Nearest neighbor search with multiple random projection trees – core method and improvements

Teemu Henrikki Pitkänen

Helsinki November 25, 2016

MSc Thesis

UNIVERSITY OF HELSINKI
Department of Computer Science

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Faculty of Science	Department of Computer Science

Tekijä — Författare — Author
Teemu Henrikki Pitkänen

Työn nimi — Arbetets titel — Title

Nearest neighbor search with multiple random projection trees – core method and improvements

Oppiaine — Läroämne — Subject
Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
MSc Thesis	November 25, 2016	54 pages + 5 appendices

Tiivistelmä — Referat — Abstract

Nearest neighbor search is a crucial tool in computer science and a part of many machine learning algorithms, the most obvious example being the venerable k-NN classifier. More generally, nearest neighbors have usages in numerous fields such as classification, regression, computer vision, recommendation systems, robotics and compression to name just a few examples. In general, nearest neighbor problems cannot be answered in sublinear time – to identify the actual nearest data points, clearly all objects have to be accessed at least once. However, in the class of applications where nearest neighbor searches are repeatedly made within a fixed data set that is available upfront, such as recommendation systems (Spotify, e-commerce, etc.), we can do better. In a computationally expensive offline phase the data set is indexed with a data structure, and in the online phase the index is used to answer nearest neighbor queries at a superior rate. The cost of indexing is usually much larger than that of performing a single query, but with a high number of queries the initial indexing cost gets eventually compensated.

The urge for efficient index structures for nearest neighbors search has sparked a lot of research and hundreds of papers have been published to date. We look into the class of structures called binary space partitioning trees, specifically the random projection tree. Random projection trees have favorable properties especially when working with data sets with low intrinsic dimensionality. However, they have rarely been used in real-life nearest neighbor solutions due to limiting factors such as the relatively high cost of projection computations in high dimensional spaces. We present a new index structure for approximate nearest neighbor search that consists of multiple random projection trees, and several variants of algorithms to use it for efficient nearest neighbor search.

We start by specifying our variant of the random projection tree and show how to construct an index of *multiple random projection trees* (MRPT), along with a simple query that combines the results from independent random projection trees to achieve much higher query accuracy with faster query times. This is followed by discussion of further methods to optimize accuracy and storage. The focus will be on algorithmic details, accompanied by a thorough analysis of memory and time complexity. Finally we will show experimentally that a real-life implementation of these ideas leads to an algorithm that achieves faster query times than the currently available open source libraries for high-recall approximate nearest neighbor search.

Some of the results have already been published in our paper [HPT$^+$].

ACM Computing Classification System (CCS):
Theory of computation → Random projections and metric embeddings
Theory of computation → Nearest neighbor algorithms
Information systems → Nearest-neighbor search

Avainsanat — Nyckelord — Keywords
random projection, machine learning, nearest neighbor

Säilytyspaikka — Förvaringsställe — Where deposited
Kumpula Science Library

Muita tietoja — övriga uppgifter — Additional information

Contents

Preface

This thesis is a result of research conducted for the Helsinki Institute for Information Technology (HIIT) during the years 2015 and 2016. The approximate nearest neighbor search method covered in this thesis is a joint effort of several colleagues and myself. I would like to thank all those people who have been involved in the research; most importantly Ville Hyvönen, who has been in the project from the start and who has been involved in all parts of the work, Risto Tuomainen for his work on sparsity and the voting in the MRPT context, and Elias Jääsaari for greatly improving the C++ implementation of our algorithms and implementing the tests presented at the end of the thesis. Researchers who also contributed to the research include Sotiris Tasoulis, Teemu Roos, Jukka Corander and Liang Wang. Finally I would like to thank all members of HIIT's Information, Complexity and Learning group for fruitful conversations during our daily meetings.

Contributions

A major part of this thesis consists of a summary of our research group's earlier findings on the approximate nearest neighbor search front. The MRPT approximate nearest neighbor method was intially covered in Ville Hyvönen's master's thesis [Hyv15], which includes an initial version of the MRPT index structure (section 2.4), a defeatist search query (section 2.5), and a comprehensive memory and running time analysis (section 2.6). Risto Tuomainen extended the method by exploring the adaptation of sparsity (section 3.1) and voting (section 3.2) to MRPT in his master's thesis [Tuo16]. Our recent paper [HPT+] added extensive comparisons (section 4) written by Elias Jääsaari. In addition, this thesis covers the following:

- Adapting a priority search strategy to MRPT (section 3.3).
- A discussion on defeatist search tuning parameters and analysis on the optimal RP-tree depth (2.7).
- Additional tests and visualizations; all experiments except those in section 4 were written for this thesis.

Notation

Linear algebra notation

Throughout the thesis vectors will be treated as row vectors and matrices will have samples as rows and features as columns. Matrices will be denoted by uppercase bold symbols such as $\mathbf{M}$, vectors with lowercase bold symbols such as $\mathbf{v}$, and scalars with light symbols such as c or C. To index matrices and vectors we use a Pythonesque syntax; $\mathbf{M}[i, j]$ refers to the j^{th} element on the ith row of $\mathbf{M}$. The indices can also be a set of integers to refer to a subset of rows/columns of $\mathbf{M}$, and : will denote "all", as in $\mathbf{M}[:, k]$, the k^{th} column. Array/vector/matrix indexing is expected to start from 1.

Common variable names

The following symbols will have a fixed meaning throughout the thesis for improved readability.

Symbol	Section	Explanation
$\mathbf{A}$	1	The approximate nearest neighbors
a	3.1	The sparsity parameter of $\mathbf{R}$
B	3.3	The number of extra leaves explored in priority search
D	1	The dimensionality of the data samples in $\mathbf{X}$ and queries $\mathbf{q}$
d	2.4	The depth of random projection trees
$\mathbf{K}$	1	The set of true nearest neighbors
k	1	The number of neighbors being searched for
M	2.5	An MRPT index, a collection of T RP-trees
N	1	The number of data samples (rows) in $\mathbf{X}$
n_0	2.4	Maximum leaf size of RP-trees, $\lceil N/2^d - 1 \rceil$ with median split
$\mathbf{P}$	2.1	The projection matrix $\mathbf{P} = \mathbf{XR}$
$\mathbf{q}$	1	The query point
$\mathbf{R}$	2.1	The random matrix, $\mathbf{R} = [\mathbf{r}_1^\intercal, \ldots, \mathbf{r}_{Td}^\intercal]^\intercal$
T	2.4	The number of trees in an MRPT index
V	3.2	The voting parameter
$\mathbf{X}$	1	The data matrix, $\mathbf{X} = [\mathbf{x}_1^\intercal \ \ldots \ \mathbf{x}_N^\intercal]^\intercal$
$\mathbf{x}_i$	1	The i^{th} sample (row in $\mathbf{X}$)

1 Introduction and related work

Nearest neighbor (NN) search, also called similarity search or closest point search, is an optimization problem where the user inputs one query object at a time, and the program's task is to find the most similar objects from a set of previously collected items. Nearest neighbors have countless uses in machine learning, statistics and data analysis. Example applications include classification, regression, computer vision [SDI06] [ML09] and face recognition [SKP15], vector quantization and compression [GG91], recommendation systems [WTRK15] [Ber16] and even path planning in robotics [Lav98].

Definition 1 (Nearest neighbors) *Consider a metric space $\mathcal{M}$, a data set $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_N) \subseteq \mathcal{M}$ and a distance metric $m : \mathcal{M} \times \mathcal{M} \to \mathbb{R}_+$. The nearest neighbor of object $\mathbf{q} \in \mathcal{M}$ in data set $\mathbf{X}$ is the sample within the smallest distance from $\mathbf{q}$,*

$$\operatorname*{argmin}_{\mathbf{x} \in \mathbf{X}} m(\mathbf{x}, \mathbf{q}).$$

Similarly the k nearest neighbors of $\mathbf{q}$ in $\mathbf{X}$ are a subset $\mathbf{K} \subseteq \mathbf{X}$ with $|\mathbf{K}| = k$ and

$$m(\mathbf{x}, \mathbf{q}) \leq m(\mathbf{y}, \mathbf{q})$$

for all $\mathbf{x} \in \mathbf{K}, \mathbf{y} \in \mathbf{X} \backslash \mathbf{K}$. If we assume all distances unique, these definitions are unique as well.

In this thesis we consider the specific case where $\mathcal{M}$ is the D-dimensional Euclidean space $\mathbb{R}^D$ and m is the Euclidean distance

$$m(\mathbf{x}, \mathbf{q}) = \sqrt{\sum_{j=1}^{D} (\mathbf{x}_j - \mathbf{q}_j)^2}.$$

In this case objects in $\mathcal{M}$ can be represented as feature vectors of length D, and the whole data set $\mathbf{X}$ can be represented as an $N \times D$ matrix.

The conventional way to solve the nearest neighbor problem is a naïve brute-force approach. The distances between the query point and samples in $\mathbf{X}$ are computed one-by-one, and the data points corresponding to the shortest distances are then picked. Although the approach is beautiful in its simplicity and requires no preprocessing of the data, its running time is linear in the size of data matrix $\mathbf{X}$ which is often too much for vast data sets of the present and applications that require real-time responses. Due to its prevalence nearest neighbor search has sparked a lot of

interest in research communities to develop methods that surpass the performance of plain linear search.

In a number of applications nearest neighbor queries are performed repeatedly within a data set $\mathbf{X}$ that is available upfront. If this is the case, we can speed up queries by building an *index data structure* for $\mathbf{X}$ in an *offline phase*. In the *online phase* we then use this structure to answer queries in superior, often sublinear time. Even with the most advanced methods discovered the retrieval of exact neighbors gets unbearably expensive for high-dimensional data, a phenomenon that is known as the *curse of dimensionality*. However, in some applications fast retrieval of neighbors is more important than exact accuracy. An example of this sort of applications are recommendation engines such as KVASIR [WTRK15] and Spotify [Ber16]. This is where *approximate nearest neighbor search* steps in.

Definition 2 (Approximate nearest neighbor search) *Let* $\mathbf{K}$ *be the* k *nearest neighbors of* $\mathbf{q}$ *in* $\mathbf{X}$*. In approximate nearest neighbor search the task is to find a good approximation for* $\mathbf{K}$*, call it* $\mathbf{A} \subset \mathbf{X}$*. To measure the accuracy of the approximation we use recall* R*, that is*

$$R = \frac{|\mathbf{K} \cap \mathbf{A}|}{k}.$$

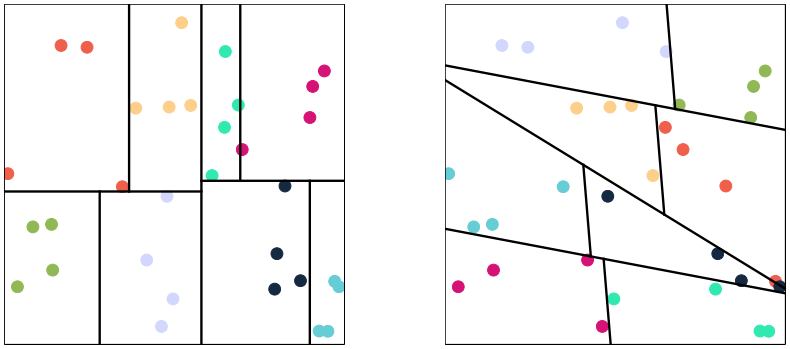
The approximate solution $\mathbf{A}$ can usually be found much faster than $\mathbf{K}$. The plethora of different algorithms for (approximate) nearest neighbor search can be roughly classified into three general categories: hashing, graph, and space partitioning algorithms.

In hashing approaches, namely locality sensitive hashing (LSH) [IM98], [GIM99], [AIL+15], a hash function is used to map each data point into a bucket. The function assigns nearby points to the same bucket with high probability, and points that are far from each other with low probability. Therefore, the hash functions used in LSH are quite the opposite to classical cryptographic hashing where the hash codes of similar-but-not-equal objects are desired to be very different. To answer a query, a simple exhaustive search can be carried out within data points in the same bucket with the query. Practical approaches often use multiple hash tables that implement different hash functions from the same *family* for improved accuracy.

Graph-based approaches [DML11], [AM93b] use a specific graph structure, the k-NN graph, as the index. In this structure each data point in the index is represented by a vertex, and each vertex is connected by an edge to nearby points, either its nearest neighbors or an approximation. The graph is searched with an algorithm

called neighbor descent: We start from some point, and iteratively move along edges that take us closer to the query object. Once such steps can no longer be taken, i.e. the current vertex is closer to the query point than all its neighbors, the algorithm terminates and the current node can be used as an approximation for the nearest neighbor.

In space partitioning methods we build an index data structure that divides the Euclidean space $\mathbb{R}^D$ into cells. As in LSH, the index allows us to quickly select a set of nearby points, those within the same cell, as candidates, and the approximate neighbors $\mathbf{A}$ are chosen from the candidates by an exhaustive search. The space partitioning data structures are inherently represented as binary trees where each node defines a rule that divides the data into two subsets, and hence the index structures are often called binary[1] space partitioning (BSP) trees. Two examples of BSP trees are shown in Fig. 1. The most renowned example is the k-dimensional tree or k-d tree [Ben75] in Fig. 1(a), where axis-aligned splits divide the space into hyper-rectangular cells. Another notable BSP-variant is the vantage point tree (also referred to as VP-tree, ball tree or metric tree) [Uhl91][Yia93] that uses spherical split boundaries. The k-means tree [ML09] partitions objects based on a k-means clustering. The random projection tree (RP-tree) (Fig. 1(b)), introduced by Dasgupta and Freund in 2008 [DF08], partitions the space with randomly chosen hyper-planes. Random projection trees will be the basis of our approximate nearest neighbor method.



(a) A simple k-d tree (b) A simple RP tree

Figure 1: Two examples of space partitioning induced by BSP trees, a k-d tree in 1(a) and a random projection tree in 1(b).

In this thesis we introduce our new variant of the RP-tree, that requires less storage

[1]Three-way and higher order splits are also possible, although most successful methods use binary.

and allows us to construct index structures of even hundreds of trees with feasible memory consumption. We represent new algorithms to efficiently use these data structures for fast and accurate nearest neighbor search. We have developed a working implementation and show experimentally that our method is the fastest alternative to reach high recall in common benchmark data sets.

2 Nearest neighbor search with random projection trees

2.1 Random projection

Random projection (RP) is first and foremost a *dimensionality reduction* technique, where we compute the orthogonal projection of high-dimensional data to a lower-dimensional random subspace. Random projections gained popularity in the late 1990s as a solution to perform computationally feasible analysis on ever growing data sets where known alternatives such as *principal components analysis* (PCA) were too expensive. Random projection has theoretical results to back its ability to preserve distances in the projected space, most importantly the Johnson-Lindenstrauss lemma [JLS86]. Empirical results, while scarce, have also shown promise. For instance, Bingham and Mannila [BM01] find the distance-preserving properties of RP to be on par with PCA with image and text data, with significant computational savings. Dasgupta [Das00] explored RP as a preprocessing step for learning Gaussians with the expectation maximization (EM) algorithm with favorable results.

Definition 3 (Random projection) *Assume data matrix* $\mathbf{X} \in \mathbb{R}^{N \times D}$ *and a random matrix* $\mathbf{R} \in \mathbb{R}^{D \times d}$ *with orthonormal columns and with* $d \ll D$*. In random projection the dimensionally reduced data* $\mathbf{P} \in \mathbb{R}^{N \times d}$ *is obtained as the orthogonal projection of the data to the subspace spanned by columns of* $\mathbf{R}$*:*

$$\mathbf{P} = \mathbf{XR}.$$

Creating a matrix $\mathbf{R}$ of random, orthonormal directions is quite straightforward. We can generate each entry in $\mathbf{R}$ from the standard normal distribution and orthonormalize the columns. However, due to the following result (Lemma 1) the expensive orthonormalization is often omitted:

Definition 4 ((Approximate) orthogonality) *Two vectors* $\mathbf{u}, \mathbf{v} \in \mathbb{R}^D$ *are said to be orthogonal, if* $\mathbf{u} \cdot \mathbf{v} = 0$. *If* $|\mathbf{u} \cdot \mathbf{v}| < \epsilon$ *for a fixed (small)* $\epsilon > 0$, $\mathbf{u}$ *and* $\mathbf{v}$ *are said to be approximately orthogonal.*

Lemma 1 *In sufficiently high dimensional spaces two independently chosen random directions are almost orthogonal with probability 1.*

Proof sketch Denote the random vectors by $\mathbf{u}$ and $\mathbf{v}$. Due to rotational symmetry we can assume $\mathbf{u} = \begin{bmatrix} 1 & 0 & 0 & \cdots & 0 \end{bmatrix}$. Now $\mathbf{u} \cdot \mathbf{v} = \mathbf{u}_1 \cdot \mathbf{v}_1$. The expected value of $\mathbf{v}_1$ is 0 due to symmetry; $\mathbf{v}$ was picked randomly from the unit sphere and for any $\mathbf{v}$ the opposite $-\mathbf{v}$ is equally likely to be chosen. Now also $\mathbb{E}[\mathbf{u} \cdot \mathbf{v}] = 0$.

Because of the length constraint $|\mathbf{v}| = \sum_{j=1}^{D} \mathbf{v}_j^2 = 1$, linearity of expectation and symmetry, $\mathbb{E}[\mathbf{v}_1^2] = 1/D$. Now also the variance $\sigma_{\mathbf{v}_1}^2 = 1/D$, and thus also $\sigma_{\mathbf{u}\cdot\mathbf{v}}^2 = 1/D$.

Applying Chebyshev's inequality yields

$$\Pr[|\mathbf{u} \cdot \mathbf{v}| \geq \epsilon] \leq \frac{1}{D\epsilon^2}$$

For any ϵ there clearly exists a D big enough to drive the probability upper bound effectively to 0.

Furthermore [Ach03] shows, that simple integer values $-1, 0$, and 1 can be used in place of Gaussian variables with respective probabilities $\frac{1}{6}, \frac{1}{3}$, and $\frac{1}{6}$. This brings additional gains through sparsity, and all projections can be computed with integer arithmetic which is valuable especially to database applications. Li et al [LHC06] go further to show that much higher sparsity, such as random matrices with only one in $\sqrt{D}$ elements non-zero can be used.

The use of random projection in applications that fundamentally rely on distances is justified by the Johnson-Lindenstrauss lemma [JLS86], which states that high-dimensional objects can be mapped into lower dimensional spaces with bounded distortions in inter-point distances. Dasgupta and Gupta [DG03] provide the form below with a lower requirement for the target dimensionality together with a simplified proof.

Lemma 2 (Johnson-Lindenstrauss) *For any* $0 < \epsilon < 1$ *and any integer* N, *let* d *be a positive integer such that*

$$d \geq 4(\epsilon^2/2 - \epsilon^3/3)^{-1} \ln N.$$

Then for any set $\mathbf{X}$ *of* N *points in* $\mathbb{R}^D$*, there is a map* $f : \mathbb{R}^D \to \mathbb{R}^d$ *such that for all* $\mathbf{x}_i, \mathbf{x}_j \in \mathbf{X}$*,*

$$(1 - \epsilon)\|\mathbf{x}_i - \mathbf{x}_j\| \leq \|f(\mathbf{x}_i) - f(\mathbf{x}_j)\| \leq (1 + \epsilon)\|\mathbf{x}_i - \mathbf{x}_j\|.$$

Furthermore, this map can be found in randomized polynomial time.

Particularly interesting to us, the map f can be assumed to be a projection to a random subspace [DG03].

2.2 Intrinsic dimensionality

Many data sets are convenient to represent as vectors in a high-dimensional Euclidean space. For instance, common choices for the dimensionality of representation are the number of pixels for images, the vocabulary size for text and the number of different items for market basket data. However, these representations may allow unnecessary degrees of freedom – the samples are often located near a much lower-dimensional manifold, and most of the space contains no actual samples. The concept of *intrinsic dimensionality* refers to the dimensionality of this manifold, the lowest dimensionality that is sufficient to describe the data. The most common formal definition of intrinsic dimensionality is the Assouad dimensionality [Ass83].

Definition 5 (Assouad Dimension) *For any point* $\mathbf{x} \in \mathbb{R}^D$ *and any* $r > 0$*, let* $B(\mathbf{x}, r) = \{\mathbf{z} : \|\mathbf{x} - \mathbf{z}\| \leq r\}$ *denote the closed ball of radius* r *centered at* $\mathbf{x}$*. The Assouad dimension of* $\mathbf{X} \subset \mathbb{R}^D$ *is the smallest integer* D_A *such that for any ball* $B(\mathbf{x}, r) \subset \mathbb{R}^D$*, the set* $B(\mathbf{x}, r) \cap \mathbf{X}$ *can be covered by* 2^{D_A} *balls of radius* $r/2$*.*

The definition is included here for completeness, although a general understanding is sufficient for our purposes. Most importantly, Dasgupta and Freund show that random projection trees have the ability to automatically adapt to the low intrinsic dimensionality of data sets [DF08].

2.3 Random projection tree

A random projection tree is a binary tree structure that partitions $\mathbb{R}^D$ by a hierarchy of split rules. It was initially proposed by Freund et al. [FDKV07] as an alternative for the k-d tree that automatically adapts to low intrinsic dimensionality. The

original paper proposes RP-trees for vector quantization, and the application to nearest neighbor search was later introduced in [DS15]. As a BSP-tree, an RP-tree is utilized in approximate nearest neighbor search to map a query point to a leaf node, which contains a set of data samples that are similar in the sense that they produce the same outputs for the tree's split rules. We can then solve the nearest neighbors within the points in the leaf by a brute force linear search. Such search methodology is sometimes called a *defeatist search* [DF08]. Notice that some nearest neighbors may be located in other leaves and the result is just an approximation, usually a very crude one if only a single tree is used.

In the offline phase, the tree is constructed by recursively partitioning the input data into two by split rules. The split rules in RP-tree nodes are defined by one dimensional random projection, i.e. a projection of the input data onto a single random vector $\mathbf{r}$ chosen uniformly by random from the unit sphere. The samples are then divided into two subsets based on these one-dimensional projection values. A split value $s \in \mathbb{R}$ is chosen, and the samples whose projections were smaller or equal to s are used to recursively construct the left subtree. The right subtree is built from the remaining samples. In the original space $\mathbb{R}^D$ the division is defined by a $D-1$-dimensional hyperplane that passes through point $s\mathbf{r}$ and whose normal vector is $\mathbf{r}$.

The general framework of an RP-tree leaves a lot of implementation specifics undefined. The split value s can be chosen in several ways, the most common choice being the median of the projection values, a choice that yields a *balanced* tree. More complicated criteria have been used and some approaches even mix multiple criteria in one tree [DF08]. Another open design choice is the depth of the recursion – it is hardly optimal to divide the data until each leaf contains just a single point, usually the recursion stops once the node size is below some threshold n_0, or all tree branches are constructed to fixed depth d; with median split criterion the two alternatives are almost equivalent.

A simple defeatist search in a single RP-tree (or BSP-tree in general) is unlikely to produce very flattering results in approximate NN-search. In the unfortunate event that the query point resides very close to a space-partitioning hyperplane, a lot of its neighbors are likely to be sliced to the other half-space and will not be found by a defeatist search strategy. High average recall values, e.g. $R > 90\%$, can not usually be reached at all, or they require extremely shallow trees, whose leaves have a high number of samples, which in turn leads to unbearably long query times. Luckily a

solution exists – we can build the same type of index several times, and combine the results. In BSP-trees we can build multiple trees, and in LSH several hash tables can be used almost equivalently. To get any advantage from the use of multiple index structures some kind of randomization has to be present to ensure that the structures do not yield identical results. In RP-trees this randomness is automatically built-in in the randomized choice of vectors $\mathbf{r}$. This is also the approach we will take; in our method we construct the index structure of multiple identically-parameterized RP-trees and call it a Multiple Random Projection Trees (MRPT) index.

2.4 MRPT index construction

Algorithm 1 Algorithm for MRPT index construction. $\mathbf{X}$ is the data set for which the index is being built represented as an $N \times D$ matrix, T is the number and d the depth of the random projection trees.

```
1: function buildMRPTIndex(X, T, d)
2:     R = RNG.gaussianRandomMatrix(shape=D × Td)
3:     P = XR
4:     trees = ∅
5:     for t = 1, ...,T do
6:         trees.append(buildTree([1, 2, ..., N], 1, t))
7:     return trees

1: function buildTree(S, l, t)
2:     if l ≤ d then
3:         node = new InnerNode
4:         node.split = median(P[S, (t − 1)d + l])
5:         node.left = buildTree({ i ∈ S : P[i, (t − 1)d + l] < node.split }, l + 1, t)
6:         node.right = buildTree({ i ∈ S : P[i, (t − 1)d + l] ≥ node.split }, l + 1, t)
7:     else
8:         node = new LeafNode
9:         node.S = S
10:    return node
```

In Algorithm 1 we present pseudo-code to build an index structure that consists of multiple random projection trees. The algorithm follows the overview given in section 2.3. In this section we discuss the various design choices that have been

made to adapt the general framework given in e.g. [DF08] for an actual working implementation. In Fig. 2 we visualize an MRPT index of four RP-trees that is constructed for a simple two dimensional data set.

2.4.1 Sharing random vectors between branches

In existing literature every random projection tree node generates its own random vector that is used to define a space-partitioning hyper-plane. To answer a query the projections in all nodes on the path from root to leaf have to be computed, and to that end we must have stored the random vectors. Furthermore, we cannot know which paths the queries will follow, and thus all $2^d - 1$ D-dimensional random vectors per tree have to be stored in the index structure. It is easy to see that for a big number of trees and high-dimensional spaces this gets unbearably expensive.

In our version a tree has the same random vector shared in each node throughout a level, i.e. for each tree t we have d random vectors $\mathbf{r}_{t,1}, \ldots, \mathbf{r}_{t,d}$. This way the number of vectors is at most logarithmic to sample size N. Trivially this change will not affect the accuracy of the NN approximation – a query point will follow a single path from root to leaf and is completely unaffected by the random vectors used in nodes not on this path. Our experiments confirm this intuition.

Now when constructing an RP tree t, each sample in $\mathbf{X}$ will get projected exactly once on each random vector $\mathbf{r}_{t,l}$, for all levels $l \in \{1, \ldots, d\}$. We notice that the random vectors can be collected into a random matrix

$$\mathbf{R}_t = \begin{bmatrix} \mathbf{r}_{t,1}^\intercal & \mathbf{r}_{t,2}^\intercal & \ldots & \mathbf{r}_{t,d}^\intercal \end{bmatrix} \in \mathbb{R}^{D \times d},$$

which allows us to compute all projections needed to construct tree t in a single matrix multiplication $\mathbf{P}_t = \mathbf{X}\mathbf{R}_t \in \mathbb{R}^{N \times d}$; a single row contains each data point's projections at all tree levels, and a single column contains the projections of all data points to the random vector used at the corresponding level. Furthermore, for an MRPT index of multiple trees we can combine the random matrices of all trees to a single matrix for the whole index by setting

$$\mathbf{R} = \begin{bmatrix} \mathbf{R}_1 & \mathbf{R}_2 & \ldots & \mathbf{R}_T \end{bmatrix} \in \mathbb{R}^{D \times Td},$$

and compute all projections needed to create *the entire index* in just one matrix multiplication; $\mathbf{P} = \mathbf{X}\mathbf{R} \in \mathbb{R}^{N \times Td}$. The result is called *the projection matrix*. A single row now contains each data point's projections at all tree levels in all trees.

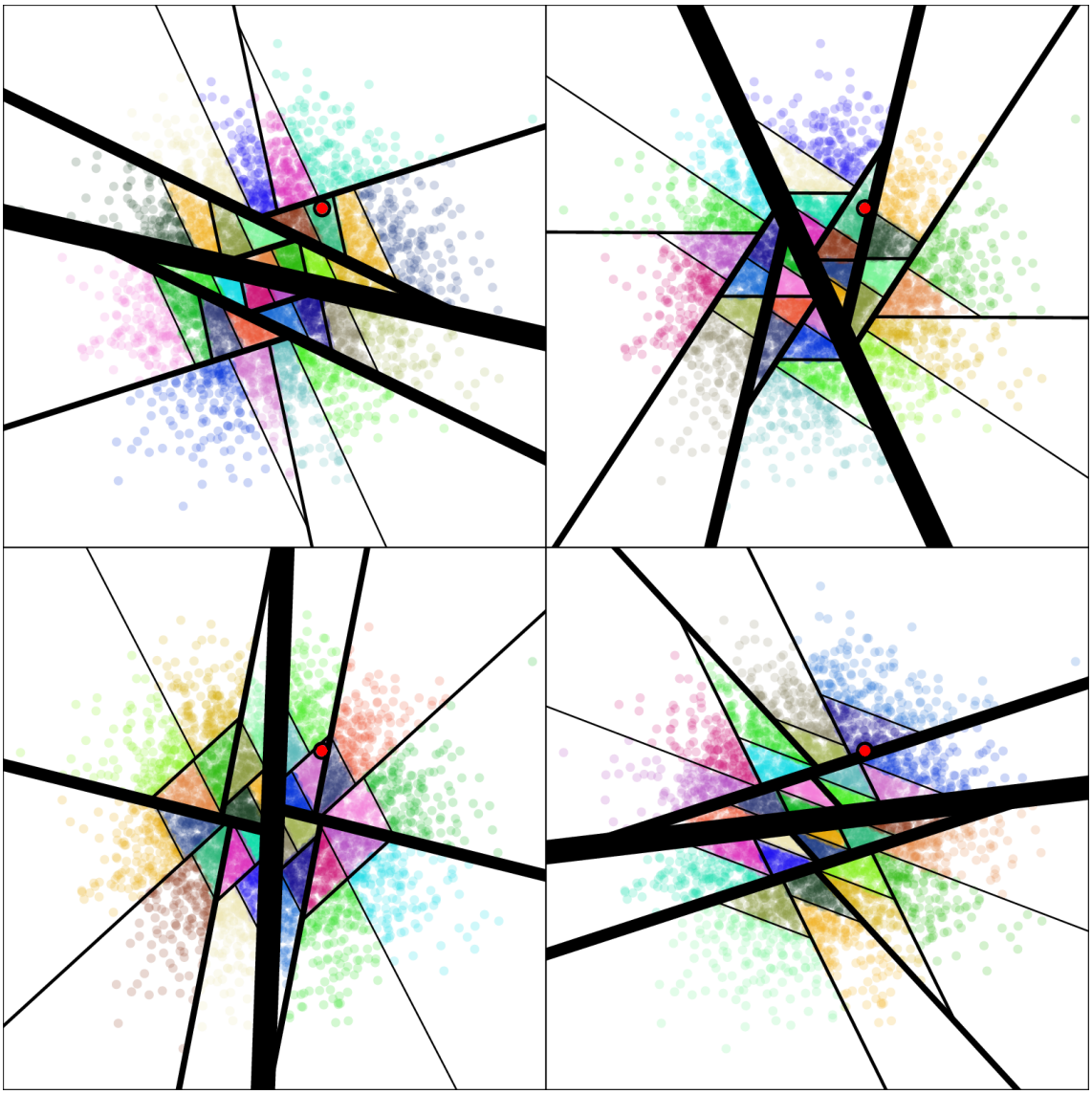


Figure 2: A visualization of an MRPT index for a simple data set in $\mathbb{R}^2$. The images show four different RP-tree space partitionings for the same data set; together they form an MRPT index of four trees. The colorful points represent the data set $\mathbf{X}$ and an example query object is shown by the bright red marker. The black lines show the split boundaries, with line thickness expressing the place in the tree hierarchy: the thickest line represents the split at the tree's root. We emphasize that the query point does not affect tree construction, and could have been placed anywhere – it is included here just to give an idea what kind of neighborhoods single RP-trees might find for a query.

The first *block* of d columns will correspond to the d levels of the first tree, the next block of d columns to the second tree and so on.

Although theoretically the amount of computation stays intact when the computations are combined this way, in real life we see a huge speed-up due to cache effects. Notice also that in the pseudo-code of Algorithm 1 the `buildTree`-function only works with the projection matrix, with no access to the original data. This perspective also allows us to see the close relation that our RP tree has to k-D trees; when working with $\mathbf{P}$ we are just doing coordinate-aligned splits, so basically the index consists of k-D trees built on a data that was preprocessed with random projection.

A final note to be made on the random projection part of the algorithm is that we do not normalize the random vectors $\mathbf{r}$, i.e., the columns of $\mathbf{R}$. In general, the projections of the data set $\mathbf{X}$ on the line spanned by vector $\mathbf{r}$ is $\frac{1}{\|\mathbf{r}\|}\mathbf{X}\mathbf{r}^\intercal$, which equals $\mathbf{X}\mathbf{r}^\intercal$ if the random vector is normalized. Notice that if $\mathbf{r}$ is not normalized to unit length, $\mathbf{X}\mathbf{r}^\intercal$ will yield projections scaled by a factor $\|\mathbf{r}\|$, which does not affect their relative order and the partitioning result does not change.

2.4.2 Split criterion

On line 4 of function `buildTree` we have to choose a *split point* to partition the projections $\mathbf{P}[\mathrm{S}, (t-1)d + l]$ into two subsets. As can be seen, our implementation uses the median value of the projections, but the choice is by no means trivial. In [DS15] Dasgupta and Sinha use a split point chosen uniformly at random from the fractile range $[1/4, 3/4]$, which was guided by the intent to prove better theoretical results.[2] Our approach was to find a criterion that works well in practice, and we tested several criteria on benchmark data sets:

- **median** – The median of the projected values.
- **mean** – The mean of the projected values.
- **longest interval between projected values** – The middle of the longest real interval that has some projection values on both sides.
- **uniform random at the middle fractile [DS15]** – A uniform random real chosen from the $[1/4, 3/4]$ fractile interval.

[2] Appendix C contains a simplification to a central lemma in Dasgupta and Sinha's paper [DS15].

- **uniform random** – A uniform random real chosen from the interval between the smallest and greatest projection values.

We found the effect of split criteria to be minimal, and chose to use the median criterion that allows for compact representation of the tree structures, provides predictable query times and accuracy due to being balanced and having consistent-sized leaves. When splitting an odd-sized input data set, we define that the 'extra' object goes to the right subtree.

2.4.3 Fixed tree depth

We have defined the recursion for tree construction except for one crucial component, the stopping criterion. Usually it is not sensible to keep splitting the data to the end until each leaf contains just one sample, but to stop earlier when some stop condition is met. A common formulation is to define a maximum leaf size $n_0 \in \mathbb{N}_+$ – a node that has more than n_0 samples will be split, and nodes with at most n_0 samples will become leaves. This approach is taken for example in [DS15], [Hyv15], [Tuo16] and [WTRK15].

We decided to use a fixed depth $d \in \mathbb{N}_+$ for each branch instead. We emphasize that we use a convention where by depth we mean the number of edges on any path from the root to a leaf, which is not to be confused with the level of leaves, which is $d + 1$. The fixed depth criterion guarantees that our RP trees are *full binary trees* which together with median split criterion introduces convenient properties in view of both theory and practical implementation.

Lemma 3 *Assume an RP tree constructed from $N \in \mathbb{N}_+$ samples and built to depth $d \in \mathbb{N}_+$ using the median split. The following properties apply:*

- *The tree has 2^d leaves $L_1, \ldots, L_{2^d}$.*
- *For any node at level $l \leq d + 1$, the leaves in its subtree contain (indices of) $\lfloor \frac{N}{2^{l-1}} \rfloor$ or $\lceil \frac{N}{2^{l-1}} \rceil$ objects.*
- *Specifically, every leaf contains $\lfloor \frac{N}{2^d} \rfloor$ or $\lceil \frac{N}{2^d} \rceil$ objects.*

Proof. The first property follows directly from properties of full binary trees – level 1 has a single node and each node has two children, so the number of nodes doubles at each level.

The second property can be easily proven with induction; As the base case we point out that a tree of depth 1 satisfies the claim – only a single split is performed, and as per the definition of splitting at the median, the subsets are sized as the lemma claims. Assume now that the claim holds for an RP-tree of depth d'. As the induction step we show that then the claim holds also for a tree of depth $d'+1$. We solve the problem in four parts.

Due to the assumption nodes at level d' have sizes $a = \lceil \frac{N}{2^{d'-1}} \rceil$ and $b = \lfloor \frac{N}{2^{d'-1}} \rfloor$.

- If $\frac{N}{2^{d'-1}}$ is an integer, $a = b$, i.e. all nodes at level d' have identical size.
  - Furthermore, if $\frac{N}{2^{d'-1}}$ is even, all nodes on level $d'+1$ will have the same size $\frac{N}{2^{d'-1}}/2 = \frac{N}{2^{(d'+1)-1}} = \lfloor \frac{N}{2^{(d'+1)-1}} \rfloor = \lceil \frac{N}{2^{(d'+1)-1}} \rceil$.
  - Otherwise for any node at level d' its children will have sizes $\lfloor \frac{N}{2^{(d'+1)-1}} \rfloor$ and $\lceil \frac{N}{2^{(d'+1)-1}} \rceil$.
- If the result of $\frac{N}{2^{d'-1}}$ is not an integer, on level d' there exists nodes of two sizes a and b, with $a = \lceil \frac{N}{2^{d'-1}} \rceil$ and $b = a - 1 = \lfloor \frac{N}{2^{d'-1}} \rfloor$.
  - If a is even, $b = a - 1$ is odd. Nodes of size a will have two children of size $a/2$, and nodes of size b will have one child of size $a/2$ and one child of size $a/2 - 1$. Now $a/2 = \lceil \frac{N}{2^{(d'+1)-1}} \rceil$ and $a/2 - 1 = \lfloor \frac{N}{2^{(d'+1)-1}} \rfloor$.
  - If b is even, $a = b + 1$ is odd. Nodes of size b will have two children of size $b/2$, and nodes of size a will have one child of size $b/2$ and one child of size $b/2 + 1$. Now $b/2 = \lfloor \frac{N}{2^{(d'+1)-1}} \rfloor$ and $b/2 + 1 = \lceil \frac{N}{2^{(d'+1)-1}} \rceil$.

The third property of the lemma is simply a special case of the second.

2.5 Querying the MRPT index

In Algorithm 2 we present pseudo-code for querying the MRPT index to find the approximate nearest neighbors of a new query object $\mathbf{q} \in \mathbb{R}^D$. As briefly visited in section 2.3, the algorithm consists of two phases; in the *tree traversal phase* we use the RP-tree structures to find a candidate set of points that are "somewhat close" to the query, and in the *linear search phase* we find the nearest neighbors within the candidates with brute force.

Much like in section 2.4.1, we first compute all projections needed to route the query point to a leaf in all trees in a single vector-by-matrix multiplication; $\mathbf{p} = \mathbf{qR} \in \mathbb{R}^{Td}$.

Algorithm 2 An example approximate nearest neighbor query using an MRPT index. $\mathbf{q} \in \mathbb{R}^D$ is the query point, k is the number of neighbors the user wants, and M is an MRPT index, basically a collection of T RP-trees. Vector $\mathbf{p} \in \mathbb{R}^{Td}$ is a concatenation of T blocks of length d, where each block contains the projection values needed for moving in a single tree. Parameter `blockidx` in `RPTreeTraversal` identifies the first index of the block that contains the values for the tree that is currently being traversed. Other variables as described in Algorithm 1.

function approximatekNN1($\mathbf{X}, \mathbf{q}, k, M$)
    $\mathbf{p} = \mathbf{qR}$
    candidates = $\bigcup_{t=1}^{T}$ RPTreeTraversal($M[t]$.root, $\mathbf{p}$, $(t-1)d$)
    return kNNLinearSearch($\mathbf{X}$[candidates], $\mathbf{q}, k$)

function RPTreeTraversal(node, $\mathbf{p}$, blockidx)
    for $l = 1, \ldots, d$ **do**
        if $\mathbf{p}$[blockidx + l] < node.split **then**
            node = node.left
        else
            node = node.right
    return node.S

The *projection vector* $\mathbf{p}$ contains the projection values of $\mathbf{q}$ on random vectors of each level in each tree, the *block* of first d entries corresponding to the first tree, the next d entries to the second tree etc. Based on these projections and the split points stored in the tree structures, we can route the query down to a leaf in every tree in the index (function `RPTreeTraversal`). We construct the *candidate set* by taking a union over all the samples that appear in the same leaf with the query in any of the trees. Fig. 3 shows the candidate set for the example in Fig. 2. In Fig. 2 we see that the query resides fairly close to a split boundary in every single tree. However, in this composite candidate set the query is not too close to the edges; for example all of the query's ten nearest neighbors are included in the candidates.

With reasonable parameters the size of the candidate set is a lot smaller than the whole data set. This allows us to use an exhaustive linear search `kNNLinearSearch` to find the nearest neighbors of $\mathbf{q}$ amongst these candidates to find the set of approximate neighbors. The linear search algorithm simply computes distances $m(\mathbf{q}, \mathbf{x})$ for all samples $\mathbf{x} \in \mathbf{X}$, and chooses the samples that correspond to the k smallest distances, i.e., all true neighbors that made it to the candidate set will be found.

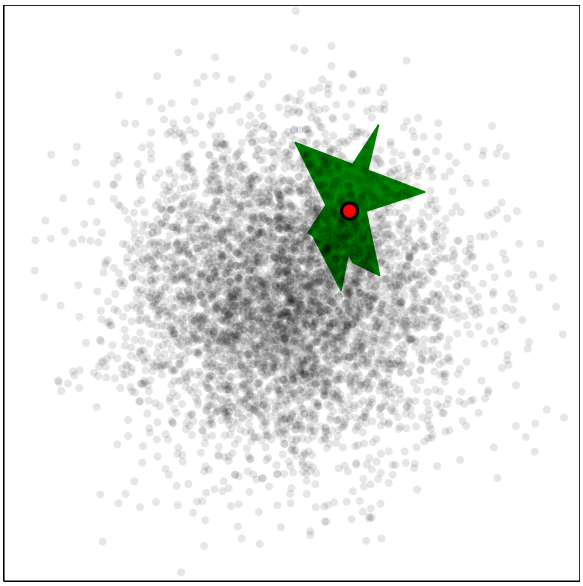


Figure 3: The green area shows which data points will be included in the candidate set when performing a standard MRPT query in the example of Fig. 2. It is simply the combined area of the leaves where the query resides in each tree.

2.6 Memory and running time analysis

As previously, consider a data set $\mathbf{X} \in \mathbb{R}^{N \times D}$ and an MRPT index of $T \in \mathbb{N}_+$ random projection trees built for $\mathbf{X}$ to depth $d \in \mathbb{N}_+$. Our primary concern is to produce an algorithm that can answer nearest neighbor queries in sublinear (in N) time. Also the index construction time and storage size have to be feasible and should be at most linear. As the following analysis shows, all these requirements hold.

Looking first at index construction (function `buildMRPTIndex` in Algorithm 1), we can see that the dominant operations are the computation of projections on line 3, and the T calls to recursive function `buildTree` on line 6. The projections are computed by multiplying a $N \times D$ matrix $\mathbf{X}$ with a $D \times Td$ matrix $\mathbf{R}$, which takes $\Theta(DNTd)$ time with the standard matrix multiplication algorithm.

During recursion, function `buildTree` is executed exactly once for each node in each tree. We can see that the function clearly runs in linear time[3] with respect to the size of input array S. According to Lemma 3, when constructing a node on level l, the size of the input array will be $\Theta(N/2^{l-1})$. In a full binary tree the l^{th} level of

[3]See [CSRL01] for an algorithm to solve median in linear time, even in the worst case.

the tree always has 2^{l-1} nodes, and finally taking a sum over levels and trees we get

$$\sum_{t=1}^{T}\sum_{l=1}^{d} 2^{l-1}\Theta(N/2^{l-1}) = \Theta(NTd)$$

as the time taken by the recursion. The time to compute the projections then dominates index construction, and the whole construction process takes $\Theta(DNTd)$.

How about the memory consumption? For each of the $2^d - 1$ inner nodes we only need to store the constant-size split value[4], so in total for all trees we need to store $T \cdot 2^d - 1 = \Theta(T2^d)$ split points. In the leaf nodes we only have to store the partitioning of the data samples. Because one sample always belongs to exactly one leaf in each tree, this can be easily achieved in $\Theta(NT)$ space[5]. Since depth d is at most logarithmic to N, the storage required by the split points is insignificant in comparison. In addition we need to store the random matrix $\mathbf{R}$, which is of size $\Theta(DTd)$. This sums up to a total memory complexity of $\Theta(NT + DTd)$.

Let us move on to the approximate NN queries, Algorithm 2. In the tree traversal phase the projections $\mathbf{p} = \mathbf{qR}$ are first computed in $\Theta(DTd)$. This is followed by the routing process through the tree structure, which consists of constant-time operations in a for-loop that is run d times.

To reason about the time requirement of the linear search phase we need to know the size of the candidate set, which is an union of the outputs of multiple trees and its size is thus nontrivial to predict exactly. However an upper bound is easily obtained by assuming all trees return unique candidates. According to Lemma 3 the tree leaf size is $\Theta(N/2^d)$, which results in candidate set size $\mathcal{O}(NT/2^d)$. We need to compute the distance between $\mathbf{q}$ and every candidate, which then takes $\mathcal{O}(DNT/2^d)$. Finally we can choose the candidates corresponding to the k smallest distances using a linear-time selection algorithm.

The total running time of an MRPT approximate nearest neighbor query is the sum of the running times of the tree traversal and linear search phases,

$$\Theta(DTd) + \mathcal{O}(DNT/2^d) = \mathcal{O}(DT(d + N/2^d)).$$

Notice the similarities in the terms corresponding to the two query phases – the tree traversal phase running time has a linear dependency on tree depth d, and the linear search phase has a dependency on the leaf size $N/2^d$ instead.

[4]Indeed, the pointers to child nodes need not be stored. See the reference implementation in appendix B. However, this does not affect the asymptotics.

[5]Notice that we are only storing the sample indices, not the samples itself and thus no dependency in dimensionality D.

To reason about the memory requirements, we exclude the space taken by the data matrix $\mathbf{X}$ – it will not be manipulated at all and the copy the user is likely to already have in memory is sufficient. In index construction we need space for matrices $\mathbf{R}$ and $\mathbf{P}$, $\Theta(DTd)$ and $\Theta(NTd)$, respectively. In addition we need space for the split values and index partitionings in RP-tree leaves, $\Theta(TN)$ and $\Theta((2^d - 1)Td)$, respectively, but the storage of the aforementioned matrices dominates. Once construction is finished, the projection matrix $\mathbf{P}$ is no longer needed. In the online phase we will only need additional memory to temporarily store the projected vector and the candidate set. The complete complexity analysis is summarized in the table below.

The number of neighbors being searched for, k, is often a relatively small number that can be treated as a constant, and is for that reason excluded from the analysis above. With the other parameters fixed, k does not change tree construction or the tree traversal phase of queries at all, and thus all changes will be in the linear search in the online phase. There distances between the query point and all points in the candidate set will have to be computed regardless of k, but in the end choosing the k shortest distances and storing the result will have a linear dependency on k – for example, using a max-heap will allow us to choose the k smallest values in $\mathcal{O}(k + N \log k)$ time. However, we point out that the value of k often affects the choice of other parameters, and thereby has an indirect effect on the running time and memory complexity.

	TIME	MEMORY
Index construction	$\Theta(DNTd)$	$\Theta(NTd + DTd)$
Index storage	N/A	$\Theta(NT + DTd + (2^d - 1)Td)$
Tree traversal phase	$\Theta(DTd)$	$\Theta(D) + \mathcal{O}(NT/2^d)$
Linear search phase	$\mathcal{O}(DNT/2^d)$	$\mathcal{O}(N/2^d)$
Query total	$\mathcal{O}((d + \frac{N}{2^d})DT)$	$\Theta(D) + \mathcal{O}(NT/2^d)$

Table 1: Asymptotic memory and time complexity of MRPT.

2.7 On tuning parameters

The MRPT approximate nearest neighbor search method, as discussed thus far, has two free tuning parameters, the number of trees in the index $T \in \mathbb{N}_+$ and the depth of the trees $d \in \mathbb{N}$. Assume we are given some initial parameters (T_0, d_0), and that

we need to increase the query recall. This can be achieved by either making the trees shallower or constructing the index of a bigger set of trees. Both approaches have their pros and cons; making shallow trees slows down the linear search phase of queries exponentially, whereas adding more trees slows down both linear search and tree traversal, but only by a linear factor. In addition both parameters have effects on memory complexity and index construction time. In the following subsection we start building an intuition for choosing parameters through a simple experiment.

2.7.1 Experiments with bounded candidate set size

Assume we have $T_0 = 1$, an index that consists of just a single tree. To reach acceptable performance, we need the tree to be very shallow as discussed in section 2.3, which in turn leads to a big leaf size (Lemma 3). Specifically, we end up in a situation with $d \ll N/2^d$. In section 2.6 we analyzed the time requirement of MRPT approximate nearest neighbor queries and found out that the tree traversal phase takes $\Theta(DTd)$ and the linear search phase takes $\mathcal{O}(DTN/2^d)$. With this knowledge we see that, with our choice of parameters, the time spent in the tree traversal phase becomes insignificant in comparison to the linear search phase. Would it have been a better choice to use a bigger number of trees to balance the time spent in the two phases? Let us seek intuition through simple experiments on a synthetic data set.

Let us define a process that updates the MRPT index parameters in a way that keeps the candidate set size bounded by a constant value C with $C = N/2^{d_0}$ for some tree depth $d_0 \in \mathbb{N}$:

1. First build one tree ($T_0 = 1$) to depth d_0, ie. the tree has leaf size C.
2. For $i = 1, 2, 3, \ldots$
   (a) Double the number of trees.
   (b) Increase tree depth by one, i.e. the leaf size will be approximately one half of the previous iteration.

Because the candidate set size remains bounded by C and at step 0 it is exactly C, the computational cost of the linear search phase never exceeds that of the first step. Contrarily, when the number of trees increases, it becomes likely that the some candidates are redundantly in the query's leaf in several trees, which leads to a smaller candidate set and correspondingly faster linear search. Since the time

spent in tree traversal in a single tree was negligible, we can expect that to be the case also if we use a few trees instead of just one. Thereby, we can expect the total query time to remain unaltered for the iterated parameters, at least for a small number of iterations. The discussion this far does not allow us to draw conclusions on how many iteration steps we can take until this observation breaks, an issue that will be addressed by experiments in this section.

Now we test the parameters at each iteration of this process to build an index for approximate nearest neighbor search in a synthetic data matrix ($N = 32\,768, D = 50$), whose each entry is sampled from the standard normal distribution. We start with $C = 4096$ ($d_0 = 3$), and run 10 iterations of the above process to update parameters. A query object was generated from an identical distribution with the data set, and the recall in 10-NN search and its standard deviation were reported. The results are averages over 100 iterations.

Fig. 4 shows the effect that the parameter updates have on the average recall. Clearly the more trees with correspondingly smaller leaves we use, the more accurate the queries will get. A single tree had a recall less than 0.3; after just five iterations the recall has more than doubled, and at the 10^{th} iteration we get more than 9 out of 10 neighbors correct on average. The standard error of the mean (SEM) was less than 0.02 for all parameter combinations, which tells that the values used to draw the curve are likely very close to the underlying true values.

In addition to high average recall, a user would likely value consistent performance; it seems likely that a method whose recall is consistently between 0.7 and 0.8 is more usable than one where the recall changes arbitrarily between queries in the range from 0 to 1, even if the average recall is the same for both methods. This is why the figure also shows the standard deviation of recall values. The more trees we use, the smaller the deviation between trials gets, and thus the multiple tree approach seems preferable also due to more consistent performance.

The query time can be expected to remain somewhat unaltered, but the iteration increases index construction time and size in memory *exponentially*. Even though our main concern is to attain a good recall/query time ratio, the index construction still has to be feasible.

Based on these results we want to perform iterations to update the parameters at least as long as the query time is unaffected, i.e. tree traversal time is no longer insignificantly small compared to the linear search. In the second experiment we start the iteration from six different parameter combinations, $T_0 = 1, d_0 = 1, 2, \ldots, 6$,

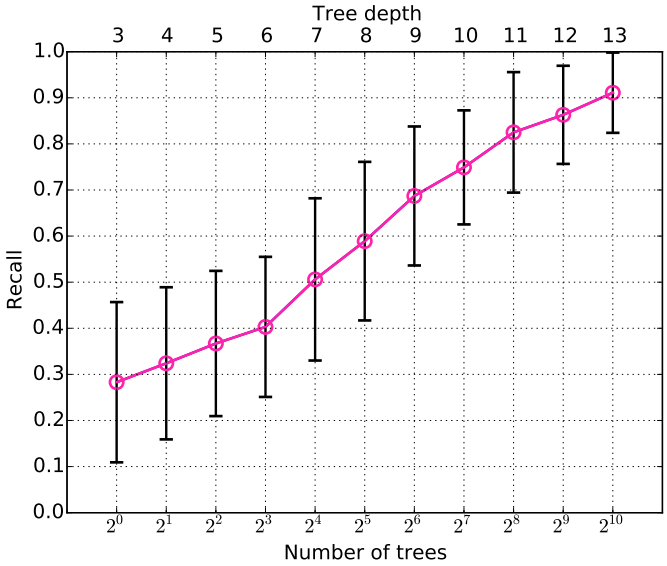


Figure 4: The average recall of approximate nearest neighbor queries for parameters produced by the iteration described in section 2.7.1.

and run ten iterations. The parameters are again used to build an index on the same data set, and the performance is tested in 10-NN search for 100 queries. This time we record both average recall and query time. The results are shown in Fig. 5, where a curve corresponding to the iteration that started from parameters of the previous experiment ($T_0 = 1, d_0 = 3$) is highlighted with a solid line. As discussed in the previous experiment, the standard error of mean for the average recall of 100 queries is fairly low, and we may assume that the results would be similar in a repeated experiment.

The results reflect the theory well. We can see that each curve starts with an almost upright improvement in accuracy, or even have a decrease in query time thanks to duplicates in the candidate set which speeds up linear search. This corresponds to the first few iterations, where the number of trees remains small enough to not affect query time noticeably. Notice that here we can use as many as approximately a hundred trees before the tree traversal time becomes an issue. After the saturation point the recall improvement slows down and the curves bend to the right.

Now our goal should obviously be to reach an as-high-as-possible recall in as-low-as-possible time, i.e. be as close to the upper left corner as we can. We can see that no single candidate set size upper bound (curve) is optimal everywhere; the highlighted curve with candidate set size bounded by $C = 4096$ is fastest to reach recall around

0.7 to 0.8. We can use an index with this candidate set size also to reach a higher recall, but as the figure shows, using a bigger candidate set will provide the same recall in less time. The iteration where the tree depth is 9 is pointed by a circle marker on each curve. As we can see these points are on the optimal performance curve shown with the green shade. This phenomenon is to be discussed in section 2.7.2.

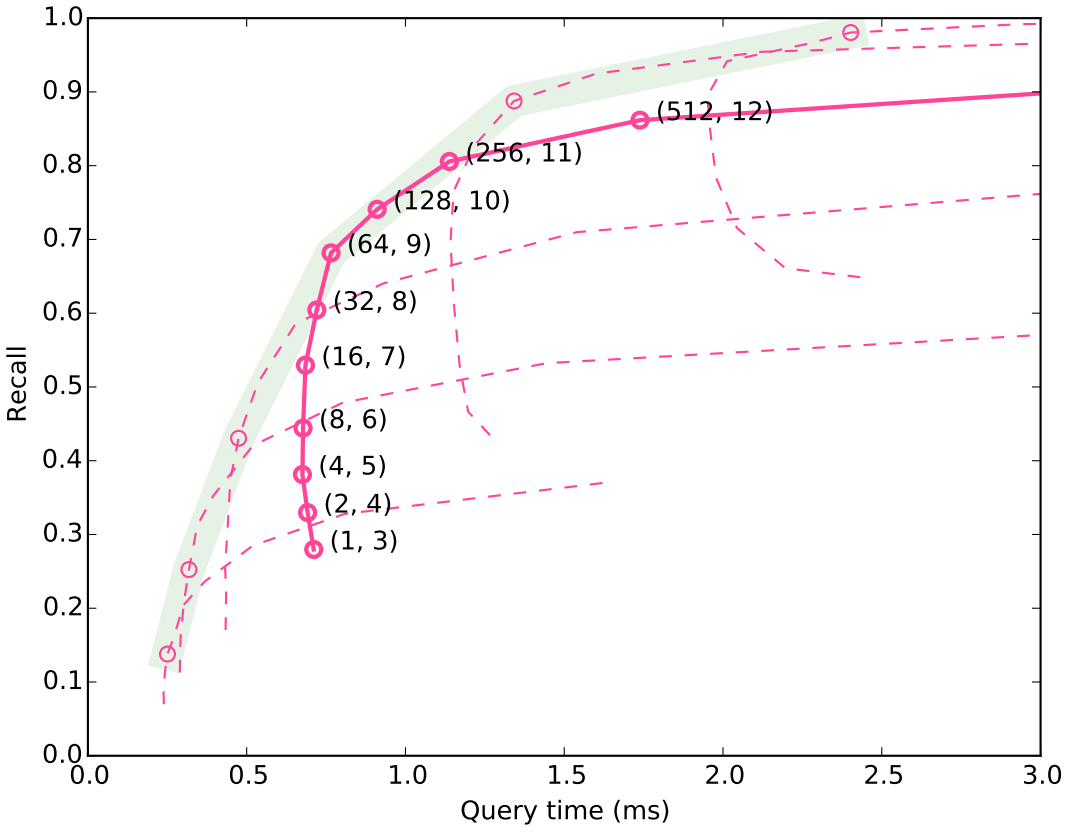


Figure 5: The average recall and query time in approximate 10-NN search in a synthetic data set ($N = 32\,768, D = 50$) for various parameters. Each curve starts with a single tree with depths $d_0 = 1, 2, \ldots, 6$, and the curves represent the changes in performance when the initial parameters are updated with the process described in section 2.7.1. The green shade gives the performance with optimally chosen parameters.

2.7.2 Optimal tree depth

As suggested by the examples in figures 4 and 5, the optimal parameters would be such, that the time spent on the tree traversals is of the same scale with the linear search phase: If the time spent on traversals is insignificant, we can improve accuracy with minimal effect on running time by using more and deeper trees. On the other hand, when the tree traversal time starts to dominate, it seems to be faster to accept a bigger set of objects in the linear search phase.

From section 2.6 we have that the linear search phase runs in $\mathcal{O}(\frac{DNT}{2^d})$. Although this is just an upper bound, our experiments show that it is reasonably tight. For the linear search phase to maintain its status as the dominating factor we want this to also be an upper bound for the running time of the tree traversals, which was analyzed to be $\Theta(DTd)$. We get

$$DTd = \mathcal{O}(\frac{DNT}{2^d})$$
$$d2^d = \mathcal{O}(N).$$

Now let us for a while ignore the $\mathcal{O}$-notation and solve $d2^d = N$ for d. Although the closed form solution is hard to obtain, both d and 2^d increase monotonically as functions of d so the solution is always unique and can be found easily by numerical methods. For the synthetic data set used in the previous subsection (2.7.1) where $N = 2^{15}$, this yields solution $d \approx 11.48$. For $d = 11$ the time requirements of the tree traversals and linear search would be approximately equal – actually that is not what we want but the tree traversal time to be insignificant in comparison, and should thereby use a slightly smaller value of d, like 9 or 10. Now we have to emphasize that the treatment is by no means rigorous in the sense that it ignores all constant factors and non-dominant terms. With this in mind the result turns out to be astoundingly useful; Going back to Fig. 5 the points where $d = 9$ (for 32, 64, 128, 256, 512 and 1024 trees) are marked on each curve, and we can see that these mark the spots where the curves start to bend. These points are connected by the thick green line, which we can see basically gives us the optimal query time/accuracy ratio. We can then simply use this fixed depth parameter and choose the number of trees to address our needs when it comes to query time and accuracy. Notice also that the number of trees does not have to be a power of two but any positive integer value can be used, allowing us to choose the desired query time/recall flexibly from the optimal curve.

In any potential use cases, the user will probably want either as accurate queries as possible within a given time, or as fast queries as possible with a given accuracy. Unfortunately our analysis does not allow to directly estimate the number of trees to satisfy these requirements. However, as any optimal MRPT index will consist of similar RP tees, just a varying number of them, we can define an easy process of finding the optimal parameters that requires the index to be built just once:

1. Build an unnecessarily large set of (e.q., 1000) optimal depth trees.
2. Test queries using subsets of trees of sizes between 1 and T to find the right balance of accuracy and query time.
3. Discard the extra trees.

3 Improving MRPT performance

In the previous section, we described how to build an efficient index structure of multiple random projection trees and how to combine the search results from trees in a simple way that achieved far superior performance to a single RP-tree. However, as we will see in section 4, some other methods still achieve superior performance. To improve even further we can introduce several clever augmentations to the core algorithmic idea of MRPT from existing NN literature – sparsity is a common way to speed up linear algebra operations, and many BSP-variants support priority search (which also has similarities to multi-probing in LSH), both to be covered in this section. Furthermore, we showcase a novel voting scheme [Tuo16] that combines the search results of single RP-trees in a more selective way to choose a smaller candidate set of higher quality, and thus achieves faster query times through faster linear search phase.

3.1 Sparsity

One weakness of RP trees is that in every tree node we need to compute a $\Theta(D)$-time distance computation. In comparison, k-D tree nodes only compare values of single features that can be done in constant time. On the other hand, each split in a k-D tree considers just a single feature, whereas in RP-trees a linear combination of all features is considered, which usually leads to a higher accuracy. The usage of

sparse random vectors in RP trees seems to provide a way to balance the positives and negatives of both approaches.

To construct the random matrix $\mathbf{R}$ with spherically random columns for random projection, we usually sample the entries from the standard Gaussian distribution. Achlioptas [Ach03] shows, that each entry r_{ij} of the random matrix $\mathbf{R}$ can be generated i.i.d. from

$$r_{ij} = \begin{cases} +1 & \text{with probability } \frac{1}{2a}, \\ 0 & \text{with probability } \frac{a-1}{a}, \\ -1 & \text{with probability } \frac{1}{2a}, \end{cases}$$

and $a = 3$ instead, without *any* decrease in the quality of the random projection. This allows the random projections to be computed with database-friendly operations, and more importantly to us, reduces the number of computation through sparsity since only a third of features needs to be considered for projection computations. Li et al. [LHC06] introduce *very sparse random projections*, where $a \gg 3$. They show that for normal-like distributions sparsity as high as $a = \frac{D}{\log D}$ can be used, although a less aggressive choice such as $a = \sqrt{D}$ is recommended for robustness.

Remember that in section 2.4 we learned that our variant of the RP-tree is actually a k-D tree applied on data pre-processed with random projection. In our version of the RP-tree the preservation of distances is clearly an important property. Obviously the smaller the distortions random projection causes in pairwise distances, the better the k-D tree will be able to find the neighbors. In our experiments we have found the sparsity of $a = \sqrt{D}$ as suggested in [LHC06] to be near optimal, and we will from here on fix this value to get rid of one tuning parameter.

With an expected $a = \sqrt{D}$ non-zero elements in each column, the random projection computation in each NN query can now be computed in $\Theta(\sqrt{D}Td)$ expected time. As in section 2.7.2 we can compare the asymptotic complexity of the query tree traversal phase and come up with formula

$$d2^d = \mathcal{O}(\sqrt{D}N),$$

which when solved for d provides the optimal tree depth. When compared to the dense case the only difference is the $\sqrt{D}$ factor on the right hand side of the equation. This makes perfect sense: using sparse random vectors for random projection cuts

down the tree traversal cost, but does not affect the linear search phase complexity at all, so we should be able to afford deeper trees. The added factor in the equation clearly yields a higher value for d.

3.2 Voting

Algorithm 3 An approximate nearest neighbor query using voting. Parameter V tells how many votes are required for a sample to be included in the candidate set for linear search. Other variables and parameters as in Algorithms 1 and 2.

```
1: function approximatekNN2(X, q, k, M, V)
2:     p = qR
3:     votes = zeros(N)
4:     for t = 1, ..., T do
5:         for i ∈ RPTreeTraversal(M[t], p, td) do
6:             votes[i] = votes[i] + 1
7:     candidates = {i : votes[i] ≥ V}
8:     return kNNLinearSearch(X[candidates], q, k)
```

Assume an MRPT index that consists of T trees of depth d built to index data set with $\mathbf{X} \in \mathbb{R}^{N \times D}$. Remember that a tree of depth d has 2^d leaves. Let

$$L_t : \mathbb{R}^D \to \{1, \ldots, 2^d\}$$

be a function that maps an object in $\mathbb{R}^D$ to the index of the leaf that represents the cell in $\mathbb{R}^D$ where the object resides in tree t. In other words, the function corresponds to the tree traversal process. Also let

$$f_t(\mathbf{x}; \mathbf{q}) = \mathbb{1}\{L_t(\mathbf{x}) = L_t(\mathbf{q})\},$$

i.e. f_t is an indicator whether $\mathbf{q}$ and $\mathbf{x}$ reside in the same leaf in tree t. This allows us to denote the total number of trees where $\mathbf{x}$ and $\mathbf{q}$ reside in the same leaf by

$$F(\mathbf{x}; \mathbf{q}) = \sum_{t=1}^{T} f_t(\mathbf{x}; \mathbf{q}).$$

In the basic MRPT query as described in section 2 we include $\mathbf{x}$ in the linear search phase if $F(\mathbf{x}; \mathbf{q}) \geq 1$. In this process we lose information; is a candidate that is in the query's leaf in just a single tree as good as one that appears in the same leaf

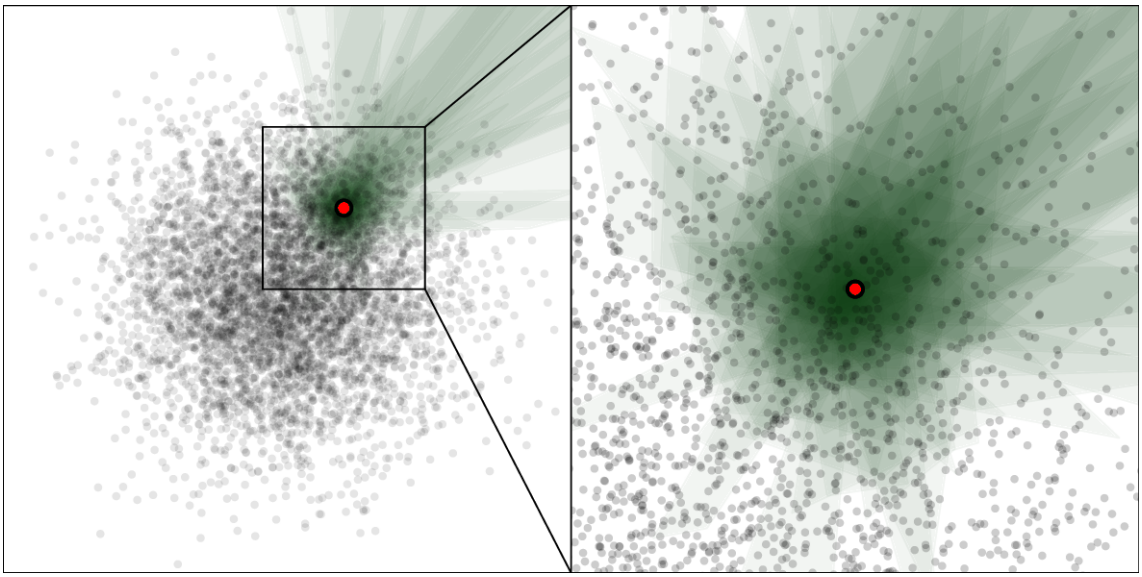


Figure 6: A visualization of voting in an MRPT index (50 RP-trees, depth 5). The query point is shown in red, and the area of the 50 leaves (one per tree) where it resides are shown with semi-transparent green polygons. Because of transparency, areas where multiple leaves overlap get darker. For reasonable k, the k nearest neighbors are located within the roughly spherical dark-green cloud around the query. The darker area thus represents a good input for linear search, and points in the light-green (and white) areas can be omitted.

10 times? Fig. 6 shows, how the points that are actually close to the query point appear often in the same leaf with it, whereas points that are very far from the query point may still be in the same leaf in a few trees. The voting scheme taps into this frequency information and allows us to cherry-pick the candidates that appear in the same leaf often and thereby seem most likely to be actually close. In this analogy the appearance in the same leaf with $\mathbf{q}$ counts as a vote for $\mathbf{x}$. A new vote requirement parameter $V \in \mathbb{N}$ is introduced, and only samples with V or more votes are included in the linear search. The candidate set for linear search is thus

$$\mathbf{C} = \{\mathbf{x} \in \mathbf{X} : F(\mathbf{x}; \mathbf{q}) \geq V\}.$$

The NN query adapted to support voting is shown in algorithm 3.

Now it is obvious that, all other parameters fixed, the basic MRPT ($V = 1$) achieves the highest accuracy, since the induced candidate set is a superset of the candidate sets for any greater V. However, for larger V the candidate set will be substantially smaller which improves speed. What we are interested in is whether the computa-

tional savings outweigh the loss of accuracy, that is, if the voting scheme achieves recall/query time ratios that are above the plain MRPT optimal performance curve.

Fig. 7 shows a simple experiment, where we compare the performance of plain MRPT to the voting variant. We are again working with the $32\,768 \times 50$ synthetic data set. We use an MRPT index with optimal depth trees ($d = 9$), a dense random matrix $\mathbf{R}$ ($a = 1$). We plot the performance of plain MRPT for number of trees T between 1 and 500. For voting, we pick one point from the plain MRPT optimal curve ($T = 380$). Case $V = 1$ is identical to the plain MRPT query, and from there we experiment with higher values of V. For $V \in \{2, 3, 4\}$ the performance of the voting queries is clearly above that reachable by means of plain MRPT. Furthermore, the MRPT index optimal for the plain queries is not optimal for voting. We can use much shallower trees and retain competitive performance, when the candidate set is pruned with voting before linear search. A formula to choose optimal parameters for voting is not known, and a grid search is currently required. A more thorough comparison of MRPT with and without voting will follow in section 4.

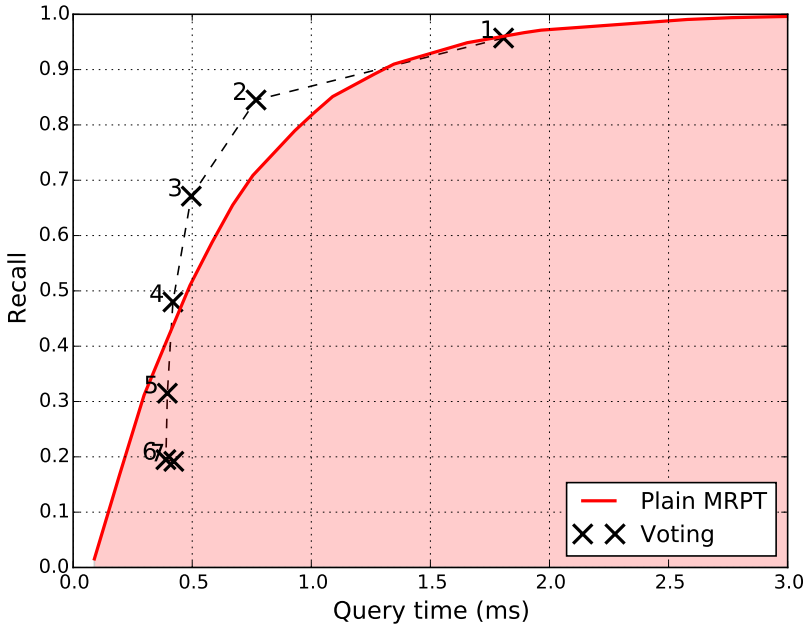


Figure 7: The reachable performance of plain MRPT and the performance of some parameters for voting. Some voting parameters are clearly capable of achieving superior performance to plain MRPT.

3.3 Priority search

In all of the approaches covered in the thesis this far, the final candidate set for linear search is constructed by using only a single leaf per tree. However, in any tree it is possible that the query resides very close to a space partitioning hyperplane that separates it from its true neighbors. In our approaches the use of multiple trees mitigates the issue, since such unfortunate hyperplane is unlikely to exist in all trees, nor even in sufficiently many to interfere with voting. Another solution that has been employed in space partitioning tree variants such as k-D trees [AM93a], *balanced box-decomposition (BBD-) trees* [AMN+98] and hierarchical k-means trees [ML09], is the use of multiple leaves per tree via priority search strategies. In addition to the single leaf where the query point resides, we include additional leaves in the order of increasing distance from the query point. This is also somewhat analogous to multi-probing in LSH [LJW+07], [AIL+15], where slightly perturbed versions are queried in addition to the query point itself, in order to find additional hash buckets that are likely to contain more neighbors. In this section we adapt a priority search strategy to the MRPT index.

Consider a tree t. When only a single leaf per tree is utilized, we choose the leaf where the query point resides according to the split criteria, $L_t(\mathbf{q})$. If we were allowed to pick an additional second leaf, which one would we choose? With the target of finding the query's nearest neighbors, we would like to pick the leaf that contains points closest to the query point. However, to have an advantage over a basic exhaustive search, this should be achieved without evaluating the actual distances between $\mathbf{q}$ and the data points. Remember the interpretation of the space partitioning induced by an RP-tree as cells (i.e., leaves) bounded by hyperplanes in the D-dimensional space. We pick the leaf whose cell contains area closest to the query, i.e., the leaf that the space partitioning induced by the tree *would allow to contain* a point closest to the query, although such point may not actually exist.

In the following, a distance between $\mathbf{q}$ and a leaf L is the shortest distance between $\mathbf{q}$ and any point in $\mathbb{R}^D$ that would be routed to leaf L,

$$m(\mathbf{q}, L) = \inf_{\mathbf{x} \in \mathbb{R}^D : L_t(\mathbf{x}) = L} m(\mathbf{q}, \mathbf{x}),$$

and the distance between $\mathbf{q}$ and an RP-subtree S is the smallest distance between $\mathbf{q}$ and a leaf in that subtree,

$$m(\mathbf{q}, S) = \min_{L \in S} m(\mathbf{q}, L).$$

Now, being allowed to pick a second leaf we would choose

$$\underset{L:L\neq L_t(\mathbf{q})}{\operatorname{argmin}}\ m(\mathbf{q}, L).$$

Similarly, if even more leaves were allowed to be picked, we would choose them in the order of increasing distance from $\mathbf{q}$. In section 3.3.1 we will see how we can compute distances between subtrees and $\mathbf{q}$ and store them in a priority queue that will allow us to quickly select the next closest leaf.

The priority search seems alluring to our method specifically for some key reasons. As we will see, to pick multiple leaves we do not need any additional projection computations. For this reason picking an extra leaf from an existing tree is much less expensive than choosing the first leaf from a completely new tree. Secondly, when the number of trees is incremented, the memory requirement grows linearly. Especially with huge data sets this is bound to be a limiting factor when choosing MRPT tuning parameters. With priority search, we can expect better contribution from individual RP-trees to the MRPT candidate set, and hopefully reach a better accuracy with a smaller set of trees.

3.3.1 Tree traversals and the priority queue

Consider first a single RP-tree t. Assume the random vectors $\mathbf{r}_1, \ldots, \mathbf{r}_d$ used to construct the tree are orthonormal; this will be required for the geometric arguments to be presented to hold. This can be satisfied if the depth of the tree is smaller or equal than the original dimensionality of the data $d \leq D$, which is satisfied always if $\log_2 N \leq D$, a condition that usually holds in potential use cases of MRPT. [6]

The priority search is enabled by a priority queue that will contain RP-subtrees with their squared distances from the query point $\mathbf{q}$ as their priorities. The contents of the queue are thus query point dependent, and a new queue is used to answer each nearest neighbor query. We use a variant where *the entry with the smallest priority value has the highest priority*, i.e. will be dequeued first. At the start of a new query, the queue is initialized to contain the root of the RP-tree. A full RP-tree always contains a single leaf where the query point resides, and thus the (squared) distance between the query point and the tree's root is 0, which will be used as the root's priority.

[6] In high-dimensional spaces random directions can be expected to be almost orthogonal even without the orthogonalization process. In any case, normalized lengths are crucial for the priority search to work; otherwise the presented distance computations between $\mathbf{q}$ and leaves will not hold.

Assume a nearest neighbor query for query point $\mathbf{q}$ using tree t. After the priority queue's initialization the tree traversals begin. Instead of just once, the tree will now be traversed multiple times where each traversal *finds* a single new leaf. Each traversal starts from the highest-priority subtree (node) in the priority queue, and thus the first traversal starts with the queue's only entry, t's root. The traversal happens in the same way as in the earlier MRPT variants, but every time we descend to a node we will add the node's unvisited sibling to the priority queue.

How do we find the distances between the query point and RP-subtrees to be used as priorities? Assume that while routing down $\mathbf{q}$ we visit node v on level l. Node v corresponds to a single split value s_v, and a single random vector $\mathbf{r}_l$ (that is shared with other RP-tree nodes on the same level). With $\mathbf{r}_l$ and s_v the node defines a unique space partitioning hyperplane, whose normal vector is $\mathbf{r}_l$ and that intersects the normal vector at point $s_v\mathbf{r}_l$. We compute the projection $\mathbf{q}\cdot\mathbf{r}_l$ to decide into which child of v to descend. Since $\mathbf{r}_l$ was the hyperplane's normal vector, the difference

$$\epsilon = |s_v - \mathbf{q}\cdot\mathbf{r}_l|$$

directly gives us the distance between the query point and the hyperplane (see Fig. 8). Like before we descend to the side of the hyperplane that contains the query point, and the subtree corresponding to the other side is then known to be ϵ away to the direction of $\mathbf{r}_l$. If this was the first traversal that started from the tree's root, the subtree corresponding to 'the other side of the hyperplane' is added to the priority queue with respective priority ϵ^2. The squared value is used to simplify mathematical operations, which will be explained later. For now it suffices to point out that, because the ϵ-values are positive, the squaring preserves the order of distances so it can be done. The same process is repeated at each level, and once the first leaf is found the priority queue will contain d entries, one for each tree level.

With later traversals things get slightly more complicated. Assume we start a second traversal, and dequeue subtree u on tree level l, that corresponds to the highest priority p. Because this is the second traversal, u is not the tree's root, $l > 1$, and because the leaf where the query resides has already been found in the first traversal, $p > 0$. We also know u is separated from $\mathbf{q}$ by a margin of width $m(\mathbf{q}, u) = \sqrt{p}$, and that this separation is to the direction of $\mathbf{r}_{l-1}$.

We route the query point down starting from u the same way as in the first traversal, but to compute the priorities we have to also account for the separation at the higher tree level contained in the priority value p. On the path from u to a leaf, assume we are at node v on level j. We can compute the difference of the node's split value and

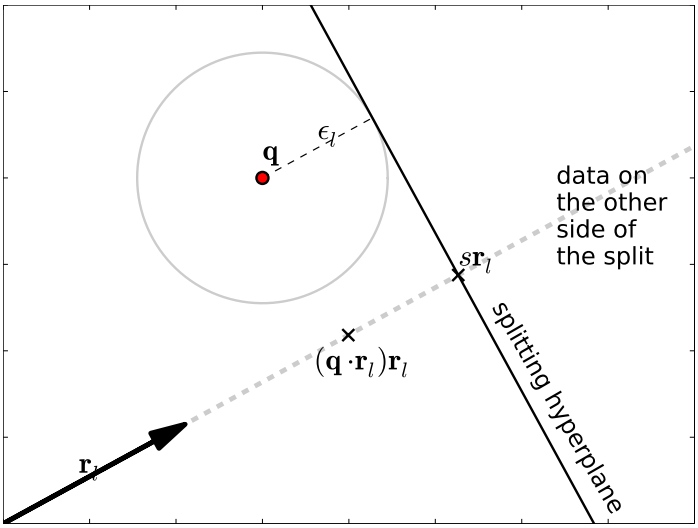


Figure 8: The difference of the projection of the query point and the split value, $\epsilon = |\mathbf{q}\cdot\mathbf{r}_l - s|$, gives the distance to the subtree, where the traversal will not go because the query point is not on the same side of the split. ϵ thus gives a lower bound to the distance between $\mathbf{q}$ and any point on the other side of the split hyperplane.

the query's projection, the ϵ-value, as in the first traversal. In the computation, the query's projection value $\mathbf{q} \cdot \mathbf{r}_l$ will be the same as in the first traversal, and does not have to be computed again, providing a serious computational advantage. Only the subtraction with the projection and the node's split value, both of which are simply one dimensional real numbers, has to be solved. Assume w is the node where we do not visit, i.e. we descend from v to w's sibling. Now we know that w is ϵ away from $\mathbf{q}$ to the direction of $\mathbf{r}_j$. Additionally, w's ancestor u and thus also w are $\sqrt{p}$ away from $\mathbf{q}$ to the direction of $\mathbf{r}_l$. Furthermore, we started with the assumption that $\mathbf{r}_1, \ldots, \mathbf{r}_d$ are orthogonal; now we get the total distance between $\mathbf{q}$ and w from the Pythagorean theorem:

$$m(\mathbf{q}, w) = \sqrt{\sqrt{p}^2 + \epsilon^2} = \sqrt{p + \epsilon^2}.$$

We use the squared distance

$$m(\mathbf{q}, w)^2 = p + \epsilon^2.$$

as the priority, and see the reason for the use of squared distance values – a lot of unnecessary operations can be avoided, since the squared Euclidean distance is coordinate-wise additive.

In the third and further traversals the priorities are computed in exactly the same way; Assume an entry u at level l is dequeued with priority p. The priority p can be interpreted as the squared distance between $\mathbf{q}$ and u (or any descendant of u) to the directions of $\mathbf{r}_1, \ldots, \mathbf{r}_{l-1}$, so we can express it as

$$p = p_1^2 + p_2^2 + \cdots + p_{l-1}^2,$$

where p_i^2 corresponds to the separation to direction of $\mathbf{r}_i$. Notice that any of the terms may be zero. Now we traverse the tree down starting from u as normally, and add the siblings of nodes visited on levels $l+1, \ldots, d$ to the priority queue. The sibling on level $j \in \{l+1, \ldots, d\}$ is enqueued with priority

$$p_1^2 + p_2^2 + \cdots + p_{l-1}^2 + \epsilon^2 = p + \epsilon^2,$$

where ϵ is the difference of the split value and the query point's projection at level $j-1$. As we can see the new priority is computed exactly the same way as in the second traversal.

The additivity of the squared distances here requires that the directions of the separations are orthogonal. The separations are always to the directions of the random vectors $\mathbf{r}_1, \ldots, \mathbf{r}_d$, which are orthogonal by definition. Furthermore, we need to ensure that separation to the direction of the same random vector is never added twice to the same priority value. It is easy to see that this holds; during any traversal all entries added to the priority queue will be on lower tree levels than the node where the traversal started. Thus, the priority value will always equal the sum of squared distances to the directions of $\mathbf{r}_1, ..., \mathbf{r}_l$ for some $l \in \{1, \ldots, d-1\}$, and the added term will be the separation to one of the directions $\mathbf{r}_{l+1}, \ldots, \mathbf{r}_d$.

When the priorities are set and tree is traversed as described, each traversal will find a new leaf in increasing order of distance from $\mathbf{q}$. The reasoning goes as follows: Initially, the priority queue contains the tree's root, so all leaves belong to the only (sub)tree that is in the queue. In any traversal we start by dequeuing a subtree S. On the path to a leaf we enqueue the unvisited sibling node (subtree) at every level, and finally find leaf L. Notice that the enqueued subtrees contain every leaf in the subtree S except L. Thereby, all leaves of the tree either a) belong to a subtree that is currently in the queue or b) have been found by an earlier traversal. When dequeueing subtree S from the priority queue it has some priority p. To be dequeued S must have been the subtree closest to $\mathbf{q}$ in the queue, which means it contains the closest leaf to $\mathbf{q}$, which is L. This is the leaf that the traversal started from S will find, notice that $m(\mathbf{q}, L) = m(\mathbf{q}, S) = \sqrt{p}$. To summarize, in every traversal

we pick the closest leaf that is still in the queue, and at the start of any traversal the queue contains (ancestors of) all leaves except the ones that were picked in an earlier traversal. Any leaf that was found in an earlier traversal has to be closer to $\mathbf{q}$ than any leaf whose ancestor remains in the priority queue due to the same argument. For a simple example see Fig. 9.

Algorithm 4 shows how the tree traversal function `RPTreeTraversal` from Algorithm 2 and the voting query `approximatekNN2` from Algorithm 3 can be extended to support priority search in the MRPT index. In the presence of multiple trees, only a single priority queue will be maintained. The roots of all trees are initially enqueued with priority 0, so the first T traversals will find the leaves where the query resides in each of the T trees (distance to any other subtree will be greater than zero). After these initial traversals using a single queue will focus the traversal efforts to the tree where the split hyperplanes bound space closest to the query point automatically.

3.3.2 Performance of priority search

We test the performance effect of priority search in 10-NN search in the MNIST data set (see section 4.1). We try 10 different values of the number of additional traversals B, $0, 25, 50, 100, 150, 200, 250, 300, 400$ and 500. For the other parameters, we fix sparsity to $a = \sqrt{D} = 28$, and the remaining parameters were chosen optimally using a grid search. Fig. 10 shows the average running time and recall in 100 queries.

We can clearly see that, with no additional constraints, priority search does not bring further benefits to reach a fixed recall level faster – the higher the value of B, the higher is the curve in the figure. Our hypothesis is, that the use of multiple trees is a sufficient and more effective means to reduce the risk of missing neighbors due to split hyperplanes that are located too near to the query point. Although priority search can choose additional leaves without increasing the number of projection computations, it always yields leaves that contain different objects than the leaf chosen in the first traversal, so the result is different than what we would get from choosing a new leaf from a completely new tree. This is a possible reason why the computational advantage of fewer projections does not make the priority search enabled variant faster.

However, the priority search variant still has use cases where it may prove advantageous. On the red curve where $B = 500$, the optimal number of trees found via grid

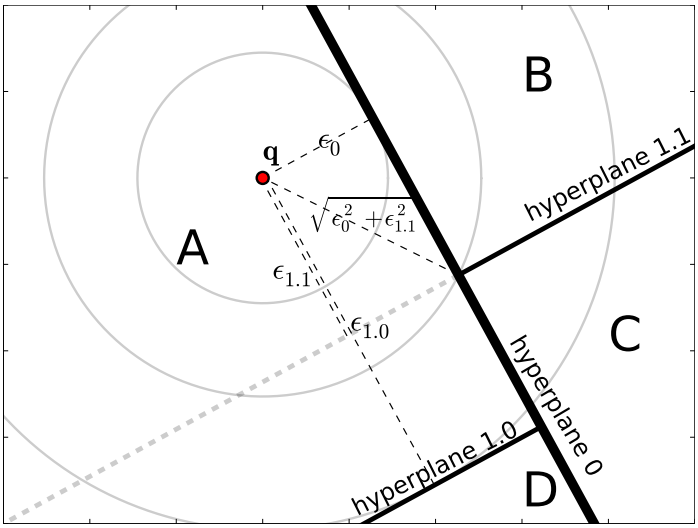


Figure 9: Priority search in a four-leaf RP-tree for query point $\mathbf{q}$. In the start of the first traversal the priority queue contains only the root node, corresponding to split hyperplane 0, where the traversal will start. In the routing process we determine distances between the query point and split hyperplanes 0 and 1.0, which are ϵ_0 and $\epsilon_{1.0}$. The corresponding nodes are added to the priority queue with the squared distances as weights. The first traversal finds leaf A where the query point $\mathbf{q}$ resides. In the second traversal we dequeue node 1.1 (corresponding to the similarly named hyperplane) from the queue, because its priority value ϵ_0^2 is the smallest (out of two) in the queue. We compare the query to hyperplane 1.1 and descend to leaf B. Leaf C is added to the queue with priority $\epsilon_0^2 + \epsilon_{1.1}^2$, the squared distance between $\mathbf{q}$ and C. In the third traversal we dequeue leaf C with priority $\epsilon_0^2 + \epsilon_{1.1}^2$, and finally in the fourth traversal D with priority $\epsilon_{1.0}$. We see that the leaves were found in order of increasing distance from $\mathbf{q}$.

search is approximately half of what is required if $B = 0$. For this reason, the memory requirement for the index is much smaller with priority search. Furthermore, for high recall, the curves get very close to each other, and the speed advantage of the $B = 0$ variant gets insignificant. Another noteworthy point is, that priority search does not require any changes to the index structure. For any MRPT index, we can run queries either with or without priority search, with different values of B if we wish, allowing for a lot of flexibility; queries with higher B will be more accurate but slower. Last, the priority values allow us to infer extra guarantees on the approximation accuracy, that will be discussed in the following section.

Algorithm 4 MRPT approximate nearest neighbor query with priority search. B is a new parameter that tells how many additional traversals are performed in the index after all trees are traversed once. Other variables and parameters as in algorithms 1, 2 and 3.

function approximatekNN3($\mathbf{q}, k, M, V, B$)
    votes = zeros(N)
    $\mathbf{p} = \mathbf{qR}$
    Q = new PriorityQueue(order=smallest first)
    for $t = 1, \ldots, T$ **do**
        Q.enqueue($(0, M[t], t, 1)$)
    for iteration $= 1, \ldots, B + T$ **do**
        leaf = prioritySearchTraversal(Q.dequeue(), $\mathbf{p}$)
        for $i \in$ leaf **do**
            votes[i] = votes[i] + 1
    candidates = $\{i : \text{votes}[i] \geq V\}$
    return kNNLinearSearch($\mathbf{X}$[candidates], $\mathbf{q}, k$)

function prioritySearchTraversal((priority, node, t, startlevel), $\mathbf{p}$)
    for $l =$ startlevel, $\ldots, d$ **do**
        $\epsilon = \mathbf{p}[(t-1)d + l] -$ node.split
        if $\epsilon < 0$ **then**
            Q.enqueue((priority + ϵ^2, node.right, t, l+1))
            node = node.left
        else
            Q.enqueue((priority + ϵ^2, node.left, t, l+1))
            node = node.right
    return node.S

3.3.3 Additional guarantees with priority search

We saw that the priority values of a single RP-tree as set by Algorithm 4 can be interpreted as the squared distances between the query point and subtrees that contain all RP-tree leaves that were not added to the candidate set. For an MRPT index of multiple tree only a single queue is maintained, but it can be thought of as a merge result of the queues for all independent trees of the index. Thus after the traversals end, the values remaining in the queue can be used to infer guarantees for

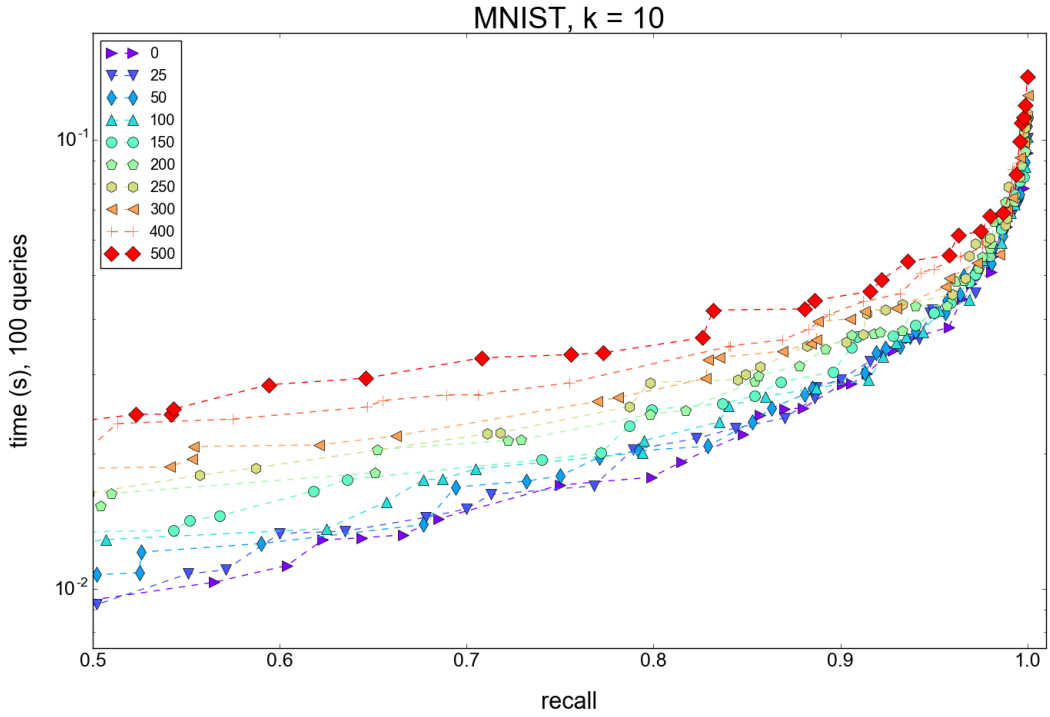


Figure 10: A performance test of priority search 10-NN queries in an MRPT index. The plot shows the average query time and recall for 100 queries in the MNIST data set (see section 4.1) for details on the data set.

distances that the data points still in the queue must be away from the query point. Denote the entries that remain in the queue after traversals as $p_1, p_2, ..., p_m$, where m is the total number of entries. Each entry in the queue is an RP-tre node that belongs to a single tree, $\text{tree}(p_i) \in \{1, ..., T\}$. We can assume, that for all trees there is at least one entry in the queue. Now

$$g_t = \sqrt{\min_{p_i:\text{tree}(p_i)=t} p_i}$$

gives the *guarantee range* for tree t. Since g_t is the shortest distance to a subtree of t that was not found in the traversals, we can guarantee that all points within g_t belong to a leaf that was found. For a 2-d visualization of guarantee ranges see Fig. 9. If only one traversal would be done, only leaf A would be found, and the guarantee range would be ϵ_0, the square root of the smallest priority value in the queue. If the traversals were stopped after two iterations, a second leaf B would be found, and the range would be the square root of the smallest priority value in the queue, $\sqrt{\epsilon_0^2 + \epsilon_{1.1}^2}$, etc.

With plain MRPT queries (voting for $V = 1$), that construct the candidate set by taking a union of the leaves found by single trees this implies that the index can guarantee the longest range that any of the trees in the index can guarantee, and the range for index M would be

$$g_M = \max_{t \in M} g_t = \max_{t \in M} \sqrt{\min_{p_i : \text{tree}(p_i) = t} p_i},$$

because for any data point it suffices for just one tree in the index to find it and it will be included in the candidate set. For voting we require $V > 1$ trees to find the points that will be included as neighbor candidates, and so to find the index guarantee range for voting we have to find the V^{th} greatest single-tree guarantee range. To find the g_t values to solve the index guarantee range we need to scan through the priority queue until at least one entry corresponding to each tree has been met. The priority queue can contain at most $\mathcal{O}((T + B)d)$ points and this operation thus takes at most $\mathcal{O}((T + B)d)$ time, which will not affect the query's asymptotic time complexity.

Where can the index guarantee ranges be used? We can provide them as additional information to the user, to show that there can not be non-found neighbors arbitrarily close to the query point. Maybe the range will be larger than the distance to some of the approximate neighbors returned, in which case they can be confirmed to be true neighbors. Additionally, with negligible changes we can provide two sorts of exact neighbor algorithms that utilize a single RP-tree.

- We can answer *exact range queries*, where all neighbors within a user-specified range will be returned. We build a single-tree MRPT index, and keep starting new traversals until the square root of the priority at the head of the queue is greater than the user specified range. Then we search all candidates in the leaves found with an exhaustive search.

- We can also answer exact k-NN queries. We again build an index of a single RP-tree. We have to alter the query so that the distance between $\mathbf{q}$ and a data point in a leaf that is added to the candidate set is evaluated instantly when the leaf is found. During traversals we keep count of the k nearest neighbors that have been found thus far. We keep starting new traversals, until the square root of the priority at the head of the queue exceeds the distance to the k^{th} nearest data point already found, at that point we know that the current approximation is in fact exact.

However, the algorithms are more of a theoretical curiosity than actually competitive methods for exact neighbor search. This is because the distance guarantees that can be given are in practice very small compared to the actual distances between data points if the dimensionality is high. Consider, for example, a 1000-dimensional data set. If we did a random projection to 1000-dimensional space, this would simply be a random rotation of the data and the distances between data points would not change. Now assume we are using a depth 20 RP-tree in the algorithm; in this case the distances to subtrees are equivalent to being estimated just in the first 20 dimensions of the 1000-dimensional random projection! In practice we found that the guarantee ranges have a usable length only for low-dimensional data, preferably $D \approx d$. With the exact RP-tree algorithms and higher dimensional data we have to perform an extremely large number of traversals to provide the guarantees.

4 Comparisons

In this section we compare the performance of MRPT to other approximate NN libraries. For the comparison we chose leading algorithms with efficient C++ implementations from all three major categories (BSP-trees, LSH and k-NN graphs), and measure the online-phase performance on data sets that represent typical use scenarios for approximate NN search.

We test the search for different number of neighbors $k \in \{1, 10, 100\}$, which covers the range used in typical applications. The performance in all cases is measured by the average value of recall R over 100 queries. With $\mathbf{K}$ being the set of exact k nearest neighbors and $\mathbf{A}$ the approximation returned, $R = |\mathbf{A} \cap \mathbf{K}|/k$, i.e. the fraction of true neighbors within the approximation result.

All experiments were performed on a single Dell PowerEdge M610 computer with 32GB of RAM and 2 Intel Xeon E5540 2.53GHz CPUs. No parallelization was utilized beyond that achieved by the use of linear algebra libraries, such as caching and vectorization of matrix operations.

4.1 Data sets

The data sets chosen for the comparison are listed in table 2 and they reflect realistic use-cases for nearest neighbor search. The only text data set, the News data set [WTRK15] consists of news articles converted to feature vector representation

using a bag-of-words model with term frequency-inverse document frequency (TF-IDF) weights followed by a dimensionality reduction with latent semantic analysis (LSA). MNIST[7], Trevi and STL-10 represent raw image data: MNIST is a collection of grayscale images handwritten digits, Trevi[8] contains grayscale image patches of images depicting three popular tourist attractions, and STL-10[9] [CLN91] is a common image classification benchmark set containing color photos of objects from 10 distinct classes. Data sets GIST and SIFT[10] [JDS11], [JTDA11] contain image descriptors extracted with similarly named algorithms, furthermore SIFT is a random sample of $N = 2\,500\,000$ data points in the data set called BIGANN in the source. Finally we also include a synthetic data set that consists of 50 000 data points sampled uniformly by random from the 4096-dimensional unit sphere. We refer to the included links for further details on the data sets.

Data set	N	D	type
News	262 144	1000	text (TF-IDF & LSA)
MNIST	60 000	768	image
Trevi	101 120	4096	image
STL-10	100 000	9216	image
GIST	1 000 000	960	image descriptors
SIFT	2 500 000	128	image descriptors
Random	50 000	4096	synthetic

Table 2: Specification of data sets that were used in the performance experiments.

4.2 Libraries included in comparison

For completeness we picked algorithms from all three main categories of nearest neighbor methods for the comparison. All these libraries are available open-source, written in C++, support Euclidean distance and have been deployed in a lot of real-life use scenarios.

- ANN [11] (*Approximate Nearest Neighbor library*) is one of the earliest approx-

[7] `http://yann.lecun.com/exdb/mnist/`
[8] `http://phototour.cs.washington.edu/patches/default.htm`
[9] `https://cs.stanford.edu/acoates/stl10`
[10] `http://corpus-texmex.irisa.fr`
[11] `https://www.cs.umd.edu/~mount/ANN/`

imate NN libraries, being released originally in 1998. It implements a set of algorithms based on k-d trees [Ben75] and balanced box-decomposition trees [AMN+98]. We include the classic k-d tree algorithm mainly to be a baseline of performance – the other libraries utilize decades of additional knowledge in the field and expectedly reach better performance.

- FLANN [ML09], [ML12], [ML14] [12] (*Fast Library for Approximate Nearest Neighbor search*) is another BSP-tree based library, which implements *hierarchical k-means trees* and an optimized version of the classic k-d tree the authors call *randomized k-d trees*. We test both methods separately.

- FALCONN [AIL+15] [13] (*FAst Lookups of Cosine and Other Nearest Neighbors*) extends cross-polytope LSH with a multi-probe search strategy that utilizes several hash buckets per table for each query. FALCONN is among the fastest LSH-based libraries.

- KGraph [14] was chosen as a representative of the graph approach. It utilizes a k-NN graph and uses a fast neighbor-descent algorithm [DML11] to search the graph for approximate nearest neighbors.

We also include three different versions of our own algorithms

- RP-trees is the "plain" MRPT query introduced in section 2.5. The query point is placed into a leaf in each tree in the index in a defeatist-search manner, and the results are combined by taking a simple set-union before a brute-force linear search.

- Sparse RP-trees improves upon the preceding by introducing the concept of sparsity as described in section 3.1. We fix the sparsity parameter to the square root of dimensionality, $\sqrt{D}$, to reduce the complexity of parameter search.

- MRPT is the complete algorithm that also utilizes the voting scheme from section 3.2 to achieve a more strictly selected candidate set in order to reduce the brute-force input size.

To choose the parameters for each method we use a grid search on the parameters that the authors report as most important, in combination with default values where

[12] `http://www.cs.ubc.ca/research/flann/`
[13] `https://falconn-lib.org`
[14] `http://www.kgraph.org`

applicable. For ANN, we vary the error bound ϵ, which controls the guaranteed ratio of distances between the query point and the true and approximate nearest neighbors. For FLANN's hierarchical k-D tree algorithm we control the number of trees used in the index structure, whereas for the k-means tree algorithm, where the number of trees is always 1, we control the number of leaves chosen from that tree (chosen from the tree similarly to our priority search strategy in section 3.3), and the maximum number of iterations used in the recursive k-means clustering. The optimal FALCONN index was searched by varying the number of hash tables and functions (through the "number of hash bits" parameter) used in the index, and for KGraph we try k-NN graphs of different degrees and varying values of the search cost parameter. For our own algorithms, we vary the number of trees T, their depth d, the voting parameter V and sparsity a. The voting parameter V was fixed to 1 for (sparse) RP-trees, the sparsity parameter was fixed to $a = \sqrt{D}$ for sparse RP-trees and MRPT, and to $a = 1$ for RP-trees. Furthermore, we adjusted the ranges being searched to each data set so that the best values are not close to the extreme values tested. As an example, the parameters and ranges handled in the grid search for the MNIST data are shown in table 3. More details on the parameter choices can be found at the test suite's GitHub page.[15]

In the grid search we build the index for each combination of parameters, and then test the performance on a sample of 100 query points. This raises the concern, whether the chosen parameters overfit to the set of query points that were used to choose the best performing model. Due to this reason the real-life performance with the parameters chosen via grid search may might be slightly lower that suggested by the grid search process itself. The overfitting issue in the choice of approximate NN algorithm tuning parameters requires further analysis, but is outside the scope of this thesis.

4.3 Results

Figures 11, 12 and 13 show the comparison results for different values of k; 1, 10 and 100, respectively. The results are also listed in tables 4, 5 and 6 in appendix A.

Between different verisons of our own algorithms, we see that sparsity improves performance especially when the dimensionality of the data is high. This is the case with STL-10, Trevi and Random, with respective dimensionalities 9216, 4096 and

[15] `https://github.com/ejaasaari/mrpt-comparison/`

Grid search parameters and ranges for MNIST		
ALGORITHM	PARAMETER	RANGE TESTED
ANN	error ϵ	$1 - 20.75$
FLANN k-d trees	trees	$1 - 32$
FLANN k-means trees	branching	$8 - 256$
	iterations	15
FALCONN	hash tables	$1 - 200$
	number of hash bits	$12 - 18$
KGraph	cost	$1 - 1600$
	graph degree k	$1 - 100$
RP-trees	T (number of trees)	1 - 600
	d (tree depth)	5 - 13
	V (voting parameter)	1
	a (sparsity)	1
Sparse RP-trees	T	1 - 600
	d	5 - 13
	V	1
	a	$\sqrt{D} = 28$
MRPT	T	$10 - 1000$
	d	$5 - 13$
	V	$1 - 100$
	a	$\sqrt{D} = 28$

Table 3: The ranges of tuning parameters optimized in the grid search.

4096. For data sets of lower dimension, e.g. SIFT, the differences are minimal. What really shines through though is the voting scheme; we achieve up to a 10-fold speedup compared to the plain MRPT approach and a 5-fold speedup in several occasions at the high ($> 90\%$) recall levels.

MRPT also outperforms all other BSP-tree and hashing methods compared in six out of seven data sets, with all values of k, with the SIFT data being the only exception where FLANN's BSP-algorithms seem to do a better job. It is important to notice

that SIFT is also the lowest-dimensionality data set within the ones tested. Our hypothesis is that the more efficient projection computations induced by sparsity combined with the flexibility of a high number of independent RP-trees allows the MRPT algorithm to thrive especially in the high-dimensionality scenario.

The KGraph library attains extremely fast queries for moderate recall, especially for high values of k, and for recall levels below 90% it is the fastest method almost consistently. However, in non-optimal problem settings its efficacy suffers dramatically; For $k = 1$ the algorithm is slower than MRPT in most cases, and sometimes its performance drops way below all BSP and hashing methods, see e.g. the $k = 1$ comparison for the Trevi data set in Fig. 11. Also when the query recall is increased beyond 0.9 the query time starts to increase steeply, and for very high recall values $R > 95\%$ MRPT is faster in a clear majority of cases. For extremely high recall, above 99%, MRPT is the fastest method in all data sets for $k = 1$, in 6/7 data sets for $k = 10$ and in 5/7 data sets for $k = 100$.

Overall, the results show admirable performance for MRPT – for high recall queries it is the fastest method in most cases, and even for lower recall levels it is among the fastest algorithms. The only notable exception was the lower-dimensional SIFT data, where MRPT is consistently outperformed by KGraph and FLANN algorithms.

5 Conclusion

We presented the novel MRPT index structure for fast approximate nearest neighbor search. We started with a discussion on existing literature, and chose random projection trees from the class of BSP data structures as the basis of our method. We showed how to build an efficient index structure of multiple RP-trees, presented a practical index construction algorithm, and several augmentations to existing work to make our index structure practically viable.

The discussion on nearest neighbor queries using the MRPT index focused on algorithmic details, which included a detailed analysis on the time and memory complexities, and we found that our algorithm reaches the bounds that are commonly required for approximate NN methods, most importantly linear index memory complexity and sublinear query time. In addition to the common defeatist search strategy we also showcased smart additions to the core idea, that utilize sparseness and the frequency information inherent in the multiple tree approach for additional speed

gains. Finally, we experimented with the addition of a priority search phase to the algorithm, that was experimentally shown to offer no savings in computational time, but is still beneficial to reach higher recall if memory is the limiting factor.

We also discussed the choice of the many tuning parameters that the proposed method has. We observed that with the defeatist search strategy empirical evidence suggests that balancing the computational time spent in tree traversal and the final linear search within possible candidates yields an optimal depth for trees. The derived optimal tree depth was shown to hold in an experimental setting.

To truly put the efficacy of our method to test, we also presented extensive comparisons between the MRPT algorithms and the leading competing approximate NN libraries. These comparisons show, that MRPT is among the fastest, if not even the fastest, algorithm for high-recall approximate nearest neighbor search in high dimensions.

Still, the problem of approximate nearest neighbor search remains far from being solved, and the open-ended research leaves a lot of directions for future work; The probability bounds for the method's success are only analyzed for 1-NN search and were not extended to cover voting. Also our novel voting scheme is by no means specific to the RP-tree setting, and would be an interesting addition to any other BSP-method with sufficient randomness. Our library still misses some core features, most importantly incremental updates to the index and auto-tuning of parameters, both topics where also a lot of theoretic work still remains. The index structure consisting of independent trees would also be ideal for a distributed implementation, that would allow even larger data sets (i.e. big data) to be processed efficiently.

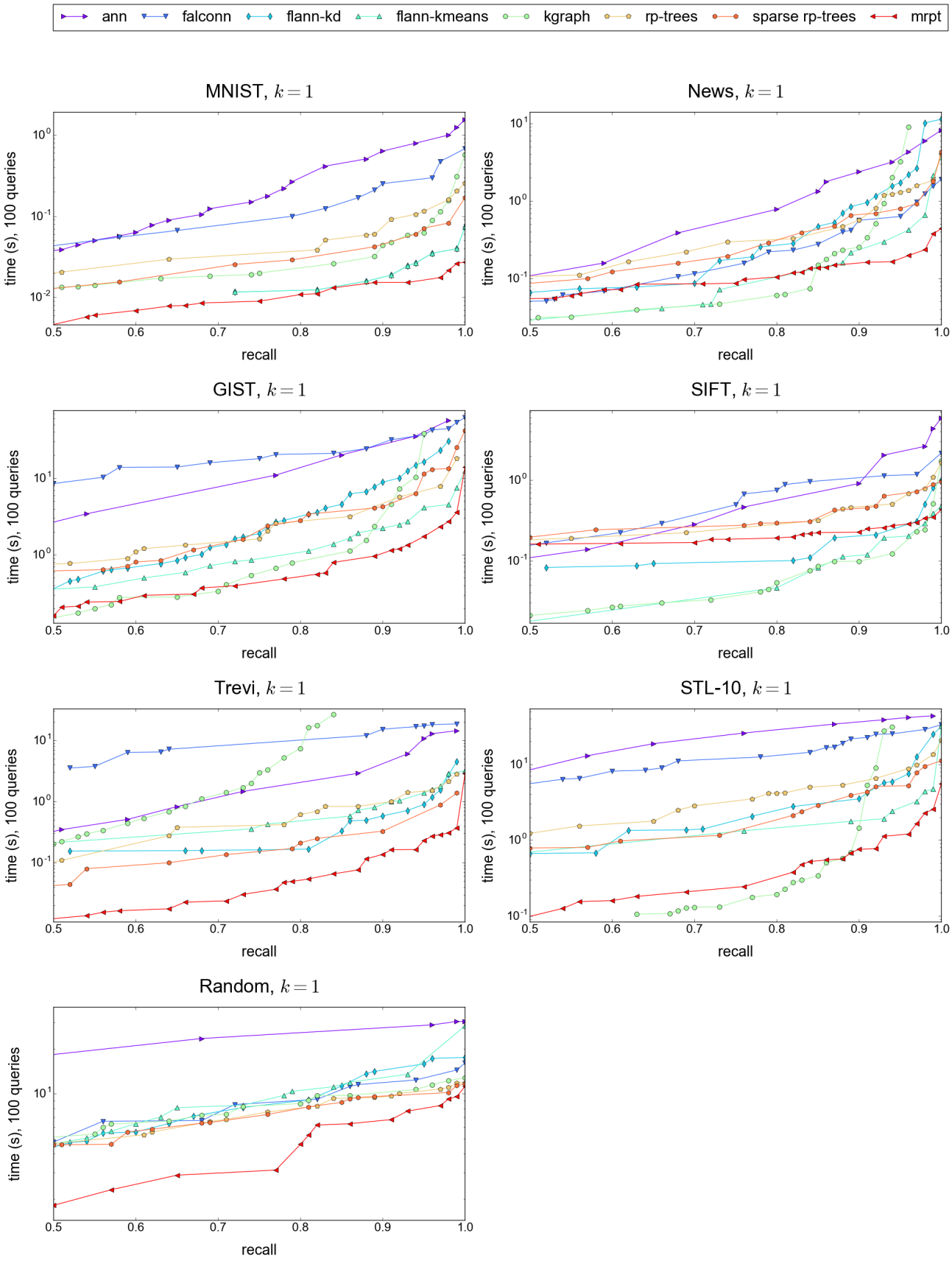


Figure 11: Comparison results for $k = 1$.

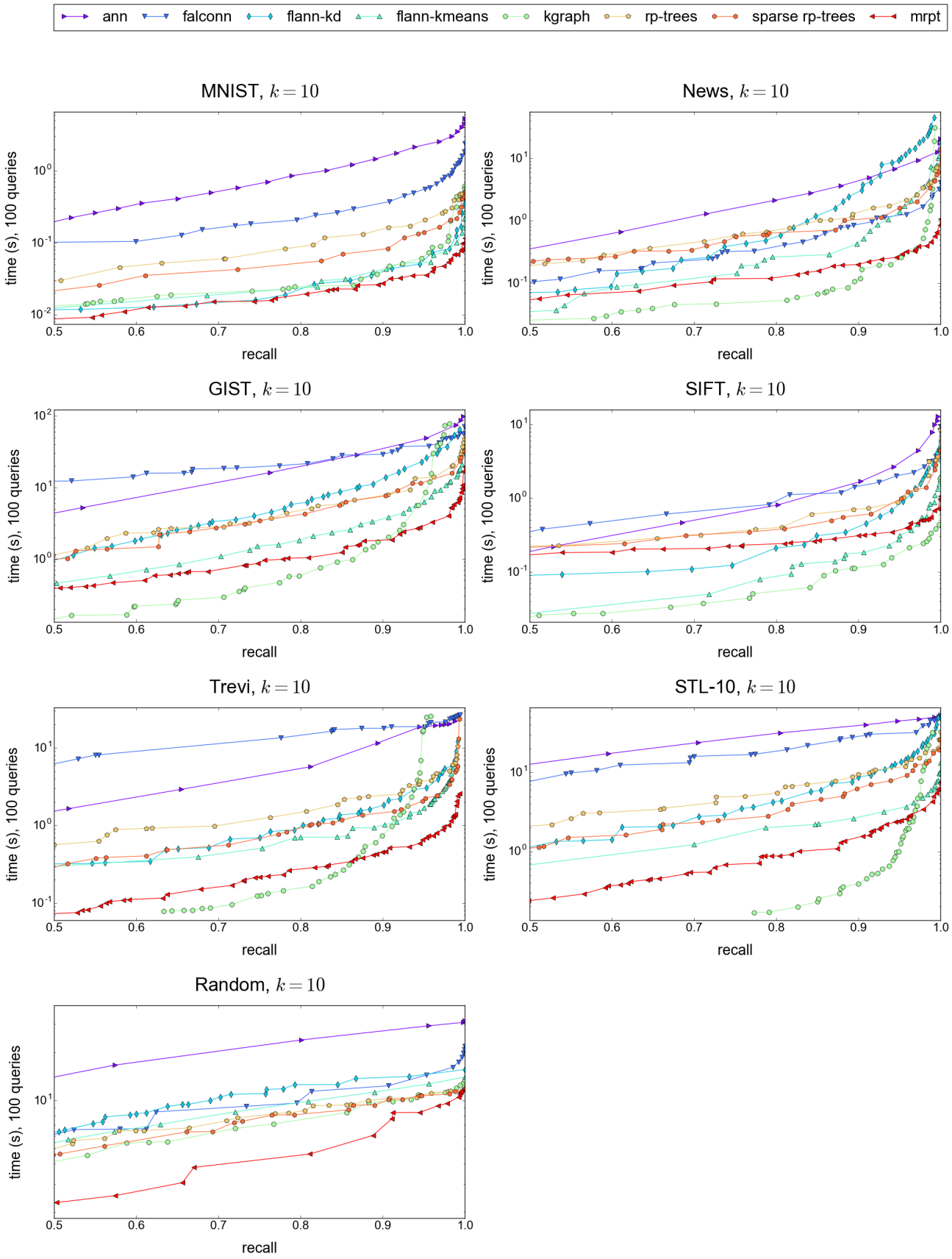


Figure 12: Comparison results for $k = 10$.

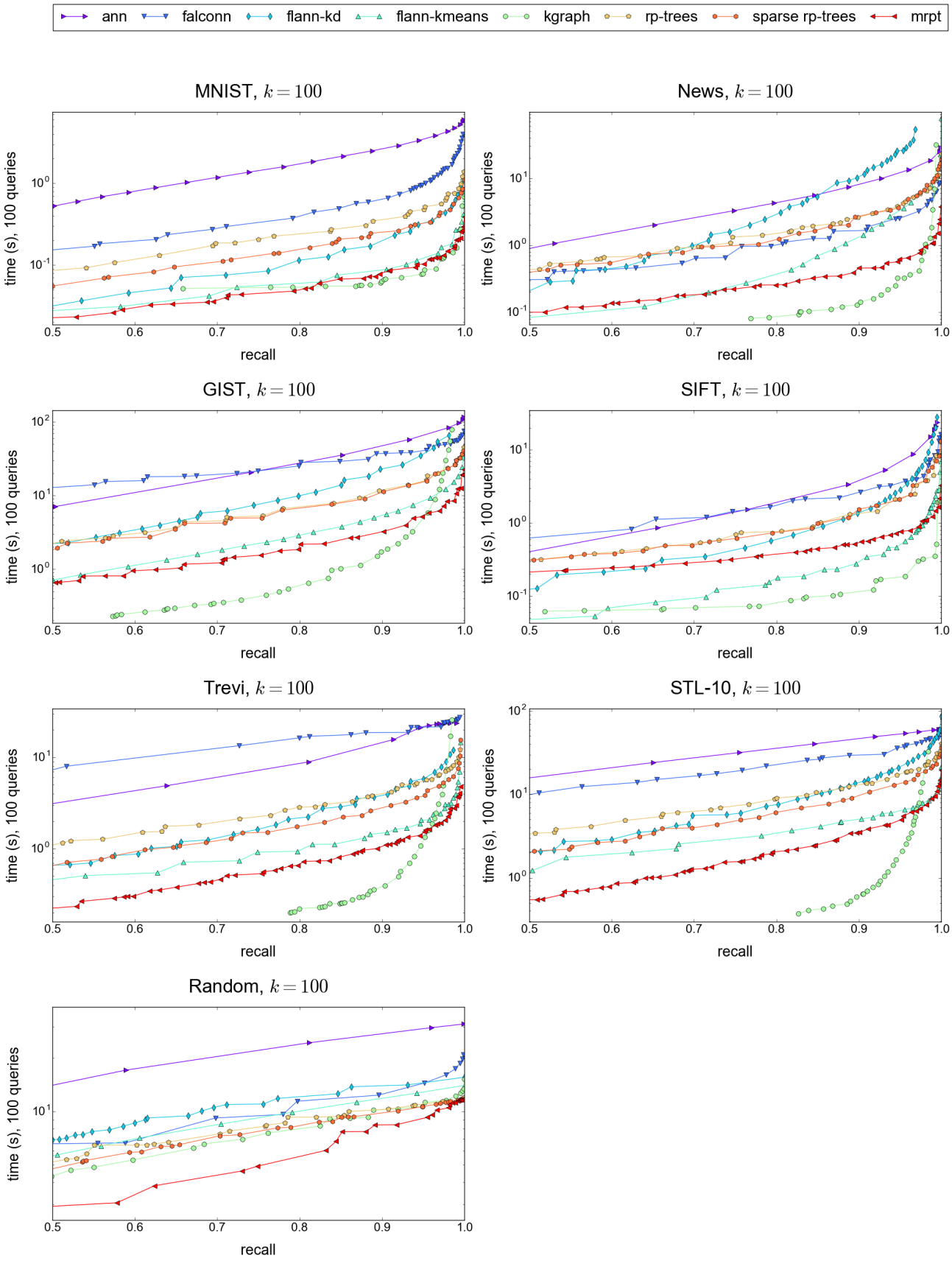


Figure 13: Comparison results for $k = 100$.

References

Ach03 Achlioptas, D., Database-friendly random projections: Johnson-lindenstrauss with binary coins. *Journal of Computer and System Sciences*, 66,4(2003), pages 671–687.

AI06 Andoni, A. and Indyk, P., Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Proceedings of the 47th Annual IEEE Symposium on Foundations of Computer Science*, FOCS '06, Washington, DC, USA, 2006, IEEE Computer Society, pages 459–468.

AIL+15 Andoni, A., Indyk, P., Laarhoven, T., Razenshteyn, I. and Schmidt, L., Practical and optimal lsh for angular distance. *Proceedings of the 28th International Conference on Neural Information Processing Systems*, NIPS'15, Cambridge, MA, USA, 2015, MIT Press, pages 1225–1233.

AM93a Arya, S. and Mount, D. M., Algorithms for fast vector quantization. *Proceedings of DCC '93: Data Compression Conference*. IEEE Press, 1993, pages 381–390.

AM93b Arya, S. and Mount, D. M., Approximate nearest neighbor queries in fixed dimensions. *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '93, Philadelphia, PA, USA, 1993, Society for Industrial and Applied Mathematics, pages 271–280.

AMN+98 Arya, S., Mount, D. M., Netanyahu, N. S., Silverman, R. and Wu, A. Y., An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *Journal of the ACM*, 45,6(1998), pages 891–923.

Ass83 Assouad, P., Plongements lipschitziens dans rn. *Bulletin de la Société Mathématique de France*, 111, pages 429–448.

Ben75 Bentley, J. L., Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18,9(1975), pages 509–517.

Ber16 Bernhardsson, E., Annoy: Approximate nearest neighbors in c++/python optimized for memory usage and loading/saving to disk, 2016. URL `https://github.com/spotify/annoy`. 2.6.2016.

BM01 Bingham, E. and Mannila, H., Random projection in dimensionality reduction: Applications to image and text data. *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '01, New York, NY, USA, 2001, ACM, pages 245–250.

CLN91 Coates, A., Lee, H. and Ng, A. Y., An analysis of single-layer networks in unsupervised feature learning. *International Conference on Artificial Intelligence and Statistics*, 1991, page 215– 223.

CSRL01 Cormen, T. H., Stein, C., Rivest, R. L. and Leiserson, C. E., *Introduction to Algorithms*. McGraw-Hill Higher Education, second edition, 2001.

Das00 Dasgupta, S., Experiments with random projection. *Proceedings of the 16th Conference on Uncertainty in Artificial Intelligence*, UAI '00, San Francisco, CA, USA, 2000, Morgan Kaufmann Publishers Inc., pages 143–151.

DF08 Dasgupta, S. and Freund, Y., Random projection trees and low dimensional manifolds. *Proceedings of the Fortieth Annual ACM Symposium on Theory of Computing*, STOC '08, New York, NY, USA, 2008, ACM, pages 537–546.

DG03 Dasgupta, S. and Gupta, A., An elementary proof of a theorem of johnson and lindenstrauss. *Random Structures & Algorithms*, 22,1(2003), pages 60–65.

DML11 Dong, W., Moses, C. and Li, K., Efficient k-nearest neighbor graph construction for generic similarity measures. *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, New York, NY, USA, 2011, ACM, pages 577–586.

DS15 Dasgupta, S. and Sinha, K., Randomized partition trees for nearest neighbor search. *Algorithmica*, 72,1(2015), pages 237–263.

FDKV07 Freund, Y., Dasgupta, S., Kabra, M. and Verma, N., Learning the structure of manifolds using random projections. *Proceedings of the 20th International Conference on Neural Information Processing Systems*, NIPS'07, USA, 2007, Curran Associates Inc., pages 473–480.

GG91 Gersho, A. and Gray, R. M., *Vector Quantization and Signal Compression*. Kluwer Academic Publishers, Norwell, MA, USA, 1991.

GIM99 Gionis, A., Indyk, P. and Motwani, R., Similarity search in high dimensions via hashing. *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, San Francisco, CA, USA, 1999, Morgan Kaufmann Publishers Inc., pages 518–529.

HD16 Harwood, B. and Drummond, T., Fanng: Fast approximate nearest neighbour graphs. *The IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2016, pages 5713–5722.

HPT+ Hyvönen, V., Pitkänen, T., Tasoulis, S., Jääsaari, E., Tuomainen, R., Roos, T., Corander, J. and Wang, L., Fast nearest neighbor search through sparse random projections and voting. To appear in IEEE International Conference on Big Data, Washington D.C., 2016. Preprint available at https://arxiv.org/abs/1509.06957.

Hyv15 Hyvönen, V., Approximate nearest neighbor search using multiple random projection trees. Master's thesis, University of Helsinki, 2015.

IM98 Indyk, P. and Motwani, R., Approximate nearest neighbors: Towards removing the curse of dimensionality. *Proceedings of the Thirtieth Annual ACM Symposium on Theory of Computing*, STOC '98, New York, NY, USA, 1998, ACM, pages 604–613.

JDS11 Jégou, H., Douze, M. and Schmid, C., Product quantization for nearest neighbor search. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 33,1(2011), pages 117–128.

JLS86 Johnson, W. B., Lindenstrauss, J. and Schechtman, G., Extensions of lipschitz maps into banach spaces. *Israel Journal of Mathematics*, 54,2(1986), pages 129–138.

JTDA11 Jégou, H., Tavenard, R., Douze, M. and Amsaleg, L., Searching in one billion vectors: Re-rank with source coding. *Proceedings of the IEEE International Conference on Acoustics, Speech, and Signal Processing, ICASSP 2011*, 2011, pages 861–864.

Lav98 Lavalle, S. M., Rapidly-exploring random trees: A new tool for path planning. Technical Report, Computer Science Department, Iowa State University, 1998.

LHC06 Li, P., Hastie, T. J. and Church, K. W., Very sparse random projections. *Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '06, New York, NY, USA, 2006, ACM, pages 287–296.

LJW+07 Lv, Q., Josephson, W., Wang, Z., Charikar, M. and Li, K., Multi-probe lsh: Efficient indexing for high-dimensional similarity search. *Proceedings of the 33rd International Conference on Very Large Data Bases*, VLDB '07. VLDB Endowment, 2007, pages 950–961.

ML09 Muja, M. and Lowe, D. G., Fast approximate nearest neighbors with automatic algorithm configuration. *International Conference on Computer Vision Theory and Application VISSAPP'09)*. INSTICC Press, 2009, pages 331–340.

ML12 Muja, M. and Lowe, D. G., Fast matching of binary features. *Computer and Robot Vision (CRV)*, 2012, pages 404–410.

ML14 Muja, M. and Lowe, D. G., Scalable nearest neighbor algorithms for high dimensional data. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 36, pages 2227 – 2240.

SDI06 Shakhnarovich, G., Darrell, T. and Indyk, P., *Nearest-Neighbor Methods in Learning and Vision: Theory and Practice (Neural Information Processing)*. The MIT Press, 2006.

SKP15 Schroff, F., Kalenichenko, D. and Philbin, J., Facenet: A unified embedding for face recognition and clustering. *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, June 2015, pages 815–823.

TCV+14 Tasoulis, S., Cheng, L., Välimäki, N., Croucher, N. J., Harris, S. R., Hanage, W. P., Roos, T. and Corander, J., Random projection based clustering for population genomics. *IEEE International Conference on Big Data*, Oct 2014, pages 675–682.

Tuo16 Tuomainen, R., A variant of the multiple random projection trees algorithm for nearest neighbors search. Master's thesis, University of Helsinki, 2016.

Uhl91 Uhlmann, J. K., Satisfying general proximity / similarity queries with metric trees. *Information Processing Letters*, 40,4(1991), pages 175 – 179.

WTRK15 Wang, L., Tasoulis, S., Roos, T. and Kangasharju, J., Kvasir: Seamless integration of latent semantic analysis-based content provision into web browsing. *Proceedings of the 24th International Conference on World Wide Web*, WWW '15 Companion, New York, NY, USA, 2015, ACM, pages 251–254.

Yia93 Yianilos, P. N., Data structures and algorithms for nearest neighbor search in general metric spaces. *Proceedings of the Fourth Annual ACM-SIAM Symposium on Discrete Algorithms*, SODA '93, Philadelphia, PA, USA, 1993, Society for Industrial and Applied Mathematics, pages 311–321.

A Comparison result tables

		time (s), 100 queries								
data set	R (%)	ANN	FALCONN	FLANN-kd	FLANN-kmeans	KGraph	RP-trees	MRPT($v = 1$)	MRPT	brute force*)
MNIST	0.80	0.3	0.11	**0.01**	**0.01**	0.02	0.04	0.03	**0.01**	2.59
	0.90	0.63	0.25	**0.02**	**0.02**	0.04	0.08	0.05	**0.02**	
	0.95	0.84	0.29	0.03	0.03	0.06	0.12	0.07	**0.02**	
	0.99	1.24	0.61	0.04	0.04	0.31	0.2	0.13	**0.03**	
News	0.80	0.78	0.22	0.27	0.11	**0.06**	0.32	0.31	0.1	23.09
	0.90	2.37	0.56	0.9	0.24	0.25	0.56	0.67	**0.16**	
	0.95	3.74	0.63	1.73	0.38	3.23	1.29	0.8	**0.18**	
	0.99	7.08	1.56	10.76	2.13	-	1.84	1.79	**0.38**	
GIST	0.80	14.43	20.73	3.2	1.09	0.84	2.77	2.81	**0.52**	52.98
	0.90	28.71	29.26	8.89	2.23	3.49	4.86	4.25	**1.05**	
	0.95	40.85	36.89	16.35	4.15	38.57	6.98	11.43	**1.74**	
	0.99	-	53.76	-	7.56	-	18.04	25.21	**3.59**	
SIFT	0.80	0.59	0.75	0.1	**0.05**	**0.05**	0.29	0.29	0.19	21.32
	0.90	0.91	1.07	0.2	0.12	**0.1**	0.47	0.45	0.23	
	0.95	2.27	1.15	0.27	0.2	**0.16**	0.59	0.68	0.28	
	0.99	4.36	1.84	0.81	0.39	0.51	1.09	0.88	**0.35**	
Trevi	0.80	2.17	10.37	0.17	0.49	7.38	0.61	0.21	**0.05**	22.15
	0.90	4.44	15.19	0.58	0.9	-	0.95	0.33	**0.14**	
	0.95	10.71	17.19	0.9	1.38	-	1.47	0.72	**0.23**	
	0.99	14.37	18.51	4.51	3.01	-	2.83	1.38	**0.37**	
STL-10	0.80	28.87	13.14	2.49	1.46	**0.19**	4.12	1.91	0.33	49.43
	0.90	36.42	22.22	3.5	1.81	1.45	5.92	4.11	**0.76**	
	0.95	40.66	26.17	6.7	2.7	-	8.22	5.2	**1.18**	
	0.99	43.75	31.14	24.93	4.7	-	13.49	10.31	**2.54**	
Random	0.80	25.67	9.05	8.99	10.54	8.61	8.05	8.01	**4.61**	10.9
	0.90	27.61	11.88	14.39	12.83	10.22	9.55	9.66	**6.65**	
	0.95	28.58	12.77	15.81	17.75	11.08	10.48	9.99	**8.01**	
	0.99	30.32	14.42	17.41	26.24	12.52	11.77	11.35	**9.59**	

Table 4: Query times required to reach at least a given recall level for recall R = 80%, 90%, 95%, 99%, and for $k = 1$. Fastest times for each recall level are emphasized with bold font.

		time (s), 100 queries								
data set	R (%)	ANN	FALCONN	FLANN-kd	FLANN-kmeans	KGraph	RP-trees	MRPT($v = 1$)	MRPT	brute force*)
MNIST	80	0.89	0.21	**0.02**	**0.02**	**0.02**	0.09	0.05	**0.02**	2.59
	90	1.57	0.36	0.04	0.04	0.04	0.16	0.08	**0.03**	
	95	2.29	0.55	0.06	0.05	0.06	0.2	0.14	**0.04**	
	99	3.46	1.23	0.15	0.1	0.44	0.39	0.22	**0.07**	
News	80	2.15	0.39	0.56	0.26	**0.05**	0.71	0.64	0.12	23.09
	90	4.42	0.88	2.92	0.45	**0.11**	1.37	1.05	0.2	
	95	7.35	1.23	9.55	1.39	**0.25**	1.98	1.62	0.28	
	99	11.83	2.58	38.81	7.5	3.24	4.3	3.87	**0.5**	
GIST	80	20.54	21.21	6.03	1.86	**0.59**	4.77	4.19	1.03	52.98
	90	36.18	29.19	13.35	3.63	1.87	7.69	7.54	**1.83**	
	95	48.13	37.92	24.41	6.22	7.94	13.6	11.89	**2.91**	
	99	76.57	50.29	59.12	14.14	-	24.11	20.33	**6.1**	
SIFT	80	0.81	0.93	0.21	0.09	**0.05**	0.4	0.38	0.24	21.32
	90	1.67	1.47	0.41	0.18	**0.11**	0.71	0.59	0.31	
	95	3.1	2.06	0.84	0.31	**0.18**	0.92	0.97	0.37	
	99	7.88	3.2	2.81	0.86	**0.35**	3.22	2.16	0.62	
Trevi	80	5.5	14.62	0.97	0.7	**0.16**	1.48	0.94	0.27	22.15
	90	12.43	17.88	1.81	1.01	0.75	2.44	1.46	**0.45**	
	95	18.64	18.8	3.02	1.69	18.68	3.74	2.15	**0.67**	
	99	-	25.54	10.54	7.61	-	8.13	5.25	**2.07**	
STL-10	80	31.31	18.62	4.29	2.04	**0.18**	5.46	3.06	0.88	49.43
	90	39.38	28.62	8.76	2.72	**0.39**	9.45	6.32	1.52	
	95	45.0	31.46	13.39	3.53	**1.13**	11.62	8.58	2.43	
	99	49.67	45.42	29.96	6.1	32.12	18.56	16.89	**4.28**	
Random	80	23.86	10.1	12.55	9.73	7.59	8.55	8.2	**4.55**	10.9
	90	27.33	12.27	13.91	11.48	9.8	10.2	9.8	**6.9**	
	95	29.07	14.28	14.5	12.55	10.93	10.88	10.71	**8.6**	
	99	30.47	17.03	15.37	13.7	12.02	11.59	11.42	**10.36**	

Table 5: Query times required to reach at least a given recall level for recall R = 80%, 90%, 95%, 99%, and for $k = 10$. Fastest times for each recall level are emphasized with bold font.

		time (s), 100 queries								
data set	R (%)	ANN	FALCONN	FLANN-kd	FLANN-kmeans	KGraph	RP-trees	MRPT($v = 1$)	MRPT	brute force*)
MNIST	0.80	1.74	0.4	0.11	0.06	**0.05**	0.24	0.16	**0.05**	2.59
	0.90	2.66	0.64	0.2	0.09	**0.06**	0.36	0.25	0.08	
	0.95	3.49	0.98	0.33	0.12	**0.08**	0.48	0.33	0.1	
	0.99	4.93	2.24	0.67	0.27	**0.16**	0.81	0.62	0.2	
News	0.80	4.33	1.01	2.92	0.38	**0.09**	1.51	1.22	0.26	23.09
	0.90	8.18	1.66	10.57	1.33	**0.14**	2.62	2.72	0.46	
	0.95	12.26	2.54	25.88	3.35	**0.32**	3.99	3.9	0.62	
	0.99	20.37	5.35	-	8.78	4.33	8.89	9.24	**1.13**	
GIST	0.80	28.64	26.61	9.99	2.91	**0.61**	7.06	6.8	2.11	52.98
	0.90	48.76	37.34	23.11	5.41	**1.75**	11.94	11.16	3.23	
	0.95	67.08	40.88	36.76	8.55	**5.13**	15.77	15.35	5.3	
	0.99	93.39	55.57	-	17.04	-	30.01	30.1	**10.75**	
SIFT	0.80	2.05	1.75	0.6	0.18	**0.08**	0.77	0.74	0.37	21.32
	0.90	3.9	2.63	1.38	0.3	**0.12**	1.36	1.54	0.54	
	0.95	7.14	3.56	2.86	0.51	**0.18**	2.22	2.12	0.76	
	0.99	17.49	8.08	15.73	1.89	**0.35**	4.45	6.15	1.27	
Trevi	0.80	8.67	16.38	2.18	0.96	**0.22**	2.84	1.76	0.68	22.15
	0.90	14.77	18.73	3.84	1.49	**0.38**	4.01	2.81	1.06	
	0.95	21.67	21.46	5.83	2.06	**1.24**	5.53	3.78	1.5	
	0.99	23.81	25.42	-	4.21	-	8.93	7.02	**2.97**	
STL-10	0.80	36.1	22.89	7.55	3.44	-	8.99	6.11	**2.06**	49.43
	0.90	47.02	29.74	13.95	5.04	**0.61**	13.05	9.75	3.5	
	0.95	53.24	36.67	21.54	6.4	**1.94**	17.62	13.55	4.94	
	0.99	59.13	47.34	42.75	9.39	-	25.84	22.26	**9.19**	
Random	0.80	24.0	11.44	12.11	10.05	7.99	9.29	8.32	**5.6**	10.9
	0.90	27.53	12.51	13.95	11.87	10.23	10.25	9.81	**8.39**	
	0.95	29.31	14.4	14.51	12.86	10.89	11.02	10.66	**9.15**	
	0.99	30.76	17.64	15.37	13.79	12.24	11.61	11.5	**11.32**	

Table 6: Query times required to reach at least a given recall level for recall R = 80%, 90%, 95%, 99%, and for $k = 100$. Fastest times for each recall level are emphasized with bold font.

B A simple implementation in Python

We have implemented the MRPT algorithms into an optimized library written in C++, available at `https://github.com/teemupitkanen/mrpt`. Here we present a simpler version written in Python, that implements the basic MRPT index construction (section 2.4), the plain query (2.5), voting (section 3.2) and priority search (section 3.3). This is meant for demonstrative purposes, and users looking for efficiency should look into the C++ version that also offers bindings for use with the Python language.

```
import numpy as np
from scipy.spatial.distance import cdist
from Queue import PriorityQueue, Empty


class MRPTIndex(object):

    def __init__(self, data, d, T):
        """
        The initializer builds the MRPT index.
        :param data: The data for which the index is built.
        :param d: The depth of the RP-trees.
        :param T: The number of trees used in the index.
        """
        self.N, self.D = data.shape
        self.data, self.d, self.T = data, d, T
        self.R = np.random.normal(size=(self.D, self.T*self.d))
        self.R /= np.linalg.norm(self.R, axis=0)  # Normalize columns
        self.splits = np.zeros((T, d, 2**d))
        self.M = []
        P = np.dot(data, self.R)
        for t in range(T):
            prev_level = [np.arange(len(data))]
            next_level = []
            for i in range(self.d):
                for j in range(len(prev_level)):
                    idxs = prev_level[j]
                    split = np.median(P[idxs, t*self.d + i])
                    self.splits[t, i, j] = split
                    next_level.append(idxs[P[idxs, t*self.d + i] < split])
                    next_level.append(idxs[P[idxs, t*self.d + i] >= split])
                prev_level = next_level
                next_level = []
            self.M.append(prev_level)

    def ann(self, q, k, V=1, B=0):
        """
        The MRPT approximate nearest neighbor query.
        :param q: The query point
        :param k: The number of neighbors being searched for
        :param V: The number votes required for candidates
        :param: b: The number of additional traversals in priority search
```

```
        :return: The approximate neighbors.
        """
        # Prepare for search
        votes = np.zeros(self.N)
        p = np.dot(q, self.R)
        pq = PriorityQueue()
        for t in range(self.T):
            pq.put((0, 0, t, 0))

        # Priority search / defeatist search if b == 0
        for iteration in range(self.T + B):
            try:
                priority, node, t, startlvl = pq.get(block=False)
                for l in range(startlvl, self.d):
                    epsilon = p[t*self.d + l] - self.splits[t, l, node]
                    if epsilon < 0:
                        node *= 2
                        pq.put((priority + epsilon**2, node + 1, t, l+1))
                    else:
                        node = node*2 + 1
                        pq.put((priority + epsilon**2, node - 1, t, l+1))
                votes[self.M[t][node]] += 1

            except Empty:
                print ('Priority queue is empty')

        # Choose candidate set, voting if V > 1
        candidates = np.array([i for i in range(self.N) if votes[i] >= V])

        # Brute-force linear search within candidates
        return candidates[np.argsort(cdist([q], self.data[candidates])[0])[:k]]
```

C A note on performance guarantees of RP-trees in NN search from [DS15]

Probably the most central theoretical result for the NN search performance of RP-trees is the upper bound for the failure probability to retrieve the nearest neighbor with a single RP-tree by Dasgupta and Sinha [DS15]. However, the RP-tree structure used in the proof is slightly different from the trees used in MRPT. Instead of performing the recursive partitioning in tree construction at the median of the projections, they also randomize the choice of the split point, which is chosen uniformly at random from the projections' $[1/4, 3/4]$ fractile interval. The authors state that the heavy use of randomness is necessary to avoid being tripped up by pathological data configurations. We have confirmed that a similar proof structure with the median split and without any additional assumptions on the data set leads to the

trivial upper bound 1. Nevertheless, we found a way to simplify a central formula in Dasgupta and Sinha's paper, that may prove useful in future work on RP theory. The proof in [DS15] builds on the following result.

Lemma 4 (Dasgupta and Sinha [DS15]) *Pick any* $\mathbf{q}, \mathbf{x}, \mathbf{y} \in \mathbb{R}^D$ *with* $\|\mathbf{q}-\mathbf{x}\| \leq \|\mathbf{q}-\mathbf{y}\|$. *Pick a direction* $\mathbf{r}$ *uniformly at random from the unit sphere. The probability, over the choice of* $\mathbf{r}$, *that* $\mathbf{y}\cdot\mathbf{r}$ *falls strictly between* $\mathbf{q}\cdot\mathbf{r}$ *and* $\mathbf{x}\cdot\mathbf{r}$ *is*

$$\frac{1}{\pi}\arcsin\left(\frac{\|\mathbf{q}-\mathbf{x}\|}{\|\mathbf{q}-\mathbf{y}\|}\sqrt{1-\cos^2((\mathbf{q}-\mathbf{x})\sphericalangle(\mathbf{y}-\mathbf{x}))}\right),$$

where

$$\cos((\mathbf{q}-\mathbf{x})\sphericalangle(\mathbf{y}-\mathbf{x})) = \left(\frac{(\mathbf{q}-\mathbf{x})\cdot(\mathbf{y}-\mathbf{x})}{\|\mathbf{q}-\mathbf{x}\|\|\mathbf{y}-\mathbf{x}\|}\right)$$

i.e. it is the cosine of the angle between vectors $(\mathbf{q}-\mathbf{x})$ *and* $(\mathbf{y}-\mathbf{x})$.

We found out that this result can be simplified to a form that has also a more intuitive interpretation.

$$\begin{aligned}&\frac{1}{\pi}\arcsin\left(\frac{\|\mathbf{q}-\mathbf{x}\|}{\|\mathbf{q}-\mathbf{y}\|}\sqrt{1-\cos^2((\mathbf{q}-\mathbf{x})\sphericalangle(\mathbf{y}-\mathbf{x}))}\right)\\ &= \frac{1}{\pi}\arcsin\left(\frac{\|\mathbf{q}-\mathbf{x}\|}{\|\mathbf{q}-\mathbf{y}\|}\sin((\mathbf{q}-\mathbf{x})\sphericalangle(\mathbf{y}-\mathbf{x}))\right) = \dots\end{aligned}$$

Now the law of sines states that for any triangle with sides A, B and C and the angles opposite to those sides a, b and c, it holds that $A/\sin(a) = B/\sin(b) = C/\sin(c)$. This yields $\sin(b) = \frac{B}{A}\sin(a)$, and we place $A = \|\mathbf{q}-\mathbf{x}\|$, $B = \|\mathbf{q}-\mathbf{y}\|$ and $C = \|\mathbf{y}-\mathbf{x}\|$.

$$\begin{aligned}\dots &= \frac{1}{\pi}\arcsin\left(\frac{\|\mathbf{q}-\mathbf{x}\|}{\|\mathbf{q}-\mathbf{y}\|}\frac{\|\mathbf{q}-\mathbf{y}\|}{\|\mathbf{q}-\mathbf{x}\|}\sin((\mathbf{q}-\mathbf{y})\sphericalangle(\mathbf{y}-\mathbf{x}))\right)\\ &= \frac{1}{\pi}\arcsin\left(\sin((\mathbf{q}-\mathbf{y})\sphericalangle(\mathbf{y}-\mathbf{x}))\right)\\ &= \frac{(\mathbf{q}-\mathbf{y})\sphericalangle(\mathbf{y}-\mathbf{x})}{\pi}\\ &= \frac{1}{\pi}\arccos\left(\frac{(\mathbf{q}-\mathbf{y})\cdot(\mathbf{y}-\mathbf{x})}{\|\mathbf{q}-\mathbf{y}\|\|\mathbf{y}-\mathbf{x}\|}\right).\end{aligned}$$

Thus, the probability is simply the angle between vectors $\mathbf{q}-\mathbf{y}$ and $\mathbf{y}-\mathbf{x}$, normalized to a probability by division by π. The lengths of the vectors do not affect the probability.