# ANN-Benchmarks: A Benchmarking Tool for Approximate Nearest Neighbor Algorithms*

Martin Aumüller[1], Erik Bernhardsson[2], and Alexander Faithfull[1]

[1] IT University of Copenhagen, Denmark, {maau,alef}@itu.dk
[2] Better, mail@erikbern.com

**Abstract.** This paper describes ANN-Benchmarks, a tool for evaluating the performance of in-memory approximate nearest neighbor algorithms. It provides a standard interface for measuring the performance and quality achieved by nearest neighbor algorithms on different standard data sets. It supports several different ways of integrating $k$-NN algorithms, and its configuration system automatically tests a range of parameter settings for each algorithm. Algorithms are compared with respect to many different (approximate) quality measures, and adding more is easy and fast; the included plotting front-ends can visualise these as images, LaTeX plots, and websites with interactive plots. ANN-Benchmarks aims to provide an constantly updated overview of the current state of the art of $k$-NN algorithms. In the short term, this overview allows users to choose the correct $k$-NN algorithm and parameters for their similarity search task; in the longer term, algorithm designers will be able to use this overview to test and refine automatic parameter tuning. The paper gives an overview of the system, evaluates the results of the benchmark, and points out directions for future work. Interestingly, very different approaches to $k$-NN search yield comparable quality-performance trade-offs. The system is available at http://sss.projects.itu.dk/ann-benchmarks/.

## 1 Introduction

Nearest neighbor search is one of the most fundamental tools in many areas of computer science, such as image recognition, machine learning, and computational linguistics. For example, one can use nearest neighbor search on image descriptors such as MNIST [18] to recognize handwritten digits, or one can find semantically similar phrases to a given phrase by applying the word2vec embedding [22] and finding nearest neighbors. The latter can, for example, be used to tag articles on a news website and recommend new articles to readers that have shown an interest in a certain topic. In some cases, a generic nearest neighbor search under a suitable distance or measure of similarity offers surprising quality improvements [8].

In many applications, the data points are described by high-dimensional vectors, usually ranging from 100 to 1000 dimensions. A phenomenon called the *curse of*

* The research of the first and third authors has received funding from the European Research Council under the European Union's 7th Framework Programme (FP7/2007-2013) / ERC grant agreement no. 614331.

*dimensionality*, the existence of which is also supported by popular algorithmic hardness conjectures (see [2,28]), tells us that to obtain the true nearest neighbors, we have to use either linear time (in the size of the dataset) or time/space that is exponential in the dimensionality of the dataset. In the case of *massive* high-dimensional datasets, this rules out *efficient* and *exact* nearest neighbor search algorithms.

To obtain efficient algorithms, research has focused on allowing the returned neighbors to be an *approximation* of the true nearest neighbors. As an example, in the case of searching for *the* nearest neighbor $p^*$ of a query point $q$, we require for the algorithm to be correct that it returns a point that is at most a factor of $1+\varepsilon$ further away from $q$ than $p^*$.

There exist many different algorithmic techniques for finding approximate nearest neighbors. Classical algorithms such as *kd*-trees [5] or M-trees [9] can simulate this by terminating the search early, for example shown by Zezula et al. [29] for M-trees. Other techniques [20,21] build a graph from the dataset, where each vertex is associated with a data point, and a vertex is adjacent to its true nearest neighbors in the data set. Others involve projecting data points into a lower-dimensional space using hashing. A lot of research has been conducted with respect to locality-sensitive hashing (LSH) [15], but there exist many other techniques that rely on hashing for finding nearest neighbors; see [27] for a survey on the topic. We note that, in the realm of LSH-based techniques, algorithms guarantee sublinear query time, but solve a problem that is only distantly related to finding the $k$ nearest neighbors of a query point. In practice, this could mean that the algorithm runs *slower* than a linear scan through the data, and counter-measures have to be taken to avoid this behavior [1,25].

Given the difficulty of the problem of finding nearest neighbors in high-dimensional spaces and the wide range of different solutions at hand, it is natural to ask how these algorithms perform in empirical settings. Fortunately, many of these techniques already have good implementations: see, e.g., [23,6,19] for tree-based, [7,11] for graph-based, and [3] for LSH-based solutions. This means that a new (variant of an existing) algorithm can show its worth by comparing itself to the many previous algorithms on a collection of standard benchmark datasets with respect to a collection of quality measures. What often happens, however, is that the evaluation of a new algorithm is based on a small set of competing algorithms and a small number of select datasets. This approach poses problems for everyone involved:

(i) *The algorithm's authors*, because competing implementations might be unavailable, they might use other conventions for input data and output of results, or the original paper might omit certain required parameter settings (and, even if these are available, exhaustive experimentation can take lots of CPU time).
(ii) *Their reviewers and readers*, because experimental results are difficult to reproduce and the selection of datasets and quality measures might appear selective.

This paper proposes a way of standardizing benchmarking for nearest neighbor search algorithms, taking into account their properties and quality measures. Our benchmarking framework provides a unified approach to experimentation and comparison with existing work. The framework has already been used for experimental comparison in other papers [20] (to refer to parameter choice of algorithms)

and algorithms have been contributed by the community, e.g., by the authors of NMSLib [7] and FALCONN [3]. An earlier version of our framework is already widely used as a benchmark referred to from other websites, see, e.g., [7,3,6,19,11].

**Related work.** Generating reproducible experimental results is one of the greatest challenges in many areas of computer science, in particular in the machine learning community. As an example, `openml.org` [26] and `codalab.org` provide researchers with excellent platforms to share reproducible research results.

The automatic benchmarking system developed in connection with the mlpack machine learning library [10,13] shares many characteristics with our framework: it automates the processs of running algorithms with preset parameters on certain datasets, and can visualize these results. However, the underlying approach is very different: it calls the algorithms natively and parses the standard output of the algorithm for result metrics. Consequently, the system relies solely on the correctness of the algorithms' own implementations of quality measures, and adding new quality measures needs changes in *every single* algorithm implementation. We instead require algorithms to expose a simple programmatic interface, which is only required to return the set of nearest neighbors of a given query, after preprocessing the set of data points. Timing and quality measure computation is conducted within our framework. This lets us add new metrics without rerunning the algorithms, if the metric can be computed from the set of returned elements.

Our benchmarking framework does not aim to replace these tools; instead, it complements them.

**Contributions.** We describe our system for benchmarking approximate nearest neighbor algorithms with the general approach described in Section 3. The system allows for easy experimentation with $k$-NN algorithms, and visualizes algorithm runs in an approachable way. Moreover, in Section 4 we use our benchmark suite to overview the performance and quality of current state-of-the-art $k$-NN algorithms. This allows us to identify areas that already have competitive algorithms, to compare different methodological approaches to nearest neighbor search, but also to point out challenging datasets and metrics, where good implementations are missing or do not take full advantage of properties of the underlying metric. Having this overview has immediate practical benefits, as users can select the right combination of algorithm and parameters for their application. In the longer term, we expect that more algorithms will become able to tune their own parameters according to the user's needs, and our benchmark suite will also serve as a testbed for this automatic tuning.

## 2 Problem Definition and Quality Measures

We assume that we want to find near neighbors in a space $X$ with a distance measure $\mathrm{dist} : X \times X \to \mathbb{R}$, for example the $d$-dimensional Euclidean space $\mathbb{R}^d$ under Euclidean distance ($l_2$ norm), or $d$-dimensional Hamming space $\{0,1\}^d$ under Hamming distance.

An algorithm $\mathcal{A}$ for nearest neighbor search builds a data structure $\mathrm{DS}_\mathcal{A}$ for a data set $S \subset X$ of $n$ points. In a preprocessing phase, it creates $\mathrm{DS}_\mathcal{A}$ to support

the following type of queries: For a query point $q \in X$ and an integer $k$, return a *result tuple* $\pi = (p_1, \ldots, p_{k'})$ of $k' \leq k$ distinct points from $S$ that are "close" to the query $q$. Nearest neighbor search algorithms generate $\pi$ by refining a set $C \subseteq S$ of candidate points w.r.t. $q$ by choosing the $k$ closest points among those doing distance computations. The size of $C$ (and thus the number of distance computations) is denoted by $N$. We let $\pi^* = (p_1^*, \ldots, p_k^*)$ denote the tuple containing the true $k$ nearest neighbors for $q$ in $S$ (where ties are broken arbitrarily). We assume in the following that all tuples are sorted according to their distance to $q$.

### 2.1 Quality Measures

We use different notions of "recall" as a measure of the quality of the result returned by the algorithm. Intuitively, recall is the ratio of the number of points in the result tuple being true nearest neighbors and the number $k$ of true nearest neighbors. However, this intuitive definition is fragile when distances are not distinct or when we try to add a notion of approximation to it. To avoid these issues, we use the following distance-based definitions of recall and $(1+\varepsilon)$-approximative recall, that take the distance of the $k$-th true nearest neighbor as threshold distance.

$$\text{recall}(\pi, \pi^*) = \frac{|\{p \text{ contained in } \pi \,|\, \text{dist}(p,q) \leq \text{dist}(p_k^*, q)\}|}{k}$$

$$\text{recall}_\varepsilon(\pi, \pi^*) = \frac{|\{p \text{ contained in } \pi \,|\, \text{dist}(p,q) \leq (1+\varepsilon)\text{dist}(p_k^*, q)\}|}{k}, \quad \text{for } \varepsilon > 0.$$

(If all distances are distinct, $\text{recall}(\pi, \pi^*)$ matches the intuitive notion of recall.)

We note that (approximate) recall in high dimensions is sometimes criticised; see, for example, [7, Section 2.1]. We investigate the impact of approximation as part of the evaluation in Section 4, and plan to include other quality measures such as position-related measures [29] in future work.

### 2.2 Performance Measures

With regard to the performance, we use the performance measures defined in Table 1, which are divided into measures of the performance of the preprocessing step, i.e., generation of the data structure, and measures of the performance of the query algorithm. With respect to the query performance, different communities are interested in different cost values. Some rely on actual timings of query times, where others rely on the number of distance computations (reflected in the accuracy of the algorithm). The framework can take both of these measures into account. However, none of the currently included algorithms report the number of distance computations.

| Name of Measure | Computation of Measure |
|---|---|
| Index size of DS | Size of DS after preprocessing finished (in kB) |
| Index build time DS | Time it took to build DS (in seconds) |
| Accuracy of a query | $N/n$ |
| Time of a query | Time it took to run the query and generate result tuple $\pi$ |

**Table 1.** Performance measures used in the framework.

## 3 System Design

ANN-Benchmarks is implemented as a Python library with several different front-ends: one script for running experiments and a handful of others for working with and plotting results. It is designed to be run in a virtual machine or Docker container, and so comes with shell scripts for automatically installing algorithms, dependencies, and datasets.

The experiment front-end has some parameters of its own that influence what algorithms will be tested: the dataset to be searched (and an optional dataset of query points), the number of neighbours to search for, and the distance algorithm to be used to compare points. The plotting front-ends are also aware of these parameters, which are used to select and label plots.

This section gives only a high-level overview of the system; see http://sss.projects.itu.dk/ann-benchmarks/ for more detailed technical information.

### 3.1 Installing algorithms and datasets

Each dataset and library has a shell script that downloads, builds and installs it. These scripts are built on top of a shell function library that defines a few common operations, like cloning and patching a Git repository or downloading a dataset and checking its integrity. Datasets may also need to be converted; we include Python scripts for converting a few commonly-used formats into the plain-text format used by our system, and the shell scripts make use of these.

The shell function library used to install algorithm libraries can also automatically apply a patch series, so we can carry patches in our repository that make algorithms available to the framework before later moving them upstream.

Adding support for a new algorithm to ANN-Benchmarks is as easy as writing a script to install it and its dependencies, making it available to Python by writing a wrapper (or by reusing an existing one), and adding the parameters to be tested to the configuration files. Most of the installation scripts fetch the latest version of their algorithm from its Git repository, but there is no requirement to do this; indeed, installing several different versions of an algorithm would make it possible to use the framework for regression testing.

**Algorithm wrappers.** To be usable by our system, each of the algorithms to be tested must have some kind of Python interface. Many libraries already provide their own Python wrappers, either written by hand or automatically generated using a tool like SWIG; others are implemented partly or entirely in Python.

To bring algorithms that do not provide a Python interface into the framework, we specify a simple text-based protocol that supports the few operations we care about: parameter configuration, sending training data, and running queries. The framework comes with a wrapper that communicates with external programs using this protocol. In this way, algorithms can be run in external front-end processes implemented in any programming language.

The protocol has been designed to be easy to implement. Every message is a line of text that will be split into tokens according to the rules of the POSIX shell, good implementations of which are available for most programming languages. The

protocol is flexible and extensible: front-ends are free to include extra information in replies, and they can also implement special configuration options that cause them to diverge from the protocol's basic behaviour. As an example, we provide a simple C implementation that supports an alternative query mode in which parsing and preparing a query data point and running a query are two different commands. (As the overhead of parsing a string representation of a data point is introduced by the use of the protocol, removing it makes the timings more representative of normal use of the algorithm.)

### 3.2 Loading datasets and computing ground truth

Once we have datasets available, we must load them and compute the ground truth for the query set: the true nearest neighbours for each query point, along with their distances. This ground truth is passed, along with the values obtained by each experiment, to the functions used by the plotting scripts to calculate the various quality metrics.

The query set for a dataset is, by default, a pseudorandomly-selected set of ten thousand entries separated from the rest of the training data. If this behavior is not wanted, datasets can declare a different number of queries in their metadata, or the user can provide an explicit query set instead.

Depending on the values of the system's own configuration parameters, many different sets may have to be computed. Each of these is stored in a separate cache file.

### 3.3 Creating algorithm instances

After loading the dataset, the framework moves on to creating the algorithm instances. It does so based on a YAML configuration file that specifies a hierarchy of dictionaries: the first level specifies the point type, the second the distance metric, and the third each algorithm to be tested. Each algorithm gives the name of its wrapper's Python constructor; a number of other entries are then expanded to give the arguments to that constructor. Figure 1 shows an example of this configuration file.

The `base-args` list consists of those arguments that should be prepended to every invocation of the constructor. Figure 1 also shows one of the special keywords, `"@metric"`, that is used to pass one of the framework's configuration parameters to the constructor.

```
float:
  any:
    annoy:
      constructor: Annoy
      base-args: ["@metric"]
      run-groups:
        one-or-two-hundred-trees:
          args: [[100, 200], [100, 200, 400, 1000]]
        four-hundred-trees:
          args: [400, [1000, 2000, 4000, 10000]]
```

**Fig. 1.** An example of a fragment of an algorithm configuration file.

Algorithms must specify one or more "run groups", each of which will be expanded into one or more lists of constructor arguments. The `args` entry completes the argument list, but not directly: instead, the Cartesian product of all of its entries is used to generate *many* lists of arguments. The `annoy` entry in in Figure 1, for example, expands into twelve different algorithm instances, from `Annoy("euclidean", 100, 100)` to `Annoy("euclidean", 400, 10000)`.

### 3.4 The experiment loop

Once the framework knows what algorithms to run, it moves on to the experiment loop. (Figure 2 gives an overview of the loop.) Each algorithm instance is run in a separate subprocess. This makes it easy to clean up properly after each algorithm – simply destroying the subprocess takes care of everything. This approach also gives us a simple and algorithm-agnostic way of computing the memory usage of an algorithm: the subprocess takes a snapshot of its memory consumption before and after initialising the algorithm's data structures and compares the difference.

The complete results of each run are sent back to the main process using a pipe. The main process performs a blocking, timed wait on the other end of the pipe, and will destroy the subprocess if the user-configurable timeout is exceeded before any results are available.

### 3.5 Results and metrics

The fact that we store the *complete* results of each run – the time taken for each query and the indices and distances of each candidate – makes it easy to add new metrics and minimises wasted computation.

For each run, we store the full name – including the parameters – of the algorithm instance, the time it took to evaluate the training data, and the near neighbours returned by the algorithm, along with their distance from the query point. (To avoid affecting the timing of algorithms that do not indicate the distance of a result, the experiment loop independently re-computes distance values after the run is

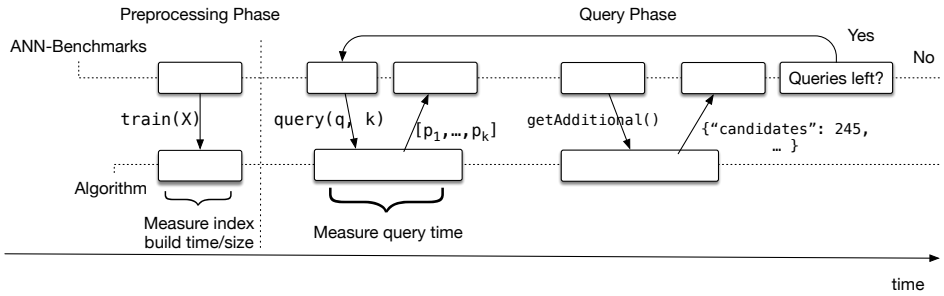


**Fig. 2.** Overview of the interaction between ANN-Benchmarks and an algorithm under test. The algorithm builds an index data structure for the dataset $X$ in the preprocessing phase. During the query phase, it returns $k$ near neighbors for each query point; after answering a query, it can also report any extra information it might have, such as the size of the candidate set.

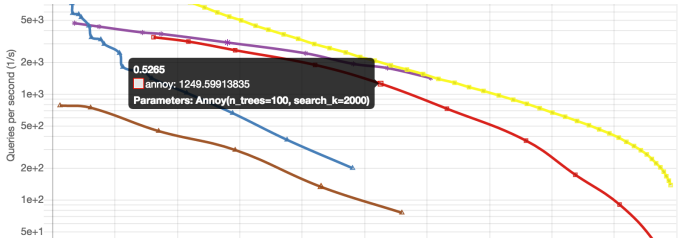


**Fig. 3.** Interactive plot screen from framework's website (cropped). Plot shows "Queries per second" ($y$-axis, log-scaled) against "Recall" ($x$-axis, not shown). Highlighted data point corresponds to a run of Annoy with parameters as depicted, giving about 1249 queries per second for a recall of about 0.52.

| Principle | Algorithms |
|---|---|
| $k$-NN graph | `KGraph` [11], `SWGraph` [21,7], `HNSW` [20,7] |
| tree-based | *FLANN* [23], *BallTree* [7] |
| LSH | `FALCONN` [3], *MPLSH* [12,7], `FAISS` [16] (+PQ codes) |
| random-projection forest | `Annoy` [6], `RPForest` [19] |
| other | `DOLPHINN` [4] (Projections + Hypercube) |
| | `Multi-Index Hashing` (MIH) [24] (exact Hamming search) |

**Table 2.** Overview of tested algorithms. Implementations in *italics* have "recall" as quality measure provided as an input parameter.

otherwise finished.) Each run is stored in a separate file in a directory hierarchy that encodes the framework's configuration. Keeping runs in separate files makes them easy to compress, easy to enumerate, and easy to re-run, and individual results – or sets of results – can easily be shared to make results more transparent.

Metric functions are passed the ground truth and the results for a particular run; they can then compute their result however they see fit. Addding a new quality metric is a matter of writing a short Python function and adding it to an internal data structure; the plotting scripts query this data structure and will automatically support the new metric.

### 3.6 Frontend

ANN-Benchmarks provides two options to evaluate the results of the experiments: a script to generate individual plots using Python's matplotlib and a script to generate a website that summarizes the results and provides interactive plots with the option to export the plot as LaTeX code using pgfplots. See Figure 3 to see a small example. Plots depict the Pareto frontier over all runs of an algorithm; this gives an immediate impression of the algorithm's general characteristics, at the cost of concealing some of the detail. When more detail is desired, the scripts can also produce scatter plots.

## 4 Evaluation

In this section we present a short evaluation of our findings from running benchmarks in the benchmarking framework.

**Experimental setup.** All experiments were run in Docker containers on *Amazon EC2 c4.2xlarge* instances that are equipped with Intel Xeon E5-2666v3 processors

| Dataset | Data/Query Points | Dimensionality | Metric |
|---|---|---|---|
| SIFT | 1 000 000 / 10 000 | 128 | Euclidean |
| GLOVE | 1 183 514 / 10 000 | 100 | Angular |
| NYTimes | 234 791 / 10 000 | 256 | Euclidean |
| Rand-Angular | 1 000 000 / 1 000 | 128 | Angular |
| SIFT-Hamming | 1 000 000 / 10 000 | 256 | Hamming |
| NYTimes-Hamming | 234 791 / 10 000 | 128 | Hamming |

**Table 3.** Datasets under consideration

(4 cores available, 2.90 GHz, 25.6MB Cache) and 15 GB of RAM running Amazon Linux. We ran a single experiment multiple times to verify that performance was reliable, and compared the experiments results with a 4-core Intel Core i7-4790 clocked at 3.6 GHz with 32GB RAM. While the latter was a little faster, the relative order of algorithms remained stable. For each parameter setting and dataset, the algorithm was given thirty minutes to build the index and answer the queries.

**Tested Algorithms.** Table 2 summarizes the algorithms that are at the moment included in our framework. More thorough descriptions of the algorithms can be found in the references provided. The scripts that set up the framework automatically fetch the most current version found in each algorithm's repository.

**Datasets.** The datasets used in this evaluation are summarized in Table 3. Results for other datasets are found on the framework's website. The NYTimes dataset was generated by building tf-idf descriptors from the bag-of-words version, and embedding them into a lower dimensional space using the Johnson-Lindenstrauss Transform [17]. Hamming space versions have been generated by applying Spherical Hashing [14] using the implementation provided by the authors of [14]. The random dataset Rand-Angular is generated by choosing 500 query points at random and putting clusters of 500 points at distance around $\alpha\sqrt{2}/3$, where $\alpha$ grows linearly from 0 to 1 with step size $1/500$. Each cluster has 500 points at distance around $2\alpha\sqrt{2}/3$ added. The rest of the dataset consists of random data points, 500 of which are chosen as the other set of query points (with closest neigbors expected to be at distance $\sqrt{2}$).

**Parameters of Algorithms.** Most algorithms do not allow the user to explicitly specify a quality target—in fact, only three implementations from Table 2 provide "recall" as an input parameter. We used our framework to test many parameter settings at once. The detailed settings tested for each algorithm can be found on the framework's website.

### 4.1 Objectives of the Experiments

We used the benchmarking framework to find answers to the following questions:

- **(Q1) Performance.** Given a dataset, a quality measure and a number $k$ of nearest neighbors to return, how do algorithms compare to each other with respect to different performance measures, such as query time or index size?
- **(Q2) Robustness.** Given an algorithm $\mathcal{A}$, how is its performance and result quality influenced by the dataset and the number of returned neighbors?
- **(Q3) Approximation.** Given a dataset, a number $k$ of nearest neighbors to return, and an algorithm $\mathcal{A}$, how does its performance improve when the returned neighbors can be an approximation? Is the effect comparable for different algorithms?

**(Q4) Embeddings.** Equipped with a framework with many different datasets and distance metrics, we can try interesting combinations. How do algorithms targeting Euclidean space or cosine similarity perform in, say, Hamming space? How does replacing the internals of an algorithm with Hamming space related techniques improve its performance?

The following discussion is based on a combination of the plots found on the framework's website; see the website for more complete and up-to-date results.

### 4.2 Discussion

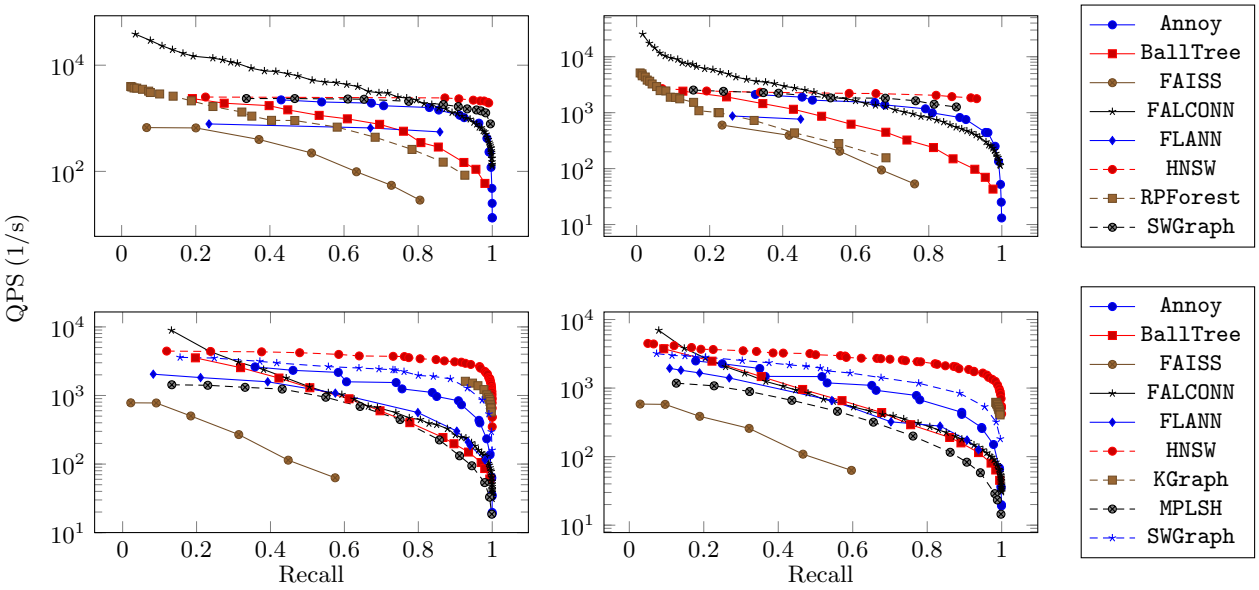


**Fig. 4.** Recall-QPS (1/s) tradeoff - up and to the right is better. Top: GLOVE, bottom: SIFT; left: 10-NN, right: 100-NN.

**(Q1) Performance.** Figure 4 shows the relationship between an algorithm's achieved recall and the number of queries it can answer per second (its QPS) on the two datasets GLOVE (cosine similarity) and SIFT (Euclidean distance) for 10- and 100-nearest neighbor queries.

For GLOVE, we observe that the graph-based algorithms `HNSW` and `SWGraph`, the LSH-based `FALCONN`, and the "random-projection forest"-based `Annoy` algorithm are fastest. For high recall values, `HNSW` is fastest, while for lower recall values, `FALCONN` achieves highest QPS. We can also observe the importance of implementation decisions: although `Annoy` and `RPForest` are both built upon the same algorithmic idea, they have very different performance characteristics.

On SIFT, almost all algorithms except `FAISS` can achieve close to perfect recall. In particular, the graph-based algorithms (along with `KGraph`) are fastest, followed by `Annoy`. `FALCONN`, `BallTree`, and `FLANN` have very similar performance.

Very few of these algorithms can tune themselves to produce a particular recall value. In particular, the fastest algorithms on the GLOVE dataset expose many parameters, leaving the user to find the combination that works best. The `KGraph` algorithm, on the other hand, uses only a single parameter, which—even in its "smallest" choice—still guarantees recall at least 0.9 on SIFT. `FLANN` manages to tune itself for a particular recall value on the SIFT dataset; for GLOVE with high recall values, however, the tuning does not complete within the time limit, especially with 100-NN.

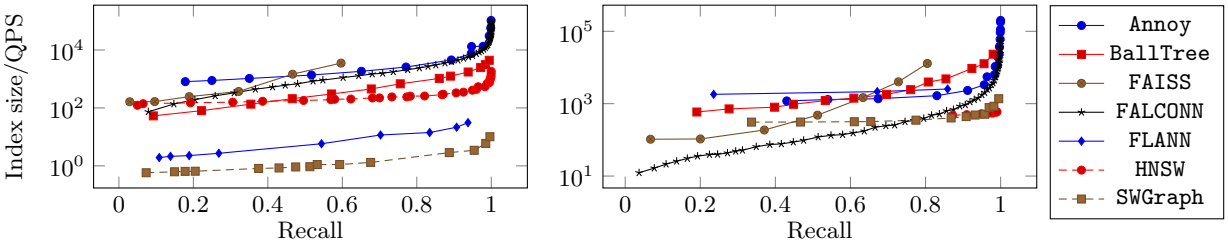


**Fig. 5.** Recall-Index size (kB)/QPS (s) tradeoff - down and to the right is better. Left: SIFT ($k$=100), right: GLOVE ($k$=10).

Figure 5 relates an algorithm's performance to its index size. High recall can be achieved with small indexes by probing many points; however, this probing is expensive, and so the QPS drops dramatically. To reflect this performance cost, we scale the size of the index by the QPS it achieves. This reveals that, on SIFT, `SWGraph` and `FLANN` achieve good recall values with small indexes. Both `BallTree` and `HNSW` show a similar behavior to each other. `Annoy` and `FALCONN` need rather large indexes to achieve high QPS. The picture is very different on GLOVE, where `FALCONN` provides the best ratio up to recall 0.8, only losing to the graph-based approaches at higher recall values.

**(Q2) Robustness.** Figure 6 plots recall against QPS for `Annoy`, `FALCONN`, and `HNSW` with fixed parameters over a range of datasets. Each algorithm has a distinct performance curve. In particular, `FALCONN` has very fast query times for low recall values; the other two algorithms appear to have some base cost associated with each query that prevents this behavior. Although all algorithms take a performance hit for high recall values, `HNSW` (when it has time to complete its preprocessing) is the least affected. All algorithms show a sharp transition for the random dataset; this is to be expected based on the dataset's composition (cf. **Datasets** above).

**(Q3) Approximation.** Figure 7 relates achieved QPS to the (approximate) recall of an algorithm. The plots show results on the NYTimes dataset for recall with no approximation and approximation factors of 1.01 and 1.1. The dataset is notoriously difficult; with no approximation, only a handful of algorithms can achieve a recall above 0.98. However, we know the candidate sets of most algorithms are very close to the true nearest neighbors, as even a very small approximation factor of 1.01 improves the situation drastically: all algorithms (except `FAISS`) get more than 0.9 recall. Allowing for an approximation of 1.1 yields very high performance for most

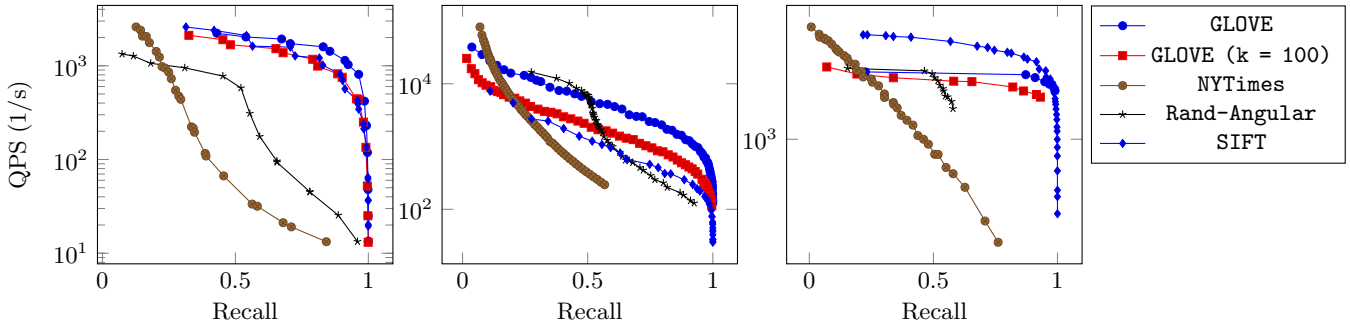


**Fig. 6.** Recall-QPS (1/s) tradeoff - up and to the right is better, 10-nearest neighbors, left: `Annoy`, middle: `FALCONN`, right: `HNSW`.

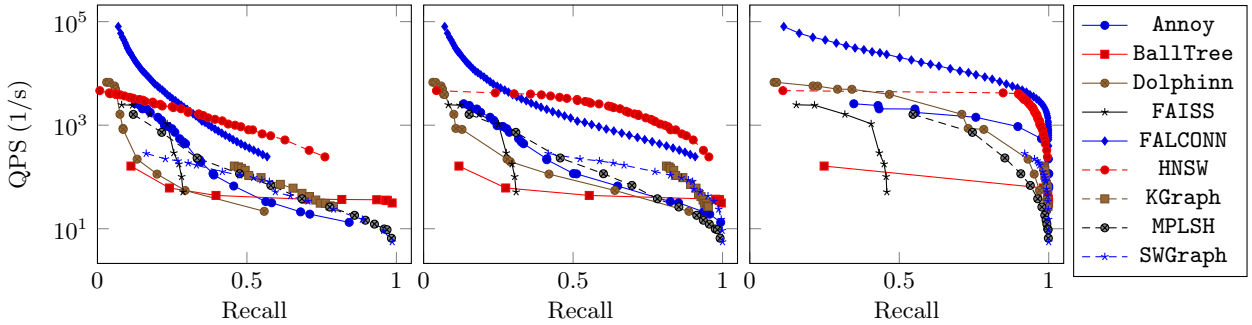


**Fig. 7.** (Approximate) Recall-QPS (1/s) tradeoff - up and to the right is better, nytimes dataset; left: $\varepsilon = 0$, middle: $\varepsilon = 0.01$, right: $\varepsilon = 0.1$

algorithms, although some benefit more than others: `FALCONN`, for example, now always outperforms `HNSW`, while `Annoy` suddenly leaps ahead of its competitors.

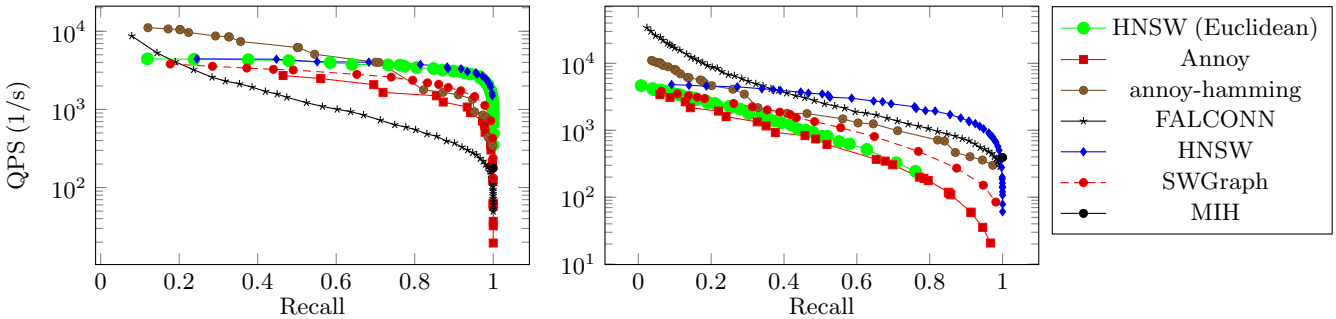


**Fig. 8.** Recall-QPS (1/s) tradeoff - up and to the right is better, 10-nearest neighbors, left: SIFT-Hamming, right: NYTimes-Hamming.

**(Q4) Embeddings.** Figure 8 shows a comparison between selected algorithms on the binary versions of SIFT and NYTimes. The performance plot for HNSW in the original Euclidean-space version is also shown. On SIFT, algorithms perform very similarly to the original Euclidean-space version (see Figure 4), which indicates that the queries are as difficult to answer in the embedded space as they are in the original space. The behavior is very different on NYTimes, where all algorithms improve their speed *and* quality. The only dedicated Hamming space algorithm shown here, exact multi-index hashing, shows good performance at around 180 QPS on SIFT and 400 QPS on NYTimes.

As an experiment, we created a Hamming space-aware version of Annoy, using the popcount intrinsic for distance computations, and sampling single bits (as in Bitsampling LSH [15]) instead of choosing hyperplanes as in random projection forests. This version is an order of magnitude faster on NYTimes; on SIFT, the running times converge for high recall values.

The embedding into Hamming space does have some consistent benefits that we do not show here. Hamming space-aware algorithms should always have smaller index sizes, for example, due to the compactness of bit vectors stored as binary strings.

## 5 Conclusion & Further Work

We introduced ANN-Benchmarks, an automated benchmarking system for approximate nearest neighbor algorithms. We described the system and used it to evaluate existing algorithms. Our evaluation showed that well-engineered solutions for Euclidean and Cosine distance exist, and many techniques allow for fast nearest neighbor search algorithms. At the moment, graph-based approaches such as HNSW or KGraph outperform the other approaches for very high recalls, whereas LSH-based solutions such as FALCONN yield very high performance at lower recall values. Index building for graph-based approaches takes a long time for datasets with difficult queries. We did not find solutions targeting Hamming space under Hamming distance, but showed that substituting Hamming space-specific techniques into more general algorithms can greatly improve their running time.

In future, we aim to add support for other metrics and quality measures, such as positional errors [29]. Preliminary support exists for set similarity under Jaccard distance, but algorithm implementations are missing. Testing Hamming space-aware versions of the graph-based methods and FALCONN could also be instructive, as could benchmarking GPU-powered nearest neighbor algorithms, one of which is already included in FAISS [16]. We also intend to simplify and further automate the process of re-running benchmarks when new versions of certain algorithms appear.

## References


1. Ahle, T.D., Aumüller, M., Pagh, R.: Parameter-free locality sensitive hashing for spherical range reporting. In: SODA'17. pp. 239–256
2. Alman, J., Williams, R.: Probabilistic polynomials and hamming nearest neighbors. In: FOCS'15. pp. 136–150
3. Andoni, A., Indyk, P., Laarhoven, T., Razenshteyn, I.P., Schmidt, L.: Practical and optimal LSH for angular distance. In: NIPS'15. pp. 1225–1233. https://falconn-lib.org/

4. Avarikioti, G., Emiris, I.Z., Psarros, I., Samaras, G.: Practical linear-space approximate near neighbors in high dimension. CoRR abs/1612.07405 (2016)
5. Bentley, J.L.: Multidimensional binary search trees used for associative searching. Commun. ACM 18(9), 509–517 (1975)
6. Bernhardsson, E.: Annoy, https://github.com/spotify/annoy
7. Boytsov, L., Naidan, B.: Engineering efficient and effective non-metric space library. In: SISAP'13. pp. 280–293
8. Boytsov, L., Novak, D., Malkov, Y., Nyberg, E.: Off the beaten path: Let's replace term-based retrieval with k-nn search. In: CIKM'16. pp. 1099–1108
9. Ciaccia, P., Patella, M., Zezula, P.: M-tree: An efficient access method for similarity search in metric spaces. In: VLDB'97. pp. 426–435 (1997)
10. Curtin, R.R., Cline, J.R., Slagle, N.P., March, W.B., Ram, P., Mehta, N.A., Gray, A.G.: MLPACK: A scalable C++ machine learning library. Journal of Machine Learning Research 14, 801–805 (2013)
11. Dong, W.: KGraph, https://github.com/aaalgo/kgraph
12. Dong, W., Wang, Z., Josephson, W., Charikar, M., Li, K.: Modeling LSH for performance tuning. In: CIKM'08. pp. 669–678. ACM, http://lshkit.sourceforge.net/
13. Edel, M., Soni, A., Curtin, R.R.: An automatic benchmarking system. In: NIPS 2014 Workshop on Software Engineering for Machine Learning (2014)
14. Heo, J.P., Lee, Y., He, J., Chang, S.F., Yoon, S.E.: Spherical hashing: Binary code embedding with hyperspheres. IEEE TPAMI 37(11), 2304–2316 (2015)
15. Indyk, P., Motwani, R.: Approximate nearest neighbors: Towards removing the curse of dimensionality. In: STOC'98. pp. 604–613
16. Johnson, J., Douze, M., Jégou, H.: Billion-scale similarity search with gpus. CoRR abs/1702.08734 (2017)
17. Johnson, W.B., Lindenstrauss, J., Schechtman, G.: Extensions of lipschitz maps into banach spaces. Israel Journal of Mathematics 54(2), 129–138 (1986)
18. LeCun, Y., Bottou, L., Bengio, Y., Haffner, P.: Gradient-based learning applied to document recognition. Proceedings of the IEEE 86(11), 2278–2324 (1998)
19. Lyst Engineering: Rpforest, https://github.com/lyst/rpforest
20. Malkov, Y.A., Yashunin, D.A.: Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs. ArXiv e-prints (Mar 2016)
21. Malkov, Y., Ponomarenko, A., Logvinov, A., Krylov, V.: Approximate nearest neighbor algorithm based on navigable small world graphs. Inf. Syst. 45, 61–68 (2014)
22. Mikolov, T., Sutskever, I., Chen, K., Corrado, G.S., Dean, J.: Distributed representations of words and phrases and their compositionality. In: NIPS'13. pp. 3111–3119
23. Muja, M., Lowe, D.G.: Fast approximate nearest neighbors with automatic algorithm configuration. In: VISSAPP'09. pp. 331–340. INSTICC Press
24. Norouzi, M., Punjani, A., Fleet, D.J.: Fast search in hamming space with multi-index hashing. In: CVPR'12. pp. 3108–3115. IEEE
25. Pham, N.: Hybrid LSH: faster near neighbors reporting in high-dimensional space. In: EDBT'17. pp. 454–457
26. Van Rijn, J.N., Bischl, B., Torgo, L., Gao, B., Umaashankar, V., Fischer, S., Winter, P., Wiswedel, B., Berthold, M.R., Vanschoren, J.: Openml: A collaborative science platform. In: ECML PKDD. pp. 645–649. Springer (2013)
27. Wang, J., Shen, H.T., Song, J., Ji, J.: Hashing for similarity search: A survey. CoRR abs/1408.2927 (2014), http://arxiv.org/abs/1408.2927
28. Williams, R.: A new algorithm for optimal 2-constraint satisfaction and its implications. Theor. Comput. Sci. 348(2-3), 357–365 (2005)
29. Zezula, P., Savino, P., Amato, G., Rabitti, F.: Approximate similarity retrieval with M-Trees. VLDB J. 7(4), 275–293 (1998)