Fast Nearest Neighbor Search through Sparse Random Projections and Voting

Hyvönen, Ville

2016

Hyvönen, V, Pitkänen, T, Tasoulis, S, Jääsaari, E, Tuomainen, R, Wang, L, Corander, J I & Roos, T 2016, Fast Nearest Neighbor Search through Sparse Random Projections and Voting. in Proceedings of the 2016 IEEE Conference on Big Data. IEEE, New York, NY, pp. 881-888, IEEE International Conference on Big Data, Washington, DC, United States, 05/12/2016. https://doi.org/10.1109/BigData.2016.7840682

http://hdl.handle.net/10138/301147
10.1109/BigData.2016.7840682

Fast Nearest Neighbor Search through Sparse Random Projections and Voting

Ville Hyvönen*, Teemu Pitkänen*, Sotiris Tasoulis†, Elias Jääsaari*, Risto Tuomainen*,
Liang Wang‡, Jukka Corander*§¶ and Teemu Roos*
*Helsinki Institute for Information Technology HIIT, University of Helsinki, Finland. Email: {firstname.lastname}@cs.helsinki.fi
†Liverpool John Moores University, UK. Email: S.Tasoulis@ljmu.ac.uk
‡Computer Laboratory, University of Cambridge, UK. Email: liang.wang@cl.cam.ac.uk
§Pathogen Genomics, Wellcome Trust Sanger Institute, Cambridge, UK
¶Department of Biostatistics, University of Oslo, Norway. Email: jukka.corander@helsinki.fi

***Abstract*—Efficient index structures for fast approximate nearest neighbor queries are required in many applications such as recommendation systems. In high-dimensional spaces, many conventional methods suffer from excessive usage of memory and slow response times. We propose a method where multiple random projection trees are combined by a novel voting scheme. The key idea is to exploit the redundancy in a large number of candidate sets obtained by independently generated random projections in order to reduce the number of expensive exact distance evaluations. The method is straightforward to implement using sparse projections which leads to a reduced memory footprint and fast index construction. Furthermore, it enables grouping of the required computations into big matrix multiplications, which leads to additional savings due to cache effects and low-level parallelization. We demonstrate by extensive experiments on a wide variety of data sets that the method is faster than existing partitioning tree or hashing based approaches, making it the fastest available technique on high accuracy levels.**

***Index Terms*—Nearest Neighbor Search; Random Projections; High Dimensionality; Approximation Algorithms**

I. Introduction

Nearest neighbor search is an essential part of many machine learning algorithms. Often, it is also the most time-consuming stage of the procedure, see [1]. In application areas, such as in recommendation systems, robotics and computer vision, where fast response times are critical, using brute force linear search is often not feasible. This problem is further magnified by the availability of inreasingly complex, high-dimensional data sources.

Applications that require frequent k-NN queries from large data sets are for example object recognition [2], [3], shape recognition [4] and image completion [5] using large databases of image descriptors, and content-based web recommendation systems [6].

Consequently, there is a vast body of literature on the algorithms for fast nearest neighbor search. These algorithms can be divided into exact and approximate nearest neighbor search. While exact nearest neighbor search algorithms return the true nearest neighbors of the query point, they suffer from the curse of dimensionality: their performance degrades when the dimension of the data increases, rendering them no better than brute force linear search in the high-dimensional regime.

In approximate nearest neighbor search, the main interest is the tradeoff between query time and accuracy, which can be measured either in terms of distance ratios or the probability of finding the true nearest neighbors. Several different approaches have been proposed and their implementations are commonly used in practical applications.

In this section, we first define the approximate nearest neighbor search problem, then review the existing approaches to it, and finally outline a new method that avoids the main drawbacks of existing methods.

A. Approximate nearest neighbor search

Consider a metric space $\mathcal{M}$, a data set $\mathbf{X} = (\mathbf{x}_1, \ldots, \mathbf{x}_n) \subseteq \mathcal{M}$, a query point $\mathbf{q} \in \mathcal{M}$, and a distance metric $m : \mathcal{M}^2 \to \mathbb{R}$. The k-nearest neighbor (k-NN) search is the task of finding the k closest (w.r.t. m) points to $\mathbf{q}$ from the data set $\mathbf{X}$, i.e., find a set $\mathbf{K} \subseteq \mathbf{X}$ for which it holds that $|\mathbf{K}| = k$ and

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

for all $\mathbf{x} \in \mathbf{K}$, $\mathbf{y} \in \mathbf{X} \setminus \mathbf{K}$.

In this work, we consider applications where $\mathcal{M}$ is the d-dimensional Euclidean space $\mathbb{R}^d$, and m is Euclidean distance

$$m(\mathbf{x}, \mathbf{y}) = \sqrt{\sum_{i=1}^{d} (x_i - y_i)^2}.$$

An approximate nearest neighbor search algorithm can be divided into two separate phases: an offline phase, and an online phase. In the offline phase an index is built, and the online phase refers to completing fast nearest neighbor queries using the constructed index. In practical applications, the most critical considerations are the accuracy of the approximation (the definition of the accuracy and the required accuracy level depending on the application), and the query time needed to reach it.

As a measure of accuracy, we use recall, which is relevant especially for information retrieval and recommendation settings. It is defined as the proportion of true k-nearest neighbors of the query point returned by the algorithm:

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

Here $\mathbf{A}$ is a set of k approximate nearest neighbors returned by the algorithm, and $\mathbf{K}$ is the set of true k nearest neighbors of the query point.

Of secondary importance are the index construction time in the offline stage and the memory requirement, which are usually not visible to the user but must be feasible. Note that in many applications the index must be updated regularly when we want to add new data points into the set from where the nearest neighbors are searched from.

B. Related work

Most effective methods for approximate nearest neighbor search in high-dimensional spaces can be classified into either hashing, graph, or space-partitioning tree based strategies.

1) Hashing based algorithms: The most well-known and effective hashing based algorithms are variants of locality-sensitive hashing (LSH) [7]–[9]. In LSH, several hash functions of the same type are used. These hash functions are *locality-sensitive*, which means that nearby points fall into the same hash bucket with higher probability than the points that are far away from each other. When answering a k-NN query, the query point is hashed with all the hash functions, and then a linear search is performed in the set of data points in the buckets where the query point falls into. Multi-probe LSH [10]–[12] improves the efficiency of LSH by searching also the nearby hash buckets of the bucket a query point is hashed into. Thus a smaller amount of hash tables is required to obtain a certain accuracy level. The choice of the hash function affects the performance and needs to be done carefully in each case.

2) Graph based algorithms: Approximate graph based methods such as [13]–[15] build a k-NN-graph of the data set in an offline phase, and then utilize it to do fast k-NN queries. For example in [15] several graphs of random subsets of the data are used to approximate the exact k-NN-graph of the whole data set. The data set is divided hierarchically and randomly into subsets, and the k-NN-graphs of these subsets are built. This process is then repeated several times, and the resulting graphs are merged and the final graph is utilized to answer nearest neighbor queries. Graph based methods are in general efficient, but the index construction is slow for larger data sets.

3) Space-partitioning trees: The first space partitioning-tree based strategy proposed for nearest neighbor search was the k-d tree [16], which divides the data set hierarchically into cells that are aligned with the coordinate axes. Nearest neighbors are then searched by performing a backtracking or priority search in this tree. k-d trees and other space-partitioning trees can be utilized in approximate nearest neighbor search by terminating the search when a predefined number of data points is checked, or all the points within some predefined error bound from the query point are checked [17].

Instead of the hyperplanes, clustering algorithms can be utilized to partition the space hierarchically; for example the k-means algorithm is used in k-means trees [1], [18]. Other examples of clustering based partitioning trees are cover trees [19], VP trees [20] and ball trees [21]. Similarly to graph-based algorithms, clustering based variants have the drawback of long index construction times due to hierarchical clustering on large data sets.

An efficient scheme to increase the accuracy of space-partitioning trees is to utilize several parallel randomized space-partitioning trees. Randomized k-d trees [22], [23] are grown by choosing a split direction at random from dimensions of the data in which it has the highest variance. Nearest neighbor queries are answered using priority search; a single priority queue, which is ordered by distance of the query point from splitting point in the splitting direction, is maintained for all the trees.

Another variant of randomized space-partitioning tree is the random projection tree (RP tree), which was first proposed for vector quantization in [24], and later modified for approximate nearest neighbor search in [25]. In random projection trees the splitting hyperplanes are aligned with the random directions sampled from the unit sphere instead of the coordinate axes. Nearest neighbor queries are answered using *defeatist search*: first the query point is routed down in several trees, and then a brute force linear search is performed in the union of the points of all the leaves the query point fell into.

C. MRPT algorithm

One of the strengths of randomized space-partitioning trees is in their relative simplicity compared to hashing and graph based methods: they have only few tunable parameters, and their performance is quite robust with respect to them. They are also perfectly suitable for parallel implementation because the trees are independent, and thus can be easily parallelized either locally or over the network with minimal communication overhead.

However, both of the aforementioned variants of randomized space-partitioning trees have a few weak points which limit their efficiency and scalability. First, randomization used in randomized k-d trees does not make the trees sufficiently decorrelated to reach high levels of accuracy efficiently. Randomization used in RP trees is sufficient, but it comes at high computational cost: computation of random projections is time-consuming for high-dimensional data sets because the random vectors have the same dimension as the data.

Second, both the priority queue search used in [22], [23] and the defeatist search used in [25] require checking a large proportion of the data points to reach high accuracy levels; this leads to slow query times. Third, because each node of an RP tree has its own random vector, the number of random vectors required is exponential with respect to tree depth. In the high-dimensional case this on the one hand increases the memory required by the index unnecessarily, and on the other hand slows down the index construction process.

Because of the first two reasons the query times of the randomized space-partitioning trees are not competitive with the fastest graph based methods (cf. experimental results in Section IV).

In this article we propose a method that uses multiple random projection trees (MRPT); it incorporates two additional features that lead to fast query times and accurate results. The algorithm has the aforementioned strengths of randomized space-partitioning trees but avoids their drawbacks.

More specifically, our contributions are:

1) We show that sparse random projections can be used to obtain a fast variant of randomized space-partitioning trees.
2) We propose the MRPT algorithm that combines multiple trees by *voting search* as a new and more efficient method to utilize randomized space-partitioning trees for approximate nearest neighbor search.
3) We present a time and space complexity analysis of our algorithm.
4) We demonstrate experimentally that the MRPT algorithm with sparse projections and voting search outperforms the state-of-the-art methods using several real-world data sets across a wide range of sample size and dimensionality.

In the following sections we describe the proposed method by first revisiting the classic RP tree method [25], and then describing the novel features that lead to increased accuracy and reduced memory footprint and query time.

II. Index construction

A. Classic random projection trees

At the root node of the tree, the points are projected onto a random vector $\mathbf{r}$ generated from the d-dimensional standard normal distribution $N_d(\mathbf{0}, \mathbf{I})$. The data set is then split into two child nodes using the median of the points in the projected space: points whose projected values are less or equal to the median in the projected space are routed into the left child node, and points with projected values greater than the median are routed into the right child node. This process is then repeated recursively for the subsets of the data in both of the child nodes, until a predefined depth ℓ is met.

For high-dimensional data, the computation of random projections is slow, and the memory requirement for storing the random vectors is high. To reduce both the computation time and the memory footprint, we propose two improvements:

- Instead of using dense vectors sampled from the d-dimensional normal distribution, we use sparse vectors to reduce both storage and computation complexity.
- Instead of using different projection vectors for each intermediate node of a tree, we use one projection vector for all the nodes on the same level of a tree, so that we can further reduce the storage requirement and maximize low-level parallelism through vectorization.

In addition, instead of splitting at a fractile point chosen at random from the interval $[\frac{1}{4}, \frac{3}{4}]$, as suggested in [25], we split at the median to make the performance more predictable and to enable saving the trees in a more compact format.

In the following subsections we briefly discuss the details of the above improvements. Pseudocode for constructing the index using sparse RP trees is given in detail in Algorithms 1–2 below. The proposed Algorithm 1 consists of three embedded **for** loops. The outermost loop (line 4 - 15) builds T RP trees by continuously calling the `GROW_TREE` function in Algorithm 2. For each individual RP tree, the two inner **for** loops (line 6 - 12) prepare a sparse random matrix $\mathbf{R}$ which will be used for projecting the data set $\mathbf{X}$ to $\mathbf{P}$ using ℓ random vectors (at line 13). Specifically, the innermost loop is for constructing a d-dimensional sparse vector for a given tree level. In Algorithm 2, the `GROW_TREE` function constructs a binary search tree by recursively splitting the high-dimensional space $\mathbf{X}$ into sub-spaces using the previous projection $\mathbf{P}$.

Algorithm 1 Grow T RP trees.

```
1:  function GROW_TREES(X, T, ℓ, a)
2:      n ← X.nrows
3:      let trees[1...T] be a new array
4:      for t in 1,...,T do
5:          let R be a new d × ℓ matrix
6:          for level in 1,...,ℓ do
7:              for i in 1,...,d do
8:                  generate z from Bernoulli(a)
9:                  if z = 1 then
10:                     generate R[i, level] from N(0, 1)
11:                 else
12:                     R[i, level] ← 0
13:         P ← XR
14:         trees[t].root ← GROW_TREE(X, [1...n], 0, P)
15:         trees[t].random_matrix ← R
16:     return trees
```

Algorithm 2 Grow a single RP tree.

```
1:  function GROW_TREE(X, indices, level, P)
2:      if level = ℓ then
3:          return X[indices, ] as leaf
4:      proj ← P[indices, level]
5:      split ← median(proj)
6:      left_indices ← indices[proj ≤ split]
7:      right_indices ← indices[proj > split]
8:      left ← GROW_TREE(X, left_indices, level + 1, P)
9:      right ← GROW_TREE(X, right_indices, level + 1, P)
10:     return split, left, right
```

B. Sparse random projections

With high-dimensional data sets, computation of the random vectors easily becomes a bottleneck on the performance of the algorithm. However, it is not necessary to use random vectors sampled from the d-dimensional standard normal distribution to approximately preserve the pairwise distances between the data points. Achlioptas [26] shows that the approximately distance-preserving low-dimensional embedding of Johnson-Lindenstrauss-lemma is obtained also with sparse random vectors with components sampled from $\{-1, 0, 1\}$ with respective probabilities $\{\frac{1}{6}, \frac{2}{3}, \frac{1}{6}\}$. Li *et al.* [27] prove that the same

components with respective probabilities $\{\frac{1}{2\sqrt{d}}, 1-\frac{1}{\sqrt{d}}, \frac{1}{2\sqrt{d}}\}$, where d is the dimension of the data, can be used to obtain a $\sqrt{d}$-fold speed-up without significant loss in accuracy compared to using normally distributed random vectors.

We use sparse random vectors $\mathbf{r} = (r_1, \ldots, r_d)$, whose components are sampled from the standard normal distribution with probability a, and are zeros with probability $1-a$:

$$r_i = \begin{cases} N(0,1) & \text{with probability } a \\ 0 & \text{with probability } 1-a. \end{cases}$$

The sparsity parameter a can be tuned to optimize performance but we have observed that $a = \frac{1}{\sqrt{d}}$ recommended in [27] tends to give near-optimal results in all the data sets we tested, which suggests that further fine-tuning of this parameter is unnecessary. This proportion is small enough to provide significant computational savings through the use of sparse matrix libraries.

C. Compactness and speed with fewer vectors

In classic RP trees, a different random vector is used at each inner node of a tree, whereas we use the same random vector for all the sibling nodes of a tree. This choice does not affect the accuracy at all because a query point is routed down each of the trees only once; hence, the query point is projected onto a random vector $\mathbf{r}_i$ sampled from the same distribution at each level of a tree. This means that the query point is projected onto i.i.d. random vectors $\mathbf{r_1}, \ldots, \mathbf{r}_\ell$ in both scenarios.

An RP tree has $2^\ell - 1$ inner nodes; therefore, if each node of a tree had a different random vector as in classic RP trees, $2^\ell - 1$ different random vectors would be required for one tree. However, when a single vector is used on each level, only ℓ vectors are required. This reduces the amount of memory required by the random vectors from exponential to linear with respect to the depth of the trees.

Having only ℓ random vectors in one tree also speeds up the index construction significantly. While some of the observed speed-up is explained by a decreased amount of the random vectors that have to be generated, mostly it is due to enabling the computation of all the projections of the tree in one matrix multiplication: the projected data set $\mathbf{P} \in \mathbb{R}^{n \times \ell}$ can be computed from the data set $\mathbf{X} \in \mathbb{R}^{n \times d}$ and a random matrix $\mathbf{R} \in \mathbb{R}^{d \times \ell}$ as

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

Although the total amount of computation stays the same, in practice this speeds up the index construction significantly due to the cache effects and low-level parallelization through vectorization.

D. Time and space complexity: Index construction

At each level of each tree the whole data set is projected onto a d-dimensional random vector that has on average ad non-zero components, so the expected[1] index construction time is $\Theta(T\ell nad)$. For classic RP trees ($a = 1$) this is $\Theta(T\ell nd)$, but for sparse trees ($a = \frac{1}{\sqrt{d}}$) this is only $\Theta(T\ell n\sqrt{d})$.

For each tree, we need to store the points allocated to each leaf node, so the memory required by the index is at least $\mathcal{O}(Tn)$. At each node only a split point is saved; this does not increase the space complexity because there are only $2^\ell - 1 < n$ inner nodes in one tree.

The expected amount of memory required by one random vector is $\Theta(ad)$ when random vectors are saved in sparse matrix form. This is $\Theta(d)$ for dense RP trees, and $\Theta(\sqrt{d})$ for sparse RP trees with the sparsity parameter fixed to $a = \frac{1}{\sqrt{d}}$. Because an RP tree has $2^\ell - 1$ inner nodes, the memory requirement for T classic RP trees, which have a different random vector for each node of a tree, is $\mathcal{O}(Tdn)$. However, in our version, in which a single vector is used on each level, there are only ℓ vectors; hence, the memory requirement for T sparse RP trees is $\mathcal{O}\left(T(\sqrt{d}\log n + n)\right)$.

III. Query phase

In many approximate nearest neighbor search algorithms the query phase is further divided into two steps: a candidate generation step, and an exact search step.

In the candidate generation step, a candidate set S, for which usually $|S| \ll n$, is retrieved from the whole data set, and then in the exact search step k approximate nearest neighbors of a query point are retrieved by performing a brute force linear search in the candidate set. In the MRPT algorithm, the candidate generation step consists of traversal of T trees grown in the index construction phase.

The leaf to which a query point $\mathbf{q}$ belongs to is retrieved by first projecting $\mathbf{q}$ at the root node of the tree onto the same random vector as the data points, and then assigning it into the left or right branch depending on the value of the projection. If it is smaller than or equal to the cutpoint s (median of the data points belonging to that node in the projected space) saved at that node, i.e.

$$\mathbf{q}^T\mathbf{r}_i \leq s,$$

the query point is routed into the left child node, and otherwise into the right child node. This process is then repeated recursively until a leaf is met.

The query point is routed down into a leaf in all the T trees obtained in the index construction phase. The query process is thus far similar to the one described in [25]. The principal difference is the candidate set generation: in classic RP trees, the candidate set S includes all the points that belong to the same leaf with the query point in at least one of the trees.

A problem with this approach is that when a high number of trees are used in the tree traversal step, the size of the candidate set $|S|$ becomes excessively large.

In the following, we show how the frequency information (i.e., how frequently a point falls into the same cell as query point) can be utilized to improve both query performance and accuracy.

[1]The following complexity results hold exactly if the algorithm is modified so that each random vector has exactly $\lceil ad \rceil$ non-zero components instead of the expected number of non-zero components being ad.

A. Voting search

Assume that we have constructed T RP trees of depth ℓ. Each of them partitions $\mathbb{R}^d$ into 2^ℓ cells (leaves) $L_1, \dots L_{2^\ell}$, all of which contain $\lceil \frac{n}{2^\ell} \rceil$ or $\lfloor \frac{n}{2^\ell} \rfloor$ data points. For $1 \leq t \leq T$, let f_t be an indicator function of data point $\mathbf{x} \in \mathbf{X}$ and the query $\mathbf{q}$, which returns 1, if $\mathbf{x}$ and $\mathbf{q}$ reside in the same cell in tree t, and 0 otherwise:

$$f_t(\mathbf{x}; \mathbf{q}) = \sum_{m=1}^{2^\ell} \mathbb{1}\{\mathbf{x} \in L_m, \mathbf{q} \in L_m\}.$$

Further, let F be a count function of data point $\mathbf{x}$, which returns the number of trees in which $\mathbf{x}$ and $\mathbf{q}$ belong to the same leaf:

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

We propose a simple but effective *voting search* where we choose into the candidate set only data points residing in the same leaf as the query point in at least v trees:

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

The vote threshold v is a tuning parameter. A lower threshold value yields higher accuracy at the expense of increased query times.

This further pruning of the candidate set utilizes the intuitive notion that the closer the data point is to the query point, the more probably an RP tree divides the space so that they both belong to the same leaf. We emphasize that our voting scheme is not restricted to RP trees, but it can be used in combination with any form of space-partitioning algorithms, given that there is enough randomness involved in the process to render the partitions sufficiently independent.

Pseudocode for the online stage of the MRPT algorithm is given in detail in Algorithms 3–4 below (the knn($\mathbf{q}, k, S$) function is a regular k-NN search which returns k nearest neighbors for the point $\mathbf{q}$ from the set S).

Algorithm 3 Route a query point into a leaf in an RP tree.

```
function TREE_QUERY(q, tree)
    R ← tree.random_matrix
    p ← q^T R
    root ← tree.root
    for level in 1, ..., ℓ do
        if p[level] ≤ root.split then
            root ← root.left
        else
            root ← root.right
    return data points in root
```

B. Time and space complexity: Query execution

When the trees are grown into some predetermined depth ℓ using the median split, the expected running time of the tree traversal step is $\Theta(T\ell ad)$ because at each level of each RP tree the query point is projected onto a d-dimensional random vector that has on average ad non-zero components. When using random vectors sampled from the d-dimensional standard normal distribution, this is $\Theta(T\ell d)$, but when using value $a = \frac{1}{\sqrt{d}}$ for the sparsity parameter, it is only $\Theta(T\ell\sqrt{d})$.

When the median split is used, each leaf has either $\lceil \frac{n}{2^\ell} \rceil$ or $\lfloor \frac{n}{2^\ell} \rfloor$ data points. Therefore, the size of the final search set satisfies $|S| \leq T\lceil \frac{n}{2^\ell} \rceil$ for both defeatist and voting search.

Because of this upper bound for the size of the candidate set, the running time of the exact search in the candidate set is $\mathcal{O}\left(T\frac{n}{2^\ell}d\right)$ for both defeatist search and voting search. Thus, the total query time is $\mathcal{O}\left(Td(\ell + \frac{n}{2^\ell})\right)$ for classic RP trees, and $\mathcal{O}\left(Td(\frac{\ell}{\sqrt{d}} + \frac{n}{2^\ell})\right)$ for the MRPT algorithm.

Algorithm 4 Approximate k-NN search using multiple RP trees.

```
function APPROXIMATE_KNN(q, k, trees, v)
    S ← ∅
    let votes[1 ... X.nrows] be a new array
    for tree in trees do
        for point in TREE_QUERY(q, tree) do
            votes[point] ← votes[point] + 1
            if votes[point] = v then
                S ← S ∪ {point}
    return knn(q, k, S)
```

TABLE I: Time and space complexity of the algorithm compared to classic RP trees.

	RP trees	MRPT $\left(a = \frac{1}{\sqrt{d}}\right)$
Query time	$\mathcal{O}\left(Td(\ell + \frac{n}{2^\ell})\right)$	$\mathcal{O}\left(Td(\frac{\ell}{\sqrt{d}} + \frac{n}{2^\ell})\right)$
Index construction time	$\Theta(T\ell nd)$	$\Theta(T\ell n\sqrt{d})$
Index memory	$\mathcal{O}(Tdn)$	$\mathcal{O}\left(T(\sqrt{d}\log n + n)\right)$

IV. Experimental results

We assess the efficiency of our algorithm by comparing it against state-of-the-art approximate nearest neighbor search algorithms. To make the comparison as practically relevant as possible, we chose widely used methods that are available in optimized libraries.

All of the compared libraries, including ours, are implemented in C++ and compiled with similar optimizations. We make our comparison and the implementation of our algorithm available as open-source[2].

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

We compare the MRPT algorithm to representatives from all three major groups of approximate nearest neighbor search algorithms: tree , graph and hashing based methods (cf. Table

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

TABLE II: Data sets used in the experiments

Data set	n	d	type
GIST	1000000	960	image descriptors
SIFT	2500000	128	image descriptors
MNIST	60000	784	image
Trevi	101120	4096	image
STL-10	100000	9216	image
News	262144	1000	text (TF-IDF + LSA)
Random	50000	4096	synthetic

TABLE III: Algorithms tested

Algorithm	Library	type
K-d tree	ANN	tree
Randomized k-d trees	FLANN	tree
K-means tree	FLANN	tree
NN-descent	KGraph	graph
Multi-probe LSH	Falconn	hash

III). In addition, we included our own implementation of classic RP-trees and sparse RP-trees to test the efficacy of our modifications.

A. Results

In the experiments we used three different values of k that cover the range used in typical applications: $k = 1, 10$ and 100. However, due to space restrictions, we present results only for $k = 10$ (the results for $k = 1$ and $k = 100$ are available in the supplementary material on GitHub).

Figure 1 shows results for $k = 10$ on six data sets. The times required to reach a given recall level are shown in Table IV. The MRPT algorithm is significantly faster than other tree-based methods and LSH on all of the data sets, except SIFT. It is worth noting that SIFT has a much lower dimension ($d = 128$) than the other data sets. These results suggest that the proposed algorithm is more suitable for high-dimensional data sets.

The performance of the MRPT algorithm is also superior to classic RP trees on all of the data sets. Of our two main contributions, voting seems the be more important than sparsity with respect to query times: sparse RP trees are only slightly faster than dense RP trees, the gap being somewhat larger on the higher-dimensional data sets, but the voting search provides a marked improvement on all data sets. This shows that the voting search, especially when combined with sparsity, is an efficient way to reduce query time without compromising accuracy.

The numerical values in Table IV indicate that for recall levels $\geq 90\%$, the MRPT method is the fastest in 14 out of 21 instances, while the KGraph method is fastest in 7 out 21 cases. Compared to brute force search, MRPT is roughly 25–100 times faster on all six real-world data sets at 90% recall level, and roughly 10–40 times faster even at the highest 99% recall level.

V. Conclusion

We propose the multiple random projection tree (MRPT) algorithm for approximate k-nearest neighbor search in high dimensions. The method is based on combining multiple sparse random projection trees using a novel voting scheme where the final search is focused to points occurring most frequently among points retrieved by multiple trees. The algorithm is straightforward to implement and exploits standard fast linear algebra libraries by combining calculations into large matrix–matrix operations.

We demonstrate through extensive experiments on both real and simulated data that the proposed method is faster than state-of-the-art space-partitioning tree and hashing based algorithms on a variety of accuracy levels. It is also faster than a leading graph based algorithm (KGraph) on high accuracy levels, while being slightly slower on low accuracy levels. The good performance of MRPT is especially pronounced for high-dimensional data sets.

Due to its very competitive and consistent performance, and simple and efficient index construction stage — especially compared to graph-based algorithms — the proposed MRPT method is an ideal method for a wide variety of applications where high-dimensional large data sets are involved.

Acknowledgements

This work was supported in part by the Finnish Funding Agency for Innovation (Project SPA), the Academy of Finland (Centre-of-Excellence COIN), and the DoCS graduate school of the University of Helsinki.

References

[1] M. Muja and D. G. Lowe, "Scalable nearest neighbor algorithms for high dimensional data," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 36, no. 11, pp. 2227–2240, 2014.

[2] D. Nister and H. Stewenius, "Scalable recognition with a vocabulary tree," in *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, vol. 2. IEEE, 2006, pp. 2161–2168.

[3] D. G. Lowe, "Distinctive image features from scale-invariant keypoints," *International journal of computer vision*, vol. 60, no. 2, pp. 91–110, 2004.

[4] Y. Amit and D. Geman, "Shape quantization and recognition with randomized trees," *Neural computation*, vol. 9, no. 7, pp. 1545–1588, 1997.

[5] J. Hays and A. A. Efros, "Scene completion using millions of photographs," in *ACM Transactions on Graphics (TOG)*, vol. 26, no. 3. ACM, 2007, p. 4.

[6] L. Wang, S. Tasoulis, T. Roos, and J. Kangasharju, "Kvasir: Scalable provision of semantically relevant web content on big data framework," *IEEE Transactions on Big Data*, vol. Advance online publication. DOI:10.1109/TBDATA.2016.2557348, 2016.

[7] P. Indyk and R. Motwani, "Approximate nearest neighbors: towards removing the curse of dimensionality," in *Proceedings of the thirtieth annual ACM symposium on Theory of computing*. ACM, 1998, pp. 604–613.

[8] A. Gionis, P. Indyk, R. Motwani *et al.*, "Similarity search in high dimensions via hashing," in *VLDB*, vol. 99, no. 6, 1999, pp. 518–529.

[9] A. Andoni and P. Indyk, "Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions," in *Foundations of Computer Science, 2006. FOCS'06. 47th Annual IEEE Symposium on*. IEEE, 2006, pp. 459–468.

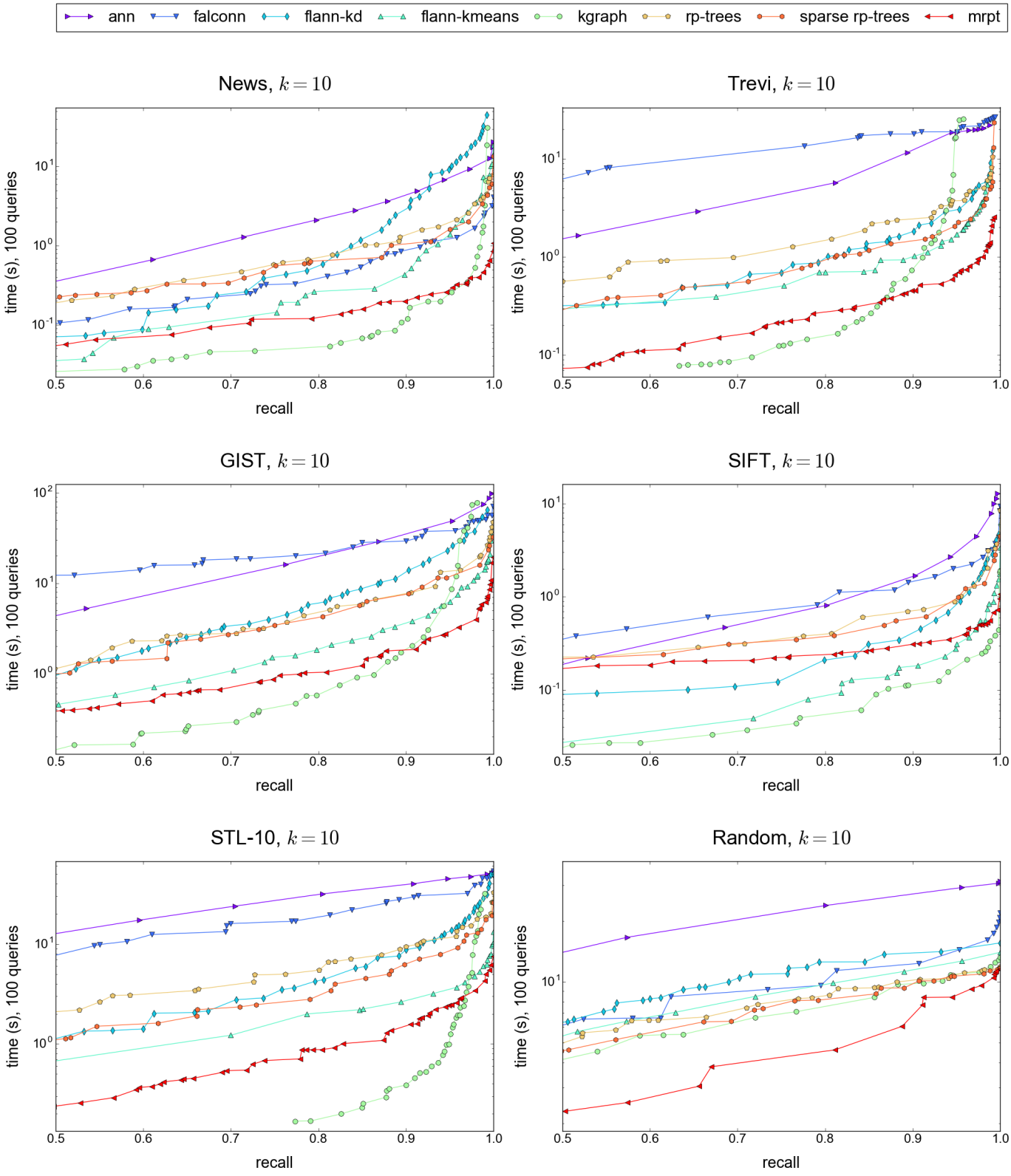

Fig. 1: Query time (log scale) for 100 queries versus recall on six different data sets with $k = 10$. The compared methods are: k-d tree (ann), multi-probe LSH (falconn), randomized k-d tree (flann-kd), hierarchical k-means tree (flann-kmeans), k-NN graph (kgraph) and our method (mrpt). We also include results for our implementation of classic RP-trees (rp trees) and RP-trees with sparsity (sparse rp trees).

TABLE IV: 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. MRPT consistently outperforms other algorithms at high recall levels. For example, for R $\geq$ 95%, 71.4% of times MPRT achieves the fastest query time (i.e., 10 out of 14) whereas it is only 28.6% for KGraph (i.e., 4 out of 14). *) R = 100% by definition for brute force search.

		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**	

[10] Q. Lv, W. Josephson, Z. Wang, M. Charikar, and K. Li, "Multi-probe lsh: efficient indexing for high-dimensional similarity search," in *Proceedings of the 33rd international conference on Very large data bases*. VLDB Endowment, 2007, pp. 950–961.

[11] W. Dong, Z. Wang, W. Josephson, M. Charikar, and K. Li, "Modeling lsh for performance tuning," in *Proceedings of the 17th ACM conference on Information and knowledge management*. ACM, 2008, pp. 669–678.

[12] A. Andoni, P. Indyk, T. Laarhoven, I. Razenshteyn, and L. Schmidt, "Practical and optimal lsh for angular distance," in *Advances in Neural Information Processing Systems 28*. Curran Associates, Inc., 2015, pp. 1225–1233.

[13] K. Hajebi, Y. Abbasi-Yadkori, H. Shahbazi, and H. Zhang, "Fast approximate nearest-neighbor search with k-nearest neighbor graph," in *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, vol. 22, no. 1, 2011, p. 1312.

[14] W. Dong, C. Moses, and K. Li, "Efficient k-nearest neighbor graph construction for generic similarity measures," in *Proceedings of the 20th international conference on World wide web*. ACM, 2011, pp. 577–586.

[15] J. Wang, J. Wang, G. Zeng, Z. Tu, R. Gan, and S. Li, "Scalable k-nn graph construction for visual descriptors," in *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*. IEEE, 2012, pp. 1106–1113.

[16] J. L. Bentley, "Multidimensional binary search trees used for associative searching," *Communications of the ACM*, vol. 18, no. 9, pp. 509–517, 1975.

[17] S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu, "An optimal algorithm for approximate nearest neighbor searching fixed dimensions," *Journal of the ACM (JACM)*, vol. 45, no. 6, pp. 891–923, 1998.

[18] K. Fukunaga and P. M. Narendra, "A branch and bound algorithm for computing k-nearest neighbors," *IEEE transactions on computers*, vol. 100, no. 7, pp. 750–753, 1975.

[19] A. Beygelzimer, S. Kakade, and J. Langford, "Cover trees for nearest neighbor," in *Proceedings of the 23rd international conference on Machine learning*. ACM, 2006, pp. 97–104.

[20] P. N. Yianilos, "Data structures and algorithms for nearest neighbor search in general metric spaces," in *SODA*, vol. 93, no. 194, 1993, pp. 311–21.

[21] B. Leibe, K. Mikolajczyk, and B. Schiele, "Efficient clustering and matching for object class recognition." in *BMVC*, 2006, pp. 789–798.

[22] C. Silpa-Anan and R. Hartley, "Optimised kd-trees for fast image descriptor matching," in *Computer Vision and Pattern Recognition, 2008. CVPR 2008. IEEE Conference on*. IEEE, 2008, pp. 1–8.

[23] M. Muja and D. G. Lowe, "Fast approximate nearest neighbors with automatic algorithm configuration." *VISAPP (1)*, vol. 2, pp. 331–340, 2009.

[24] S. Dasgupta and Y. Freund, "Random projection trees for vector quantization," *IEEE Transactions on Information Theory*, vol. 55, no. 7, pp. 3229–3242, 2009.

[25] S. Dasgupta and K. Sinha, "Randomized partition trees for nearest neighbor search," *Algorithmica*, vol. 72, no. 1, pp. 237–263, 2015.

[26] D. Achlioptas, "Database-friendly random projections," in *Proceedings of the twentieth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*. ACM, 2001, pp. 274–281.

[27] P. Li, T. J. Hastie, and K. W. Church, "Very sparse random projections," in *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 2006, pp. 287–296.