# The Fault in Our Stars

## Designing Reproducible Large-scale Code Analysis Experiments

**Petr Maj**
Czech Technical University, Prague, Czech Republic

**Stefanie Muroya**
Institute of Science and Technology Austria (ISTA), Klosterneuburg, Austria

**Konrad Siek**
Czech Technical University, Prague, Czech Republic

**Luca Di Grazia**
Università della Svizzera italiana (USI), Lugano, Switzerland

**Jan Vitek**
Charles University, Prague, Czech Republic
Northeastern University, Boston, MA, USA

**Abstract**

Large-scale software repositories are a source of insights for software engineering. They offer an unmatched window into the software development process at scale. Their sheer number and size holds the promise of broadly applicable results. At the same time, that very size presents practical challenges for scaling tools and algorithms to millions of projects. A reasonable approach is to limit studies to representative samples of the population of interest. Broadly applicable conclusions can then be obtained by generalizing to the entire population. The contribution of this paper is a standardized experimental design methodology for choosing the inputs of studies working with large-scale repositories. We advocate for a methodology that clearly lays out what the population of interest is, how to sample it, and that fosters reproducibility. Along the way, we discourage researchers from using extrinsic attributes of projects such as stars, that measure some unclear notion of popularity.

**2012 ACM Subject Classification** Software and its engineering

**Keywords and phrases** software, mining code repositories, source code analysis

**Digital Object Identifier** 10.4230/LIPIcs.ECOOP.2024.27

**Funding** This work was supported by the Czech Ministry of Education, Youth and Sports under program ERC-CZ, grant agreement LL2325, BigCode (reg. no. CZ.02.1.01/0.0/0.0/15_003/0000421). NSF grants CCF-1910850, CNS-1925644, and CCF-2139612, as well as the GACR EXPRO grant 23-07580X.

**Acknowledgements** We would like to thank Digital Ocean for their involuntary contribution of computational resources during the early data gathering phase of our research. We acknoweldge the reviewers of ICSE'22, and thank the reviewers of ECOOP'23 for their encouragments and for sticking around until 2024.

## 1 Introduction

And so it begins...

> *count the number of stars associated with each repository. The number of stars relate to how many people are interested in that project. Thus, we assume that stars indicate the popularity of a project. We select the top 50 projects in each language*

Sentences like these appear in the methodology sections of software engineering papers, with sometimes, little more in terms of experimental design. This paper aims to convince readers that using extrinsic features of projects, such as popularity, may limit applicability of results of the studies relying on them. Instead one should select projects based on their intrinsic features and spell out expectations as well as threats to validity.

Empirical software engineering studies are experiments performed on a corpus of software to validate some hypotheses. For instance, one could take projects written in various languages and attempt to show that some language feature has an impact on the quality of the code written using it. The value of large-scale corpus study often does not lie in what we learn about the projects that were analyzed, but rather in what these can teach us about the larger population. There is limited value in, say, finding out that a new Java language feature is beneficial in a handful cases if we cannot generalize that result to a broader portion of the Java ecosystem.

Yet, many papers in the field do not articulate how broadly applicable their results are expected to be. Even the simple question of how the projects that were analyzed were selected is not clear. While large-scale code repositories, such as GitHub, are a boon to the software engineering community, their sheer size requires care. We argue that better experimental design will strengthen research done in the field.

Consider Table 1 which has a meta-study of three years of the *Mining Software Repositories* conference. Forty-one papers relied on subsets of GitHub. Out of those, five papers lacked sufficient information about their dataset to determine how they selected their inputs, twenty-one used GitHub stars to obtain a subset of projects, ten used simple combinations of attribute thresholds and only five relied on random sampling over the entire population.

**Table 1** Experimental design in MSR 2019, 2020 and 2021.

| papers | class | description | # of projects |
|---:|---|---|---|
| 5 | Unknown | Unknown or proprietary | 1–35K |
| 21 | Stars | Filter projects using stars | 5–2M |
| 10 | Other | Other filter for projects | 7–290K |
| 5 | Random | Filter and sample randomly | 6–51K |

What is the right choice here? None of the papers analyzed the entire ecosystem as that would mean tens to hundreds of millions of projects. The question thus becomes how to sample the population of projects hosted on GitHub. In this paper, we criticize the use of GitHub stars as they are appealing and popular, yet also dangerous. But, really, our point generalizes to any extrinsic attributes of a project. So, again, what is the right choice? The answer is, of course, that it depends. The rest of this paper attempts to shed some light on how to make a reasoned choice of inputs.

First, let us return to stars and ponder why they play such an important role in our experimental methodology? We believe expectations and pragmatics are the explanation. Community standards are largely set by the papers we publish. The literature codifies expectations for authors of the next batch of papers. These expectations slowly evolve in response to reviewer attitudes. So, we use stars because our peers do. And just as importantly for the pragmatic reason that GitHub does not provide an index of projects, nor does it allow to query over intrinsic attributes of code. Finding inputs is thus hobbled by limitations of our tools. Stars play a double role. They are a queryable index of projects as GitHub does provide an interface to obtain them. They also come with an expectation that starred projects enjoy some notion of quality [8]. This paper will show that stars do not necessarily correlate with quality and that they introduce reproduction barriers.

We propose a methodology for designing reproducible software engineering experiments over large-scale repositories with the explicit goal of improving the generalizability of our results. The methodology is in line with evolving community standards [23] but specific to large-scale code analysis. We propose to follow the following protocol when designing a new experiment:

1. **Population Hypothesis:** Give a brief description of the population of interest the research should generalize to; it may be narrow such as "programs written by students learning JavaScript" or as broad as "commercial code".
2. **Frame Oracle:** Give a procedure for deciding if a project belongs to the population of interest. The procedure should be efficiently computable over intrinsic attributes of a project. An oracle could, say, return projects with a single JavaScript file created by a user with no previous commits.
3. **Sampling Strategy:** Describe a strategy for selecting a subset of the entire population. Ideally, specified algorithmically. An example is random sampling without replacement from a known seed.
4. **Validity:** Give an argument as to how the oracle and the sampling strategy are valid means to obtain representative samples. This can be a discussion of how to check result quality, such as manual inspection of samples written by beginners, and threats to validity.
5. **Reproduction Artifacts:** Publish an artifact that reproduces exactly the reported results, and supports changes to either the input or the details of the experiment.

Reproducibility has nuances. Our emphasis is on providing support for the following three use cases: *Repetitions* which run the reproduction artifact to obtain bit-for-bit equal results. This is the most stringent use case and often requires a reproduction artifact that bundles code and inputs. *Reanalysis* alters either the method or its input, it requires an executable artifact and a method for acquiring new inputs. Finally, *reproductions* are independent implementations that require the paper to have an unambiguous description of all experimental details.[1] Supporting reproducibility can be greatly simplified with appropriate tooling. Our work builds on the CodeDJ infrastructure (`codedj-prg.github.io`). Our contributions are:

1. **A dataset** of 2Mio+ projects with intrinsic attributes precomputed.
2. **A characterization of stars** as a means to select inputs for code analysis experiments.
3. **A methodology** that can be readily adopted to improve reproducibility.
4. **A reproduction** that highlights challenges to generalization due to project selection.

Our community has been moving towards broader adoption of the practice of artifact evaluation [11]. While artifacts are clearly helpful as they make papers providing them easier to reproduce, the selection of inputs is often hardwired and not considered part of the reproduction. The impact of our proposal, if adopted, would be to encourage authors of large-scale code studies to consider the collection of the inputs to their work to be part of the experiment and thus make it easy to change the way inputs are selected.

**Road map.** The structure of the paper is as follows.

- Section 2 begins with a short overview of the state with respect to methodologies for project selection and tooling to support it.
- Section 3 takes four practical examples, papers published at the Mining Software Repositories conference, and attempts to couch their experimental design in the terms introduced above. These example suggest that authors are not always clear about their intent and strategy. While looking at these papers we found a number of practical impediments to reproducibility.

[1] The terminology comes from [24] and was used by SIGPLAN artifact evaluation committees.

- Section 4 describes characteristics of the projects hosted on Github and argues that stars cannot yield a representative sample of developed projects.
- Section 5 outlines our proposal for how to design large-scale program analysis experiments.
- Section 6 follows our guidelines and attempts to repeat the studies of Section 3 while perturbing the experimental inputs.
- Section 7 is responsive to reviewers of this paper and their request to reproduce an experiment from a paper with a verified artifact.
- Section 8 concludes and gives some parting thoughts. This paper improves on the state of the art in that it argues for a structured experimental design that relies on tooling for input selection.

## 2 Related work

We review relevant advice, warnings and the state of tooling.

### 2.1 Community standards

A push towards reproducibility is underway. The standards framework of Ralph et al. [23] includes a section on experimental design and specifically on sampling. This is further explored by Baltes and Ralph [1]. They argue that software engineering faces a generalizability crisis. In their meta-analysis of 120 papers, they find that convenience sampling[2] is widely used to select projects to analyze from a large population. Convenience sampling rarely leads to representative samples, and – without a careful study of potential sources of bias – can lead to conclusions that do not generalize. They explain this state of affairs by a fundamental challenge: the lack of appropriate sampling frames to access elements of the population of interest. Earlier work by Nagappan et al. [19] already attempted to address this problem by defining the notion of sample coverage as a way to assess the quality of the data used as input to an experiment. Even closer to our paper is the study by Cosentino et al. [4] which reported that out of 93 large corpus papers, 63 papers failed to provide replication datasets. Most papers did not use random samples and omitted mentions of any limitations.

### 2.2 Mining repositories

GitHub is a popular data source. Warnings about its perils go back to the work of Kalliamvakou et al. [10] which highlighted "noise" among hosted projects. In particular, they point out, tiny and inactive projects dominate the platform. Lopes et al. [13] poured oil on that fire, showing that up to 95% of the files containing code in some language ecosystems were copies of one another and filtering by stars reduces the proportion of duplicates without eliminating them. One way researchers have strived to find signal in GitHub's sea of noise is to use stars. But what do stars mean? We would like them to be correlated with quality code, code worth analyzing. Borges and Valente [2] conducted a user survey that found the most common reasons for starring a project was to show appreciation (e.g. *starred this repository because it looks nice*) and bookmark it (e.g. *starred it because I wanted to try it later*). They also warn against promotional campaigns to drive up ratings. Popularity of projects was studied by Han et al. [8], they found that while users believe stars are a measure of a project's popularity, intrinsic attributes such as branches, open issues and

[2] The Wikipedia definition of convenience sampling is a type of non-probability sampling that involves the sample being drawn from that part of the population that is close to hand.

contributors are better predictors. Expending on that result, Munaiah et al. [18] propose classifier for *engineered projects*, which they define as projects that leverage sound software engineering principles. They show that the classifier outperforms stars. Pickerill et al. [22] further improved classification with an approach based on time-series clustering.

### 2.3 Tools for miners

A number of infrastructures have been developed to assist researchers in the field. The most ubiquitous was, the now defunct, GHTorrent [6]. The project offered a continuously updated database of metadata about public projects that was a valuable building block for other tools. Boa is complementary as it lets users write sophisticated queries over source code [5]. CodeDJ is a newer infrastructure that supports queries over both meta-data and file contents and is language agnostic [15]. Recent works address performance issues of querying at scale [14, 17]. Of these, only CodeDJ ensures reproducible queries.

## 3 State of practice

How do people design experiments for large-scale code studies? This section gives some examples that we believe to be representative which we will revisit later when we attempt to reproduce the results with different inputs. For each paper, we provide a brief summary of the scientific claims made by the authors. Then, we attempt, with our best understanding of the work, to reverse engineer a version of the protocol laid out in the introduction. We, thus, give an account of each paper's population hypothesis, a description of the frame oracle, sampling strategy, validation and reproduction artifacts. We conclude the section with some observations general reproduction issues that show up in these papers.

### 3.1 MSR 2020: What is software

"Software" has an intuitive definition, namely code, but there is more. The paper by Pfeiffer [21] classifies the content of repositories in categories such as code, data and documentation. They, then, observe that software is more than just code. Documentation is an integral constituent of software, and software without data is often correlated with libraries, and finally that software without code is rare, but exists. The paper answers the question *"what are the constituents of software and how are they distributed?"* The paper argues that existing definitions of the term are non-descriptive, inconclusive and even contradictory.

**Population Hypothesis:** Implicitly, the population is all inclusive.

**Frame Oracle:** Given the lack of details, we assume all projects on GitHub belong.

**Sampling Strategy:** the authors carry out convenience sampling by choosing popular repositories. Stating *"by popularity we mean the starred criteria with which GitHub users express liking similar to likes in social networks."* Most-starred projects in 25 languages were acquired by executing one query by language, saying that *"without language qualifier, the API returns only 1,020 repositories in total, which we decided is not enough for our study."*

**Validity:** No discussion of relevant issues or threats.

**Reproducibility Artifacts:** A listing of files and repositories is provided along with the code of the classifier and a notebook. Repository contents were not preserved.

## 3.2 MSR 2020: Method chaining

In an object-oriented language, a *method chain* occurs when the result of a method invocation is the receiver of a subsequent invocation. In Java, chains manifest as sequences of calls connected by dots. Nakamura et al. [20] analyze trends in usage of method chains and conclude that they increase over a period of eight years.

**Population Hypothesis:** Java projects developed *"by real-world programmers."* The authors state that they *"did not apply any filter to the collected repositories. This supports the generalizability of our results."* The authors also consider generalization beyond Java, saying *"our results are more likely to be applied to a language that does not provide such a construct (e.g. PHP and JavaScript). The empirical study of this hypothesis is future work."*

**Frame Oracle:** Implicitly, all Java projects hosted on GitHub.

**Sampling Strategy:** The authors use convenience sampling, taking 2,814 projects that appeared at least once in the list of the 1K most-starred projects in November 2019. Projects were deduplicated and filtered for syntactically invalid files.

**Validity:** No discussion of relevant issues.

**Reproducibility Artifacts:** Project metadata and computed chain lengths are available. Communication with the authors reveals that their reproduction package is not available.

## 3.3 MSR 2019: Style analyzer

Each software project seems to develop its own formatting conventions. Markotsev et al. [16] demonstrate that an unsupervised learning algorithm can automate project-specific code formatting. They reproduce styles with a high degree of precision for a set of repositories.

**Population Hypothesis:** The authors speak of *"real projects"* and their artifact support JavaScript, so we assume an expectation that the projects "developed" in a sense similar to [18].

**Frame Oracle:** All developed JavaScript projects hosted on GitHub.

**Sampling Strategy:** Convenience sampling yielded 19 JavaScript projects with high star counts.

**Validity:** Authors manually inspected projects in the selection.

**Reproducibility Artifacts:** A GitHub repository containing the tool and a file with project URLs along with their head and base commits is provided. Contents of repositories are not included. Run scripts did not run out of the box.

## 3.4 MSR 2020: Code smells

Code smells are programming idioms correlated with correctness or maintenance issues. Jebnoun et al. [9] contrast code smells in projects related to deep learning and general purpose software. Their claim is that for large and small projects there is a statistical difference in the occurrence of code smells, whereas medium sized projects are indistinguishable.

**Population Hypothesis:** The paper focuses on two populations: projects related to deep learning, and general purpose software. For pragmatic reasons, they focus on Python as it is popular for machine learning.

**Frame Oracle:** Python projects with keywords indicating machine learning, discarding tutorials. Furthermore, the authors *"also carefully select popular and mature DL projects from them by employing maturity and popularity metrics (e.g., issue count, commit count, contributor count, fork count, stars)."*

**Sampling Strategy:** A staged strategy was employed. The authors relied on judgment sampling to manually select 59 deep learning projects. For general purpose projects, they used a top-starred list of 106 Python projects from [3] and randomly sampled 59 projects. Projects were further clustered into small ($\leq 4,000$), medium, and large ($\geq 15,000$) based on size.

**Validity:** No issues were discussed.

**Reproducibility Artifacts:** A listing of the 59 deep learning projects is provided.

## 3.5 Summary and discussion

The papers we have reviewed do not explicitly talk about any of the four points in our protocol, in all cases we had to reverse engineer (or guess) some of them. This suggests that our proposal would improve the generalizability of the research.

While the mentioned research projects were done with care, there were challenges reproducing them out of the box. Common sources of reproducibility failures that occur in the papers we have reviewed are:

- *Missing descriptions:* Failure to specify either one of: population hypothesis, frame oracle or sampling strategy. Reproduction is fraught with perils and an apple-to-apple comparison between papers is difficult. This affects [21, 20, 16, 9] as their descriptions are open to interpretation.
- *Missing projects:* Even with a list of URLs, the corresponding projects may vanish at any time (e.g., deleted or made private). Reproductions are partial at best, we have seen a project disappear while being downloaded. This affects [21, 20, 16, 9].
- *Fading stars:* Stars are volatile. [20] observed close to 3,000 projects in the top 1K during a period of two months. Without a history of star attribution and a timestamp, reconstructing the star listings is not possible. Stars volatility also caused problems for [9].
- *Shifting contents:* The contents of a project change with new commits. To reconstruct the data, ids of the last observed commit must be specified. Even that is not foolproof as Git histories can be updated destructively. This affects [21, 20, 9].
- *Language attribution:* Projects contain code in many languages. For reproduction attribution must be specified. While delegating to, e.g. GitHub, is reasonable, one should be aware that GitHub has changed their attribution algorithm several times. Double counting a project is sometimes valid. This affects [21, 20, 16].
- *Deterministic replay:* Non-determinism must be limited. Random sampling seeds should be specified. This affects [16].

## 4 Mapping the GitHub landscape

The meta-study of Table 1 highlights the dominant position of GitHub as a data source in large-scale code analysis studies. The size of GitHub is such that it is necessary to resort to sampling to yield manageable datasets. As shown in the previous section, authors often look for some notion of "developed" projects, that is, they want projects that contains code of some quality.

We claimed that convenience sampling using stars as a proxy for various other characteristics of "real-world" software is flawed. While this may sound plausible to some readers, it should be backed up with data. Given the size of GitHub, this section uses sampling to answer the following questions: *Are starred projects a representative sample of all projects?*

and *Are starred projects a representative sample of developed projects?* where what it means for a project to be developed is purposefully left open as there is no agreement on a precise definition of the term.

Since the later parts of this paper require Java, Python and JavaScript, we acquire samples of these three ecosystems. We use CodeDJ to do this. It is an open source project that allows users to create a dedicated input project database and ensure reproducibility of queries.

We used random sampling over the entire GHTorrent dataset to select which projects to acquire in each of the languages of interest. The number of downloaded projects is somewhat arbitrary as it is based on available hardware during the acquisition phase. The datastore has 1,111,950 Java projects, 216,602 Python projects and 1,259,856 JavaScript projects. To give an idea of the scale, our Java dataset accounts for 20% of all non-forked GitHub Java projects. To get a manageable size, we down-sample further, randomly selecting 1Mio Java and JavaScript projects, and 200K Python projects.

## 4.1 Attributes

With CodeDJ, it is easy to write queries that compute project attributes. For this paper, we calculate five attributes that highlight the differences between projects:

- **C-index:** A developer handle has a C-index of $n$ if that developer was party to at least $n$ commits to $n$ projects (i.e. $n^2$ commits). The C-index of a project is the highest such number across developers. This measures developer expertise.
- **Age:** The age of a project is the number of days separating the first commit and the most recent commit. This correlates with the maturity of a project.
- **Devs:** The count of unique developer handles in the git logs; includes both the author of a code change and the committer of that change. Devs approximates the size of a team, as some individuals may have more than one handle this is an upper bound.
- **Locs:** The total number of lines in files that are recognized as code, in any language, and appear in the head of the default branch.
- **Versions:** A version is implicitly created for each commit touching a file, be that for creation, deletion or update. This counts versions in the entire project's history including branches. Versions measure the activity in a project.

While we make no claims that these attributes suffice to fully describe a software project, we have found them to be an effective summary in many interesting dimensions.

## 4.2 Stars v. All

What do these attributes tell us about the overall population and about starred projects? If starred projects were representative of the entire population, they would share the same statistical distribution.

Fig. 1 is a histogram of each attribute; the x-axis is log scaled values in the unit of each attribute, the y-axis is the proportion of projects normalized for the maximum height. Grey denotes the whole population, red and blue denote the 1K most starred projects written in Java and Python respectively. The black, red and blue bars denote the median of their respective populations.

Consider the grey bars for the whole population, when comparing Java and Python, we see the same general shapes. The C-index is low, with a median of 2 for Java and Python. This means that half of the projects hosted on GitHub, have developers who have made at most two commits. The median age of Java projects is 7 days, while Python projects trend

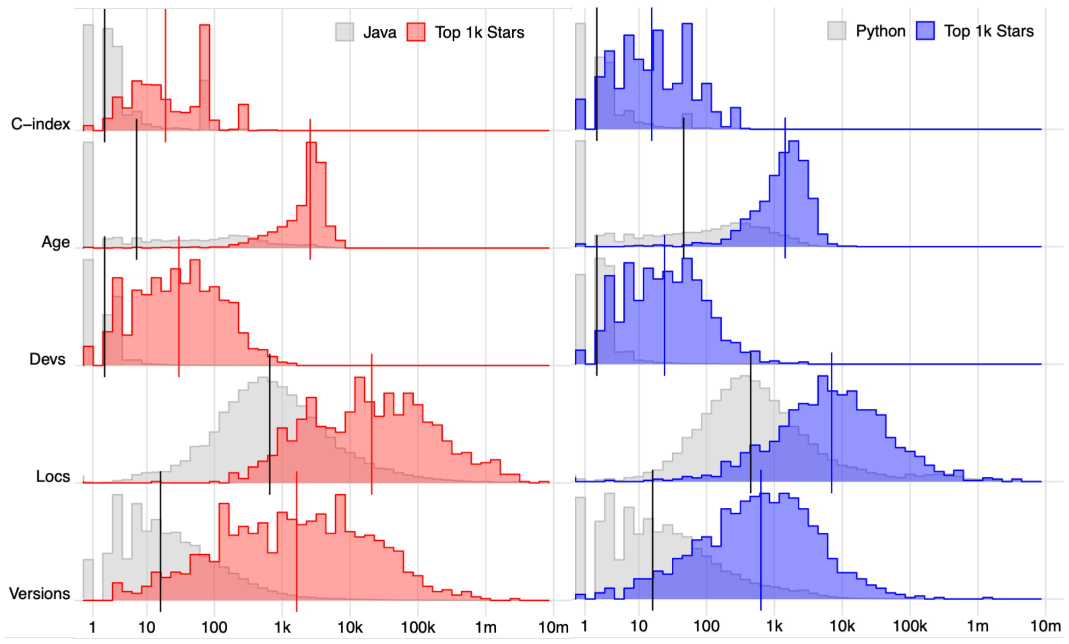

**Figure 1** Comparing datasets.

slightly older, 46 days. The median number of developers for both languages is 2. As for median lines of code, Java project are slightly larger than Python, 655 compared to 448. The median number of commits (versions) is 16 for both languages. Overall, this confirms that most projects are small, short-lived and created by relative newcomers.

The top 1K starred projects have a very different make up. Visually it is clear from the fact that every distribution is shifted right. Starred projects are larger, older, with more experienced developers. While there are slight differences between languages, the overall picture is consistent.

Consider for instance, the C-index and age attributes. While many starred projects are team efforts, a significant number of projects have few contributors. Their C-index is high, with median of 19 for Java and 15.5 for Python, suggesting that experienced developers tend to contribute to popular repositories. The median age projects is more surprising with 2,581 and 1,440 days. Manual inspections suggest that many starred projects are indeed long lived but also have been inactive for years. Projects rarely "loose" stars, so if a project gets to the top there is a chance it will stay there long past its useful lifetime.

The answer to our first question is clearly negative. Starred projects are not representative of the overall population. This is not necessarily a bad thing, as folklore suggests that most of GitHub is uninteresting. Perhaps it is the case that starred projects are more "interesting."

## 4.3 Stars v. Developed

Researchers often look for *engineered* [18, 22] or *developed* projects – informally, projects created with some care – alas there is little agreement on a precise definition.

Slightly easier, perhaps, is to settle on what we do not want, the projects that are uninteresting, one that are clearly of little value for any reasonable research question. Moreover, one could hope that the complement of uninteresting projects are the projects researchers look for. Let us define a project as *uninteresting* if it has less than 100 lines of code, fewer than 7 days old, and fewer than 10 commits. When this definition is used to filter projects, this rather low bar manages to eliminate 71% of Java and 55% of Python projects. For the purpose of this discussion we term the remaining projects are developed or interesting. It would be nifty if stars were a proxy for filtering out uninteresting projects.

Fig. 2 shows the distribution of the whole population in Grey (in the same way as in Fig. 1), the interesting projects in Black and the top 1K starred projects in Red. Clearly, the shape of the Black and Red distributions do not align suggesting that stars do not represent interesting projects.

Manual inspection of the starred project highlights their main issue – stars are extrinsic properties without a direct connection to any attributes of a project. Unlike our computed attributes, stars grow monotonically. Their meaning is unclear as users award them for various reasons including humor and shock value. Some projects earned stars because of a joke not fit for this audience (e.g. `github.com/dickrnn/dickrnn.github.io`), or have dared users to star junk (`github.com/gaopu/java`). Stars do not measure quality or usefulness of repositories.

To further illustrate the limitation of stars as a filter, we take, for each attribute, the 20 projects with the lowest score for that attribute. Table 2 shows a manual classification of these projects. None of these projects is useful: externals lack histories, widgets are small and biased by their application domain, babies are too small to yield much insights, and the remaining ones only have code snippets.

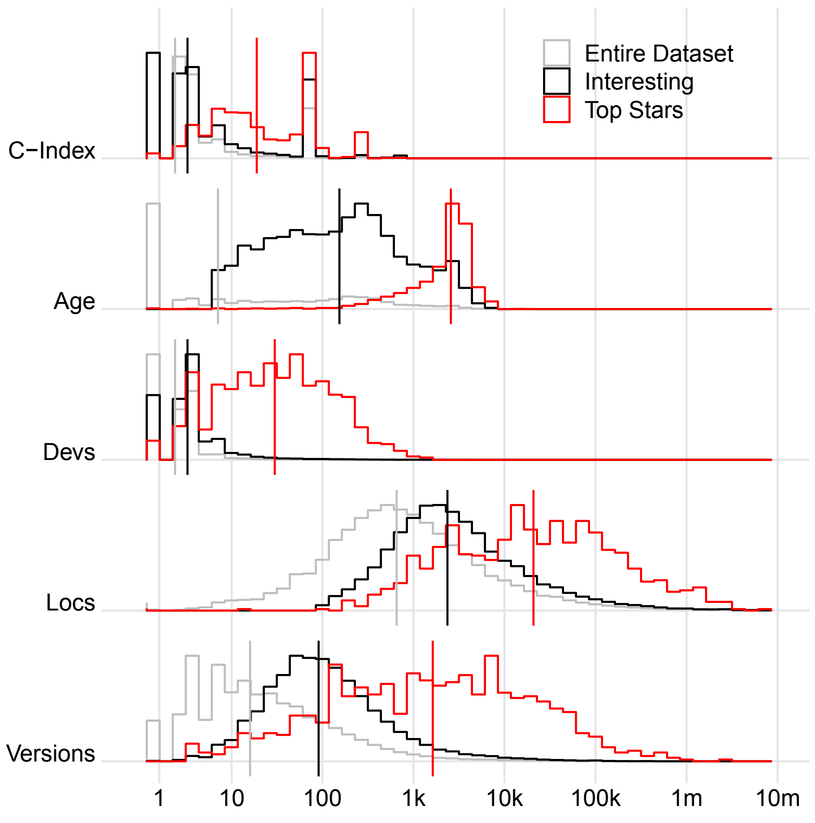

**Figure 2** Comparing developed and starred projects.

**Table 2** Categorizing 200 starred projects.

| Category | Java | Python | Description |
|---:|---:|---:|---|
| **Externals** | 9% | 5% | Infrequent synchronization with another repository. |
| **Widgets** | 43% | 0% | Tiny projects with little activity, popular UI widgets or plugins. |
| **Docs** | 4% | 15% | Interview questions, course materials, games, knitting patterns. |
| **Tutorials** | 17% | 9% | Educational materials, tutorials and example applications. |
| **Babies** | 16% | 32% | Valid but extremely small projects with little activity. |
| **Artifacts** | 0% | 21% | Research artifacts developed elsewhere and deposited for sharing. |
| **Deprecates** | 1% | 5% | Deprecated projects, no code on the main branch. |

The answer to our second question is also negative. Starred projects are not representative of the interesting ones. To summarize what we learned about stars, they capture extrinsic characteristics of GitHub projects and are at best an indirect and noisy proxy for a robust frame oracle.

## 4.4 How to select projects?

What to use for project selection if not stars? We argue that selection must be based on intrinsic features – measurable attributes of a project's contents. While one may use machine learning [18, 22] to build classifiers, in this paper we will use our five computed attributes (we leave machine learning as an interesting area of future work).

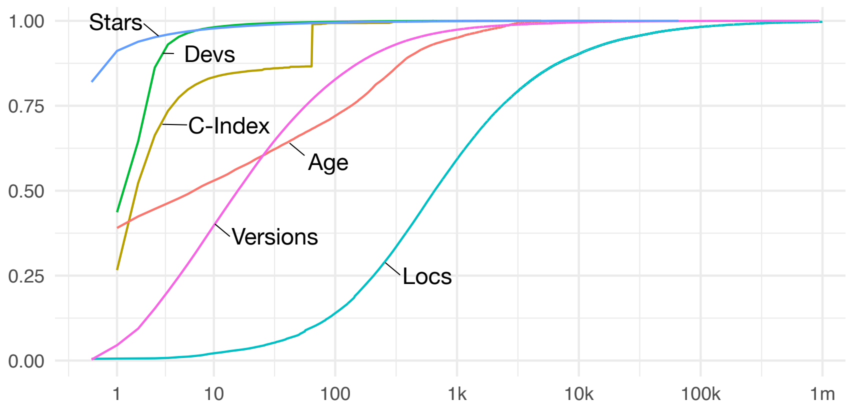

**Figure 3** Cumulative Density Functions.

Fig. 3 is the cumulative density function of the various attributes for Java (the shapes of the curves for Python are similar). The interpretation of each line is what percentage of the dataset is filtered for a particular attribute value. Project selection can be performed by a combination of attributes with cutoffs. We do not argue for a particular formula; researchers must make their own choices in this respect.

For instance, if one were to use 10 days of age as a cutoff, then 52% of the dataset would be filtered out. Whereas picking a 10 star cutoff, filters out 98% of projects.[3]

[3] The discontinuity of C-Index at 65 is odd. After manual investigation, we found that there is a single developer with that C-index, it turns out that the "developer" is a bot doing automated updates.

## 4.5 Validity of our dataset

We noticed an oddity around project ages in our dataset. Experience with GitHub trained us to expect the unexpected. Our investigation started with a plot of creation dates.

Fig. 4 shows the log scaled counts of new projects over time. While there is a steady progression in the count of projects created each year, we see a significant drop in 2015 and a plateau until 2019.

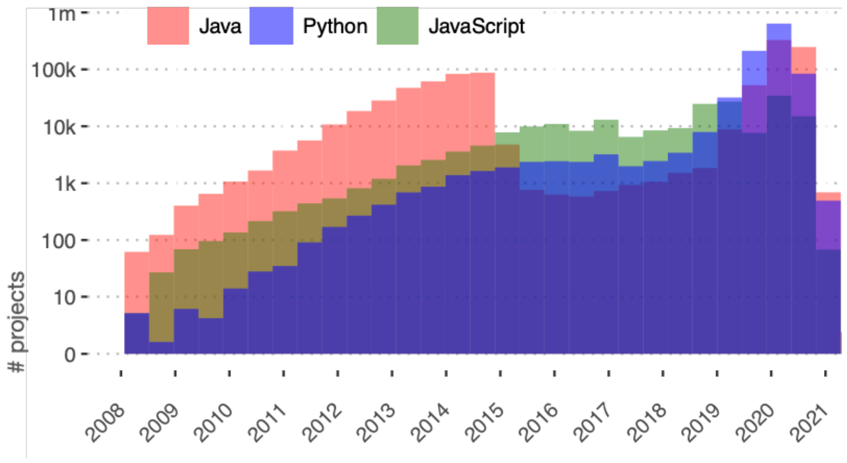

**Figure 4** Creation date.

We reviewed our pipeline to no avail. We use GHTorrent to acquire all available URLs. Then, we randomly sample projects from that list. We validated both acquisition and sampling. This leaves us with two hypotheses. First is a consistent flaw in the CodeDJ downloader causing some projects to fail to download. 17% URLs obtained from GHTorrent point to dead projects, but there is no apparent bias. Second some projects could be missing in GHTorrent.

Another issue showed up on inspection, JavaScript project ages are significantly higher than those of other languages. We found that GitHub timestamps are frequently inconsistent. Why should JavaScript be more affected? Until an explanation can be found, we removed JavaScript from the overall comparison and use JavaScript projects in the reproduction with extreme care.

## 5 Reproducible large-scale analysis experiment design

This paper proposes that researchers conducting experiments over large-scale software repositories follow a specific experimental design methodology to ensure their work can be reproduced and increase chances that their results generalize as expected. While the mechanics of reproducibility of the actual experiment itself vary, the setup of the experiment is a common problem. The proposed methodology has five steps, we encourage researchers to document each of these steps explicitly.

## 5.1 Population hypothesis

Formulating a population hypothesis lets researchers stake a claim about the applicability of their work. This represents the population to which the result of an experiment should generalize to. The statement of that hypothesis can be brief and appeal to intuition, the other parts of the description flesh out the details.

Ideally, we would like our results to be as broadly applicable as possible, but pragmatically designing experiments that back up overly broad claims is difficult. Some populations of interest are difficult to sample, for instance "commercial software" is a relatively simple and unambiguous description but one that we typically cannot sample from as most of the commercial software is not in the public domain. Other populations can be difficult to identify. Imagine a study of the challenges linked to retraining imperative programmers to use functional idioms. Finding code written by such developers can be done manually but is difficult to automate. It is often easier to describe a population by intrinsic features of projects such as the language used to write the code or some estimate of the size of the project.

## 5.2 Frame oracle

A frame oracle is a, possibly noisy, deterministic algorithm for deciding if a project belongs to the population of interest. The oracle is our best approximation of the population of interest. An executable and reproducible oracle allows to compare different papers with the same selection. The description of the oracle should specify the data source along with any information required to acquire projects. The procedure for evaluating a project should be clear and based on intrinsic attributes. A paper should at least have a short description of the oracle, full details should be given in the reproduction artifact.

## 5.3 Sampling strategy

The literature has an abundant advice on sampling (see e.g. [12]). Briefly, a sampling strategy picks the type of sampling (probabilistic or non-probabilistic) and describes the steps used to obtain a sample. The sampling implementation is expected to be found in the reproduction artifact.

Many works use convenience sampling as it is simpler, cheaper and less time consuming. A better alternative is some form of probabilistic sampling as it is more likely to yield a representative sample. Probabilistic sampling can be staged if the structure of the population is more complex. The simplest approach is random sampling where each element has the same chance of being picked. We often have to resort to stratified sampling when the population is divided into subgroups of different sizes. Typically we sample without replacement as we do not want to pick the same project multiple times.

## 5.4 Validity

The validity section argues, when there are reasons for doubt, why using the frame oracle and the sample strategy results in representative samples of the population of interest. This section should address potential sources of bias and attempts by the authors to control for them. This section also should address any foreseen challenges to reproducibility and offer means to mitigate them.

## 5.5 Reproducibility artifacts

Finally, we advocate to link the paper to a reproduction artifact that contains code and data to support experimental repeatability and reanalysis.

Section 3 listed issues with reproducibility. In some cases, the authors did not give a precise description of the steps needed for reproduction. Following our proposed methodology along with a reproduction artifact will greatly help.

The second category of issues are more pragmatic, it is difficult to repeat the analysis of a paper because some aspect of the data used is not available. We suggest that research infrastructures should support this task.

An example of an infrastructure is CodeDJ which is both a continuously updated datastore and a database that can be queried by a DSL. We adopted it for our work and illustrate how it helps with reproducibility. The implementation of a frame oracle and sampling strategy can be combined into a single expression. Fig. 5 shows a query which starts by filtering out projects containing fewer than 80% JavaScript code, then it uses pre-computed attributes Locs, Age and Devs to filter further. The last stage of filtering involves computing an attribute on the fly, here we sum up the commits in the project, before performing random sampling.

```
database.projects().filter(|p| {
      p.language_composition().map_or(false, |langs| {
         langs.into_iter().any(|(lang, rate)| { lang == JavaScript && rate >= 80 })
      })
  })
  .filter_by(AtLeast(Locs, 5000))
  .filter_by(AtLeast(Age, Duration::from_months(12)))
  .filter_by(AtLeast(Devs, 2))
  .filter_by(AtLeast(Count(Commits), 100))
  .sample(Random(30, SEED)))
```

**Figure 5** Project selection with CodeDJ.

CodeDJ is split between a persistent datastore in which every data item is timestamped, and an ephemeral database used to service queries. A reproducible query is a Rust crate archived in a git repository associated to the datastore. Running the query produces a *receipt* which is the hash of a commit automatically added to the archive repository. The receipt can be used to share the query (exactly as executed) and its results (exactly as produced). It can be used to retrieve the Rust crate and re-execute the code. Code re-execution is helped by the fact that queries are deterministic and the crate contains a list of all dependencies, a timestamp, and all random seeds. When a query with an embedded date is executed, CodeDJ accesses the exact state of the datastore at the specified date. Since CodeDJ stores the contents of files, entire experiments can be fully reproduced.

## 6 Reproductions

We illustrate the use of the proposed methodology by revisiting the papers we discussed in section 3. For each paper, we attempt reproduction where we vary the input. The original papers used stars in their selection, we will explore different inputs based on intrinsic attributes.

If the results of the reproduction match the original results, then it suggests that stars were an appropriate filter. If the reproduction departs, this suggests that there may be need to conduct further experiments to be confident in the results.

For each paper, we picked a subset of the scientific claims to fit the reproduction in the available space. We use our proposed methodology to describe the details of the reproduction.

One may wonder how we selected the paper to reproduce. Our criteria were (1) papers that used automated techniques to analyze properties of the code contained in Github projects, (2) their population of interest was a large subset of Github, (3) a working artifact could be located, and (4) the input could be changed with ease. We did not cherry-pick as even positive results would be interesting. Our choice was limited by the fact that many

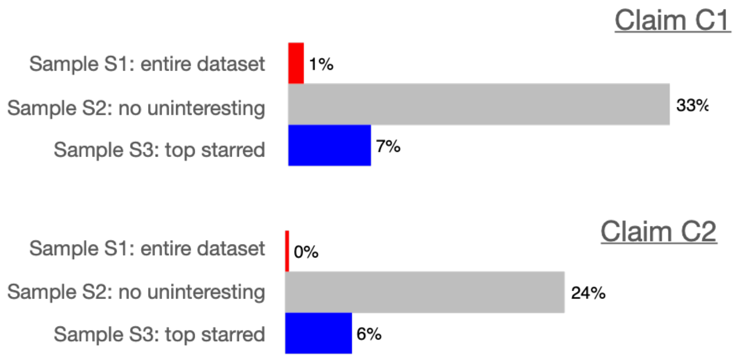

**Figure 6** Proportion of projects without code, data or documentation.

papers either did not have artifacts or that they were not working anymore. Furthermore, some papers had hardwired they selection of projects by, for example, preprocessing the input data and omitting to include the tooling to repeat that processing. Given more time, more works could be reproduced.

## 6.1 Reproduction: What is software

This reproduction aims to validate two findings of [21]: (C1) 4% of repositories do not contain code, data and documentation; (C2) 2% of repositories do not contain documentation.

**Population Hypothesis:** The universe of software projects.

**Frame Oracle:** To understand the impact of project selection we consider two oracles. O1 accepts any project hosted on GitHub. O2 removes uninteresting projects (as defined above).

**Sampling Strategy:** We report on three samples. S0 is a convenience sample of starred following [21]. S1 and S2 are random samples without replacement from O1 and O2 respectively, stratified by language and deduplicated.

**Validity:** Our reproduction differs in the number of languages (3 v. 25) and by categorizing files based on the file path alone. We tested stability of our results with multiple samples of varying sizes and manually inspected the produced labels.

**Reproduction Artifact:** Our artifact contains a CodeDJ receipt for this query.

### 6.1.1 Results

Fig. 6 shows results for claims C1 and C2. Compare the percentages between S0 (original) and S1 (target population). Statistical analysis is not required to see that the difference is significant. The sample S2 (without uninteresting projects) is there to illustrate the impact of slightly more developed population, but even these are still quite different.

Would the results agree if we included more languages? The three languages we downloaded account for most of GitHub, it is conceivable that other languages could affect results, but that would just push the generalizability issue somewhere else as the claims would become language-specific.

## 6.2 Reproduction: Method chaining

Nakamura et al. [20] claim that 50% of projects in 2018 had method chains longer than 7 while in 2010 that number was 42%. They state that “chains of length 8 are unlikely to be composed by programmers who tend to avoid method chaining, this result is another supportive evidence for the widespread use of method chaining.”

**Population Hypothesis:** The universe of real-world Java programs.

**Frame Oracle:** We accept any Java project hosted on GitHub and delegate to Github for language attribution.

**Sampling Strategy:** Stratified sampling to randomly select projects with commits in 2010 and 2018.

**Validity:** To reproduce the original results, we performed stratified sampling to get top starred projects active in the target years. The authors used a different sample of top stars. The original paper had different sample sizes for each year, but those are not specified. We fix the sample size to 250. The authors could not locate the code of their chain detector, so we use our own implementation.

**Reproduction Artifact:** Our artifact contains a CodeDJ receipt for this query.

### 6.2.1 Results

Fig. 7 shows the difference in proportion of projects at various chain lengths. The solid line uses stars, colors represent different random samples. For instance, if we pick chains of length 8, the number used by [20], the difference is a 13% increase in the number of projects between 2011 and 2018. The differences for our random samples are -2%, 0.6% and 0.7%. In other words, the samples from this particular population do not seem to show the effect expected by the authors. We surmise that some notion of developed project may show more favorable results, but without more guidance in the population hypothesis it is hard to guess which to pick.

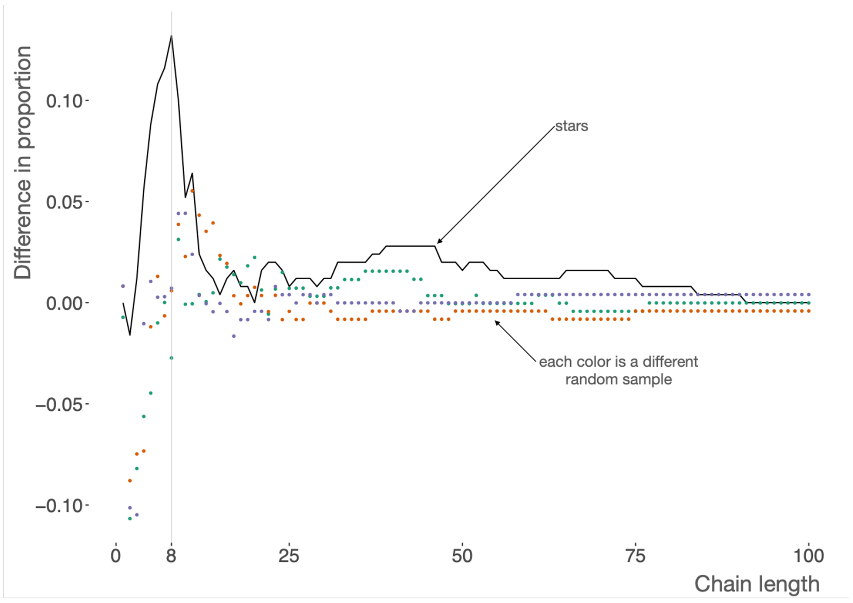

**Figure 7** Difference in chain lengths.

### 6.3 Reproduction: Style analyzer

Markovtsev et al. [16] builds a model of the style of a repository and apply this model on a held-out part of that repository to produce corrections. Their experiment uses 19 top-starred JavaScript project to gauge the precision with which the tool flags formatting discrepancies and the relationship between this precision and the size of the project. They report a precision of 94% (average, weighed by project size) and better overall performance for large projects and projects with better style guidelines.

**Population Hypothesis:** Developed JavaScript projects.

**Frame Oracle:** JavaScript projects with at least 80% JavaScript code, Loc $\geq 5000$, Age $\geq 12 * 31$ and Devs $\geq 2$.

**Sampling Strategy:** We randomly select 10 sets of 30 projects. This is more projects than the original sample to account for errors in processing. After processing is finished, following the original paper, we randomly select 19 projects out of the pool of successfully processed projects.

**Validity:** Given that the author's artifact lacks a configuration, we used the default one. This increases project size, as compared to the published numbers, by 38% per project (up to a maximum of 154%) and causes precision to diverge by 2.2% on average, and up to 7.9%.

The tool failed to process 4 projects: `freecodecamp` and `atom` due to errors in unicode processing, `express` due to a programming bug, and `30-seconds-of-code` due to bad file identification. Three of the missing projects were located close to the median in terms of precision, prediction rate, and project size in the original paper, while `axios` was in the lower quartile for sample count.

Style analyzer analyzes each project at two points in its history specified by a base commit and a head commit. The base commit is a point in the past which the tool checks out to learn the project's formatting style. The head commit is a more recent point used to evaluate the model and calculate precision. The original paper provides head and base commits for each project in their experiment, but does not specify the method of selecting these commits. We pick the current head of the default branch as the head commit. For base commit we pick one that lies at an offset equal to 10% of the number of all commits in the default branch from the head commit. This retrieves different commits than the original paper, which causes a 3% median change in precision (up to 17%– `telescope`) and a median project size increase of 76%, and up to 311% (`reveal.js`).

**Reproduction Artifact:** Datasets, receipts from submitted queries, style analyzer's reports and scripts for the entire experimental pipeline are included in the artifact.

#### 6.3.1 Results

We recreate a plot of the effect of the number of items in the training set on precision from the original paper in Fig. 8. The training set consists of snippets created around tokens/AST nodes relevant to formatting (whitespace, indentation, quotes, zero-length gaps). We plot the selection from the original paper along three selections from our interesting project frames. In addition, we plot the distributions of precision in each selection in

In Fig. 9, we compare the precision scores in each sample with the selection used in the original paper using a Mann-Whitney U test to show which samples performed statistically differently from the original. The scatter plots show a different grouping of results from the original paper. The groupings in the scatter plot visibly differ between selections. The distribution comparison shows that our selections generate significantly smaller training

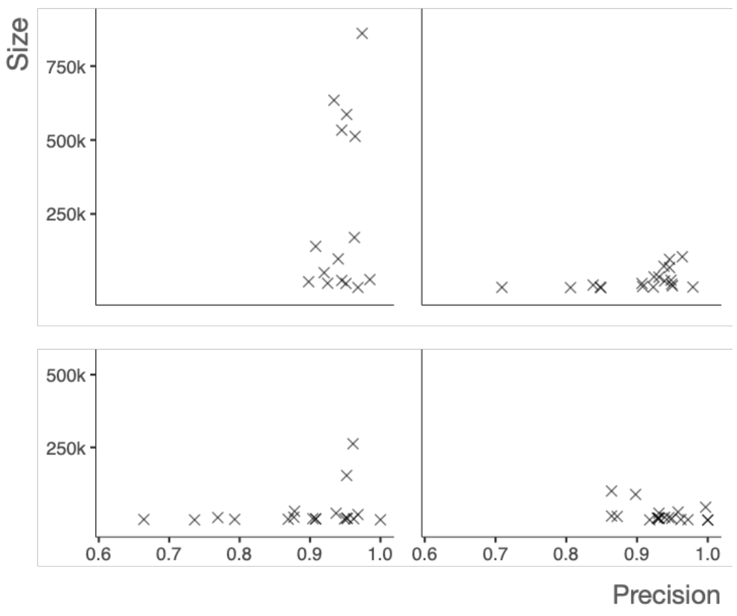

**Figure 8** Relationship between label groups and precision.

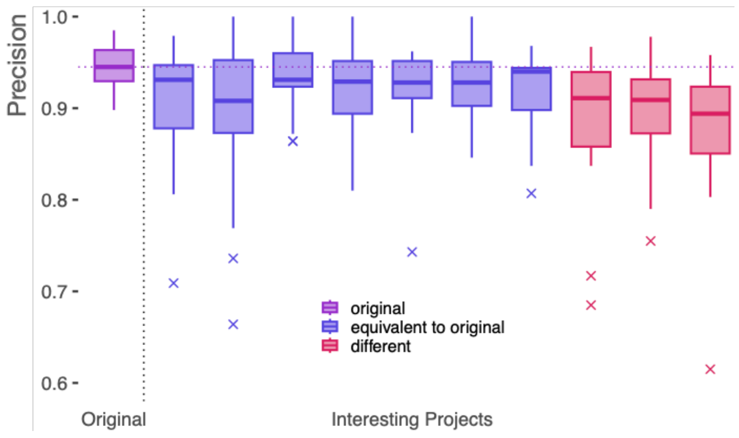

**Figure 9** Comparing label group count and precision.

sets in all cases and yield lower precision. In addition, 3 out of the 10 interesting project selections produced significantly lower precision, with the remainder producing a statistically equivalent distribution.

Overall, we see our selections yielding precision between 0.9 and 0.95 (the paper sets a precision of 0.95 as a benchmark for success). We also do not see a clear relationship between the number of label groups and precision, such as the one the authors note in the original paper.

## 6.4 Reproduction: Code smells

We seek to validate the claim of [9] that for large and small projects there is a statistical difference in the occurrence of code smells between machine learning and most popular Python repositories, whereas medium sized projects are indistinguishable.

**Population Hypothesis:** Mature Python projects in all application domains including machine learning.

**Frame Oracle:** Projects with C-Index $\geq 5$, or Age $\geq 180$, or Locs $\geq 10,000$, or Versions $\geq 100$.

**Sampling Strategy:** The deep learning projects were provided by the authors. Out of 59 projects, 57 were still accessible on August 2nd 2021. At download time there were 6 small, 13 medium, and 38 large deep learning projects. For the reproduction of the original results, we used a staged strategy, first convenience sampling the top starred Python projects and amongst those used stratified sampling to select 57 projects with a similar distribution of sizes. To generalize the results we used quota sampling to match the size distribution.

**Validity:** Our reproduction uses the Locs reported by CodeDJ. The date the authors downloaded the repositories is unknown. We use the content of the main branch of each repository as of April 1st, 2020. The authors say "*each of repositories is pre-processed and prepared for code smell detection*", however details are missing. We used the default thresholds of their tool.

**Reproduction Artifact:** A CodeDJ receipt is included in our reproduction package along with code to run the experiment.

### 6.4.1 Results

Fig. 10 contrasts the distribution of code smells for deep learning projects, top starred projects, and three random samples. Computing the p-values with the non-parametric Mann-Whitney Wilcox shows that while we were able to reproduce the statistically significant results for the small projects, we disagree on the large most popular projects with the original. The disagreement is even greater in the random samples where no large projects and only one small project is statistically different. Generalizability of the results is thus questionable.

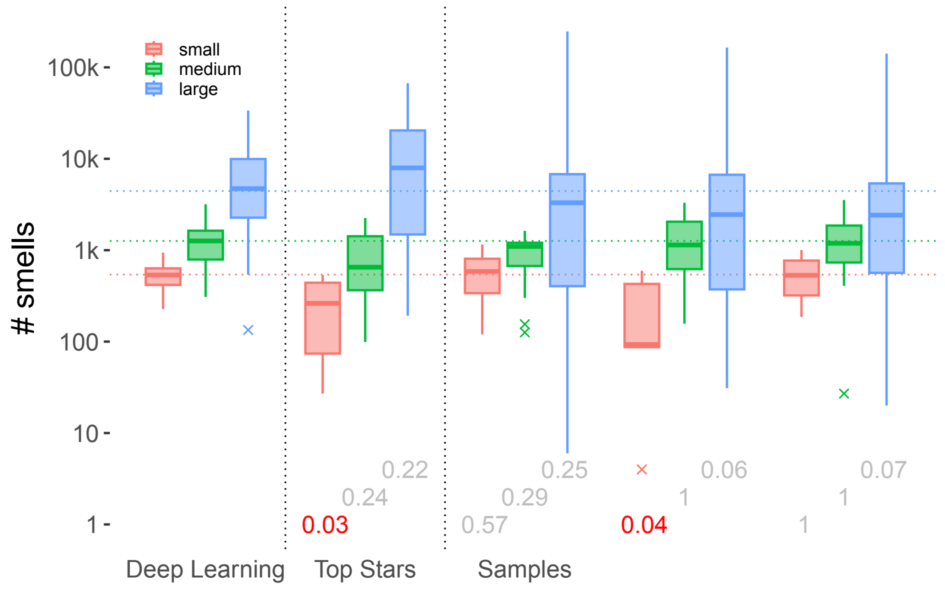

**Figure 10** Comparing Smells. Numbers are p-Values indicating a significance of the difference from the deep learning projects. Statistically significant different p-Values (cutoff at 0.05) are shown in color, insignificant ones are in gray.

## 7 Collaborative Reproduction

We perform one last reproduction in which we obtain the assistance of the study's authors to validate whether stars are a good input selection strategy. For this reproduction we selected a distinguished paper from the Foundations of Software Engineering conference (2020), "The Evolution of Type Annotations in Python" [7]. The paper has a reusable artifact for repetition of the original results. Our reproduction only required to change the list of GitHub URLs used as input in the analysis. The authors helpfully allowed us to run the rather computationally intensive workload on their machine.

The original study reported the following protocol for input selection. *"We group projects by creation date, considering projects created in the years 2010 to 2019, into ten groups. We sort each group by number of stars and select the top-1000 per group, which yields a total of 10,000 projects. The rationale for first grouping and then sampling is to avoid biasing our study toward projects created in a particular time frame, e.g., mostly old projects. Removing projects that we could not clone, e.g., because they became unavailable since the beginning of our study, the total number of analyzed repositories is 9,655."*

The first research questions was related to the evolution in the number of type annotations in projects. The main insight from the work was that *"better developer training and automated techniques for adding type annotations are needed, as most code still remains unannotated, and they call for a better integration of gradual type checking into the development process."*

For this reproduction, we discussed input selection criteria with the authors and arrived at the following formulation.

**Population Hypothesis:** Python projects in all application domains with earliest commit date in 2015[4] as this was the year when early adoption of type annotations began.

**Frame Oracle:** Python projects whose life span is longer or equal to 7 days and that have over 100 lines of Python code. The study authors intended their work to be representative of most of the Python ecosystem, but closer inspection of some of the small projects suggested that they would introduce noise. The particular cutoffs were chosen heuristically.

**Sampling Strategy:** Projects were grouped by year active and, for each year, a random sample of 1200 projects was selected. The goal was to get close to a thousand usable projects for each year. As some of the projects in our database are no longer available, the sample size is increased heuristically.

Fig. 11 illustrates the reproduction results (and mirrors Fig. 2 in [7]). The plot on the left shows the number of type annotations found per thousand lines of code in the projects being looked at. The red line has the new data, the blue one is for the original study. The y-axis is logarithmic. The two lines start at zero in 2015. Both data sets tell a similar story: type annotations are gradually added to projects. The plot on the right shows the total number of annotations found each year in all projects. The y-axis is in thousands. In 2021, the original data had close to 800,000 annotation while the new data is under 250,000.

Both data sets are large. The original one contains 1,123,393 commits and the new data set 1,535,824 – suggesting that projects are slightly larger in the randomly selected data then in the most starred projects. In both cases, only a fraction of the repositories have types. In the old data 668 repositories are type-annotated, whereas in the new data 1,040 projects have at least one type. The fraction of commits that change a type annotation is small in both cases 5.5% in the original data and 2.1% in the new data.

[4] 2015 is when type annotations were added to Python.

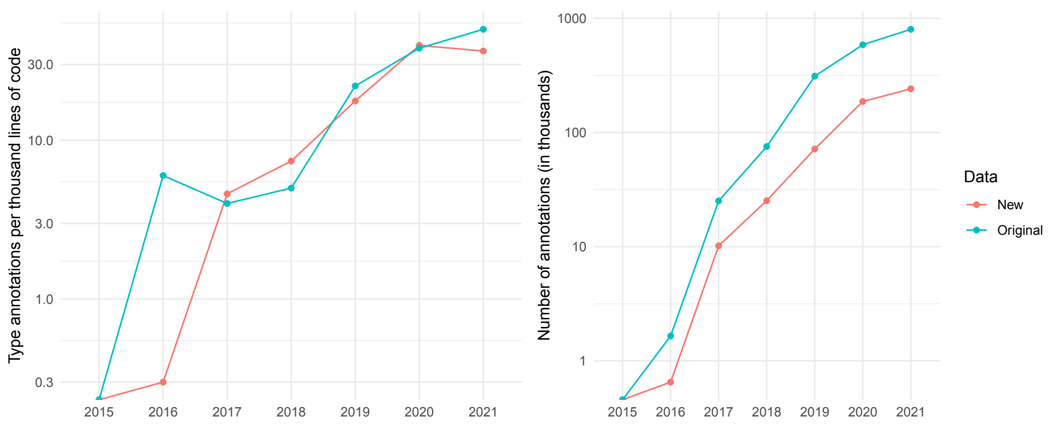

**Figure 11** Types in Python.

Overall, the reproduction verifies and, even, strengthens the conclusion of the original paper. Five years after introduction of type annotations, their use remains rather limited. Having said this, it is true that actual values reported are different enough to be noticeable.

## 8 Conclusions

Sometimes, doing it wrong is so much easier than the alternative, that we convince ourselves that a little wrong can be right enough. Our paper is unusual. While it purports to contain a call to arms for better experimental practices, it is just as much a record of our own journey to that goal. What reads as criticism is really written in self-reflection. So, what can a researcher take away from this paper? There are three ideas we would like to leave the reader with.

**Generalizability.** The value of an experiment often lies as much in what it generalizes to, as in the experiment's outcome. We found that many researchers rely on GitHub stars to pick representative samples of software projects, yet starred projects tend to be larger in most dimensions than typical ones, also that they are more likely to be inactive, and that their ranking is not a measure of intrinsic qualities of the code. Hopefully, this paper is the last nail in that coffin. More generally, we advocate for the use of probabilistic sampling over populations defined by intrinsic attributes of software, and also for clear and standardized documentation of experimental design.

**Reproducibility.** The value of a scientific experiment also lies in our ability to reproduce it. Carrying out reproducible experiments over large-scale software repositories is hard. Especially when aiming to support the three reproduction modalities: repetition, as practiced in artifact evaluation, where an artifact is re-executed to obtain identical results; reanalysis, where the artifact or its input are modified; and independent reproduction, where the entire experiment is re-implemented from scratch. The first modality requires faithful replay and is best served if all data used is included with the artifact. The second, requires support for automatically acquiring new representative samples. The third needs an unambiguous description of all experimental steps. We advocate for reproductions artifacts that supports the first two modes, and a detailed description of the experiment for the last.

**Tooling.** Generalizability and reproducibility, while worthy goals, represent much work, and they are work that is orthogonal to the scientific goals of researchers. The only reasonable answer is to provide tooling that automates acquisition of representative samples and generation of reproduction artifacts. In this paper, we used `CodeDJ` and found it helpful as it let us specify queries over attributes of the code for many projects, while also supporting experimental repetition and reanalysis through historical queries. It has its limitations, we found execution times to be somewhat long and doubt it will scale to the whole of GitHub.

Our vision for a brighter future is one where the community agrees on standard tools and techniques for this kind of experiment, tools which automate the acquisition and packaging of input datasets and the re-execution of entire experiments.

## References

1 S Baltes and P Ralph. Sampling in software engineering research: a critical review and guidelines. *Empir. Softw. Eng.*, 27(4):94, 2022. `doi:10.1007/s10664-021-10072-8`.

2 H Borges and M Tulio Valente. What's in a github star? understanding repository starring practices in a social coding platform. *Journal of Systems and Software*, 2018. `doi:10.1016/j.jss.2018.09.016`.

3 Z Chen et al. Understanding metric-based detectable smells in python software. *Information and Software Technology*, 2018. `doi:10.1016/j.infsof.2017.09.011`.

4 V Cosentino, J Izquierdo, and J Cabot. Findings from GitHub: Methods, datasets and limitations. In *Mining Software Repositories (MSR)*, 2016. `doi:10.1145/2901739.2901776`.

5 R Dyer, H Nguyen, H Rajan, and T Nguyen. Boa: A language and infrastructure for analyzing ultra-large-scale software repositories. In *Int. Conf. on Software Engineering (ICSE)*, 2013. `doi:10.5555/2486788.2486844`.

6 G Gousios and D Spinellis. GHTorrent: GitHub's data from a firehose. In *Mining Software Repositories (MSR)*, 2012. `doi:10.1109/MSR.2012.6224294`.

7 L Di Grazia and M Pradel. The evolution of type annotations in python: An empirical study. In *European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2022. `doi:10.1145/3540250.3549114`.

8 J Han et al. Characterization and prediction of popular projects on GitHub. In *Computer Software and Applications Conf. (COMPSAC)*, 2019. `doi:10.1109/COMPSAC.2019.00013`.

9 H Jebnoun et al. The scent of deep learning code. In *Mining Software Repositories (MSR)*, 2020. `doi:10.1145/3379597.3387479`.

10 E Kalliamvakou et al. The promises and perils of mining GitHub. In *Mining Software Repositories (MSR)*, 2014. `doi:10.1145/2597073.2597074`.

11 S Krishnamurthi and J Vitek. The real software crisis: repeatability as a core value. *Commun. ACM*, 58(3), 2015.

12 S Lohr. *Sampling: Design and Analysis.* Cengage Learning EMEA, 2010.

13 C Lopes et al. Déjà Vu: A map of code duplicates on GitHub. *Proc. ACM Program. Lang. (OOPSLA)*, 2017. `doi:10.1145/3133908`.

14 Y Ma et al. World of code: enabling a resarch workflow for mining and analyzing the universe of open source vcs data. *Empirical Softw. Eng.*, 2021. `doi:10.1007/s10664-020-09905-9`.

15 P Maj et al. CodeDJ: Reproducible queries over large-scale software repositories. In *European Conf. on Object-Oriented Programming (ECOOP)*, 2021. `doi:10.1145/2658987`.

16 V Markovtsev et al. Style-analyzer: fixing code style inconsistencies with interpretable unsupervised algorithms. In *Mining Software Repositories (MSR)*, 2019. `doi:10.1109/MSR.2019.00073`.

17 T Mattis, P Rein, and R Hirschfeld. Three trillion lines: Infrastructure for mining github in the classroom. In *Conf. on Art, Science & Eng. of Programming <Programming>*, 2020. `doi:10.1145/3397537.3397551`.

18 N Munaiah et al. Curating github for engineered software projects. *Empirical Software Engineering*, 2017. doi:10.1007/s10664-017-9512-6.

19 M Nagappan, T Zimmermann, and C Bird. Diversity in software engineering research. In *Foundations of Software Engineering (FSE)*, 2013. doi:10.1145/2491411.2491415.

20 T Nakamaru et al. An empirical study of method chaining in Java. In *Mining Software Repositories (MSR)*, 2020. doi:10.1145/3379597.3387441.

21 R Pfeiffer. What constitutes software? In *Mining Software Repositories (MSR)*, 2020. doi:10.1145/3379597.3387442.

22 P Pickerill et al. Phantom: curating github for engineered software projects using time-series clustering. *Empir Software Eng*, 2020. doi:10.1007/s10664-020-09825-8.

23 P Ralph. SIGSOFT empirical standards released. *Softw. Eng. Notes*, 46(1):19, 2021. doi:10.1145/3437479.3437483.

24 J Vitek and T Kalibera. R3: Repeatability, reproducibility and rigor. *SIGPLAN Not.*, 2012. doi:10.1145/2442776.2442781.