High-dimensional approximate nearest neighbor: k -d Generalized Randomized Forests

Yannis Avrithis, Ioannis Z. Emiris, Giorgos Samaras

High-dimensional approximate nearest neighbor: *k*-d Generalized Randomized Forests

Yannis Avrithis*, Ioannis Z. Emiris*, and Georgios Samaras

Dept of Informatics & Telecommunications, University of Athens, Greece
iavr@image.ntua.gr,emiris@di.uoa.gr,georgesamarasdit@gmail.com

Abstract. We propose a new data-structure, the generalized randomized k-d forest, or k-d GeRaF, for approximate nearest neighbor searching in high dimensions. In particular, we introduce new randomization techniques to specify a set of independently constructed trees where search is performed simultaneously, hence increasing accuracy. We omit backtracking, and we optimize distance computations, thus accelerating queries. We release public domain software `GeRaF` and we compare it to existing implementations of state-of-the-art methods including BBD-trees, Locality Sensitive Hashing, randomized k-d forests, and product quantization. Experimental results indicate that our method would be the method of choice in dimensions around 1,000, and probably up to 10,000, and pointsets of cardinality up to a few hundred thousands or even one million; this range of inputs is encountered in many critical applications today. For instance, we handle a real dataset of 10^6 images represented in 960 dimensions with a query time of less than 1sec on average and 90% responses being true nearest neighbors.

Keywords: data-structure, randomized tree, space partition, geometric search, open software, practical complexity

1 Introduction

After a couple of decades of work, Nearest Neighbor Search remains a fundamental optimization problem with both theoretical and practical open issues today, in particular for large datasets in dimension well above 100. An exact solution using close to linear space and sublinear query time is impossible, hence the importance of approximate search, abbreviated *NNS*. We focus on the Euclidean metric but extensions to other metrics should be possible.

Definition 1 *Given a finite dataset $X \subset \mathbb{R}^d$ and real $\epsilon > 0$, $x^* \in X$ is an ϵ-approximate nearest neighbor of query $q \in \mathbb{R}^d$, if $dist(q, x^*) \leq (1+\epsilon) dist(q, x)$ for all $x \in X$. For $\epsilon = 0$, this reduces to exact NNS.*

Despite a number of sophisticated methods available, it is still open which is best for various ranges of the input parameters. Here, we propose a practical data-structure, generalizing k-d trees, which should be the method of choice in dimension roughly in the range of 1,000 to 10,000 and inputs of a few hundred thousand points, and up to a million. By taking advantage of randomization and new algorithmic ideas, we offer a very competitive open software for (approximate) NNS in this range of inputs, which provides a good trade-off between accuracy and speed. Our work also sheds light into the efficiency of k-d trees, which is one of the most common data structures but whose complexity analysis is far from tight.

High-dimensional NNS arises naturally when complex objects are represented by vectors of d scalar features. NNS tends to be one of the most computationally expensive parts of many algorithms in a variety of applications, including computer vision, knowledge discovery and data mining, pattern recognition and classification, machine learning, data compression, multimedia databases, document retrieval and statistics [7,11,13,16,17]. Large scale problems are quite common in such areas, for instance more than 10^7 points and more than 10^5 dimensions [12].

* The first two authors are partially supported by the European Social Fund and Greek National Fund through the National Strategic Reference Framework, research funding program "ARISTEIA", project "ESPRESSO".

Previous work. There are many efficient approaches to NNS. We focus on the most competitive ones, with emphasis on practical performance.

An important class of methods consists in data-dependent methods, where the decisions taken for space partitioning are based on the given data points. The Balanced Box Decomposition (BBD) tree [3] is a variant of the quadtree, closely related to the fair-split tree. It has $O(\log n)$ height, and subdivides space into axis-aligned hyper-rectangles, containing one or more points with bounded aspect ratio. It achieves query time $O(d^{d+1} \log n/\epsilon^d)$, using space in $O(dn)$, and preprocessing time in $O(dn \log n)$. The implementation in library `ANN`[1] seems to be the most competitive method for roughly $d < 100$. Recently, a novel dimensionality reduction method has been combined with BBD-trees to yield NNS with optimal space requirements and sublinear query time [1].

The performance of BBD-trees in practice is comparable to that of k-d trees. The latter lack a tight analysis but it is known that search becomes almost linear in n for large d because of backtracking. Randomization is a powerful idea: in [14], a random isometry is used with k-d trees; in [15], tree height is analyzed under random rotations; Random Projection trees [5] take another tack. R-trees and their variants are most frequent in database applications: they are comparable in performance to k-d trees, but lack complexity and error bounds.

k-d trees are probably the most common data-structure for NNS, having implementations in libraries `ANN`, with performance comparable to BBD-trees, and `CGAL`, which is competitive only for small inputs. A successful contribution has been library `FLANN` [10,11], considered state-of-the-art for d about 100; the method has been most successful on SIFT image descriptors with $d = 128$. `FLANN`[2] constructs a forest of up to 6 randomized k-d trees and performs simultaneous search in all trees. It chooses the split coordinates adaptively but all leaves contain a single point. The implementation adopts some optimization techniques, such as unrolling the loop of distance computation, but our software goes significantly further in this direction.

In high dimensional space, tree-based data structures are affected by the curse of dimensionality, i.e., either the running time or the space requirement grows exponentially in d. An important method conceived for high dimensional data is Locality Sensitive Hashing (LSH). LSH induces a data independent space partition and is dynamic, since it supports insertions and deletions. The basic idea of LSH is to hash the points of the data set so as to ensure that the probability of collision is much higher for objects that are close to each other than for those that are far apart. The existence of such hash functions depends on the metric space. In general, LSH requires roughly $O(dn^{1+\rho})$ space and $O(dn^{\rho})$ query time for some $\rho \in (0, 1)$. It is known [2] that in the Euclidean case, it is possible to bound ρ by $\rho \leq 1/(1 + \epsilon)^2$. One implementation that we use for comparisons is in library `E`^{2}`LSH`[3].

A different hashing approach is to represent points by short binary codes to approximate and accelerate distance computations. Recent research on learning such codes from data distributions is very active [16]. A more general approach is to use any discrete representation of points, again learned from data points. A popular approach is product quantization (PQ) [7], which both compresses data points and provides for fast asymmetric distance computations, where points remain compressed but queries are not. A powerful non-exhaustive search method inspired by PQ is the inverted multi-index [4]. There are several recent extensions, and the current state of the art in up to 10^9 points in 128 dimensions is locally optimized product quantization (LOPQ) [8].

Contribution. Our main contribution is to propose a new, randomized data-structure for NNS, namely the k-d *Generalized Randomized Forest* (k-d GeRaF), which generalizes the k-d tree in order to perform fast and accurate NNS in high dimensions and dataset cardinality in thousands or millions. We employ adaptive and randomized algorithms for choosing the split coordinate, and further randomization techniques to build a number of independent k-d trees. We also pro-

[1] `http://cs.umd.edu/~mount/ANN`
[2] `http://cs.ubc.ca/research/flann`
[3] `http://www.mit.edu/~andoni/LSH`

vide automatic configuration of the parameters governing tree construction and search. All trees are searched simultaneously, with no need for backtracking. We examine alternative ideas, such as random shuffling of the points, random isometries, leaves with several points, and methods for accelerating distance computation. By keeping track of encountered points, we avoid repeated computations [11].

We have implemented all of the above techniques within a public domain C++ software, `GeRaF`. This has also allowed us to experiment with different alternatives and provide a simple yet effective automatic parameter configuration. We compare to the main existing alternative libraries on a number of synthetic and real datasets of varying dimensionality and cardinality. We have experimented with parameters of all methods and observed the difficulty, in general, to optimize them. Automatic configuration works fine for `GeRaF`, which is the fastest in building. `GeRaF` also scales very well, even for $d = 10^4$ or $n = 10^6$, and, at the same accuracy, it is faster than competition for d roughly in the range $(10^3, 10^4)$, and n in the hundreds of thousands or millions.

The paper is structured as follows. Section 2 discusses the data structure and method, including randomization factors, building the forest, searching, and improvements that we introduce. Section 3 focuses on more technical implementation issues. Section 4 presents experimental evaluation and comparisons, while conclusions are drawn in Section 5.

2 The *k*-d GeRaF

The limitations of a single k-d tree for high d are overcome by searching multiple, randomized trees, simultaneously. This section discusses randomization, and algorithms for parameter configuration, building, and searching. Overall, m different randomized k-d trees are built, each with a different structure such that search in the different trees is independent; *i.e.*, neighboring points that are split by a hyperplane in one, are not split in another. Search is simultaneous in the m trees, *i.e.*, nodes from all trees are visited in an order determined by a shared priority queue. There is no backtracking, and search terminates when c leaves are visited.

2.1 Randomization

The key insight is to construct substantially different trees, by randomization. Multiple independent searches are subsequently performed, increasing the probability of finding approximate nearest neighbors. Randomization amounts to either generating a different randomly transformed pointset per tree (*e.g.*, rotation or shuffling), or choosing splits at random at each node (*e.g.*, split dimension or value). As discussed below, we investigate four randomization factors, which we use either independently or in combination.

Rotation. For each k-d tree, we randomly rotate the input pointset or, more generally, apply a different isometry [14]. Each resulting tree is thus based on a different set of dimensions. Only the transformation matrix R is stored for each tree, and not the rotated set. In fact, not even the entire matrix needs to be stored, as discussed in section 2.2. During search, the query is rotated using R before descending each tree. However, distances are computed between the original stored points and the original query.

Split dimension. In a conventional k-d tree, the pointset is halved at each node along one dimension; dimensions are examined in order even for high d. Here, we find the t dimensions of highest variance for the input set and then choose uniformly at random one of these t dimensions at each node. Thus, different trees are built from the given pointset.

Split value. The default split value in a conventional k-d tree is the median of the coordinates in the selected split dimension. `FLANN` uses the mean for reasons of speed. Here, we compute the median, which would yield a perfect tree, and then randomly perturb it [13]. In particular,

Algorithm 1: k-d GeRaF: building

input : pointset X, #trees m, #split-dimensions t, max #points per leaf p
output: randomized k-d forest F

begin
  $V \leftarrow \langle$VARIANCE of X in every dimension$\rangle$
  $D \leftarrow \langle t$ dimensions of maximum variance $V\rangle$
  $F \leftarrow \emptyset$ ▷ forest
  for $i \leftarrow 1$ **to** m **do**
    $f \leftarrow \langle$random transformation$\rangle$ ▷ isometry, shuffling
    $F \leftarrow F \cup (f, \text{BUILD}(f(X)))$ ▷ build on transformed X, store f
  return F

function BUILD(X) ▷ recursively build tree (node/leaf)
  if $|X| \leq p$ **then** ▷ termination reached
    return $leaf(X)$
  else ▷ split points and recurse
    $s \leftarrow \langle$one of dimensions D at random$\rangle$
    $v \leftarrow \langle$MEDIAN of X in dimension $s\rangle$
    $(L, R) \leftarrow \langle$SPLIT of X in dimension s at value $v\rangle$
    return $node(c, v, \text{BUILD}(L), \text{BUILD}(R))$ ▷ build children on L, R

the split value equals the median plus a quantity δ uniformly distributed in $[-3\Delta/\sqrt{d}, 3\Delta/\sqrt{d}]$, where Δ is the diameter of the current pointset; δ is computed at every node during building [15].

Shuffling. When computing the split value at each node in a conventional k-d tree, the current pointset at the node is used. Even if the split value is randomized, it is still possible that the same point is chosen if the same coordinate value occurs more than once in the selected dimension. This is particularly common when points are quantized; for instance, SIFT vectors are typically represented by one byte per element. We thus randomly shuffle points at each tree. Hence, different splits occur despite ties.

2.2 Building

The overall building algorithm for k-d GeRaF, consisting of m trees, is outlined in Alg. 1 (Appendix). For simplicity, only the random split dimensions are included, while the split value is the standard median. There is a random data transformation f per tree, which may include either an isometry, shuffling, or both; in case of an isometry, it is stored for use during search.

Given a dataset X, the t dimensions of maximum variance, say D, are computed. For each tree, X is transformed according to a different function f and then the tree is built recursively. At each node, one dimension (coordinate), say s, is chosen uniformly at random from D and X is split at the median in s. The two subsets of X, say L, R, are then recursively given as input datasets to the two children of the node. The split node so constructed contains the split dimension s and the split value v. Splitting terminates when fewer than p points are found in the dataset, in which case the point indices are just stored in a leaf node. When n is much higher than d, the bottleneck of the algorithm is finding the median, which is $O(n)$ on average. Otherwise, the bottleneck is computing the variance per dimension, which is $O(d)$. The space requirement for the entire data structure is $O(nd)$ for the data points and $O(nm)$ for the trees, including both nodes and indices to points, for a total of $O(n(d+m))$.

Each random isometry can be a rotation [15] or reflection, and in general requires the generation of a random orthogonal matrix R. We rather use an elementary Householder reflector P for efficiency [14]. In particular, given unit vector $u \in \mathbb{R}^d$ normal to hyperplane H, the orthogonal projection of a point x onto H is $x - (u^\top x)u$. Its reflection across H is twice as far from x in the same direction, that is, $y = x - 2(u^\top x)u = Px$, where $P = I - 2uu^\top$. Although P is orthogonal,

Algorithm 2: k-d GeRaF: searching.

input : query point q, forest F, #neighbors k, max #leaf-checks c
output: k nearest points

begin
  Q.INIT() ▷ min-priority queue, initially empty
  for $i \leftarrow 1$ **to** m **do**
    DESCEND($q, F[i]$, FALSE) ▷ descend i-th tree, store path in Q, no checks
  $\ell \leftarrow 0$ ▷ # of leaves checked
  H.INIT(k) ▷ min-heap of size k
  while $\neg Q$.EMPTY() $\wedge\ \ell < c/(1+\epsilon)$ **do**
    $(N, d) \leftarrow Q$.EXTRACT-MIN() ▷ (node, distance)
    DESCEND(q, N, TRUE) ▷ descend again, but check leaves now
    $\ell \leftarrow \ell + 1$ ▷ increase leaves checked
  return H

function DESCEND(q, *node* N, *check*) ▷ descend node N for query q
  $d \leftarrow N$.DIST(q) ▷ signed distance to boundary
  if $d < 0$ **then** ▷ q is in negative half-space
    Q.INSERT($N.right, |d|$) ▷ remember right child
    DESCEND($q, N.left$, *check*) ▷ descend left child
  else
    Q.INSERT($N.left, |d|$) ▷ and vice versa
    DESCEND($q, N.right$, *check*)

function DESCEND(q, *leaf* N, *check*) ▷ test query q on leaf N
  if $\neg$*check* **then return for** $i \in N$.POINTS **do**
    H.INSERT($i, \|q - X_i\|^2$) ▷ distances to points X_i in leaf N

the computation of reflection Px is $O(n)$, involving a dot product and an element-wise multiplication and addition. This is because $uu^\top$ is of rank one. We only need to store vector u for each tree.

2.3 Searching

Searching takes place in parallel in all trees; this does not refer to independent search per tree, but rather that nodes from all trees are visited in a particular order using a shared min-priority queue Q. The idea is that given a bound c on the total leaves to be checked, the query iteratively descends the most promising nodes from all trees, and the criterion is the distance of the query to the hyperplane specified by each node.

A shown in Alg. 2, the query initially descends all trees of forest F while all visited nodes are stored in Q, without checking any leaves. Then, for each node extracted from Q, the query descends again, this time computing distances to all points in the leaf. For each decision made at a node while descending, the other one is stored in Q. In particular, the *signed* distance $d = N.\text{DIST}(q)$ of query q to the hyperplane specified by node N is

$$N.\text{DIST}(q) = N.tree.f(q)_{N.c} - N.v \tag{1}$$

where $N.tree.f$ is the isometry of the tree where N belongs, and $N.c$, $N.v$ are the split dimension (coordinate) and value of N, respectively. One child of N is chosen to descend according to the sign of d, and the other is stored in Q with the absolute distance $|d|$ as key. This key is used for priority in Q.

Results are stored in a min-heap H that holds up to k points, where k is the number of neighbors to be returned. For each leaf visted, the distance between q and all points stored in the leaf is computed. For each point X_i of the dataset X, H is updated dynamically such that it always contains the k nearest neighbors to q. The key used for H is the computed (squared)

distance $\|q - X_i\|^2$. A separate array keeps track of points encountered so far, such that no distance is computed twice; this detail is not shown in Alg. 2.

For each tree built under isometry f, the transformed query $f(q)$ is used in all tests at internal nodes, but the initial query q is rather used in all distance computations with points stored at leaves. Similarly, the transformed dataset is used only for building the tree but is not stored. This is possible since the isometry leaves distances unaffected. In practice, unlike (1), the query is transformed according to isometries of all trees prior to descending.

Although no backtracking occurs, visiting new nodes is an implicit form of backtracking. However, given the bound on the number of leaves to be visited, search is approximate. In particular, apart from the case when Q is empty, search terminates when $c/(1+\epsilon)$ leaves have been checked. That is, up to c leaves are checked for $\epsilon = 0$, while this bound decreases for $\epsilon > 0$, making search faster and less accurate.

3 Implementation

This section discusses our C++ implementation of k-d GeRaF, which is available online[4]. The project is open source, under the BSD 2-clause license. Important implementation issues are discussed here, focusing on efficiency.

Parameters. Our implementation provides several parameters to allow the user to fully customize the data structure and search algorithm:

- m Number of trees in forest. A small number yields fast building and search, but may reduce accuracy; a large m covers space better and enhances accuracy, but slows down building and search.
- t Number of dimensions used for splits. As d increases, a larger t is better, until accuracy begins to drop. The optimum t depends on the input.
- p Maximum number of points per leaf. A large p means short trees, and saves space; a small p accelerates search, but may reduce accuracy.
- c Maximum number of leaves to be checked during search. The higher this number, the higher the accuracy and search time.
- ϵ Determines search accuracy (Definition 1); more accurate search comes at the expense of slower query.
- k Number of neighbors to be returned for a query; specified during search.

Configuration. We provide a simple and fast automatic configuration method for parameter tuning. Given a dataset and ϵ we automatically configure all parameters above, except k. In particular, taking into account n, d and the five coordinates of greatest variance, we configure parameters p, c, t, m, limiting their values to powers of two. The particular values chosen are piecewise constant functions of ϵ, n, d, where constants have been obtained by experience, *i.e.* by manually setting parameters on a number of datasets. This kind of tuning is largely subjective. The runtime is negligible, since the variances are computed by the algorithm anyway. However, the resulting parameter set is not optimal, *e.g.* in terms of accuracy or speed.

Tree structure. Every tree consists of split nodes and leaves. A split node contains the split dimension and value, while a leaf contains a number of point indices. Points are stored only once, regardless of forest size. We store trees in arrays to benefit from contiguous storage. As discussed in section 4, split value randomization is not beneficial so we disable it. In this case, we split at medial and trees are perfect, thus space is optimized. No re-allocation is needed because we know the size of the tree in advance.

Median, variances. The median is found efficiently by the quickselect algorithm, with average complexity $O(n)$. Variance is computed by an extension of Knuth's online algorithm [9, p.232],

[4] `https://github.com/gsamaras/kd_GeRaF`

Algorithm 3: Modified Knuth's online variance algorithm

input : sequence x of real vectors in $\mathbb{R}^d$
output: variance on each dimension of the vectors in x

```
1 begin
2     if x.SIZE() < 2 then return return 0;                ▷ zero vector in R^d
3     μ ← 0; v ← 0                                         ▷ zero vectors in R^d
4     for n ← 1 to x.SIZE() do
5         α ← 1/n                                          ▷ α: scalar
6         δ ← x[i] − μ                                     ▷ δ: vector in R^d
7         μ ← μ + αδ                                       ▷ μ: vector in R^d
8         v ← v + δ ∘ (x[i] − μ)                           ▷ ∘: Hadamard product; v: vector in R^d
9     return v/(n − 1)
```

as shown in Alg. 3. In particular, we extend the algorithm to operate in parallel on a sequence of vectors rather than scalars. In doing so, we replace vector division with scalar n by multiplication with $\alpha = 1/n$. This choice provides significant speed-up.

Distance computation. This is the most expensive task during search in high dimensions. To speed it up, we note that squared Euclidean distance between point x and query q is $\|q-x\|^2 = \|q\|^2 + \|x\|^2 - 2q^\top x$, where $\|q\|$ is constant, while $\|x\|$ can be stored for all points. Thus distance computation reduces to dot product, providing a speed-up of $> 10\%$ in certain cases. The space overhead is one scalar per point, which is negligible in high dimensions since all points are stored in memory.

Parallelization. The building process is trivially parallelizable: we just assign building of individual trees to different threads, making sure that the work is balanced among threads. Searching is not performed in parallel: due to use of a single priority queue for all trees, more work would be required for communication between different threads. It would be interesting to investigate this extension in future work.

4 Experiments

This section presents our experimental results and comparisons on a number of synthetic and real datasets. All experiments are conducted on a processor at 2.40 GHz×4 with 3.8 GB memory, except for GIST dataset with $n = 10^6$, for which we use a processor at 3 GHz×4 with 8 GB. We compare to BBD-trees as implemented in `ANN`, LSH as implemented in `E`^{2}`LSH`, `FLANN`, and our implementation of PQ.

Datasets. We use five datasets of varying dimensionality and cardinality. To test special topologies, the first two, *Klein bottle* and *Sphere* are synthetic. We generate points on a Klein bottle and a sphere embedded in $\mathbb{R}^d$, then add to each coordinate zero-mean Gaussian noise of standard deviation 0.05 and 0.1 respectively. In both cases, queries are nearly equidistant to all points, which implies high miss rates.

The other three datasets, *MNIST*[5], *SIFT* and *GIST*[6] [7], are common in computer vision and machine learning. MNIST contains vectors of 784 dimensions, that are 28×28 image patches of handwritten digits. There is a set of 60k vectors, plus an additional set of 10k vectors that we use as queries. SIFT is a 128-dimensional vector that describes a local image patch by histograms of local gradient orientations. GIST is a 960-dimensional vector that describes globally an entire image. SIFT and GIST datasets each contain one million vectors and an additional set for queries, that are 10^4 for SIFT and 1000 for GIST. For GIST, we also use the first 10^5 vectors as a separate smaller dataset.

[5] `http://yann.lecun.com/exdb/mnist/`
[6] `http://corpus-texmex.irisa.fr/`

Parameters. Most experiments use the default parameters provided by existing implementations but, on specific inputs, we have optimized the parameters manually. This improves performance, but is quite impractical in general. `FLANN` and `GeRaF` determine automatically the parameters given the dataset and ϵ, while `ANN` uses default parameters regardless of ϵ. `E`^{2}`LSH` provides automatic parameter configuration, but not for the most important one, R, used in solving a randomized version of R-near neighbor. This is a major drawback, since the user has to manually identify R at every input. As discussed below, accuracy measurements only refer to the first nearest neighbor, so we always set $k = 1$ in Alg. 2. The same holds for `BBD` and `FLANN`, but not for `LSH` where the number of neighbors is only controlled by R.

In k-d GeRaF, we have observed that rotation does not seem to affect search performance, despite the time penalty, *i.e.* build time increases from 0.26 to 1.35sec on Klein bottle with $n = 10^4, d = 10^4$. Similarly, split value randomization brings no benefit, despite its cost: build time increases from 0.06 to 1.92 (89.8) sec for approximate (exact) diameter computation, while search accuracy decreases for approximate computation. We have therefore disabled these two randomization factors.

Implementation. Before presenting experimental comparisons to other methods, we measure the effect of two implementation issues discussed in section 3, in particular parallelization and distance computation. Both are measured on SIFT dataset with $d = 128$. On four cores, parallelization reduces build time from 15 to 9msec for $n = 10^4$, and from 3.32 to 1.48sec for $n = 10^6$: the speedup is higher for larger forests. On the other hand, reduction of distance computation to dot product reduces build time from 3.06 to 2.67μsec per point. However, this approach appears to be effective only when $d > 100$ in practice.

	Sphere $n = 10^3, d = 10^4$				Klein $n = 10^4, d = 10^2$				MNIST $n = 60\text{k}, d = 784$			
ϵ	0	0.1	0.5	0.9	0	0.1	0.5	0.9	0	0.1	0.5	0.9
`BBD`	1.25	1.26	1.30	1.25	0.13	0.14	0.17	0.14	187.5	184.3	185.1	185.6
`LSH`	0.21	0.16	0.18	0.31	0.11	0.07	0.03	0.05	1.47	69.76	48.47	14.35
`FLANN`	25.0	25.4	25.5	25.6	–	–	–	–	244.6	217.2	157.3	142.0
`GeRaF`	0.06	0.06	0.06	0.06	0.06	0.06	0.06	0.08	8.167	8.567	8.579	8.565

Table 1. Build time (s) for three representative datasets. `FLANN` does not finish after 4hr on Klein bottle, which is indicated by '–'.

Preprocessing. For all methods this includes building, but for `FLANN` and `GeRaF` it also includes automatic parameter configuration. Build time is related to the required precision as expressed by ϵ. For `LSH`, ϵ is failure probability and its build time is the most sensitive to ϵ. Despite requesting the user to manually determine parameter R, `LSH` performs an automatic parameter configuration as well, which is included in the building process.

Table 1 shows representative experiments. `FLANN` has difficulties with automatic configuration, which does not terminate after 4hr on Klein bottle and is quite slow in general. `LSH` is unexpectedly fast on MNIST for $\epsilon = 0$, which may be due to the parameters chosen by autotuning. `GeRaF` works well with automatic configuration, and is typically one order of magnitude faster than other methods. Its preprocessing time may increase with ϵ since this requires fewer points per leaf, hence more subdivisions.

We additionally carry out an experiment with product quantization (in particular, IVFADC) [7] on SIFT, implemented on Matlab with `Yael`[7] library. Its off-line processing includes codebook learning, which takes 440sec for 50 k-means iterations and encoding/indexing, which takes 45sec. The latter time is competitive if codebooks are existing from a similar dataset, but the total time given a new unknown dataset is quite higher than `GeRaF` and `LSH`; and even higher than `FLANN` with default values.

[7] `https://gforge.inria.fr/projects/yael`

	Sphere $n = 10^3, d = 10^4$				Klein $n = 10^4, d = 10^2$				Sphere $n = 10^5, d = 10^2$			
ϵ	0	0.1	0.5	0.9	0	0.1	0.5	0.9	0	0.1	0.5	0.9
	miss %											
BBD	0	100	100	100	0	59	59	59	1	100	100	100
LSH	45	45	45	45	1	1	20	63	2	2	2	2
FLANN	0	0	0	0	–	–	–	–	100	100	100	100
GeRaF	0	24	24	100	2	3	3	5	2	26	40	81
	search (ms)											
BBD	9.100	0.210	0.220	0.200	0.470	0.043	0.046	0.052	12	0.024	0.028	0.026
LSH	17.000	16.000	18.000	17.000	2.700	2.400	1.900	0.850	28.000	24.000	22.000	22.000
FLANN	0.310	0.280	0.350	0.320	–	–	–	–	0.021	0.021	0.020	0.021
GeRaF	0.400	0.200	0.150	0.100	0.100	0.083	0.083	0.070	3.900	2.900	1.500	1.300

Table 2. Search accuracy and times for synthetic datasets. Search times in gray represent failure cases where miss rate is 100%. Queries are nearly equidistant to points, which explains high miss rates, especially for BBD and FLANN; '–' indicates preprocessing does not finish after 4hr.

Search. We report query times and miss rates for four representative values of ϵ. The miss rate is the percentage of queries where the reported neighbor is *not* the exact one. In case of ties, any point at the same distance as the nearest neighbor is accepted as correct. Table 2 shows results for all methods on three representative synthetic datasets. BBD and FLANN have problems with high miss rate or having failed in automatic preprocessing. LSH is at least one order of magnitude slower than GeRaF. In most cases GeRaF is faster (especially for large $d = 10^4$), with competitive miss rate, except for FLANN on Sphere with $d = 10^4$, which is the best dimension for FLANN.

Figure 1 presents four representative datasets with real data. BBD and FLANN have problems with either running out of memory or not completing automatic build. GeRaF is typically faster than LSH by at least an order of magnitude at the same accuracy. In all cases, FLANN preprocessing does not terminate after 4hr so we manually configure parameters because default ones yield even higher miss rates. FLANN is generally the fastest method but with low accuracy. On GIST, with $d = 960$, GeRaF shows best performance. LSH has 0.5% better miss rate for $n = 10^5$, but is quite slower; it also fails with 100% miss rate for $n = 10^6$. With automatic configuration, GeRaF always yields a good trade-off between accuracy and speed.

We also experiment with PQ (IVFADC) in these datasets, which is known to outperform FLANN [7] when combined with re-ranking. For instance, it takes 7 (70) msec for a miss rate of 1% on SIFT (GIST $n = 10^6$). However its training is slow, as noted above.

n	d	miss %				search (μsec)			
		BBD	LSH	FLANN	GeRaF	BBD	LSH	FLANN	GeRaF
10^3	100	100	0	16	0	1	212	12	199
	1000	100	50	100	50	5	1850	34	14
	5000	100	0	100	0	39	8675	149	122
	10000	100	37	100	2	276	17000	289	520
1000	10^3	100	50	100	50	5	1850	34	14
10000		100	0	100	0	5	1780	–	390
100000		100	8	100	0	276	–	–	10900

Table 3. Klein bottle search for $\epsilon = 0.1$, for varying n or d, where the other parameter is fixed. Search times in gray represent failure cases where miss rate is 100%. Queries are nearly equidistant from the points, which explains high miss rates. '–' indicates preprocessing does not finish after 2hr.

Table 3 displays, for all methods, the miss rate and search time as a function of n or d when the other parameter is fixed. In cases where miss rate is not 100%, GeRaF is an order of magnitude faster. The only exception is $d = 100$, where the situation is inversed with FLANN.

Approximate search evaluation. We also measure for GeRaF the percentage of queries where the reported nearest neighbor does *not* lie within $1 + \epsilon$ of the nearest distance. This a more natural measure than miss rate when approximate search is requested given a specific ϵ. For Klein bottle with $n = 10^4, d = 10^2$, this rate is 2% and 0%, for $\epsilon = 0$, and $\epsilon \in \{0.1, 0.5, 0.9\}$, respectively. In order for the output to always lie within $1 + \epsilon$ of optimal, one may set $c = n$, thus disabling

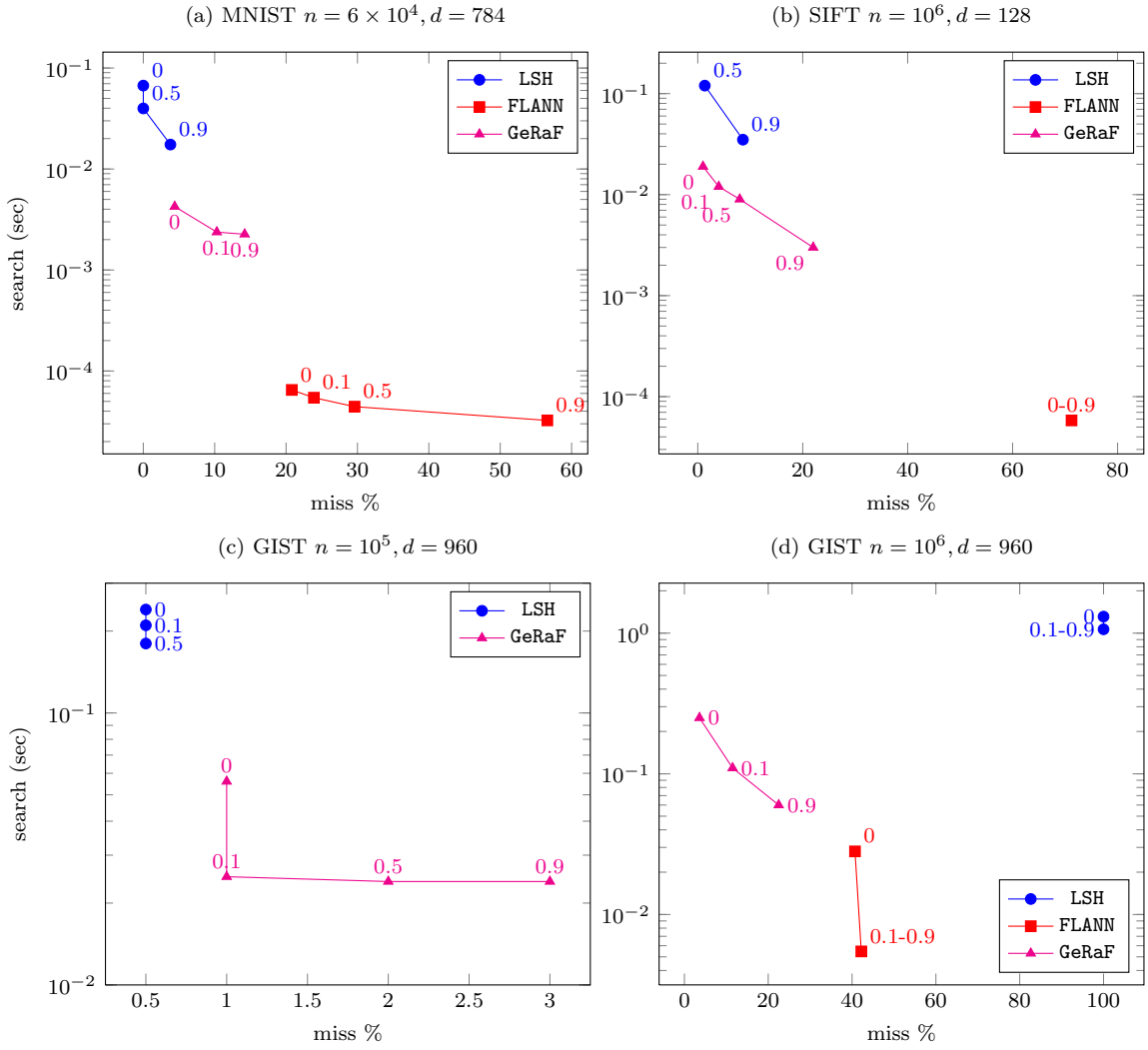

Fig. 1. Search accuracy and times on real datasets. In (b), `LSH` is out of memory for $\epsilon \in \{0, 0.1\}$. In all cases, `BBD` is out of memory and `FLANN` does not preprocess after 4hr for any ϵ. Its measurements in (a),(b),(d) refer to manually configured parameters.

the termination condition of leaves to be checked. However, due to the curse of dimensionality, performance nearly reduces to brute force in this case. For GIST with $n = 10^5, d = 960$ for instance, search takes 140ms, whereas miss rate is 0% and 0.4% for $\epsilon = 0$ and $\epsilon \in \{0.1, 0.5, 0.9\}$ respectively.

p	256	128	64	32	16	4	2	1
build (s)	0.0592	0.0618	0.0674	0.0695	0.0860	0.1159	0.1543	0.1587
search (ms)	0.2324	0.1863	0.1198	0.0941	0.0712	0.0592	0.0743	0.0928
miss %	1	1	2	7	6	10	14	22

Table 4. `GeRaF` build and search measurements for Klein bottle dataset with $n = 10^4, d = 10^2$ for varying points per leaf p.

Points per leaf. Finally, we measure the effect of storing multiple points per leaf on the Klein bottle dataset. The results are shown in Table 4. It is clear that search time improves when there are less points per leaf, and this is why a single point per leaf is a common approach. However, the build time and most importantly the miss rate also increase significantly. We therefore provide a reasonable trade-off by automatically adjusting parameter p.

5 Discussion

We have presented an efficient data structure for approximate nearest neighbor search that explores different randomization strategies, and an efficient implementation, `GeRaF`, that is found competitive against existing implementations of several state-of-the-art methods. We provide a simple but effective automatic parameter configuration that yields the fastest preprocessing, including both configuration and building, as well as a successful trade-off between accuracy and speed. Most competing methods have difficulties in running out of memory at large scale (*e.g.*, `BBD`), slow or non-terminating parameter configuration (*e.g.*, `FLANN`), or unstable search behavior between accurate (but slow) or fast (but inaccurate) search (*e.g.*, `LSH` and `FLANN`). PQ is consistently faster and more accurate at search, but is significantly slower to build, which is impractical when the dataset is updated. Our findings are consistent on both synthetic and real datasets of a wide range of dimensions and cardinalities.

Interesting open questions include whether and how `GeRaF` can be fully dynamic, supporting insertions and deletions, as well as handling batch queries in an optimized manner. Other future directions include performing parallel or distributed search and more principled parameter configuration with discrete optimization. In fact, recent experiments with parameter tuning by generic algorithms indicate that build time for large datasets such as SIFT can drop by a factor of 100 without significantly affecting search time while reducing miss rate [6].

References

1. E. Anagnostopoulos, I.Z. Emiris, and I. Psarros. Low-quality dimension reduction and high-dimensional approximate nearest neighbor. In *Proc. Annual Symp. on Computational Geometry*, pages 436–450, 2015.
2. A. Andoni and P. Indyk. Near-optimal hashing algorithms for approximate nearest neighbor in high dimensions. *Commun. ACM*, 51(1):117–122, 2008.
3. S. Arya, D.M. Mount, N. Netanyahu, R. Silverman, and A. Wu. An optimal algorithm for approximate nearest neighbors in fixed dimension. *J.ACM*, 45:891–923, 1998.
4. A. Babenko and V. Lempitsky. The inverted multi-index. In *Computer Vision and Pattern Recognition*, pages 3069–3076. IEEE, 2012.
5. S. Dasgupta and Y. Freund. Random projection trees and low dimensional manifolds. In *Proc. ACM STOC*, pages 537–546, 2008.
6. N. Giachoudis. Report for k-d GeRaF parameter auto tunning. Technical report, 2015.
7. H. Jégou, M. Douze, and C. Schmid. Product quantization for nearest neighbor search. *IEEE Trans. Pattern Analysis & Machine Intell.*, 33(1):117–128, 2011.
8. Y. Kalantidis and Y. Avrithis. Locally optimized product quantization for approximate nearest neighbor search. In *Comp. Vision & Pattern Recogn.*, 2014.
9. D. Knuth. *The Art of Computer Programming*, volume 2. Addison-Wesley, 1998.
10. M. Muja and D.G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *Proc. VISAPP: Intern. Conf. Computer Vision Theory & Appl.*, pages 331–340, 2009.
11. M. Muja and D.G. Lowe. Scalable nearest neighbour algorithms for high dimensional data. *Pattern Analysis and Machine Intelligence*, 2014.
12. F. Perronnin, Z. Akata, Z. Harchaoui, and C. Schmid. Towards good practice in large-scale learning for image classification. In *Computer Vision and Pattern Recognition*, 2012.
13. J. Philbin, O. Chum, M. Isard, J. Sivic, and A. Zisserman. Object retrieval with large vocabularies and fast spatial matching. In *Comp Vision & Pattern Recogn*, 2007.
14. C. Silpa-Anan and R. Hartley. Optimised kd-trees for fast image descriptor matching. In *Proc. IEEE Computer Vision & Pattern Recognition*, 2008.
15. S. Vempala. Randomly-oriented kd-trees adapt to intrinsic dimension. In *Proc. Foundations Software Techn. & Theor. Comp. Science*, pages 48–57, 2012.
16. J. Wang, H. T. Shen, J. Song, and J. Ji. Hashing for similarity search: A survey. Technical Report 1408.2927, Arxiv, 2014.
17. Y. Weiss, A. Torralba, and R. Fergus. Spectral hashing. In D. Koller et al., editor, *Proc. NIPS 21*, pages 1753–1760, 2008.