Reproducibility Paper for ANN-Benchmarks: A benchmarking tool for approximate nearest neighbor search algorithms[⋆,⋆⋆]

Martin Aumüller[a], Erik Bernhardsson[b], Alexander Faithfull[a]

[a] *IT University of Copenhagen, Denmark*
[b] *Better, Inc., United States*

Abstract

In (Aumüller, Bernhardsson, Faithful, *Information Systems*, 2020), a benchmarking framework for nearest neighbor search implementations was introduced. This framework was used to evaluate a selection for nearest neighbor search algorithms on different datasets. This companion paper details the experimental setup and provides a step-by-step description to reproduce the original results.

1. Introduction

Nearest neighbor search is one of the central techniques in many diverse areas of computer science such as image processing, recommender systems, data mining, and machine learning. The task of a nearest neighbor search algorithm is to preprocess a dataset $X \subseteq \mathbb{R}^d$ of n d-dimensional data points to answer nearest neighbor queries: Given a query point $x \in \mathbb{R}^d$, return the k nearest neighbors to x in X. While this can be efficiently solved for low-dimensional settings, such as $d \in \{2, 3\}$, exact algorithms often fall back to being similar (or worse) than a linear scan in high dimensions, a phenomenon called the "curse of dimensionality".

This paper is a companion paper to [1], in which a benchmarking framework for implementations of nearest neighbor search algorithms is presented. [1] focused on a succinct description of the general approach of the framework, and presented an evaluation of state-of-the-art nearest neighbor search algorithms at the end of 2017 until submission in mid-2018.

Relation to current state of ANN-Benchmarks. This paper details the exact steps needed to reproduce the plots in the paper [1]. Since the submission of the paper, many new algorithms were added to ann-benchmarks, and existing ones

⋆DOI of original article: https://doi.org/10.1016/j.is.2019.02.006
⋆⋆Code repository: https://github.com/maumueller/ann-benchmarks-reproducibility
Research artifacts: https://doi.org/10.5281/zenodo.4607761

were refined. See `http://ann-benchmarks.com` for an up-to-date overview of nearest neighbor search implementations.

A note to the reviewers. To speed up the reproducibility process, we would appreciate if problems are directly reported as issues via `https://github.com/maumueller/ann-benchmarks-reproducibility`.

2. Experiments in Aumüller et al. [1]

We invite the reader to first take a look at [1] to get an idea about the scope of the framework. In a nutshell, we used our framework to compare many state-of-the-art nearest neigbor search algorithms on a broad collection of high-dimensional datasets. From each dataset, a certain collection of points was chosen as queries and presented to the algorithms.[1] Implementations were measured on their ability to quickly return a "good approximation" of the true nearest neighbors. Usually, this means that the throughput (measured in *queries per second*) was put into relation to the *average recall* (the average of the fraction of correct nearest neighbors among the query answers over all queries).

Results were reported on these performance/quality measures, but also on questions such as "how long does it take to build an index that will allow to achieve a recall of at least .9?" and "how adaptive are algorithms?"

3. Framework Overview

3.1. Code base

ANN-Benchmarks is primarily written in `Python`. It takes care of setting up and running an experiment. An experiment consists of running k-NN queries for a specified implementation on a predefined dataset. All implementations considered in [1] are listed in Table 1. One run consists of building the index for a list of dataset points (using parameters related to *index building*), and running queries with parameters related to *query processing*. The authors of the individual implementations provided these parameter choices themselves, and ann-benchmarks just carries out the experiment using these parameters. The actual wrappers for the implementations are found in `ann_benchmarks/algorithms/`, a standard set of parameters can be found in `algos.yaml`.

3.2. An Overview over the Architecture

ANN-Benchmarks uses Docker to encapsulate different implementations. This was a necessary step to allow easy handling of different implementations, which each have their own dependencies. It also allows to limit the resources of each container, since we run all implementations single-threaded and enforce a limit

[1] For [1] those queries were chosen at random, but more refined approaches were later introduced in [2].

Principle	Algorithms
graph-based	`KGraph (KG)` [3], `SWGraph (SWG)` [4, 5], `HNSW` [6, 5], `PyNNDescent (NND)` [7], `PANNG` [8], `ONNG` [8, 9]
tree-based	*FLANN* [10], *BallTree (BT)* [5], `Annoy (A)` [11], `RPForest (RPF)` [12], `MRPT` [13]
LSH	*MPLSH* [14, 5]
other	`Multi-Index Hashing (MIH)` [15] (exact Hamming search), `FAISS-IVF (FAI)` [16] (inverted file)

Table 1: Overview of tested algorithms (abbr. in parentheses).

Dataset	Internal Name	#Data/#Query	Dim.	Metric
SIFT	`sift-128-euclidean`	1 000 000 / 10 000	128	Euclidean
GIST	`gist-960-euclidean`	1 000 000 / 10 000	960	Euclidean
GLOVE	`glove-100-angular`	1 183 514 / 10 000	100	Angular
NYTimes	`nytimes-256-angular`	234 791 / 10 000	256	Euclidean
Rand-Euclidean	`random-10nn-euclidean`	1 000 000 / 10 000	128	Angular
SIFT-Hamming	`sift-256-hamming`	1 000 000 / 1 000	256	Hamming
Word2Bits	`word2bits-800-hamming`	399 000 / 1 000	800	Hamming

Table 2: Datasets under consideration.

on the memory usage. The main controller that manages all experiments lives outside the Docker environment and invokes different Docker containers based on the experiment that is run. During setup of the container, it mounts the Python module and the *data/* folder containing all datasets as read-only into the container, and mounts the *results/* directory as read-write.

Since docker-in-docker environments and parallel docker environments are difficult to manage, we assume in this reproducibility that we start with a fresh Ubuntu 18.04 VM.

3.3. Dataset format

Table 2 gives an overview over the datasets considered in [1]. Generating these datasets is done by the script `create_datasets.py`, which internally calls `ann_benchmarks/dataset.py`. Each dataset is internally stored as an `hdf5` file, that contains the dataset points, the query points, the ids of the 100-nearest neighbors to each query point, and the distances of these points to the query. Since building this ground truth takes a considerable amount of time, all datasets are stored on `https://ann-benchmarks.com`. Before creating the dataset locally, the script always tries to download the `hdf5` file first.

3.4. Result format

After finishing a run, a single `hdf5` file containing the results of the run is written to the file system in the *results/* folder. See Figure 1 for a partial snapshot of this directory. We stress that we write the *raw answer* of the query algorithm of the algorithm under consideration, that means the identifiers of the

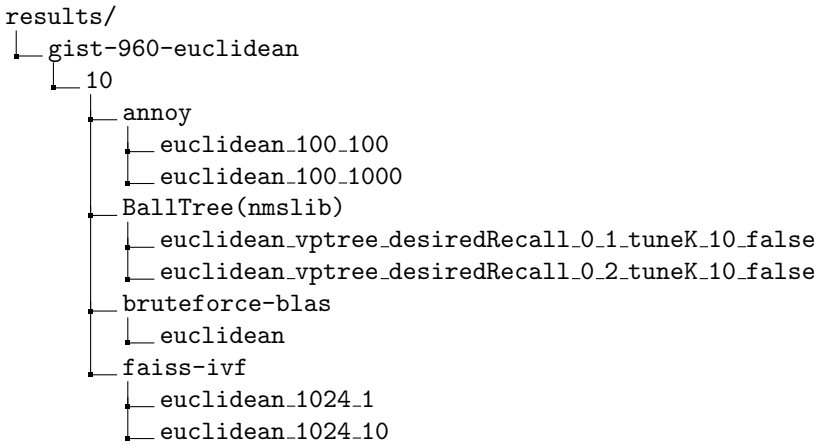

Figure 1: Overview over results in results folder

nearest neighbors returned for the individual queries. Only later on, this data is used to compute metrics such as (approximate) recall. The hierarchy it uses is *dataset/number_of_nearest_neighbors/algorithm/*. Each file contains the results of a single run (i.e., one set of query parameters). Each file contains general information of the run and details of the measurements, such as the identifiers of the nearest neighbors that were returned, query times, build time of the index, etc. The result file can be explored in the interactive Python console as shown in Figure 2.

3.5. Post-Processing Results

After running the experiments, there are a couple of choices to visualize or process the data for another setting. The script `plot.py` creates a single PNG plot on a specific dataset for two gives metrics, such as recall and throughput. `create_website.py` generates a website that visualizes *all runs* that can be found in the *results/* directory, split up by dataset (with varying algorithms) or algorithm (with varying datasets). For reproducing [1], the main scripts are *data_export.py* and *reproducibility/generate_result_tables.py*, which exports the raw data to a csv file and generates data that can be plotted via *pgfplots*. The latter script also has special code to generate Figures 10 and 13 in [1].

4. Reproducibility Experiment

This section describes the workflow to reproduce the experimental results from [1]. The original experiments were carried out on an Amazon EC2 c5.4xlarge instance, which was equipped with Intel Xeon Platinum 8124M CPU (16 available cores, 3.00 GHz, 25MB L3 Cache) and 32 GB of RAM using Amazon Linux. For the sake of reproduction, we ran the scripts below on a local machine with 2x 14-core Intel Xeon E5-2690v4 (2.60GHz) with 512 GB of RAM using Ubuntu

```
>>> f = h5py.File("euclidean_reverse_1_true_100")
>>> dict(f)
{'metrics': <HDF5 group "/metrics" (3 members)>, '
    ↪ distances': <HDF5 dataset "distances": shape
    ↪ (1000, 10), type "<f4">, 'times': <HDF5 dataset
    ↪  "times": shape (1000,), type "<f4">, '
    ↪ neighbors': <HDF5 dataset "neighbors": shape
    ↪ (1000, 10), type "<i4">}
>>> dict(f.attrs)
{'algo': 'kgraph', 'distance': 'euclidean', '
    ↪ run_count': 2, 'batch_mode': False, 'dataset':
    ↪ 'gist-960-euclidean', 'build_time':
    ↪ 2678.368325471878, 'count': 10, 'name': 'KGraph
    ↪ (euclidean)', 'best_search_time':
    ↪ 0.014966160058975 22, 'index_size': 2022140.0, '
    ↪ expect_extra': False, 'candidates': 10.0}
>>> f["times"][:10] # individual query times of the
    ↪ first 10 queries
array([0.0064466 , 0.01282668, 0.00558615, 0.0106256
    ↪ , 0.01036716, 0.005831  , 0.00877213,
    ↪ 0.01525331, 0.00530338, 0.01109028],
      dtype=float32)
```

Figure 2: Structure of the HDF5 result file.

16.10 with kernel 4.4.0. In Section 5, we report on the differences between these two architectures.

4.1. Reproduction from a fresh Ubuntu 18.04 installation

Figure 3 provides an overview over the whole reproducibility process.

Installation. We provide a *Vagrantfile* in the research artifacts discussed in Section 7 that automatically sets up a VM ready to carry out the experiments.

To reproduce all steps, start from a fresh installation of Ubuntu 18.04. Next, install Docker as outlined on `https://docs.docker.com/engine/install/ubuntu/`. Then, install Python 3.6 as follows:

```
$ sudo apt-get update
$ sudo apt-get install -y python3-pip build-essential git
```

Finally, clone the repository and install necessary dependencies and compile and install the nearest neighbor search implementations mentioned in Table 1. The `--proc` flag specifies that the installation should use five processes in parallel.

```
$ git clone https://github.com/maumueller/ann-benchmarks-
    reproducibility
```

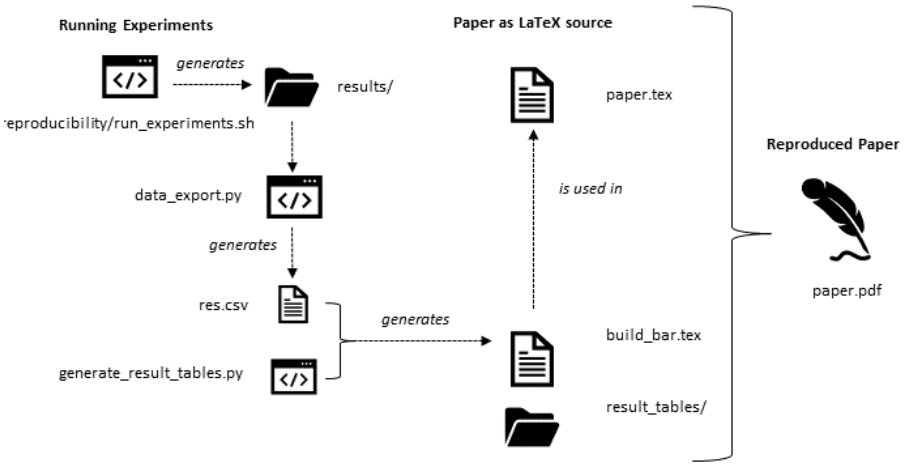

Figure 3: Overview of the reproducibility process.

```
$ cd ann-benchmarks-reproducibility
$ pip3 install -rrequirements.txt
$ python3 install.py --proc 5
```

We note that these steps will work for more recent versions of Python, but the file `requirements.txt` has to be updated to contain recent versions of libraries. At the time of writing this article, the most recent choices for each library worked well. At the end of running the installation script, the individual implementations will report on a successful or failed installation. It is necessary that all installations succeeded before proceeding.

Running CPU-based experiments. We are now ready to run all CPU-based experiments by running

```
$ PY=python3 PARALLELISM=10 bash reproducibility/
    run_experiments.sh
```

All individual runs of experiments in this part are carried out on a single CPU using Docker. The environmental variable *PARALLELISM* can be used to spawn multiple containers in parallel. For a 10-core machine, we suggest using a value of 5. On the machine specified above, we used *PARALLELISM*=20. The environmental variable *PY* can be used to point to a custom Python 3.6 installation, e.g., provided by Anaconda. Note that for the largest dataset GIST-960-Euclidean, around 20GB of RAM are necessary per process, which meant in our setup that we had a peak memory usage of 400 GB. The environmental variable *GISTPARALLELISM* controls the number of parallel instances run for the GIST dataset. Invoking

```
$ PY=python3 PARALLELISM=10 GISTPARALLELISM=3 bash
    reproducibility/run_experiments.sh
```

corresponds to 3 parallel runs on GIST, and 10 on all other datasets. A machine used for running this experiment should contain at least 64GB of RAM. On our test machine, reproducing all CPU based experiments with the parameters above took around 4 days.

Running GPU-based experiments. The paper [1] contains a single run of a GPU-based variant in Figure 12. This run was carried out in a local environment outside a docker container. To reproduce this run, we provide a script in *reproducibility/run_gpu.sh*. A Linux-based environment with a CUDA runtime of at least 10.0 is necessary. This can be checked by inspecting the output of `nvidia-smi`. Furthermore, the `nvidia-runtime` for docker must be installed, as detailed in `https://github.com/NVIDIA/nvidia-docker`. If these requirements are met, the GPU run is reproduced by running:

```
$ python3 install.py --algorithm faissgpu
$ bash reproducibility/run_gpu.sh
```

Our machine was equipped with a Quadro M4000 with compute engine 5.2 and all runs where finished within 10 minutes. If the reproducibility environment features an older GPU, the version of the compute engine must be manually set during compilation of FAISS in `install/Dockerfile.faissgpu` by editing the flag `DCMAKE_CUDA_ARCHITECTURES="75;72;52"`.

Reproducing the Paper From The Results. If all runs above have been carried out, we can start reproducing the plots in the paper. Run

```
$ sudo python3 data_export --out res.csv
$ mkdir -p paper/result_tables/
$ python3 reproducibility/create_result_tables.py res.csv
    paper/result_tables/
```

to create all the raw tables used by *pgfplots* during the final LaTeX compilation. Since exporting the results will compute all quality metrics, it took around 1 hour on our machine. (However, results are cached, so this cost applies only once.) Now, compile the paper by changing to the *paper* directory and compiling *paper.tex*, i.e.,

```
$ cd paper && latexmk -pdf paper.tex
```

This requires a standard latex installation for scientific writing. If such a system is not present, we provide another Docker container in *paper*. The reproducibility steps are then

```
$ docker build . -t ann-benchmarks-reproducibility-latex
$ docker run -it -v "$(pwd)"/:/app/:rw ann-benchmarks-
    reproducibility-latex:latest
```

from within the *paper* directory. The compilation will fail (or miss certain plot lines) if some runs did not finish in time. The compilation log will contain the name of all runs that are missing, which allows to individually re-run some experiments, for example with longer timeouts.

The final PDF can be seen in *paper/paper.pdf* and the plots can be compared to the original paper [1].

4.2. Reproduction from the original raw results

To avoid rerunning all experiments, the raw result of the original runs can be accessed from the research artifacts (see Section 7). It is required to complete the *Installation* step in Section 4.1. Unpack the results into the *ann-benchmarks* folder, and run the same steps as above.

5. Differences between [1] and our reproducibility experiment

Since most of our experiments consider the raw throughput achieved by the implementations, the compute architecture has a big influence on the individual plots. The throughput results on the Intel Xeon E5-2690v4 are roughly 1.5 to 2 times slower than the architecture used in the original paper. However, general trends translate well into the new setting. Figure 4 and Figure 5 compare Figure 7 in [1] to the measurements on the machine used for reproduction. While absolute performance decreases, performance trends are comparable. We added two versions of [1] using the original results and the results from the reproducibility work to research artifacts discussed in Section 7.

We noticed the following differences between the reproduced plots and [1].

1. **Performance of `NND`.** The implementation of `PyNNDescent` [7] performed worse (in relation to others) on the new setup. We tried other (old) versions, but the results were the same. More recent versions perform much better (being on par with HNSW in many cases), but we decided to report using an old version that is closer to the original performance.
2. **Omitted data points.** To improve the readability of the plots, we manually removed some data points in the original paper. For example, Figure 6 contained many data points with recall close to 1 which were removed. The reproduced version does not clean such data points.
3. **Differences in Figure 9.** `PANNG` is much faster in the reproducibility setup than in the original paper. This is because `PANNG` and `ONNG` are part of one library, and we had to use a more recent version to include `ONNG`. In the original paper, `PANNG` experiments where carried out in spring 2017, whereas the `ONNG` runs were done in autumn 2018. Furthermore, the line for `Annoy (eucl.)`on the plot to the left was wrong in the original paper. The performance is much better, as reported in the reproducibility experiment. One can see the mistake by a careful comparison between Figure 4 (bottom, left) and Figure 9 in [1].
4. **Longer build times.** We were not able to build indices that would allow for the same recall of HNSW on the reproducibility architecture. We increased the timeout to 12 hours (from 6 in the original paper) for an individual experiment.

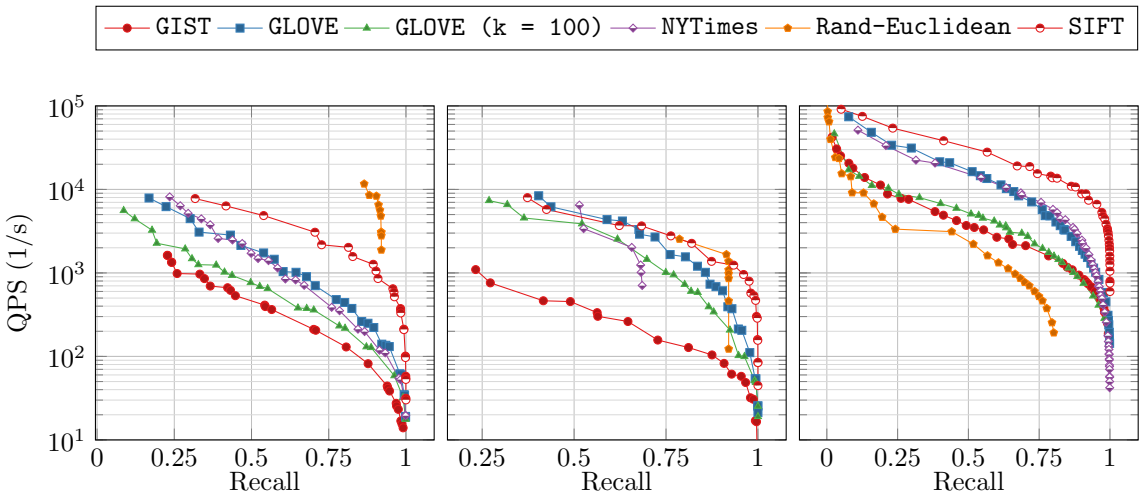

Figure 4: **Original:** Recall-QPS (1/s) tradeoff - up and to the right is better, 10-nearest neighbors unless otherwise stated, left: `Annoy`, middle: `FAISS-IVF`, right: `HNSW`.

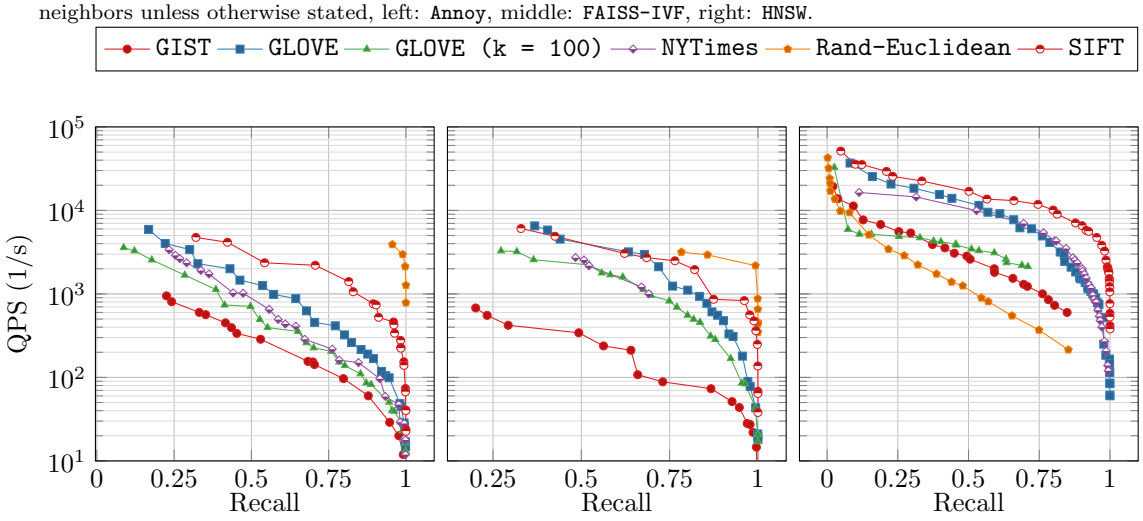

Figure 5: **Reproduced:** Recall-QPS (1/s) tradeoff - up and to the right is better, 10-nearest neighbors unless otherwise stated, left: `Annoy`, middle: `FAISS-IVF`, right: `HNSW`.

5. **Differences in Figure 12.** The reproducibility machine has more cores and thus batch runs on the CPU are faster. On the other hand, its GPU is worse, so the GPU runs are slower. This means that the differences between CPU and GPU runs in Figure 12 are not as pronounced as in the original paper.

6. Reflection on the Reproducibility Setup

Given the use of Docker in the ann-benchmarks setup, it proved difficult to provide a fully dockerized environment. We thus had to resort to providing a VM image which uses docker internally. However, this makes it difficult to run the GPU-based experiments. On the other hand, ann-benchmarks comes with a very lightweight set of dependencies and is easy to install locally.

ANN-Benchmarks is a work in progress. Many parts of the benchmarking framework and the benchmarked implementations changed over time. This present paper describes the steps to reproduce [1], but the very same setup works for producing all results on more recent versions. In particular, we used the version of ann-benchmarks from January 2021 to reproduce the old results from 2017 and 2018. The main difficulty was in tracing the exact versions of the nearest neigbor search implementations in their GitHub repositories.

Of the time of writing, ann-benchmarks compares 26 different nearest neighbor search implementations while the experiments in this paper used only 14. See `ann-benchmarks.com` for an up-to-date overview of nearest neighbor search algorithms.

7. Research Artifacts

All research artifacts are provided in [17]. It fixes the version of the code used to produce the results in this reproducibility study. It also contains tar archives containing (i) all datasets used in the study, (ii) the original raw results used to produce [1], (iii) the raw results that we got from this reproducibility work, and (iv) a *Vagrantfile* that spawns an Ubuntu VM ready to run all experiments.

References

[1] M. Aumüller, E. Bernhardsson, A. J. Faithfull, Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms, Inf. Syst. 87 (2020).

[2] M. Aumüller, M. Ceccarello, The role of local intrinsic dimensionality in benchmarking nearest neighbor search, in: SISAP, Vol. 11807 of Lecture Notes in Computer Science, Springer, 2019, pp. 113–127.

[3] W. Dong, KGraph.
URL `https://github.com/aaalgo/kgraph`

[4] Y. Malkov, A. Ponomarenko, A. Logvinov, V. Krylov, Approximate nearest neighbor algorithm based on navigable small world graphs, Inf. Syst. 45 (2014) 61–68.

[5] L. Boytsov, B. Naidan, Engineering efficient and effective non-metric space library, in: SISAP'13, pp. 280–293.

[6] Y. A. Malkov, D. A. Yashunin, Efficient and robust approximate nearest neighbor search using Hierarchical Navigable Small World graphs, ArXiv e-prints (Mar. 2016). arXiv:1603.09320.

[7] L. McInnes, PyNNDescent.
URL https://github.com/lmcinnes/pynndescent

[8] Y. J. Corporation, NGT: PANNG.
URL https://github.com/yahoojapan/NGT

[9] M. Iwasaki, D. Miyazaki, Optimization of Indexing Based on k-Nearest Neighbor Graph for Proximity Search in High-dimensional Data, ArXiv e-prints (Oct. 2018). arXiv:1810.07355.

[10] M. Muja, D. G. Lowe, Fast approximate nearest neighbors with automatic algorithm configuration, in: VISSAPP'09, INSTICC Press, pp. 331–340.

[11] E. Bernhardsson, Annoy.
URL https://github.com/spotify/annoy

[12] Lyst Engineering, Rpforest.
URL https://github.com/lyst/rpforest

[13] V. Hyvönen, T. Pitkänen, S. Tasoulis, E. Jääsaari, R. Tuomainen, L. Wang, J. Corander, T. Roos, Fast nearest neighbor search through sparse random projections and voting, in: Big Data (Big Data), 2016 IEEE International Conference on, IEEE, 2016, pp. 881–888.
URL https://github.com/teemupitkanen/mrpt

[14] W. Dong, Z. Wang, W. Josephson, M. Charikar, K. Li, Modeling LSH for performance tuning, in: CIKM'08, ACM, pp. 669–678.
URL http://lshkit.sourceforge.net/

[15] M. Norouzi, A. Punjani, D. J. Fleet, Fast search in hamming space with multi-index hashing, in: CVPR'12, IEEE, pp. 3108–3115.

[16] J. Johnson, M. Douze, H. Jégou, Billion-scale similarity search with gpus, CoRR abs/1702.08734 (2017).

[17] M. Aumüller, E. Bernhardsson, A. Faithfull, Research Artifacts for Reproducibility Paper "ANN- Benchmarks: A benchmarking tool for approximate nearest neighbor search".
URL https://doi.org/10.5281/zenodo.4607761