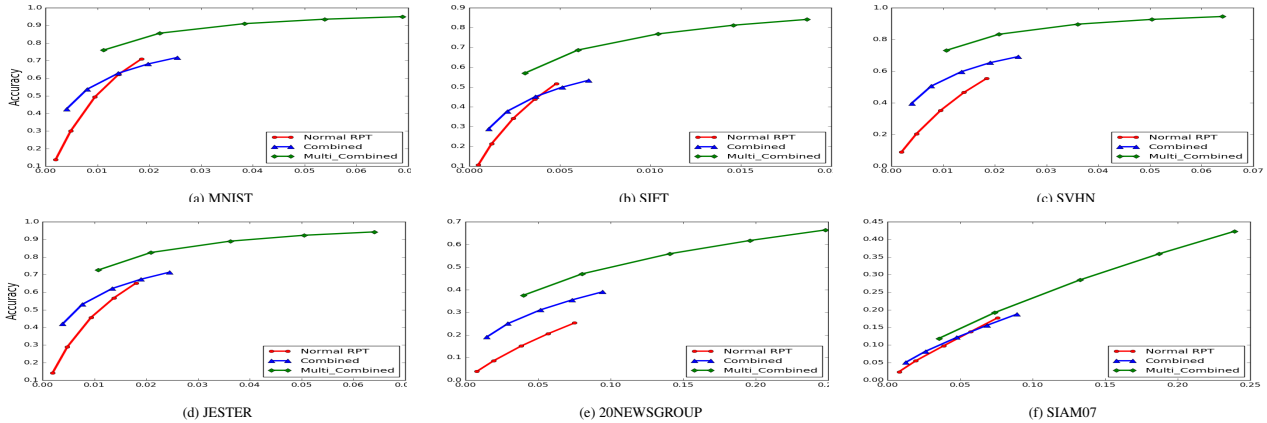

Figure 4. 10-NN accuracy for six datasets. The x-axis represents ratio of # of retrieved points to the total number of instances. The markers from left to right corresponding to 2, 5, 10, 15 and 20 iterations (for combined and Multi-Combined method) / trees (for normal RPT).

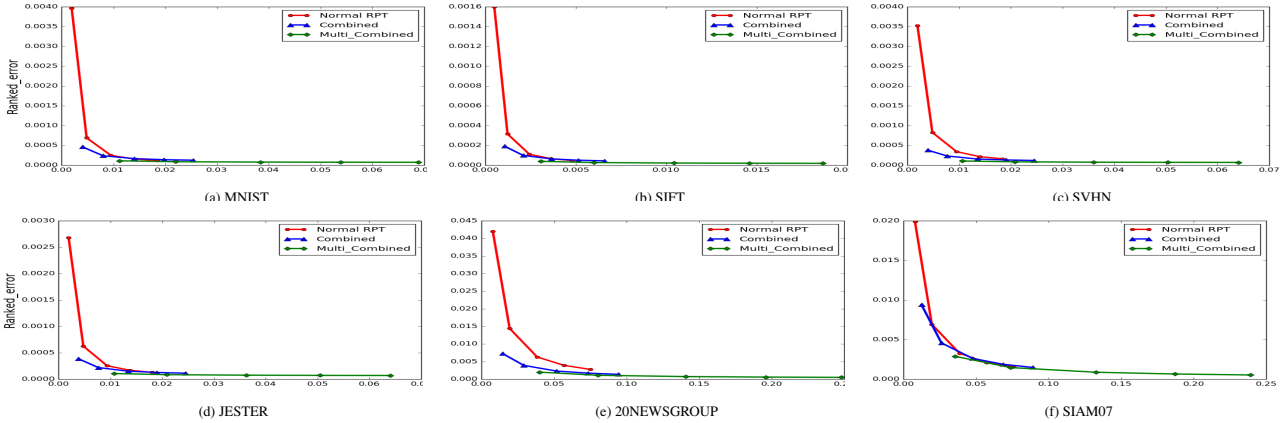

Figure 5. Rank error for six datasets. The x-axis represents ratio of # of retrieved points to the total number of instances. The markers from left to right corresponding to 2, 5, 10, 15 and 20 iterations (for combined and Multi-Combined method) / trees (for normal RPT).

Note that each additional normal RPT requires $O(nd)$ space, while additional space requirement for each Multi-combined tree is $\tilde{O}(n+d)$. We present our results in Figure 4. Our base line is normal RPT, where we use multiple {2,5,10,15,20} RPTs accessing one leaf node per tree using defeatist search (red curve). We compare it against our proposed combined approach (one tree) but using different iterations {2,5,10,15,20} (blue curve). Finally we compare multi-combined approach (green curve) which is same as blue curve but we use 3 trees. Form Figure 4, we see that using only three (Multi-Combined) trees we can achieve higher accuracy while requiring less space compared to normal RPT. In addition, we also report another natural metric known as rank-error (Ram et al., 2012) to evaluate effectiveness of our proposed method. For a query q and a returned nearest neighbor candidate $p \in \mathcal{S}$, rank error τ is defined as $\tau = |\{r \in \mathcal{S} : \|q-r\|_2 < \|q-p\|_2\}|/|\mathcal{S}|$. For 10-NN search problem, we use average rank-error of closest 10 retrieved points. We present the results in Figure 5. As can be seen from Figure 5, combined approach always yields lower rank-error compared to normal RPT, and Multi-Combined tree always yields lower rank-error compared to combined method (using a single tree).

(L,iter)	(1,20)	(2,10)	(3,7)	(4,5)	(5,4)
MNIST	72%	84%	89%	91%	93%
SIFT	55%	66%	74%	77%	80%
SVHN	63%	77%	84%	87%	89%
JESTER	72%	82%	87%	89%	90%
20Newsgroup	39%	47%	52%	54%	56%
SIAM07	19%	21%	24%	24%	25%

Table 4. 10-NN search accuracy using Mult-Combined method with different # of trees (**L**) and # of iterations per tree (**iter**).

5. Conclusion

In this paper we presented various strategies to improve nearest neighbor search performance using a single space partition tree, where basic tree construct was an RPT. Exploiting proprieties of random projection, we demonstrated how to store auxiliary information of additional space complexity $\tilde{O}(n+d)$ at the internal nodes of an RPT that helps to improve nearest search performance using defeatist search and guided prioritized search. Empirical results on six real world demonstrated that our proposed method indeed improve the search accuracy of a single RPT compared to baseline methods. Our proposed method can also be used for efficiently solving related search problems that can be reduced to an equivalent nearest neighbor search problem and solved using RPT, for example, maximum inner product search problems (Keivani et al., 2017; 2018).

References

Andoni, A. and Indyk, P. Near-Optimal Hashing Algorithms for Approximate Nearest Neighbor in High Dimensions. *Communications of the ACM*, 51(1):117–122, 2008.

Babenko, A. and Lempitsky, V. Product split trees. In *International Conference on Computer Vision and Pattern Recognition (CVPR)*, 2017.

Bentley, J. L. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, 1975.

Beygelzimer, A., Kakade, S., and Langford, J. Cover Trees for Nearest Neighbor. In *23rd International Conference on Machine Learning (ICML)*, 2006.

Ciaccia, P., Patella, M., and Zezula, P. M-tree : An Efficient Access Method for Similarity Search in Metric Spaces. In *23rd VLDB International Conference*, 1997.

Dasgupta, S. and Sinha, K. Randomized partition trees for exact nearest-neighbor search. In *26th Annual Conference on Learning Theory (COLT)*, 2013.

Dasgupta, S. and Sinha, K. Randomized Partition Trees for Nearest Neighbor Search. *Algorithmica*, 72(1):237 – 263, 2015.

Datar, M., Immorlica, N., Indyk, P., and Mirrokni, C. S. Locality-Sensitive Hashing Based on p-Stable Dis. In *The 20th ACM Symposium on Computational Geometry*, 2004.

Gionis, A., Indyk, P., and Motwani, R. Similarity search in high dimensions via hashing. In *25th International Conference on Very Large Databases (VLDB)*, 1999.

Houle, M. E. and Nett, M. Rank based similarity search: Reducing the dimensional dependence. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 37(1): 136–150, 2015.

Johnson, W. B. and Lindenstrauss, J. Extensions of Lipschitz Mapping into Hilbert Space. *Contemporary Mathematics*, 26:189 – 206, 1984.

Katayama, N. and Satoh, S. The ST-tree : An Index Structure for High-dimensional Nearest Neighbor Queries. In *Annual SIGMOD Conference*, 1997.

Keivani, O., Sinha, K., and Ram, P. Improved maxium inner product search with better theoretical guarantee. In *International Joint Conference on Neural Networks (IJCNN)*, 2017.

Keivani, O., Sinha, K., and Ram, P. Improved maximum inner product search with better theoretical guarantee using randomized partition trees. *Machine Learning*, 107(6):1069–1094, 2018.

Krauthgamer, R. and Leel, J. Navigating Nets : Simple Algorithms for Proximity Search. In *15th Annual Symposium on Discrete Algorithms (SODA)*, 2004.

Li, K. and Malik, J. Fast k-nearest neighbor search via dynamic continuous indexing. In *33rd International Conference on Machine Learning (ICML)*, 2016.

Li, K. and Malik, J. Fast k-nearest neighbor search via prioritized DCI. In *34th International Conference on Machine Learning (ICML)*, 2017.

Liu, T., Moore, A. W., Gray, A., and Yang, K. An Investigation of Practical Approximate Nearest Neighbor Algorithms. In *18th Annual Conference on Neural Information Processing Systems (NIPS)*, 2004.

Muja, M. and Lowe, D. G. Fast approximate nearest neighbors with automatic algorithm configuration. In *4th International Conference on Computer Vision Theory and Applications*, 2009.

Omohundro, S. M. Bumptrees for Efficient Function, Constraint and Classification Learning. In *4th Annual Conference on Neural Information Processing Systems (NIPS)*, 1990.

Ram, P., Lee, D., and Gray, A. G. Nearest neighbor search on a time budget via max margin trees. In *SIAM International Conference on Data Mining (SDM)*, 2012.

Sinha, K. LSH vs Randomized Partition Trees : Which One to Use for Nearest Neighbor Search? In *13th International Conference on Machine Learning and Applications (ICMLA)*, 2014.

Sinha, K. Fast l1-norm nearest neighbor search using a simple variant of randomized partition tree. *Procedia Computer Science*, 53:64–73, 2015.

Sinha, K. and Keivani, O. Sparse randomized partition trees for nearest neighbor search. In *20th International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2017.

Sproull, R. F. Refinements to nearest neighbor searching in k-dimensional trees. *Algorithmica*, 6:579 – 589, 1991.

Uhlmann, J. K. Satisfying General Proximity/Similarity Queries with Mteric Trees. *Information Processing Letters*, 40:175–179, 1991.