Running experiments with confidence and sanity

Martin Aumüller[1] and Matteo Ceccarello[2]

[1] IT University of Copenhagen, Denmark
maau@itu.dk
[2] Free University of Bozen, Italy
mceccarello@unibz.it

Abstract. Analyzing data from large experimental suites is a daily task for anyone doing experimental algorithmics. In this paper we report on several approaches we tried for this seemingly mundane task in a similarity search setting, reflecting on the many errors and consequent mishaps. We conclude by proposing a workflow, which can be implemented using several tools, that allows to analyze experimental data with confidence.

Keywords: Experimental algorithmics; Experimental analysis

1 Introduction

One of the peculiar aspects of *experimental algorithmics* [18] is that the object of the study (an algorithm and its implementation) is often crafted by the same people carrying out the analysis. This has the advantage that the insights obtained from preliminary investigations of early versions of an algorithm can be used to improve the algorithm itself. In fact, the understanding required for an implementation may uncover features of the algorithms that would otherwise go unnoticed [18], giving insights about aspects not easily described by theoretical models of computation [16]. At the same time, this feedback-based process leads to the accumulation of obsolete data, referring to old versions of algorithms and their implementations. Not mixing results from different versions of an algorithm or implementation is an obvious requirement, which however requires some care in practice. In fact, a study often involves different algorithms and datasets, each evolving at a different pace: weeks-old results might be up to date for one algorithm, and obsolete for another.

As we shall see, the literature is mainly concerned with the design and analysis of experiments and with reproducibility. In this position paper, instead, we report on our experience with the day to day tasks that have to be carried out in between those three tasks, and the approaches we developed to tackle the perils and frustrations of this often menial work.

We do not advocate for any specific technology. Rather, we propose a workflow that can be implemented with a variety of tools that can be easily integrated into existing setups. We demonstrate such a setup with a toy project that concerns an efficient implementation of a brute-force nearest neighbor search.

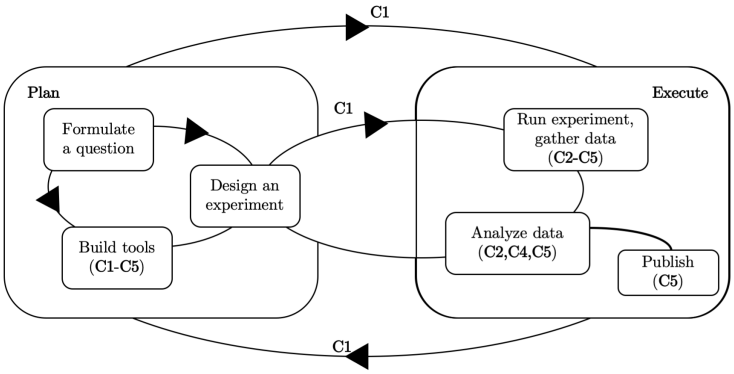

Fig. 1. Overview of the different stages of an experimental study; adapted from [17].

2 Related work

Moret and Shapiro [18] advocate for the importance of complementing the theoretical analysis of algorithms with their implementation. McGeoch [17] gives several guidelines on how to design and carry out experimental analyses of algorithms. The book [3] collects several contributions on the characterization and analysis of algorithm performance. Earlier, a Dagstuhl seminar was devoted to the discussion of the experimental evaluation of algorithms [11]. More recently, a structured approach to experimental analysis was discussed in [4].

In recent years there has been a discussion about the lack of reproducibility of research findings in several areas, including computer science [9,13]. Much effort has been devoted to finding a solution to this issue. Several contributions have been collected in [20] and [14]. Among the tools to support reproducible research, VisTrails [7] allows to explicitly define reproducible workflows. `knitr` and `Jupyter` take a *literate programming* approach, allowing experiment's code, analysis, and text to be interleaved in a single "executable" document. To solve the issues deriving from software dependencies, some tools aim at capturing the execution environment at runtime [12,10,19], while others such as Docker [5] and Singularity [15] follow a *declarative* approach, where the description of the execution environment is part of the code base.

3 Challenges in large scale experimental evaluation

We define the following challenges of running large-scale experiments:

(C1) **Feedback Loops-by-Design.** Implementations and tools support the iterative nature of an experimental study.

(C2) **Economic Execution.** Exactly those experiments that change through code changes have to be re-run, but nothing else. Moreover, only the changing parts of the experimental evaluation should be recomputed.

(C3) **Versioning.** The ability to go back in time and compare old results to more recent ones, finding regressions or bugs; the workflow is *append-only*.
(C4) **Machine Independence.** Code and tools are designed in a way that allow them to run in a general setting.
(C5) **Reproducibility-by-Design.** We strive for an automatic workflow that processes an experimental setup into measurements used for evaluating the experiment. Results to be included into a publication should not require manual work to transform these measurements into tables and plots.

Typically, an experimental evaluation spans several weeks, if not months. An overview of a typical experimental evaluation is given in Figure 1. During this time, the experiments being run have different meanings: early on during initial development, experiments are useful to find out the most appropriate parameter ranges, find bugs, and check assumptions; later on, experiments collect the results of the study. This is not a process that proceeds linearly from start to finish. Rather, the analysis of the results might prompt the modification of an algorithm or dataset, or the introduction of new algorithms and datasets into the study, followed by a new round of experiments. Together with the algorithms and the datasets, also the parameterizations and the quantities being measured are subject to evolution during the lifetime of a project (C1).

For the analysis to be sound, it is of paramount importance not to mix results related to different versions of the algorithms and datasets (C3), in particular when experiments are run on a set of different machines (C4). The simplest solution would be to re-run the entire experimental suite whenever something is modified. This usually takes a very long time, and a change might affect only a small part of the results, making this solution wasteful of time, energy, computational resources and money if computing resources are rented (C2). A potential solution might be to divide the experimental suite in smaller components, each investigating a particular aspect, re-running only those affected by a change. While this works in the short term, as the experimental study progresses the subdivision of the experimental suite will evolve with it, leading to the need of re-arranging the results. On the other hand, manually re-running only parts of an experimental suite, while reusing results from old runs, requires much care in order to exclude obsolete results from the analysis, undermining the confidence in the soundness of the whole analysis. The situation worsens in the rushed final days preceding a submission: some last minute changes are made, there is no time to re-run all the experiments, the possibility of erroneously mixing results is very concrete. Additionally, reviewers will often demand running a new set of experiments, reporting on some other quality measures, or experimentation using different computer architectures (C2, C3, C4). *Reproducibility-By-Design* (C5) requires that such wishes can be accommodated since the whole process from starting at an experimental design to a published table or figure is automated.

As for the analysis itself, it is usually executed on a machine different from the experimental code, using a different programming language (C4). The input of the analysis is the set of results produced by the experimental suite, which is usually quite large due to the fact that many parameter combinations need to be

evaluated. While the analysis code may not need the computational resources of the experimental code, it still needs to execute reasonably fast, in order to be able to examine the results interactively. This implies that the results produced by the experiments need to be stored in a convenient format that is at the same time easily manageable, convenient to transfer, and efficient to access (C2, C4).

4 Case study: Engineering a Linear Scan

As our toy project, we engineer a nearest neighbor search algorithm that just carries out a linear scan over the dataset[3].

Formally, we are given a dataset $S \subset \mathbb{R}^d$ of n points in a d-dimensional space with a distance measure $\text{dist}\colon \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$, such that given a query $q \in \mathbb{R}^d$ we want to return a point $p \in S$ that minimizes $\text{dist}(p', q)$ over all $p' \in S$. Solving this problem via a linear scan is a straight-forward exercise in an introduction to programming class: Compare all points $p' \in S$ one by one to q, and keep track of the point that is closest to q. This results in a running time $O(nd)$ per query.

To make this problem more interesting, we consider engineering choices to speed up a linear scan under inner product similarity $\text{dist}_{\text{IP}}(p, q) = \sum_{1 \le i \le d} x_i y_i$ on unit vectors, similar to Cosine similarity. For the purpose of this project, we consider (i) input representation, (ii) parallelization, and (iii) saving distance computations as factors of the experiment.

Input representation. A vector in $\mathbb{R}^d$ is traditionally represented as d 64-bit floating point values (`double`) or 32-bit floating point (`float`). Since we guarantee $0 \le x_i \le 1$ for normalized vectors, we also consider a 16-bit representations of the value $\lceil x_i \cdot 2^{16} \rceil / 2^{16}$ (which could of course affect the accuracy of the result.)

Parallelization. Naïvely, the CPU has to carry out d multiplications and $d - 1$ additions to compute the distance of two vectors. However, we notice that the structure is inherently parallel because the multiplications are data independent. This is an ideal setup for using so-called SIMD instructions (single instruction multiple data). We split up each vector into blocks of size B, and carry out d/B parallel multiplications, d/B parallel additions to aggregate terms in a register of size B, and one horizontal sum. Depending on the CPU architecture used in the experiment, B is usually 128, 256, or—very recently—512 bits.

Saving Distance Computations. Computing the distance between two vectors is certainly the most expensive operation in our linear scan. Hence, if we could decide for a data point p' that it probably is not the nearest neighbor faster than carrying out a distance computation could increase the performance of our linear scan. We include experiments with a 64-bit sketch using SimHash with probabilistic quality guarantees in our experiments (see Appendix B).

We consider this toy project representable for an experimentation task in a similarity search setting. The different choices of input representation, parallelization, and distance filter methods provide an evolutionary setting in which

[3] We would like to thank Michael Vesterli for the many code optimizations that we are using, that he developed for PUFFINN [2].

Fig. 2. Dimensions for running large-scale experimental evaluations.

we start with a standard linear scan and add features to the code base one by one. From starting with a measurement of running time, we quickly end up focusing on the quality of the achieved result when using a low-precision input representation, or analyzing the effectivity of the sketch by counting distance computations. The experiment has to be carried out on different machines because of the hardware dependencies, which might mean to rent cloud instances to carry out measurements on recent hardware with $B = 512$ bit AVX512 support.

Our code is provided at `https://github.com/Cecca/running-experiments`. For the scope of this paper, we consider the *support code* that takes care of handling the setup as the main contribution. For the interested reader, the evaluation of the toy project is given in Appendix C.

5 Approaches to experimental evaluation

We now describe a workflow we developed to address the challenges outlined in the previous sections, demonstrating it with our case study. We split up the discussion into different dimensions of running a successful experimental study. These dimensions are summarized in Figure 2. In the following, each dimension will be introduced with general guidelines and a discussion of our actual solution.

5.1 Manage the datasets and workloads efficiently

- Dataset download and preprocessing should be automated as much as possible, ideally with a single script responsible to manage all the datasets. This makes reproducibility easier, allows to share preprocessing steps between similar datasets, makes it easy to relocate the experiments on a different machine, and makes all the decisions about datasets explicit. Furthermore, it enables the community to change the datasets to observe how these changes are reflected in the experiments.
- It must be possible to create all datasets locally, but the preprocessed datasets should also be shared, for instance using plain `http` or a service such as `S3`. This makes it easier for collaborators, reviewers, and the community to re-run the experiments without incurring the set-up cost of the datasets.
- Datasets should be annotated with meta-data necessary in the evaluation, such as workloads and the related ground truth answers.

- To ease debugging, a small dataset of random data that can be created in a few seconds should also be included. This dataset can be run via Continuous Integration (CI), and results on it can be stored to enable regression testing.

In our code, the main C++ code calls a Python script (`datasets.py`) that takes care of preprocessing datasets in a well-defined manner. It checks for the existence of a shared dataset (of the same version) and computes it locally if such a dataset is not available. It supports that creation of tiny random datasets that allows to run all parts of the workflow on actual data. The query set that is latter used for experimentation is created in this process as well, and data is stored as an `HDF5` file for efficient processing in many different programming languages.

5.2 Manage the experimental configurations clearly

- Never run experiments from the command line. Direct command line execution should be limited to testing.
- Experiments should be described in one or more files. This makes it easier to reproduce the entire experimental suite. There are several options, which we both demonstrate in the associated code:
  - Files in a declarative language such as YAML listing all the combinations of parameters to be tested. These files are then interpreted by a script that spawns the appropriately configured experimental code. This approach has the advantage of being declarative, and the disadvantage of requiring some additional software.
  - Shell scripts that directly invoke the experimental code using the appropriate parameters. This is a more procedural approach, which however has the advantage of requiring very little setup.
- All the aforementioned experimental files should be tracked with version control along with the code. Before running the experiments, any pending changes should be committed.
- There should be a mechanism allowing to skip already-run configurations. This allows both to save time (C2) without having to continuously edit the configuration files to remove the configurations that do not need to be run.

We provide the example files that we used in Appendix A. While a direct experimental file written in Bash is straight-forward, the YAML structure gives a much more structured overview. The YAML file is run through an additional Python script that invokes the main implementation with the correct parameters. Using versioning and the result database (Subsection 5.5) the code can decide whether an algorithm has to be rerun.

5.3 Infrastructure management

Any implementation will likely depend on many different environmental settings, such as the correct versions of libraries/compiler/OS. To allow a machine independent workflow, we suggest to:

- Provide a containerized development environment[4].
- Consider different container formats for running experiments [1].
- Use continuous integration to test all parts of the workflow.

Our code provides a `Dockerfile` that installs a well-defined Linux environment and sets up the correct compilers and libraries. Each component is run from the local system via a `dockerrun` script that will run the intended process within the container.

5.4 Version everything

To address challenges (C2) and (C3), version control systems might not be sufficient, since source code revisions lack both a semantic meaning and a total order. Furthermore, different components of a project might evolve independently, thus needing independent versioning to address challenge (C2). Therefore, we suggest to keep track of the versions of individual components of the project, including datasets, algorithms, and database schemas, alongside the versioning provided by the version control system.

In our code, each dataset and database schema provides their own version number. Additionally, components of the implementation such as input representation type or SIMD definitions are versioned. This allows us to map each parameter set for the linear scan to a unique identifier. As an example, during continuous integration we found a bug that only affected the AVX2 inner product computation with floating point numbers. An update of the version number of this part of the code led to a re-run of all parameter configurations that used that particular combination. Each measurement obtained is versioned with its *git identifier*, the algorithm version in question, and the dataset version.

5.5 Manage the experimental results thoughtfully

As for the management of experimental results, structured text file formats like CSV address challenge (C4), but are expensive to parse and require to be fully loaded in main memory prior to the analysis, even when only a subset is needed. Moreover, it is hard to evolve the structure of these files together with the project.

- Use a database to store the results: it presents data conveniently indexed and removes the need for expensive parsing.
- Using schema migrations the database can evolve along with the rest of the project (C3), as demonstrated in our case study's code.
- For simple projects, an embeddable database like SQLite addresses challenge (C4): the results are stored in a single file which can be easily moved between machines, and many languages used for the analysis (like Python and R) provide facilities to access it, as shown in our code. For larger projects, where experimental code is executed on different machines, a database with a client-server architecture (such as PostgreSQL) might be more suitable.

[4] Recently, such environments are included in programming IDE such as `https://code.visualstudio.com/docs/remote/containers`

- The experimental code can query the database to detect whether an experiment has already been run with the current version (C2).
- Track the *provenance* [6] of each result, by storing alongside the parameters also the configuration file (and its version) that generated the result (C5).
- By means of database views we can enforce that the analysis code has access only to the most recent results related to each algorithm/dataset (C3): our code demonstrates how to embed a query in the database so to present only the results related to the most recent version of algorithms and datasets.

References

1. Arango, C., Dernat, R., Sanabria, J.: Performance evaluation of container-based virtualization for high performance computing environments. arXiv:1709.10140 (2017)
2. Aumüller, M., Christiani, T., Pagh, R., Vesterli, M.: PUFFINN: parameterless and universally fast finding of nearest neighbors. In: ESA 2019
3. Bartz-Beielstein, T., Chiarandini, M., Paquete, L., Preuss, M. (eds.): Experimental Methods for the Analysis of Optimization Algorithms. Springer (2010)
4. Bartz-Beielstein, T., Preuss, M.: Experimental analysis of optimization algorithms: Tuning and beyond. In: Theory and Principled Methods for the Design of Metaheuristics. Springer (2014)
5. Boettiger, C.: An introduction to docker for reproducible research. Operating Systems Review **49**(1) (2015)
6. Buneman, P., Khanna, S., Wang-Chiew, T.: Why and where: A characterization of data provenance. In: ICDT (2001)
7. Callahan, S.P., Freire, J., Santos, E., Scheidegger, C.E., Silva, C.T., Vo, H.T.: Vistrails: visualization meets data management. In: SIGMOD 2006
8. Charikar, M.: Similarity estimation techniques from rounding algorithms. In: STOC 2002
9. Collberg, C.S., Proebsting, T.A.: Repeatability in computer systems research. Commun. ACM (2016)
10. Davison, A.P., Mattioni, M., Samarkanov, D., Telenczuk, B.: Sumatra: a toolkit for reproducible research. In: Implementing reproducible research. CRC Press (2014)
11. Fleischer, R., Moret, B.M.E., Schmidt, E.M. (eds.): Experimental Algorithmics, From Algorithm Design to Robust and Efficient Software, LNCS, vol. 2547. Springer
12. Guo, P.J.: CDE: A tool for creating portable experimental software packages. Comput. Sci. Eng. **14**(4) (2012)
13. Hutson, M.: Artificial intelligence faces reproducibility crisis **359**(6377) (2018)
14. Kitzes, J., Turek, D., Deniz, F.: The practice of reproducible research: case studies and lessons from the data-intensive sciences. Univ of California Press (2017)
15. Kurtzer, G.M., Sochat, V., Bauer, M.W.: Singularity: Scientific containers for mobility of compute. PloS one **12**(5) (2017)
16. McGeoch, C.C.: Experimental algorithmics. Commun. ACM **50**(11), 27–31 (2007)
17. McGeoch, C.C.: Experimental methods for algorithm analysis. In: Encyclopedia of Algorithms. Springer (2008)
18. Moret, B.M.E., Shapiro, H.D.: Algorithms and experiments: The new (and old) methodology. J. UCS **7**(5) (2001)
19. Rampin, R., Chirigati, F., Shasha, D.E., Freire, J., Steeves, V.: Reprozip: The reproducibility packer. J. Open Source Softw. **1**(8) (2016)
20. Stodden, V., Leisch, F., Peng, R.D.: Implementing reproducible research. CRC Press (2014)

```yaml
environment:
    dataset: ['glove-100-angular']
    seed: 4132
    force: False
experiments:
    - name: baseline
      storage: ['float_aligned']
      method: ['simple', 'avx2', 'avx512']
    - name: i16_alignment
      storage: ['i16_aligned']
      method: ['simple', 'avx2', 'avx512']
    - name: sketches
      storage: ['float_aligned', 'i16_aligned']
      method: ['simple', 'avx2', 'avx512']
      sketches: True
      recall: [0.001, 0.1, 0.2, 0.5, 0.7, 0.9, 0.99]
```

Fig. 3. An example YAML file for running all experiments.

A Experimental Example: Bash vs. YAML

Figure 3 and Figure 4 present the same experimental setup file to produce the results we report on in Appendix C. We find the YAML configuration much more readable and easier to edit. However, it requires an additional program `run_yaml.py` that translates the YAML file into the correct system calls to the executable. Those calls are directly present in the bash file.

B Review: Linear scan using 1-bit sketches via SimHash

Charikar described in [8] the well-known *SimHash* scheme that maps a unit vector $x \in \mathbb{R}^d$ to $\{0,1\}$. It works by choosing a random d-dimensional normal vector $a \sim \mathcal{N}(0,1)^d$ and mapping x to the indicator variable $[ax \geq 0]$. For two unit vectors x, y, the probability of mapping to the same bit is $1 - \arccos(xy)/\pi$.

Assume we choose 64 independent SimHash functions to produce a 64-bit sketch. The number of differences between two vectors x and y is distributed as $\mathrm{Bin}(64, \arccos(xy)/\pi)$ and it is easy to derive thresholds τ for a failure probability $\delta > 0$ such that with probability at least $1 - \delta$, the number of differences between x and y is at most τ. (In the simplest case, we can just use standard Chernoff bounds that say that one such choice for τ is $64\arccos(xy)/\pi + \sqrt{192 \ln(1/\delta) \arccos(xy)/\pi}$.)

To find a nearest neighbor of a given point q with probability at least $1 - \delta$, we carry out a linear scan by first checking the 64-bit sketch of the current data point p and the query point, and only if we cannot rule out that p could be the nearest neighbor of q, we carry out the actual distance computation. The threshold τ is set based on the inner product of the current closest nearest point

```
FN=$(basename $0)
SEED=4132
DATASET=glove-100-angular

 ./demo --dataset $DATASET --storage float_aligned --
     experiment-file $FN --seed $SEED # produce baseline
 ./demo --dataset $DATASET --storage i16_aligned --method
     simple --experiment-file $FN --seed $SEED
 ./demo --dataset $DATASET --storage i16_aligned --method
     avx2 --experiment-file $FN --seed $SEED
for recall in 0.001 0.1 0.2 0.5 0.7 0.9 0.99; do
    for alignment in i16_aligned float_aligned; do
        for method in simple avx2; do
             ./demo --dataset $DATASET --filter --storage
                 i16_aligned --recall $recall --method
                 $method --experiment-file $FN --seed $SEED
        done
    done
done
```

Fig. 4. An example bash file for running all experiments.

and the query point as above. Initially, we set $\tau = \infty$. We remark that this allows us to control the expected recall, since a failure probability of δ translates to an expected recall of $1 - \delta$.

C Evaluation of our Engineered Linear Scan

Experimental setup. All experiments are carried out on a single core of a compute node equipped with 2 Intel Xeon Gold 6136 CPU @ 3.00GHz with 24 cores each and 192 GiB of RAM running in a shared HPC cluster. We report on results for the *Glove.27B.twitter* dataset (`https://nlp.stanford.edu/projects/glove/`) containing around 1.2 million 100-dimensional data points. We chose 100 data points at random as query points and removed them from the dataset.

Of course, evaluating a linear scan does not depend on many characteristics of the dataset except the number of vectors and the number of dimensions of each vector. However, working with low precision floats and using the sketches from above depend very much on the distance distribution of the query to the points in the dataset.

Parameters. We consider both 16-bit and 32-bit input representations, and inner product computation with a naive loop, an avx2 optimized loop, and an avx512 optimized loop. We use the SimHash 64-bit sketch described in Appendix B with recall guarantees $r \in \{0.001, 0.1, 0.2, 0.5, 0.7, 0.9, 0.99\}$.

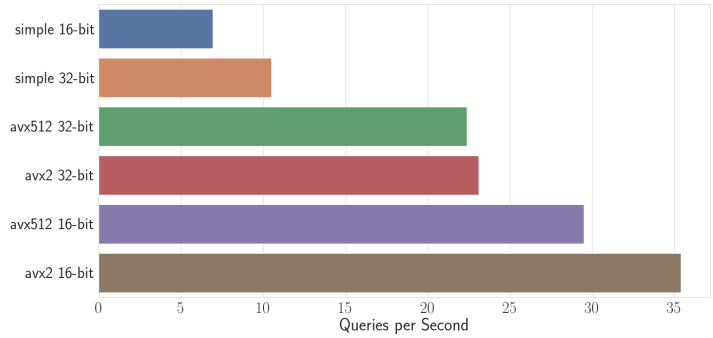

Fig. 5. Comparison of different linear scan approaches.

Evaluation. The plot in Figure 5 relates different input representations and inner product computations for a linear scan without sketches to each other. A standard linear scan on the dataset results in a throughput of around 10 queries per second. It can be sped up to 35 queries per second using the 16-bit representation and avx2 instructions. In general, a naïve scan is slower in the 16-bit representation because there are internal conversions needed to carry out the multiplication. If we use vectorized instructions, there is a huge gain from using the 16-bit instead of the 32-bit representation. Somehow interestingly, using 256 bits registers is *faster* than using the newer AVX 512 registers. This is true for both input representations, but with different penalties between the two.

In Figure 6(top), we fix a 16-bit representation and avx2-optimized inner product computation. We relate different recall guarantees to the performance and neglect the actual quality of the resulting answers. As we can see, there is big increase in throughput possible by the use of these sketches. We achieve around 50 queries per second for recall guarantee of 0.99, 100 queries per second for 0.7, and close to 350 queries per second for a recall quality of 0.001 (i.e., no guarantee).

Finally, Figure 6(bottom) relates throughput to the actual achieved recall for both input representation and simple inner product and avx2-optimized inner product. First, we note that the 16-bit representation indeed shows a small quality loss: The average recall is only 0.98, which in our setting means that two true nearest neighbors were lost due to precision losses. In general, updating the sketch threshold (see Appendix B) via a Chernoff bound is far too pessimistic: Even a guarantee of 0.001 results in actual quality guarantees of 0.66 to 0.83.[5] This suggests an even larger speed-up being possible when evaluating the cdf of the binomial distribution. With regard to throughput, we can serve up to 300 queries per second while still maintaining a quality above 0.8. The 16-bit representation is a bit faster than its 32-bit counterpart in the high recall range,

[5] We note that the τ threshold from Appendix B actually guarantees a recall of at least 50% because it is never smaller than the expectation.

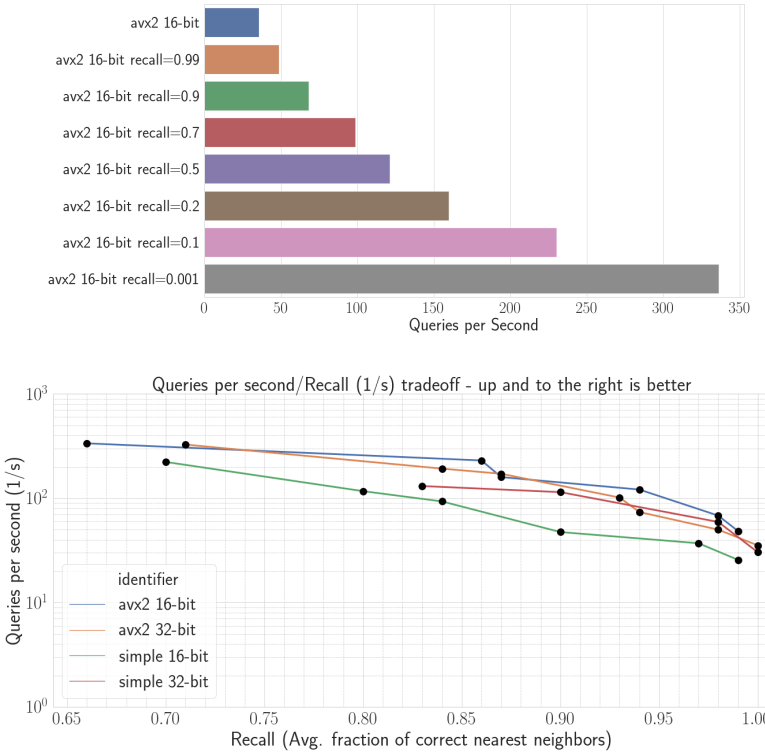

Fig. 6. top: Throughput impact of sketches; bottom: Queries-per-Second/Recall trade-off.

but these differences shrink with lower recall guarantees (probably due to the fact that many points are filtered out by the sketches). Comparing our results to standard benchmarks such as `ann-benchmarks`, obtaining 300 queries per second for recall of 0.8 is as fast as other well-engineered index approach such as annoy, see `http://ann-benchmarks.com/glove-100-angular_10_angular.html`.

Workflow discussion. The run of experiments and the analysis of results is so general that we run the whole analysis on a tiny random dataset each time code is committed to the main branch, see for example the artifacts at `https://github.com/Cecca/running-experiments/actions/runs/140873083` with its configuration at `https://bit.ly/2ChUYvy`.