External Factors in Sustainability of Open Source Software

Marat Valiev

CMU-ISR-21-103

February 2021

Institute for Software Research
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Thesis Committee:
James Herbsleb (Chair)
Bogdan Vasilescu (Co-Chair)
Audris Mockus (University of Tennessee)
Vladimir Filkov (UC Davis)

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Societal Computing.

Keywords: Collaborative Software Development, Empirical Software Engineering, Open Source

Abstract

Modern software development is heavily reliant on Open Source. It saves time and money, but, as any other non-commercial software, it comes on as-is basis. If not properly maintained or even abandoned by its community, Open Source Software (OSS) might impose extra costs or introduce bugs to the downstream projects. While software developers are well aware of these premises, existing techniques of mitigating such risks assume sustainability to be an intrinsic property of OSS, largely ignoring external factors. With plenty of examples of even high profile projects failing because of bugs in upstream dependencies, funding issues, lost competition or key developers left, it is important to understand the effect of these factors on OSS sustainability.

Using a combination of quantitative and qualitative methods, this dissertation explores effects of external factors in OSS sustainability, the mechanisms behind them, and proposes tools to make certain risk factors more visible. The findings indicate that multiple external factors, such as reused libraries, dependent projects and organizational involvement, play a significant role in OSS projects sustainability. Projects serving end users and end programmers are particularly at risk to be overwhelmed by excessive number of requests, especially questions. We found, however, that there are simple heuristics that can help getting additional insight into the structure of effort demand in OSS. For example, since established users of software mostly report bugs and new adopters mostly ask questions, we can estimate project's lifecycle stage and user base structure using already existing issue classification. Finally, this work shows that in many cases simple tools, such as autoencoder-based embeddings, can be used to detect less visible sustainability factors, such as competition and surrounding communities.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Open Source Software (OSS) plays increasingly important role in our lives. OSS products are responsible for over 70% of mobile OS market share, they power over 80% of active web servers, and make over 70% of server OS. Even in Microsoft cloud platform, Azure, over 60% of customer machines are powered by Linux[1]. Already in 2015, annual Black Duck Survey of commercial companies about their usage of Open Source reported that less than 3% of respondents *do not* use OSS in any way, after which they stopped reporting this metric. 2020 Open Source Security and Risk Analysis (OSSRA) report indicated that 99% of audited commercial code bases contained Open Source components[2].

Not only we rely on OSS a lot, but also it is often used in critical applications. Most of commonly used cryptography libraries are developed using Open Source model[3]. Besides dominating some critical domains, some key players extensively rely on OSS as well. For example, a Department of Defense (DoD) study concluded that "FOSS software plays a far more critical role in the DoD than has been generally recognized" [15].

Such degree of reliance on OSS products, including those in critical areas, means that consequences of failures in these projects will also be broad and critical. Examples of such critical failures include exploitation of a know vulnerability in Apache Struts server resulting in exposure of nearly 150 million people's private data from a consumer credit report agency Equifax. Another example is the infamous bug in OpenSSL library disclosed in 2014, letting attackers to read potentially sensitive data from servers memory. This vulnerability, Heartbleed, reportedly affected over half of active Internet servers at the time.

Despite these incidents, OSS importance continues to grow. Internet-wide scan of IPv4 revealed that not only OpenSSL remained to be the most popular TLS library used on the Internet, but its popularity continued to grow, reaching 70% share of all servers and 85% of top 1M Alexa domains in 2017 [119]. This suggests that it is even more important to develop tools to predict and prevents such failures in OSS than ever before.

[1] According to Adir Ron, Open Source Lead at Microsoft https://www.linkedin.com/posts/adirron_linuxonazure-progress-activity-6665502370795003905-DY1G/

[2] page 5, https://www.synopsys.com/content/dam/synopsys/sig-assets/reports/2020-ossra-report.pdf

[3] 15 out of 17 most popular TLS implementations are Open Source https://en.wikipedia.org/wiki/Comparison_of_TLS_implementations

Unlike commercial products, OSS comes without any warranty of support. In many cases community provided support allows to address problems reasonably fast, for example an infamous disruption caused by deletion of a minor package in NPM, leftpad, was fixed by NPM maintainers in a matter of few hours. However, often problems in an OSS project might gradually build up and stay unnoticed for a very long time. For example, the aforementioned Heartbleed bug was introduced two years before its discovery. The OpenSSL project at the time only had one full time engineer, who was overworked and underpaid. The project did not have enough resources to implement proper analysis and reviews. These organizational premises eventually led to the critical failure of the most common encryption library on the Internet[50]. However, with enough visibility and transparency of the project issues, this failure could be prevented.

However, even a perfect model would not be enough to estimate sustainability risks given just project environment. In the Leftpad incident most disruption was caused not to relatively few direct users of the package, but further in the downstream dependencies through packages like `babel`, in turn used by other projects. In the Heartbleed case, technical dependency chain was even harder to trace, as OpenSSL was implicitly included as a part of server environment, *e.g.*, OS or web server package. Arguably, external factors, such as issues in upstream dependencies and external funding, played critical role in these failures.

While a lot of existing Software Engineering body of knowledge can be at least at some extent applied to OSS, there was little work on the role of external factors in sustainability of OSS projects. For example, prior studies dependency network properties of software ecosystems [30, 31, 42] were mostly focused on evolution of the network properties, rather than their influence on sustainability. Prior studies of competing libraries we fairly limited and often did not take technical dependencies into account. For example, a study of three competing database access libraries concluded that competing projects coexist, rather than replace each other, while overlooking technical dependencies across the studied libraries, making their standalone usage technically impossible [40]. Some other studies explored effects of external factors, such as developers collaboration network, as a determinant of project success [143]. While the existing work was mostly focused on project success, project sustainability, a largely orthogonal factor, remains a relatively underexplored area. This proposal aims to fill the gap of research on the role of external factors in sustainability of OSS projects, including dependency factors, competition, surrounding community structures, and more.

This thesis describes three studies completed in satisfaction of PhD program in Societal Computing at Carnegie Mellon University. The first study aims to confirm significance and measure effects of external factors in OSS projects. The second study frames sustainability of OSS as a problem of balancing effort supply and demand, looking further into signals that can be used to manage effort demand. The third, and the final, study seeks to develop methods to estimate effects of dependency factors, such as presence of competing projects, availability of developers and finding potential substitutes for upstream dependencies.

Chapter 2

Significance of external factors in OSS

2.1 Abstract

Open-source projects do not exist in a vacuum. They benefit from reusing other projects and themselves are being reused by others, creating complex networks of interdependencies, *i.e.*, software ecosystems. Therefore, the sustainability of projects comprising ecosystems may no longer by determined solely by factors internal to the project, but rather by the ecosystem context as well.

In this paper we report on a mixed-methods study of ecosystem-level factors affecting the sustainability of open-source Python and Javascript projects. Quantitatively, using historical data from 46,547projects in the PyPI ecosystem and 209,895projects in the npm ecosystem, we modeled the chances of project development entering a period of dormancy (limited activity) as a function of the projects' position in their dependency networks, organizational support, and other factors. Qualitatively, we triangulated the revealed effects and further expanded on our models through interviews with PyPI project maintainers. Results show that the number of project ties and the relative position in the dependency network have significant impact on sustained project activity, with nuanced effects early in a project's life cycle and later on.

2.2 Introduction

While only twenty years ago open-source software (OSS) was simply a curiosity that attracted the attention of a few academics and was not seriously considered in the software industry, OSS infrastructure today is ubiquitous,[1] powering applications in virtually every domain. Economists refer to OSS as "digital dark matter" [64], to signify both its invisibility and importance. They also report valuations of OSS in the billions of dollars per year [38, 64], in terms of both direct reuse value and boosted productivity and efficiency.

Given the importance of OSS digital infrastructure to so much of the economy, one might expect that it is adequately staffed and maintained, *i.e.*, sustainable. Yet, this is often not the case. As a recent Ford Foundation report investigating the sustainability of OSS "digital infrastructure" [50] notes, most users of OSS infrastructure take it for granted, and society at large is unaware of the risks. A vivid example is OpenSSL, the OSS project critical to the secure operation of the majority of websites, recently in the spotlight for the "heartbleed" security bug [48]; at that time, Open SSL was severely understaffed. Another

[1] Already in 2015, less than 3% of respondents to a Black Duck Survey reported they *do not* use OSS in any way, https://bit.ly/2NgQNRH (slide 9).

notable example is leftpad [1], the trivial 11-LOC JavaScript package that, when deleted by its author from the npm[2] registry, caused cascading disruption in thousands of other projects that relied on it being accessible on npm.

Besides emphasizing the importance of OSS sustainability issues, both examples illustrate a challenge with modern code reuse. Indeed, with vast amounts of high-quality OSS code available for reuse, one can declare dependencies on others' code instead of copying it into their own, taking advantage of the functionality without assuming the burden of maintenance. This leads to the formation of large code interdependency networks. However, this also means that local sustainability issues around individual projects can have widespread network effects. For example, breaking changes—changes that are not backwards compatible—are a significant source of instability, causing negative consequences for dependents downstream [13].

Developer disengagement is another issue that haunts OSS sustainability. 41% of failed open source projects cite a reason involving the developer team, such as lack of interest or time of core contributors [26]. Contributors to and maintainers of OSS are often overworked volunteers, who can decide to stop contributing at any time [138, 170]. Developers disengage from OSS for many reasons. Aside from personal reasons such as family planning or job changes, OSS contributors and maintainers also suffer from lack of community support. This can be understood from two aspects: developers who work on less popular projects are more likely to disengage [**?**]; female developers are more likely to leave a project due to lack of diversity among the project contributors and new developers who have few prior connections with the OSS community are at a higher risk of disengagement [132]. As a result, some projects have a difficult time keeping core developers engaged and/or attracting new developers so that they could be groomed to become core developers. Consequently, OSS projects risk knowledge loss [118, 136], quality degradation [52], or even extinction [26], again with reverberations downstream.

These challenges are particularly visible in OSS package ecosystems like npm, PyPI,[3] and CRAN,[4] where packages form complex and often brittle dependency chains [41, 88]. With reuse so enticing and so much OSS code available, how can one make informed decisions about which packages to use? While OSS projects can be long-lived (*e.g.*, Linux, Apache, and Eclipse), relatively few reach a mature state [18, 29] and many that are active for a period of time are eventually abandoned [87], even once-popular ones [26]. Will a package still be maintained in a year? Which packages are sustainable and which are at risk? How does a package's place in the ecosystem influence its survival chances? How do developer choices, ecosystem community norms, and social processes contribute to sustainability or extinction? The empirical evidence for the mechanisms and predictive factors of OSS project survival *in an ecosystem context* is, at best, fragmented and incomplete.

In this paper, we report on a mixed-methods study of the Python PyPI and Javascript npm ecosystem, that makes a step towards filling this gap. Specifically, we conduct a mixed-methods study on the PyPI ecosystem; we further validate our conclusions drawn from the PyPI ecosystem by replicating the quantitative study on the npm ecosystem. PyPI is the official third-party registry for Python packages and one of the most popular OSS ecosystems, with over 250,000 published packages as of November 2020. npm is the world's largest software registry, containing over 1.4M packages and is used by more than 11,000,000 developers.

We study the ecosystem-level factors impacting the chances of a package becoming dormant, *i.e.*, having very low or no development activity after some time. While not all dormant projects are abandoned (*e.g.*, some simply do not require any additional maintenance because they are feature complete [26]), being

[2]Node.js Package Manager, https://www.npmjs.com
[3]Python Package Index, https://pypi.python.org
[4]Comprehensive R Archive Network, https://cran.r-project.org

in an inactive state could signal sustainability risk. For example, for an external observer, lack of project activity may indicate abandonment and increased uncertainty about whether potential issues or feature requests would be dealt with.

We interview maintainers of PyPI packages; integrate data from PyPI, and GitHub, mining repositories and their interdependencies to assemble an ecosystem-level longitudinal data set; identify which packages became dormant; and estimate Cox proportional hazards survival regressions [33, 110] to model the factors affecting a package's chances of entering this dormant state. We then validate the results obtained from the PyPI mixed-methods study on the npm ecosystem.

We find that the number of connections and the relative position in the dependency network are significant factors affecting the chances of a project becoming dormant; the organizational support a package receives, if any, has different effects depending on the type of supporting organization; and the practice of producing backwards compatible releases does not appear to influence project dormancy under our definition. The direction of the discovered effects was consistent across ecosystems across the board, even though the effect of external factors was somewhat smaller in npm.

In summary, we contribute (1) a dependency-network-based survival analysis of packages in the OSS ecosystem; (2) a series of interviews with project maintainers from the PyPI ecosystem to triangulate and refine the discovered relationships; and (3) an in-depth discussion of the effects revealed by the mixed-methods analysis.

2.3 Development of Hypotheses

As with natural systems, the sustainability of OSS projects (much like the success of OSS projects [35]) is also clearly a multi-faceted concept; *e.g.*, projects may be considered sustainable from a code maintainability perspective if they conform to modular and extensible architectures, from a community perspective if they successfully attract and retain newcomers; and from an economic perspective if they ensure low total cost of ownership and high added value. Our perspective in this paper is that of *OSS supply chains* [12]: Given the choice, should one depend on some OSS package? Will it be actively maintained in a year or will it show no signs of life?

To develop our hypotheses, we start by reviewing the literature on factors impacting the survival of OSS projects, defined here as the state of being actively maintained. We distinguish between project-level factors, of which having an appropriate supply of contributor effort is arguably most important, and ecosystem-level factors, induced by projects' position in an ecosystem and their relationship with other projects up and downstream. The project-level factors are relatively well studied, therefore we use the literature review to identify relevant control variables in our regression models. The ecosystem-level factors constitute our main contribution. For these we derive, and later test, explicit hypotheses.

Project-level Factors. In order to survive and thrive, OSS projects typically require a steady supply of contributor effort, and projects with more contributors tend to have higher survival rates [139]. But not all contributions are created equal. The communities supporting OSS projects are typically organized in layers, with different roles being recognizable among participants [81, 117]. Usually, project activity is driven by a few *core contributors*, who have commit (*i.e.*, "write") access to the repository and do most of the work. Ascension into the core group is a socio-technical process; earning committer status involves socializing with the core group [47, 57, 151] and demonstrated commitment through repeated, high-quality contributions [36, 158]. The next layer, larger, comprises *external contributors*, who submit occasional patches; on GitHub, these occasional contributions are popular with the pull-based development model [60, 127]. Next, there is typically a layer of *contributing users*, who may participate in discussions

or report issues without contributing code [173]. Finally, the outermost layer consists of *external users* of the software, who do not necessarily participate in any project activities.

Core contributors, or maintainers, are paramount to the survival of OSS projects. They are highly active and have the deepest knowledge of the code base, making them the hardest to replace. Activity in OSS projects typically follows the Pareto principle [58, 114, 169], by which 20% of contributors are responsible for 80% of all activity; to capture this phenomenon, different measures of risk of knowledge loss due to developer turnover have been proposed [136, 154], including the popular "truck factor" [6, 32]. Other contributors and users are also important: future maintainers are frequently groomed or ascend from among external contributors [117]; external contributors also provide much needed testing and quality assurance ("given enough eyeballs, all bugs are shallow" [134]); and without users the software would quickly become obsolete.

When contributions come is also important. OSS projects, as with projects generally, have a life cycle, from inception to abandonment. The motivations for contributing to a project, the amount of effort a project may need, and the chances of attracting contributors will likely vary with the stage in the life cycle. The code growth curve of OSS projects often follows the typical pattern of rapid growth slowing and flattening as projects reach maturity and require less effort for adding features, as shown, *e.g.*, in GNOME [91] and Linux stable releases [152]. Relatively few OSS projects reach maturity [18, 29] and even once-popular projects can get abandoned, *i.e.*, no longer maintained [26, 87]. However, the factors associated with sustained activity can be different in early-stage projects compared to later on [142, 161]. For example, Comino *et al.* [29] found that fewer than 2% of a sample of SourceForge projects reached maturity, and that early-stage projects risked abandonment due to restrictive licenses and smaller communities. In contrast, Coelho and Valente [26] found that common reasons why mature and once-popular OSS projects are abandoned include losing out to a competitor, having become obsolete, and lack of time and interest from contributors.

Ecosystem-level Factors. OSS ecosystems have been an active research topic (for a review, see Franco-Bedoya *et al.* [53]) and different definitions exist [17, 79, 99, 102]. Here we follow Lungu's broad definition [100] of OSS ecosystems as collections of related software projects that co-evolve in the same environment. Python and Javascript packages published on PyPI and npm, respectively, fit this definition: they coexist in the same ecosystem and, as we show below, are often interdependent.

Within an OSS ecosystem, developers frequently contribute to multiple projects [155], often at the same time [157]. In addition to building social capital, bridging sub-communities also creates connections between projects, which can impact project sustainability. For example, Casalnuovo *et al.* [19] found that GitHub contributors are more likely to join projects with which they have prior social connections. Singh *et al.* [143] showed, using a longitudinal panel of 2,378 SourceForge projects, that social network ties between developers impact OSS project survival. Finally, Wang [161] analyzed 2,220 SourceForge projects to model survival factors at various project lifecycle stages. The author found that member social connections with other projects and active engagement of contributors present survival advantages at any stage, while permissive licenses and large contributor bases help especially early on.

A significant missing puzzle piece in prior research on OSS sustainability is the impact of a project's position in dependency networks on survival or dormancy, though the heartbleed and leftpad incidents discussed above suggest this might be significant. Indeed, ecosystem connections between projects extend well beyond the social, with complex dependency networks being formed, *i.e.*, one project reusing functionality from another [1, 11, 41, 88].

We argue that the survival of an OSS project in an ecosystem, in addition to all the factors reviewed above, depends also on the survival of its dependencies upstream (*i.e.*, other packages the current package depends on) and downstream (*i.e.*, packages depending on the current package). Specifically, while

the benefits of depending on others' code in an ecosystem—reusing functionality without assuming the responsibility of maintenance—are clear, we expect that in general having more upstream dependencies may create more points of failure, because of the costs associated with responding to breaking (*i.e.*, backward incompatible) changes. Indeed, while different OSS ecosystem communities have different practices in planning and deploying breaking changes [13], it is clear that in all ecosystems developers must constantly make dependency management decisions. We expect that:

$\mathbf{H}_1$. *The number of upstream dependencies is related to a lower probability of project survival.*

However, depending on an upstream package may also create an incentive for users of downstream packages to step up to contribute changes or to help more generally because doing so benefits them as well. An ill-maintained upstream project could increase maintenance effort downstream. Therefore, the more downstreams a project has, the larger the pool of potential resources an upstream has available. This may help explain why the original issues were resolved within hours for leftpad and days for OpenSSL:

$\mathbf{H}_2$. *The number of downstream dependencies is related to a greater probability of project survival.*

Following from this, we would expect leftpad and OpenSSL to have a large number of downstream dependencies. Interestingly, this was not the case. In both cases, issues in a these libraries caused massive disruption farther down the dependency chain, because widely-used packages with many users depended on them. For OpenSSL these were popular web servers used by many websites, while leftpad was used by Babel, a core JavaScript package. Thus, the number of direct dependents may not fully reflect a package's importance. Indirect connectivity in the package dependency network is also important because of the transitive dependencies aforementioned. Consequently, we posit:

$\mathbf{H}_3$. *Structural properties indicating more indirect connectivity through transitive dependencies are related to a greater probability of survival.*

In contrast, there are upstream practices that can help mitigate these downstream costs. we consider backporting as a concrete evidence of expending effort to support a community of users. If a project makes the extra effort to support backporting, it may indicate the project does not lack developing resource, and hence, have a high probability of survival. Framing this using dependency network, we believe this indicates more intense involvement with other projects in the extended dependency network. Hence, we posit that:

$\mathbf{H}_4$. *Backporting is related to a higher probability of project survival.*

There is also a social organizational perspective to thriving in an ecosystem. Besides differences in roles, OSS contributors are also diverse in terms of background, demographics [156], and employment; they can be a mixture of volunteers, academics, and paid contributors [23, 174], with different motivations to contribute to and maintain OSS [44, 70, 94]. In particular, as ecosystem dependency management costs may become significant over time as the software and its dependencies continue to evolve, we posit that whether a big organization (commercial, non-profit, or even academic) supports an OSS project will affect the project's survival chances, as this level of investment (*e.g.*, assigned employees) is likely beyond that found among volunteers:

$\mathbf{H}_5$. *Projects supported by large organizations have a higher probability of survival.*

2.4 Methodology

To test our hypotheses we designed a mixed-methods study following a *concurrent triangulation strategy*, a common mixed-methods design [49]. We collected both quantitative and qualitative data concurrently and used findings from one source as cross-validation for findings from the other. Quantitatively, we collected a

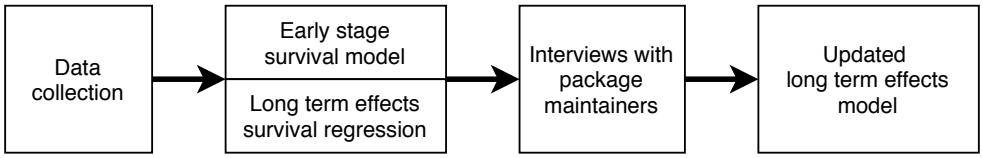

Figure 2.1: The workflow of concurrent triangulation strategy.

panel data set of Python PyPI packages and used survival analysis to model the factors that explain projects becoming dormant. Qualitatively, we interviewed package maintainers to triangulate the model results and refine the discovered effects. Because of a concurrent rather than sequential triangulation strategy, we could revisit and enhance the model to account for potential effects revealed by the interviews, which is a particular strength of this design [34]. We validate our conclusions by replicating the quantitative analysis on the Javascript npm ecosystem.

2.4.1 Data Set

We assembled a large panel data set[5] of OSS packages part of the PyPI (Python Package Index) ecosystem. A distinctive feature of our data set is that it *accurately represents the network of dependencies between member packages* (details below).

We chose Python as it is a popular general purpose language; it is the second most popular on Github by number of pull requests.[6]PyPI is the official registry of Python packages, forming an ecosystem comprising over 130,000 packages (as of March 2018), with declarative-style dependencies. Unlike other languages (*e.g.*, Haskell), Python has only one package repository; as Figure 2.2, based on our data (details below), shows, PyPI is increasingly popular.

We picked Javascript for further validation because Javascript shares several similarities with Python. Both are high-level interpreted programming languages. Javascript is widely-used for web development. It is used by 95.2% of 10 million most popular web pages.[7]and has the largest number contributors on Github.[8] npm is the default package repository for Node.js, the most widely-used framework according to 2018 StackOverflow Developer Survey. [9] Despite npm is released in 5 years later than PyPI in 2010, it hosts more packages than PyPI and is also growing more popular (Figures 2.2, 2.3).

Initial Filtering. Assembling our PyPI dataset involved integrating data from two sources: metadata from the PyPI registry and the packages' development history from their GitHub repositories.

Linking the two required several steps. First, we obtained a list of all PyPI packages using PyPI's JSON API[10] on January 21, 2018, for a total of 125,699 packages, 116,687 of which had at least one release. Next, for each package, we checked if its *home page* field in the PyPI metadata matches any popular code hosting platforms (github.com, bitbucket.org, gitlab.com). If this failed, we extended the lookup to all other metadata fields. If no URL was found, we downloaded the last package release and looked for mentions of code hosting platform URLs across all package files, with the repository name, if any, matching the package name on PyPI.

[5]Available online at https://zenodo.org/record/1419788.
[6]https://octoverse.github.com/, retrieved February 2018
[7]https://w3techs.com/technologies/details/cp-javascript/all/all, retrieved May 2019
[8]https://octoverse.github.com/, retrieved May 2019
[9]https://insights.stackoverflow.com/survey/2018, retrieved May 2019
[10]https://wiki.python.org/moin/PyPIJSON

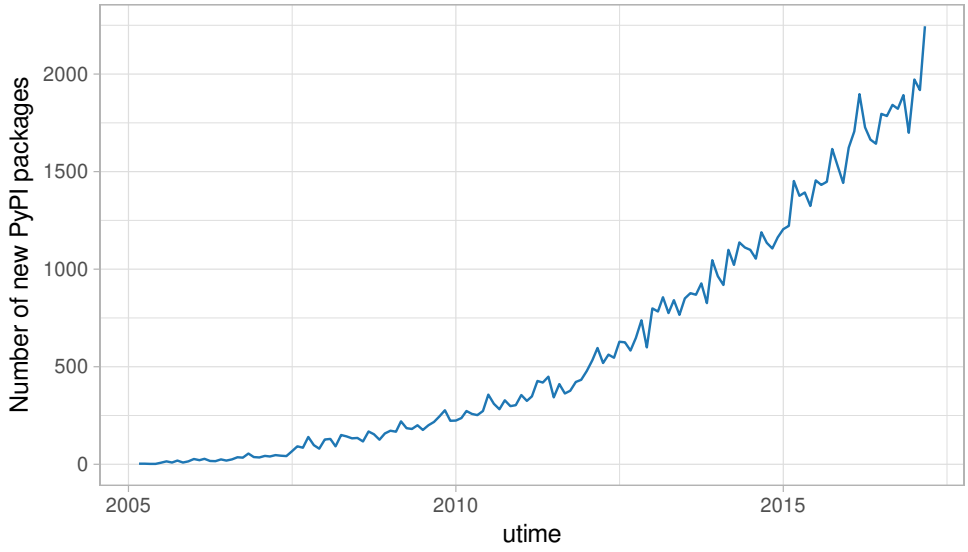

Figure 2.2: Number of new PyPI packages per month.

This approach revealed 91,728 package repositories overall, 89% (or 81,802) of which were hosted on GitHub; for simplicity, we subsequently only mined packages with GitHub repositories. These repositories were checked for existence and uniqueness to filter out project foundries (*i.e.*, repositories hosting hundreds of projects,[11] since these all point to the same PyPI package and the metadata would be impossible to disentangle), leaving 71,903 code repositories (packages) total. We further filtered out packages created before 2012, when GitHub became popular, as the data pre-2012 is sparse, and packages with less than a year of observable history, *i.e.*, created after January 2017, since we could not confidently label them as dormant or still maintained (see §2.4.2). Our final sample contains 46,547packages. For npm, we curated the dataset following the PyPI dataset curation procedure as detailed in the previous paragraph. Our final sample contains 209,895packages.

Dependency Network. By "dependency", we mean a *declared technical dependency on another package*. That is, we do not count required system libraries nor packages copied to the source tree of a package. We also exclude optional extras and test dependencies, as the vast majority of installations do not use them. Given a package, we call "upstreams" those packages used by this package, *i.e.*, packages that the focal package depends on; conversely, we call "downstreams" those packages dependent on the focal one.

To extract dependencies, we mined the package metadata whenever possible, and used a sandbox installation as a fallback. PyPI supports several packaging formats, two of which (.egg and .whl) store machine-readable dependencies. For other formats, *e.g.*, source archives, we executed a package installation in a sandbox environment with an instrumented version of the package installer, logging the requested dependencies. Unlike Python, npm package dependencies are stored in a machine-readable format (JSON) and can be extracted trivially.

To capture network dynamics we extracted dependency information from *all releases of all packages*. Then, we generated historical snapshots of the network, using the latest non-testing, non-backported release at each time. Testing releases were inferred from version numbers, using semantic versioning assumptions (*i.e.*, not matching a pattern of dots and numbers only). Backporting releases are defined as a release with lower version number than the highest non-testing release (*e.g.*, 1.10.0 is a backporting release if 2.0 was

[11] E.g., https://github.com/micropython/micropython-lib, 220 projects

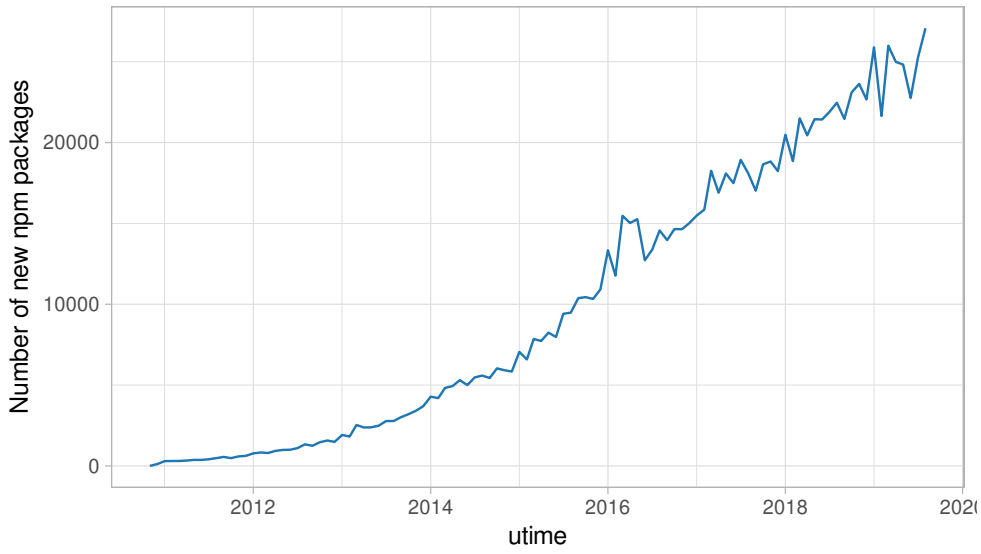

Figure 2.3: Number of new npm packages per month.

released earlier). Note that we extracted dependencies over the whole network, not just for packages with GitHub repositories; this means our network structural measures are robust to the initial filtering above.

2.4.2 Operationalizations of Concepts

In preparation for the quantitative survival analysis below (details in §2.4.3), we introduce the following operationalizations of the different project- and ecosystem-level factors discussed in §2.3 above.

Dormant projects. We consider project as dormant if it is no longer being maintained, *i.e.*, when it stops development (commit) activity, in line with prior work [26, 143, 161]. This suggests a straightforward approach to detect dormant projects: look for a long period of inactivity in the git history, and consider the timestamp of the latest commit as the dormant date. However, this is not always accurate: *e.g.*, there are instances when a seemingly dormant project, with an activity gap of one year or more, is "revived" by (a few) commits to officially indicate that the project has been deprecated; dormant projects may have also had little activity to begin with (thus long gaps are not unusual), changed owner, or were not as much abandoned as they were "completed", *i.e.*, they continue to deliver the intended value without active maintenance; 11% of developers interviewed by Coelho and Valente [26] reported this.

To increase robustness to residual development activity (*i.e.*, fewer false positives), we instead only label a project as dormant if it had less than one commit per month on average in the 12 months prior to its most recent commit. This fits well with our manual inspections of small samples of the data (tens of packages), though we do acknowledge this definition as a potential threat to validity.

Project-level Control Variables. As indicated in our literature review, several factors are known to impact survival rates, and we use the following variables to include these factors in our models:

Project age: the number of months since the first commit in the repository. Note that the earliest Git commit is sometimes dated unreasonably early, *e.g.*, because of a system time reset on a developer machine (dead CMOS battery). A true first commit is one without a parent in the Git history graph. We identified true first commits and filtered out outlying commits dated before their timestamps.

Number of commits: obtained via the GitHub API and aggregated per calendar month; consequently,

Table 2.1: PyPI dataset summary statistics (Dec 2017 snapshot)

Quantile	Min	Mean	0.5	0.7	0.8	0.9	0.95	0.97	0.99	Max
All projects										
Commits	0	1.9	0	0	0	2	8	15	42	1144
Core team	0	0.07	0	0	0	0	0	1	2	97
Non_dev_issues	0	0.2	0	0	0	0	1	1	4	245
Upstreams	0	1.7	1	2	3	4	6	8	14	117
Downstreams	0	1.5	0	0	0	1	2	4	15	5487
Katz centrality	0	3e-4	4e-4	4e-4	4e-4	5e-4	5e-4	6e-4	1e-3	0.33
Social ties	0	1.0	0	0	0	0	1	3	12	240
University	0	0.01	0	0	0	0	0	0	0.14	1
Commercial	0	0.03	0	0	0	0	0	0.5	1	1
Non-dormant projects										
Commits	0	11.6	4	8	14	27	48	66	125	1144
Core team	0	0.42	0	0	1	1	2	3	4	97
Non_dev_issues	0	0.9	0	0	1	2	4	6	14	245
Upstreams	0	2.2	0	2	4	6	9	12	20	100
Downstreams	0	5.3	0	0	0	2	5	12	67	5487
Katz centrality	0	5e-4	4e-4	4e-4	4e-4	5e-4	6e-4	1e-3	4e-3	0.33
Social ties	0	6.2	0	1	3	6	15	32	211	240
University	0	0.05	0	0	0	0	0.375	1	1	1
Commercial	0	0.18	0	0	0.33	1	1	1	1	1

since the first month may be incomplete, we exclude it from further analysis.

Number of contributors: counted as the number of GitHub users having authored commits within a given calendar month.

Size of the core team: the number of people responsible for 90% of contributions in a given month. This threshold was selected empirically as a typical "elbow" point in distribution of OSS activity, much like in other OSS projects (*e.g.*, Apache [114]).

Number of issues: obtained via the GitHub API, with pull requests filtered out, aggregated per calendar month. We distinguish between developer-reported issues, likely occurring internally during development, and non-developer-reported issues, likely reported by external users, as the latter are more indicative of the size of the user base; we call "developers" those contributors who authored prior commits and "non-developers" the rest.

Number of non-developer issue reporters: the number of non-developer GitHub users reporting issues in a given calendar month; may help distinguish communities with higher user engagement from those where few users do most issue reporting.

License type: extracted from package metadata; categorical variable, indicating whether a project is distributed under a strong copy-left license (GPL, Affero etc.), weak copy-left (LGPL, MPL, OPL, etc.) or non-copy-left license (Apache, BSD etc.), cf. [161].

Social ties: the total number of packages that contributors to the focal package also contributed to this month, as a proxy for the amount of OSS embeddedness of the contributors. Projects with more "seasoned" contributors may be more sustainable.

Table 2.2: npm dataset summary statistics (Jan 2018 snapshot)

Quantile	Min	Mean	0.5	0.7	0.8	0.9	0.95	0.97	0.99	Max
All projects										
Commits	0	0.44	0	0	0	0	0	3	11	1416
Core team	0	0.02	0	0	0	0	0	0	0	185
Non_dev_issues	0	0.04	0	0	0	0	0	0	1	503
Upstreams	0	2.67	1	3	4	6	10	13	22	787
Downstreams	0	1.4	0	0	1	2	3	6	20	6037
Katz centrality	0	1e-4	1e-4	1e-4	1e-4	1e-4	1e-4	2e-4	3e-4	0.07
Social ties	0	2.47	0	0	0	0	0	0	6	939
University	0	5e-4	0	0	0	0	0	0	0	1
Commercial	0	0.008	0	0	0	0	0	0	0.13	1
Non-dormant projects										
Commits	0	9.0	4	8	11	19	32	42	78	1416
Core team	0	0.33	0	0	0	1	1	2	3	185
Non_dev_issues	0	0.48	0	0	0	1	2	3	9	503
Upstreams	0	3.53	1	3	5	9	15	21	38	185
Downstreams	0	2.26	0	0	1	2	5	9	31	2021
Katz centrality	0	1e-4	1e-4	1e-4	1e-4	1e-4	1e-4	2e-4	5e-4	0.03
Social ties	0	50.3	1	3	6	36	305	816	819	939
University	0	0.01	0	0	0	0	0	0	0.33	1
Commercial	0	0.16	0	0	0.17	1	1	1	1	1

Variables for Ecosystem-level Hypotheses.

Number of upstreams (out-degree centrality): number of upstream dependencies used by the project (**H1**).

Dormant upstreams: binary variable indicating whether any of the upstream dependencies is itself dormant at the time (**H1**).

Number of downstreams (in-degree centrality): number of projects directly dependent on the focal project (**H2**). Both here and in the upstreams case, we do not differentiate between versions of a dependency, so even dependencies on old and unsupported versions of the package are counted. This is because more potential contributors dependent on this package and having a vested interest in its sustainability may positively affect survival.

Katz centrality: as a proxy for the importance of a package's position in the network structure (**H3**). Katz centrality [86] for node i is defined as: $C_{Katz}(i) = \alpha \sum_{j \in J} C_{Katz}(j) + \beta$, where J is the set of adjacent nodes to i. Parameter β is an initial centrality (usually 1), and α is a discriminating factor applied at each step to down-weigh farther nodes (0.1 in this study). Developers further downstream in the dependency chain may be less likely to step in if needed than people from immediately dependent projects. We use Katz centrality to capture this and down-weigh farther downstream projects depending on their dependencies. In the rare case of circular dependency, we randomly break the cycle.

Backporting: binary variable indicating whether the project produced a backporting release in the last 12 months (**H4**).

Organizational account: binary variable indicating that the project is hosted under an organizational (rather than personal) GitHub account; organizations, even informal, may possess more resources than an

individual developer, affecting survival differently (**H5**).

University involvement: share of commits where the top-level domain of the author's email is a university domain.[12] During modeling, we binarized this variable by using 10% as a cut-off point: a project with more than 10% of commits contributed from university email domains is considered as having high university involvement and, vice versa. This is a conservative operationalization, as the list may be incomplete and not all academically affiliated contributors configure their Git clients with their institutional emails. Predominantly academic projects may be subject to specific survival risks, *e.g.*, student graduation, funding cycles, and shifting research interests (**H5**).

Commecrial involvement: the share of commits from non-university, non-public, non-personal email domains. Public email providers (*e.g.*, gmail.com) were excluded based on a public list.[13] During modeling, we binarized this variable by using 10% as a cut-off point: a project with more than 10% of commits contributed from organizational email domains is considered as having high organization involvement and, vice versa. Personal domains are defined as those with only one known user across the entire ecosystem data. We manually validated the top-100 domains (by number of emails) labeled organizational and found no obvious mislabelling. Commercial companies or open-source foundations, both of which are "organizational", may act as a driving force and supply resources, *e.g.*, their employees' time (**H5**).

2.4.3 Survival Analysis (Quantitative)

We use survival analysis to model the effects of the different factors above on packages becoming dormant. Survival analysis, also known as event history analysis, is a branch of statistics that specializes in modeling of time to event data [110]. Typically only a single event occurs for each subject; in our case, the event is the package suspending its development activity. Survival analysis techniques are designed to deal with so-called right-censored observations: the time of the occurrence of the event of interest can only be recorded reliably for members of the population that already experienced the event; for others, all we know for certain is that the event hasn't happened yet; for some, it may never happen (hence the term right-censorship). In software engineering, survival analysis has been used to model, *e.g.*, defect survival in Eclipse (time to bug fixes) [163] and contributor survival in OSS projects [30, 98, 121].

Cox Proportional-Hazards Model. Different survival analysis techniques exist. The most common regression modeling framework for survival analysis is the Cox proportional-hazards model [33], which allows to estimate the effect of any one independent variable on the outcome, *while holding other covariates fixed*. This allows us to precisely isolate the effects of any given factor on survival.

In a general case, one may be interested in modeling state transitions in some system. Say we have a number of observation of some system, entering (*e.g.*, birth) and leaving (*e.g.*, death) a state of interest. For each alive subject, we thus have a *survival time* T on record. The probability of reaching a given survival time t will be defined by the *survival function* $S(t) = P(T > t)$. The probability of leaving the state at time t will be given by *hazard rate* $h(t) = \frac{P(T<t+\Delta t|T\geq t)}{\Delta t}$. Given enough data, one can build a non-parametric regression to estimate all these functions.

Our goal, however, is to estimate the effect of some independent variables X on the hazard rate: $h(t, X) = \theta(t)f(X)$. The problem in this case is that the baseline hazard rate $\theta(t)$ is non-parametric and thus does not have a functional equivalent. Cox's proportional hazard model allows to estimate coefficients of the regression $h(t, X) = \theta(t)\exp(\beta^T X)$ using partial likelihood, without any assumptions about the baseline hazard rate [80, 110].

[12] As per a public list https://github.com/Hipo/university-domains-list

[13] https://gist.github.com/tbrianjones/5992856/

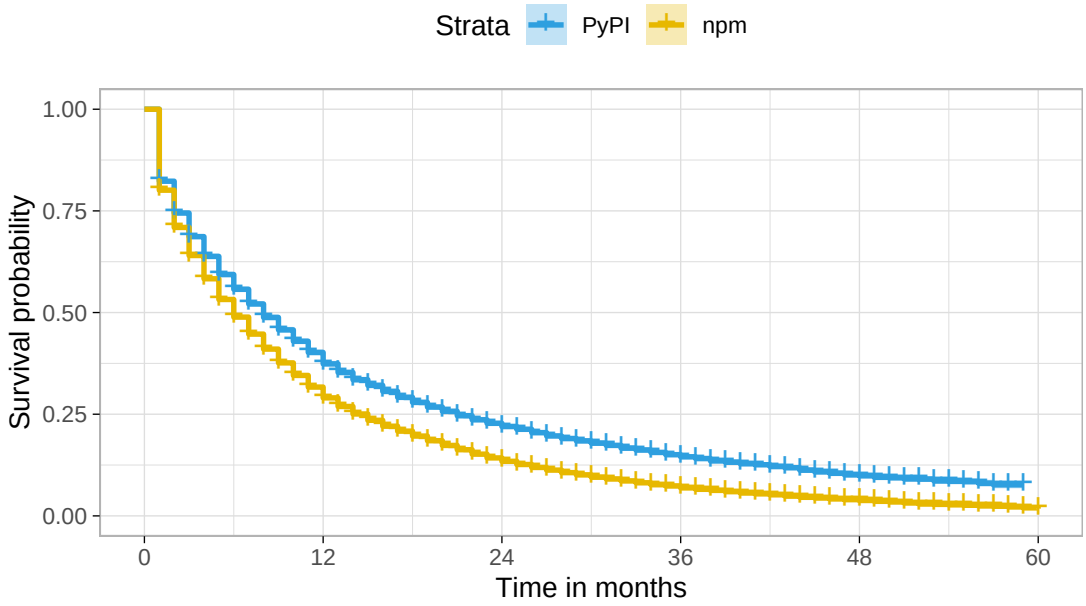

Figure 2.4: Overall probability of packages survival in PyPI and npm.

A nice property of this model is that one can directly interpret the coefficients β. For example, if $\beta_i = 2$, then every unit increase in X_i will increase the probability of death by $\exp(2) = 7.4$ times.

Modeling Considerations. Recall that our data set is longitudinal, organized in monthly windows. Measures derived from OSS data (*e.g.*, number of commits) tend to be quite noisy, with high variation from one observation (*e.g.*, month) to the next. To increase the robustness of our models to potentially high window-to-window variance, we first smoothed out all numerical variables using a six-months sliding window, where each value was replaced by the average of the previous six.[14] Then, we split project data into six-month periods, taking only the last observation from each period.

As expected (§2.3), we also observed during initial exploration of our data that many packages become dormant early; see Figure 2.4 which shows the probability of survival (*i.e.*, staying non-dormant) over time, across all PyPI and npm packages in our sample. To model how the different factors contribute to explaining the variability in package survival rates differently early-stage compared to later on, we split the data set into two parts: early-stage shutdowns (*i.e.*, stopping or nearly stopping development in the first six months, which matches well our sliding-window smoothing approach) and the rest. Similar trends may be observed for npm packages, as demonstrated in Figure 2.4.

The early-stage shutdowns, by definition given our smoothing, only contribute one observation each, while the other packages contribute more. Therefore, we model this group using logistic regression (`glm` in R), with the response variable being the likelihood of a project becoming dormant. For the remaining packages, the data contains monthly observations. A package's `dormant` variable is `True` in the last month, if we labeled the package as having stopped activity (see above), and `False` otherwise; surviving packages have, therefore, `dormant` = `False` in all windows. To model these, we estimate a Cox proportional-hazards model (`coxph` in R).

In both cases, we follow a similar procedure for model fit and diagnostics. First, for predictors with highly skewed distributions, we conservatively removed the top 1% of values as high-leverage outliers,

[14] Or fewer, in the first five months of observation.

in line with statistical best practice [124]; high-leverage points would disproportionally affect regression slopes and reduce the model's robustness. Second, we log-transformed variables with skewed distributions, as needed, to reduce heteroscedasticity [55]; this helps stabilize the variance and can improve model fit. Third, we test for multicollinearity between the explanatory variables using the variance inflation factor (VIF), and remove variables, if any, with VIF scores above the recommended maximum of 5 [28]. We also performed graphical diagnostics: deviance residual plots for the logistic regression and Schoenfeld residuals [62] for the Cox model (which test the assumption of constant hazard ratios over time); none displayed any obvious signs of violations. In the Cox model, to account for the non-independence of repeated observations per package, we explored different options, all of which produced qualitatively similar results: transforming the data into count process format [147]; or using a cluster term for package.

When interpreting the models, we consider coefficients important if they are statistically significant at 0.05 level or lower, and we estimate their effect sizes from ANOVA type-2 analyses (column "LR Chisq" in Table 2.3). For the logistic model we also report McFadden's pseudo R^2 measure of goodness of fit.

2.4.4 PyPI Maintainer Interviews (Qualitative)

To triangulate and enhance our modeling results we conducted 10 semistructured interviews with PyPI package maintainers. Recruitment was done by soliciting via email, using addresses collected from GitHub profiles or personal websites. We used stratified sampling to identify potential interviewees: 3 packages with extreme feature values (large size), 2 projects that recently became inactive, 4 randomly selected to stratify the sample by project size, and 1 highly productive individual contributing to many projects of different sizes. For each project, the most active person in the last two years was solicited via email. We sent 32 emails, received 12 responses, and conducted 10 interviews; we reached theoretical saturation, meaning roughly that subsequent data all fit within the categories derived from the previous interviews, around the sixth interview.

The interview protocol was centered around the model features, asking about the predictive power of these features, their expected effect, and comments on the effects discovered by the model. Interviews also included several open ended questions about the definition of sustainability in OSS, project context, and missing factors that should be incorporated into the model.

Interviews were recorded, transcribed, translated (two cases), and coded. For three interviews, interview recordings were partially or completely corrupted due to technical glitches. Transcripts for these interviews were restored from partial recordings and notes, confirming accuracy with participants when necessary.

A lightweight qualitative coding was made by one author and discussed with the others. Codes were designed to match model features, their possible explanations, and threats to validity.

2.5 Case Study I: PyPI Results (Mixed-Methods)

We present an integrated discussion of quantitative and qualitative results, combining the survival analysis with interview insights. Table 2.3 presents the regression results. The first two models, logistic regression for projects becoming dormant in the early-stage (first six months), and Cox proportional-hazards for those becoming dormant later on, comprise the factors we reviewed or hypothesized in §2.3. The third model extends the Cox proportional-hazards model, to test for potential interaction effects emerging during our qualitative analysis. Logistic regression coefficients are odds ratios. Cox model coefficients are hazard ratios; a hazard ratio above 1 indicates a covariate that is positively associated with the event probability, and thus negatively associated with the length of survival.

Table 2.3: Regression models for early-stage survival and later-stage survival in PyPI.

	Early-stage survival response: *dormant = TRUE* Pseudo $R^2 = 43.6\%$		**Later-stage survival** response: *dormant = TRUE* $R^2 = 17\%$		$R^2 = 17.2\%$	
	Coeffs (Err.)	LR Chisq	Coeffs (Err.)	LR Chisq	Coeffs (Err.)	LR Chisq
(Intercept)	$4.04\ (0.05)^{***}$					
Log number of commits	$0.77\ (0.02)^{***}$	7675.27^{***}	$1.77\ (0.01)^{***}$	3317.06^{***}	$1.84\ (0.01)^{***}$	3326.17^{***}
Log number of contributors	$0.87\ (0.08)^{***}$	8276.50^{***}	$0.19\ (0.05)^{***}$	1374.33^{***}	$0.19\ (0.05)^{***}$	1372.20^{***}
Log number of non-developer issues	$1.01\ (0.07)^{***}$	79.22^{***}	$0.55\ (0.04)^{***}$	222.92^{***}	$1.07\ (0.11)$	222.93^{***}
Social ties	$1.07\ (0.03)^{***}$	16.42^{***}	$1.09\ (0.02)^{***}$	17.58^{***}	$1.09\ (0.02)^{***}$	18.30^{***}
Number of downstream projects	$0.67\ (0.04)^{***}$	178.37^{***}	$0.89\ (0.02)^{***}$	68.05^{***}	$0.89\ (0.02)^{***}$	68.08^{***}
Number of upstream dependencies	$0.91\ (0.01)^{***}$	380.27^{***}	$0.95\ (0.01)^{***}$	68.25^{***}	$0.95\ (0.01)^{***}$	19.05^{***}
Some upstreams are dormant	$1.63\ (0.05)^{***}$	50.39^{***}	$1.11\ (0.03)^{***}$	13.23^{***}	$1.11\ (0.03)^{***}$	13.00^{***}
Katz centrality	$1.72\ (0.02)^{***}$	35.29^{***}	$1.27\ (0.02)^{***}$	221.55^{***}	$1.26\ (0.02)^{***}$	171.36^{***}
High university involvement (>10%)	$0.65\ (0.06)$	1.32	$0.75\ (0.05)^{***}$	30.88^{***}	$0.76\ (0.05)^{***}$	30.35^{***}
High commercial involvement (>10%)	$0.97\ (0.04)^{***}$	125.22^{***}	$1.15\ (0.03)^{***}$	24.51^{***}	$1.15\ (0.03)^{***}$	23.47^{***}
Had a backporting release in the last 12 months	$1.49\ (0.12)^{***}$	16.63^{***}	$0.97\ (0.07)$	0.21	$0.97\ (0.07)$	0.17
Strong copy-left license (vs none)	$0.7\ (0.06)^{***}$	78.42^{***}	$0.83\ (0.04)^{***}$	35.83^{***}	$0.84\ (0.04)^{***}$	34.54^{***}
Weak copy-left license (vs none)	$0.78\ (0.10)^{***}$		$0.75\ (0.07)^{***}$		$0.75\ (0.07)^{***}$	
Non-copy-left license (vs none)	$0.84\ (0.04)^{***}$		$0.98\ (0.03)$		$0.98\ (0.03)$	
Hosted under organizational account on GitHub	$0.48\ (0.04)^{***}$	259.95^{***}	$0.77\ (0.03)^{***}$	84.62^{***}	$0.78\ (0.03)^{***}$	79.91^{***}
Number of upstream dependencies, squared					$1.00\ (0.00)$	0.08
Log num. commits × log num. issues					$0.81\ (0.03)^{***}$	43.60^{***}
Log num. issues × log num. contributors					$0.91\ (0.11)$	0.75

$^{***}p < 0.001$, $^{**}p < 0.01$, $^{*}p < 0.05$

2.5.1 Survival Models and Interview Insights

Project-level Effects. The **number of commits** is associated with higher chances to become dormant in the next time period in both groups (early- and late-stage), *i.e.*, packages with higher commit activity are more likely to become dormant, other variables held constant. Interviewees pointed out that commit squashing (merging large contributions into a single commit), which would reduce the number of commits, may help explain this effect: mature projects may use this practice more often, but are less likely to become dormant; at the same time, a contributor with direct commit access can contribute many small commits, which often happens in smaller projects, which are more likely to become dormant. Three interviewees indicated using recency of commits as the main indicator of sustainability. Future work should consider counting commit message lines instead (squashed commits tend to contain all original messages), and replacing the absolute number with commit dynamics, such as stability of monthly contributions. Two participants also suggested adjusting the number of commits to the project size, since bigger projects may need more maintenance.

The **number of contributors** has a negative effect on chances of becoming dormant in both groups, *i.e.*, packages with more contributors are less likely to become dormant. Six interviewees agreed unconditionally that having more contributors improves sustainability. Explanations included, in decreasing order of popularity: larger recruitment pool for the core team, better code reviews, and an indication of healthy onboarding practices. An important addition to the number of committers as a sustainability metric, pointed out by three interviewees (all are maintainers of big projects) is that it does not capture non-code contributions. For example, people contributing code reviews, issue triaging, and even evangelism and securing funding are essential for project sustainability.

The **size of the core team** is collinear with the number of contributors in our models, hence not included. Our interviewees also perceive it similarly. It was unanimously viewed as a positive factor by maintainers of big projects. All explanations of the effect either directly referred to the "bus factor" or closely resembled its definition. For small projects, this metric was either equivalent to the total number of contributors and still considered positive, or did not apply because the project was considered feature-complete.

The **number of issue reporters** was not included in our models due to multicollinearity. For our interviewees, this was perceived as another way to measure user base.

Table 2.4: Npm regression models for early-stage survival and later-stage survival.

	Early-stage survival response: *dormant* = *TRUE* Pseudo $R^2 = 45.3\%$		**Later-stage survival** response: *dormant* = *TRUE* $R^2 = 18.3\%$		$R^2 = 18.5\%$	
	Coeffs (Err.)	LR Chisq	Coeffs (Err.)	LR Chisq	Coeffs (Err.)	LR Chisq
(Intercept)	$1.63\ (0.05)^{***}$					
Log number of commits	$0.77\ (0.01)^{***}$	29724.62^{***}	$1.78\ (0.01)^{***}$	10520.18^{***}	$1.82\ (0.01)^{***}$	10572.79^{***}
Log number of contributors	$5.14\ (0.04)^{***}$	40999.01^{***}	$0.24\ (0.02)^{***}$	5072.14^{***}	$0.25\ (0.02)^{***}$	4950.71^{***}
Log number of non-developer issues	$0.23\ (0.04)^{*}$	5.26^{*}	$0.68\ (0.02)^{***}$	475.26^{***}	$1.06\ (0.05)$	488.01^{***}
Social ties	$0.96\ (0.01)$	1.87	$1.07\ (0.00)^{***}$	202.65^{***}	$1.07\ (0.00)^{***}$	193.03^{***}
Number of downstream projects	$0.88\ (0.01)^{***}$	961.38^{***}	$0.99\ (0.00)^{***}$	34.56^{***}	$0.98\ (0.00)^{***}$	41.89^{***}
Number of upstream dependencies	$0.94\ (0.00)^{***}$	751.51^{***}	$0.99\ (0.00)^{***}$	33.78^{***}	$1.01\ (0.00)^{**}$	8.78^{**}
Some upstreams are dormant	$1.09\ (0.01)^{***}$	201.21^{***}	$1.03\ (0.01)^{***}$	16.17^{***}	$1.04\ (0.01)^{***}$	18.12^{***}
Katz centrality	$1.71\ (0.01)^{***}$	746.00^{***}	$1.08\ (0.01)^{***}$	144.92^{***}	$1.07\ (0.01)^{***}$	118.98^{***}
High university involvement (>10%)	$0.86\ (0.08)$	0.64	$0.90\ (0.06)$	3.89^{*}	$0.90\ (0.06)$	3.49
High commercial involvement (>10%)	$0.79\ (0.02)^{***}$	340.20^{***}	$1.05\ (0.02)^{**}$	10.61^{**}	$1.05\ (0.02)^{**}$	9.93^{**}
Had backporting release	$0.9\ (0.08)^{***}$	18.78^{***}	$1.03\ (0.04)$	0.38	$1.02\ (0.04)$	0.29
Strong copy-left license(vs none)	$0.71\ (0.06)^{***}$	571.49^{***}	$0.72\ (0.05)^{***}$	124.22^{***}	$0.72\ (0.05)^{***}$	125.32^{***}
Weak copy-left license(vs none)	$0.72\ (0.06)^{***}$		$0.69\ (0.05)^{***}$		$0.69\ (0.05)^{***}$	
Non-copy-left license(vs none)	$0.82\ (0.02)^{***}$		$0.86\ (0.02)^{***}$		$0.85\ (0.02)^{***}$	
Hosted under org account	$0.5\ (0.02)^{***}$	440.84^{***}	$0.88\ (0.01)^{***}$	83.12^{***}	$0.88\ (0.01)^{***}$	82.18^{***}
Num upstream dependencies, squared					$1.00\ (0.00)^{***}$	57.77^{***}
Log num. commits × log num. issues					$0.89\ (0.02)^{***}$	52.47^{***}
Log num. issues × log num. contributors					$0.83\ (0.05)^{***}$	13.05^{***}

$^{***}p < 0.001$, $^{**}p < 0.01$, $^{*}p < 0.05$

The **number of issues** was expected to be a positive indicator of user engagement. However, it was estimated as a risk factor in the first six months (*i.e.*, the early-stage model), while still decreasing the chances of becoming dormant in the later-stage survival model. In the interviews, the discussion revealed several layers of interpretation for this metric. First, reported issues were unanimously considered to be a positive sign of an active user community. Two small project maintainers also noted that no issues in a small project could be an indicator of a project without quality issues rather than low usage. Four participants suggested looking into issue handling and discussions. Average response time, number of responses, and number of closed issues were proposed as indicators of activeness for the project team.

Another interpretation for issues, coming from three maintainers of big projects, suggested that issue triaging, although helpful for end users, takes resources and sometimes slows down development activities. Some projects were known to stop responding to issues completely to conserve developers' effort.[15] Based on this discussion, in the third model we introduce an *interaction between the number of commits and number of contributors*. The interaction effect was significant, rendering number of issues as a positive factor in projects with high volume of commits. Controlling for this interaction, the number of reported issues itself is not significant.

H1. Upstreams. Upstream dependencies were expected to increase the chances of projects becoming dormant. The modeling effects are nuanced. In the first six months, a higher number of upstreams correlates with higher chances of dormancy, as hypothesized, although the presence of a *dormant* upstream reduces this risk. Later on, a higher number of upstreams correlates with lower chances of dormancy, but the presence of a *dormant* upstream increases this risk.

The interviews with utility library project maintainers offer a potential explanation. Such projects are considered dormant per our definition, while in fact they are feature complete. Reusing feature complete libraries can boost development of a new project, hence the reduced dormancy risk in the first six months. In the long-term, however, projects may start to incur higher costs of maintaining compatibility with dormant upstreams, hence the increased risk.

At the same time, having more upstreams overall in the long term enables more reuse, compensating for increased dormancy risks. Still, interviewees were cautious, stressing that it is better to limit upstream

[15]E.g., the npm CLI team: https://twitter.com/maybekatz/status/953402549293350913

dependencies in the long-term to those that are really necessary, as there is a trade-off between development effort (lower with more reuse) and potential compatibility issues later on. The mention of "as few as possible within reason" suggested a non-linear effect, which we added to the third model as a *quadratic term of the number of upstreams*; however, this term did not have a statistically significant effect. We further illustrate this trade-off.

On the one hand, the positive effect of dependency adoption comes from saving resources on implementation. A striking example comes from an interview with a project maintainer who was able to reuse a domain-specific library from a similar project. The maintainer claimed that most projects in this domain die in an attempt to implement this very expensive piece of functionality, so adoption of this dependency was essential for the project success.

On the other hand, the compatibility issues come from the Python package installer (pip) not having a version resolver. For example, if package A requires B version 1.0 and C version 1.0, and B 1.0 requires C 2.0, after installing package A a user might end up with an incompatible setup B 1.0 and C 1.0. The Python community developed tools to build isolated, non-contradicting sets of dependent packages (*e.g.*, virtualenv and pipenv), but even compatibility within these environments requires effort from package developers.

All interviewees seem well aware of this trade-off and reported using different heuristics to find the right balance. Several maintainers, especially of larger packages, indicated that they have to spend substantial effort to stay compatible with a wide range of environments by supporting outdated versions of upstream packages. In case of smaller upstreams, it is often considered a lesser evil to either reimplement a dependency or copy a compatible version under the package source tree.

In summary, an indication of compatibility with potential upstreams is the biggest factor in dependency adoption, but overall the evidence for or against **H1** is inconclusive.

H2. Downstreams. Downstream dependencies were expected to have a positive effect on project sustainability. Our models indicate that indeed they have a positive effect in the long term, but not in the first six months. Building a community of downstream projects takes time. The most likely scenario for a project to have downstream dependencies early in their lifecycle is to be chipped off from a bigger project into a small utility library, used by other projects of the same maintainer. Such projects are usually limited in scope, do not require further maintenance, and thus will be considered dormant. In the later stage, however, this metric is dominated by "natural" downstream dependencies, working as a positive survival factor in the later-stage survival models.

Across all interviews, downstream dependencies were characterized as a mostly positive factor, with the main trade-off between extra maintenance effort and resources brought by the dependent projects. Main benefits brought by downstream projects were described as code contributions (three participants), free testers (two), secure funding (two participants from academic projects). Two participants considered the number of dependent projects a proxy for user base, and two maintainers of feature-complete projects stated that it is not important at all.

The negative side of downstream dependencies was described as extra effort to maintain compatibility and triage issues. Only one participant stated that contributions from downstream projects are not worth this effort. These explanations, together with modeling results, mostly support **H2**.

H3. Structural properties. Relative position in the dependency network was only relevant for four projects in the interview pool having transitive downstream dependencies, and even for those the dependency network was not directly observable. Due to these constraints the role of the relative position in the dependency network was mostly discussed from a theoretical perspective rather than personal experience.

In most cases, transitive downstreams were interpreted in the same way as direct ones, as a user base. Two participants stated that projects higher in the dependency chain need to put more resources into sustainability because of their special position. One of them motivated this with an example of "extremely

painful" debugging of a second-level upstream dependency (*i.e.*, an upstream of an upstream). For two maintainers of feature-complete projects the relative position in the dependency network did not matter. Limitations of the interview sample prevent us from building a more robust qualitative interpretation of this metric.

However, in all three models higher Katz centrality correlates with increased chances of projects becoming dormant. This effect could be possibly explained by a higher reuse rate of feature complete libraries. It could also be attributed to an increased maintenance effort required in projects higher in the dependency chain, as indicated by interviewees. Overall, based on the modeling and interview results, we could not confirm **H3**.

H4. Backporting. Backporting was used as an indicator of project practices aimed at reducing the maintenance cost of dependent projects [13]. It was estimated to substantially reduce chances of becoming dormant in the first six months, where practical importance of backporting releases is questionable. The model estimate, suggesting increased likelihood of survival, could be explained by projects occasionally mislabeling releases; however, the fact that these projects produce multiple releases in the first six months is serving as an indicator of sustainability. In subsequent later-stage survival models this feature did not have a significant effect. We could not confirm **H4**.

H5. Organizational Support. University involvement has a special role in the Python community. Many signature Python projects are related to traditionally academic domains: Data Science, Machine Learning, Artificial Intelligence, Natural Language Processing, etc. Our expectation was that university projects have extra risks, such as students leaving after graduation, end of funding cycles, etc. However, modeling results indicated that university involvement is not a significant dormancy risk factor in the first six months, but in later-stage survival models it reduces the chance of becoming dormant by approximately 25%.

In our interview sample, four participants were university affiliates. Overall, all four indicated that their OSS work is currently funded through a university through a grant or contract, and their contributions are related to their position in academia. Two of them started their projects as students, one joined an existing project, and the last project was created as a practical tool to support existing research. Two interviewees transitioned into faculty positions during their projects, and thus had an extended perspective on the evolution of an academic project. They commented on three survival challenges in the lifecycle of a university project: surviving the student graduation cycle, surviving the academic funding cycle, and growing outside academia. Two out of four university affiliates we interviewed assumed that the diversity of institutions involved might also be used as an indicator of sustainability, where projects with multiple institutions involved are expected to be truly owned by the research community, in contrast to local research projects. It was also suggested that the university involvement effect might vary depending on the area of science.

The explanation above and the modeling results partially support **H5**. However, increased sustainability of university projects might be specific to the Python ecosystem due to its high share of "academic" projects and should be further tested in different ecosystems by future work.

Commercial involvement was expected to have a positive effect on sustainability (lower risk of dormancy), but the models suggest otherwise; the feature also elicited somewhat controversial interview explanations. A shared opinion about commercial involvement among interviewees was that companies bring resources to the project, but this support is not sustainable long term. Common concerns include misalignment of companies' priories with the project goals and sustainability issues caused by companies withdrawing from a project.

The overall extent of commercial involvement in PyPI seems small. Contrasting the results of the 2016 Future of Open Source Survey, which states that "1 in 3 companies have a full-time resource dedicated to open-source projects" and "67% of companies actively encourage developers to engage in and contribute

to open-source projects",[16] only two participants knew about cases of companies paying developers to contribute to OSS projects of their choice. One participant also indicated that their project benefits from commercial contributions, but "... those engineers will have a finite time with us. So we can't put them on the critical path". This explanation, coupled with modeling results, is at odds with **H5**.

Another aspect to commercial contributions is licensing. One participant stated that commercial companies are sensitive to licensing terms. In particular, many product companies will not be able to work with GPL products, though service companies might.

Adding **license restrictiveness** as a control variable in our models, we find that presence of a license, whether strong-, weak- or non-copy-left, is a positive survival indicator. Although one interview participant indicated that strong copy-left licenses, such as GPL, restrict project adoption, the model indicates that strong- and weak copy-left licenses have higher positive impact on project chances of survival than non-copy-left licenses or no license.

Hosting under an organizational account on GitHub has a substantial positive effect. Otherwise equal, such projects are 22% less likely to be become dormant, which partially supports **H5**.

2.5.2 Other Indicators of Sustainability

During the interviews other indicators of sustainability emerged. For example, competition was listed as a major driving force behind one project. Users in this domain can easily switch between projects, so this project had to implement new features added by their competitors. Such competition increases the required maintenance effort to stay up to date with the user needs.

Across many interviews, participants indicated that maturity of project practices plays an important role in an evaluation of the project's sustainability from the end user perspective. Prior work by Trockman *et al.* [149] also found that developers rely on observable signals when making decisions about which project to use or contribute to. The factors mentioned during our interviews were related to software quality, backward compatibility, developer onboarding, and support for end users. The indicators of such practices include implemented autotests, CI and test runner configurations, documentation, number of pull requests, GITHUB stars, project website, etc. Three participants mentioned the number of downloads as an indicator of an active user community, although noisy ("even one order of magnitude doesn't tell you very much").

2.6 case study II: npm (quantitative)

Using the same models as for PyPI packages, we conducted survival analysis of npm packages. The results of this analysis, both for early stage and long-term survival, are consistent with PyPI, with only few minor differences. The estimated effects are presented in Table **??**.

Overall, the baseline survival of packages in npm is somewhat lower than in PyPI (Figure 2.4). An average npm package has smaller core team, fewer core contributors and reported issues per month than its PyPI counterpart (Tables 2.1, 2.2). At the same time, npm packages have more upstream dependencies on average, which goes in line with previous observations [**?**].

The direction of external effects on project survival in npm is consisten with the one obseved in PyPI, with only one exception. The long term effect the number of upstream projects, while increasing chances of survival by 5% in PyPI per unit increase, in npm had the reverse effect, decreasing chances of survival by 1%. Given the differences in upstream dependency statistics, this change, yet having the smallest effect of all external factors, might be caused by underlying dependency adoption practices.

[16] https://www.blackducksoftware.com/2016-future-of-open-source, slide 26

Other external factors, while demonstrating the same effect in both ecosystems, were somewhat smaller in size. For example, presence of downstream dependencies increased chances of survival by 33% in the early stage and 11% later on in PyPI, but only by 12% and 2% in npm, respectively. In other cases, the extent of observed effects was fairly close, *e.g.*, projects hosted under an organizational account had 52% higher chances of survival in the early stage and 23% later on in PyPI, close to 50% and 12%, which we observed in npm.

2.7 Implications

Our study provides a quantitative way (supported by qualitative insights) to identify and predict which OSS projects may become dormant and therefore pose a risk to developers choosing dependencies. Our survival models show that in addition to known project-level factors impacting sustainability, such as the number of contributors and the number of users, a project's chances of becoming dormant (having limited activity) is influenced by a series of ecosystem-level factors, such as its position in the ecosystem dependency network. These results have several implications.

First, these results may be *actionable for OSS researchers and platform designers*. The ecosystem-level variables we found to correlate with a project's risk of becoming dormant, despite being aggregations of publicly accessible data, are not readily observable on platforms like GITHUB. One of the defining features of GITHUB is transparency [37]; developers rely on *signals* [149] displayed on GITHUB repository and user pages (*e.g.*, counts of stars and followers, repository badges) to form impressions about each other and their work [105]. Our approach shows how new signals to display these otherwise unobservable ecosystem-level qualities, such as a project's position in the ecosystem interdependency network or its level of organizational support, could be developed.

In turn, displaying these signals may help developers identify sustainable projects and projects at risk, steer developers and organizations towards contributing to central projects most in need of support, and overall help nudge the ecosystem towards more efficient allocation of contributor effort. Recall the Open SSL and leftpad examples discussed in the introduction. In these and other similar cases, arguably the centrality of these projects for the health of the rest of the ecosystem was not as clear before their respective prominent incidents as it has become after the fact. Newly developed signals, such as the ones our approach can inform, could have been used to reduce the information asymmetry. Therefore, it is not surprising that recently both GITHUB and PyPI have started displaying information on dependents and dependencies for some OSS packages hosted or published there. We expect other signals will become available in the future.

Our results may also be *actionable for OSS practitioners*. If future research confirms that there is a causal relationship, not just the correlation we demonstrated in our work, between the variables we identified and a project's risk of becoming inactive, that may provide means for suggesting how to extend the life of projects that become inactive without being feature complete.

Still, we emphasize that the response variable in this study (dormancy, or low development activity) is not always indicative of project abandonment, and it therefore requires careful interpretation and should be adjusted to project context. While all abandoned projects are dormant, not all dormant projects are abandoned. For example, utility libraries with a well defined scope do not require further maintenance and thus will also be rendered as dormant, even though they are not abandoned. One interviewee used SMTPlib to illustrate the issue. This library implements a standard unchanged for 30 years and does not require maintenance. This claim is supported by prior research on reasons for OSS project failure, indicating that 11% of seemingly abandoned projects are just considered feature complete by their authors [26]. This suggests that when interpreting sustainability indicators, one should adjust at least to project class and size. *E.g.,* existing dependent projects early in the lifecycle might indicate a chipoff from another project, feature

complete from birth, which is not a negative sustainability indicator. Likely, a dormant upstream project might not be an issue if it is feature complete, but can be a problem if it requires constant maintenance. In practice, it means that the presented survival model does not fully apply to feature complete projects and one should consider qualitative methods instead.

Future research should try to further distinguish feature complete projects from abandoned ones. Suggested ways to determine if a dormant project is complete rather than abandoned may include looking at: maintainers' activity outside the project, non-development activity (mailing lists, issue trackers, and community forum discussions), and dynamics of the project user base; some anecdotal evidence also suggests that projects abandoned by their maintainers continue to be used by existing adopters, but are rarely adopted by new projects, in contrast to feature complete projects, which continue to be adopted as dependencies in new projects.

2.8 Conclusions

Prior work revealed a number of project characteristics related to the sustainability of open-source projects. In this mixed-methods study, we have extended those results to include ecosystem factors. We theorized about expected effects and used survival analysis on a large set of PyPI projects hosted on GITHUB, modeling risk of dormancy early in their life cycle and later on. We then triangulated the models through interviews with project maintainers, and modeled interaction effects informed by the qualitative analysis.

Our work shows the real impact ecosystem context has in how software is developed, and suggests that it brings new risks as well as clear benefits. As open-source projects are increasingly incorporated into software supply chains, organizations need to improve their ability to evaluate the risks they are taking on and learn strategies for mitigating them. It is also becoming clear, for example in our results about the effects of corporate participation, that it can have a destabilizing effect as well as simply providing more resources. In addition to becoming more informed consumers of open-source software, commercial firms should carefully consider the impact that inconsistent participation can have on the ecosystem.

2.9 Threats to Validity

While in this study we aimed to study the effect of external factors on project sustainability, our use of dormancy is only a proxy for project's ability to sustain itself long term. One situation when this operationalization might fail is when projects stop development not because they are abandoned, but because they implemented all intended features, becoming feature complete. It is conceivable under this definition that projects implementing wrapper around external APIs with long release cycle will also be recognized as dormant under this definition. While handling of such situations should be improved, at the time we do not have a good way to recognize projects' feature completeness at scale. Given that only about 11% become feature complete [26], we consider our simplified definition of dormancy as an acceptable risk.

Due to the size of our dataset, not all new features suggested by the interviewees were practical to test. Remaining, untested effects, left open for future work, include: issue response time, issue discussions, quality of commit messages, quality of documentation, automated tests, and use of CI. Future research should generalize and contrast our results on different ecosystems, better account for feature complete projects, and test the remaining features suggested by interviewees. For now, we note all these issues as potential threats to validity.

Chapter 3

Issue sources: balancing effort supply and demand

3.1 abstract

Ensuring that maintainers of Open Source Software are capable of putting enough effort to address all reported issues is an essential condition of project sustainability. To understand factors generating such demand for maintainers' effort, we use a combination of interviews with project maintainers and issue reporters, and mining GITHUB data. Specifically, we look into how maintainers prioritize their effort and what external factors they consider, in projects facing high demand for maintainers effort. We further interviewed 64 issue reporters from 12 popular projects, asking them what context their issue originated from and what was their usage status at the time of reporting. We then developed a set of signals to automatically differentiate between issue sources and evaluated their predictive power using the previous interviews as the ground truth.

We found that projects facing high effort demand tend to be at the very end of the dependency chain, serving either end users or end programmers. We found that in our sample issues originated dominantly from work-related environment, and were mostly reported by established users, as opposed to new adopters and one time users. While we found no significant relation between reporter's usage status and the origin context of their issues, we found that established users dominantly report bugs, as opposed to new adopters and one time users, who mostly ask questions. Finally, we found that while there is no a single decisive signal that can differentiate issue sources, there is a set of medium strength signals that can be used jointly. We hope that our findings would help inform projects' practices of effort demand prioritization, improving sustainability in projects with inadequate supply of effort.

3.2 Introduction

Sustainability of open source infrastructure, on which we heavily rely, is often called into question [50]. To remain relevant, open source infrastructure, like all other software [51, 96], needs to be maintained and needs to evolve: to fix bugs, to patch vulnerabilities, to update dependencies, and to adapt to evolving requirements, new use cases, and changing environments. However, the supply of developers maintaining open source infrastructure is limited. At the same time, broad and often controversial discussions on funding, fairness, corporate involvement, stress, and even burnout have started to emerge [56, 109, 174]. Maintainers, volunteers and paid developers alike, report being overwhelmed with the amount of support

requests, bug reports, feature suggestions, and patch submissions, all demanding their time and all highly visible. For example, in a widely read blog post "What it feels like to be an open-source maintainer"[1], Lawson describes his struggle with a large amount of incoming requests in the issue system: "There are still more than a hundred waiting in line. But by now you're feeling exhausted; each person has either had a complaint, a question, or a request for enhancement." Sustainability requires that demand and supply of maintenance effort are balanced.

On the supply side, the motivation for maintainers to contribute to open source have been well studied [93, 109, 144, 159], surfacing many aspects of OSS contributors, such as motivation and barriers to join projects, factors for sustainable contribution and reasons to leave. Also the technical and social mechanism of how maintainers interact with other stakeholders have been studied in depth (*e.g.*, how pull requests or code reviews are managed and prioritized [56, 61, 151]).

In contrast, in this paper, we explore the external demand for maintenance effort, about which little is known. We explore both the what and the who of maintenance demand, that is, what kind of requests are posed to the projects and who poses those requests. To that end, we manually code a large sample of 237 GitHub issues in 14 Python projects different across a number of key dimensions. We additionally survey the creators of the issue to learn about their demographics and their relationship to the project. Among others, we observe that over a half of issues originated from commercial context, and less than a half of all cases software adoption is public. In our study, we focus on issue discussions on GitHub and analyze both the content of the issue, characteristics of the user posting it and their interactions. We focus on issues because they are the dominant form of communication to request maintenance work in many open source projects. While some support work may be done in other channels (e.g., Slack or StackOverflow, by the maintainers or other community members) most external maintenance requests, including bug reports and feature requests, are communicated through issue trackers.

Understanding the demand for maintenance effort will provide an important grounding for the research community and an important step in understanding and potentially influencing the dynamics in open source toward more sustainable practices in future work. For example, it will provide context for discussions such as:

- What are strategies to place more of the maintenance cost on the person creating the work? Examples currently explored by some open source projects include enforcing issue templates [126], requiring community discussions and responses[2], or interactive deduplicating of issues [172]. Strategies like this could curb demand either by reducing less important requests or by making requests easier to handle.

- What are sustainable practices to prioritize maintenance work? For example, one could argue that requests from corporate users should be prioritized or that those requests should not be prioritized unless the corporation supports the project financially. Once we understand both supply and demand better, larger discussions around fairness [56] and free riding [122] in open source can be revised.

- How can we encourage a better balance between maintenance supply and demand in open source projects? One could explore whether detecting and signaling an unbalance between maintenance supply and demand encourages more developers to contribute, for example through "Project in Need" badges or other interventions.

[1] https://nolanlawson.com/2017/03/05/what-it-feels-like-to-be-an-open-source-maintainer/

[2] E.g. as done by npm CLI team https://twitter.com/maybekatz/status/953402549293350913

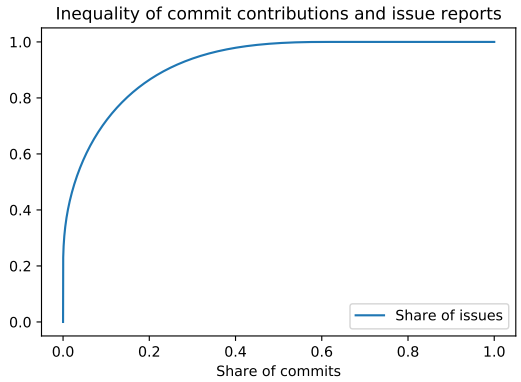

Figure 3.1: Inequality of issue reporting and code contributions on GITHUB
The curve is built over 28 million users who made at least one commit or reported an issue.

Importantly, the who behind an issue is often not very transparent, beyond a link to a more or (often) less descriptive GitHub user account. In 53% the use is not publicly visible in the user's profile, in 72% the issuer does not share an affiliation in the profile, and 70% of all requests are posted by users with empty GITHUB profiles, essentially anonymously. However, for several potential interventions it may be valuable to signal more clearly the issue reporter's affiliation and relationship to the project (*e.g.*, to prioritize issues by sponsors or non-profits, or to take the reporter's experience into account). To that end, we explore (with the ground truth of our survey) what kind of signals we can automatically infer and amplify from public repository data that may be useful for such considerations, such as prevalence of certain types of user activity, or reporter's job profile.

Besides transparency issues, there is also a good deal of imbalance in the amount of contributions effort appropriators bring to the project. On GITHUB, over 83% of all reported issues were created by users responsible for less than 17% of all commits on that platform. At the same time, 72% of code contributors (*i.e.*, effort producers) did not report a single issue (see Figure 3.1).

Historically, maintainers of OSS projects deployed various strategies to prevent effort overappropriation, *e.g.*, by accepting only certain types of requests. `TensorFlow`, a machine learning framework, limited issue reporting to only bugs and performance issues, leaving out user support requests.[3] A similar policy was adopted by other projects, *e.g.*, `numpy`[4] and `requests`[5] diverted users to StackOverflow for general support questions. While drawing effort appropriation barriers along issue types worked for these packages, as we show further, it might be suboptimal in other contexts.

Limiting effort demand by only certain type of requests is not the only strategy used by maintainers. Some projects that are often offered as a part of Infrastructure-as-a-Service (IaaS) model, used special licenses to impose barriers on certain types of users, rather than using effort demand type. In classic software projects, like `tensorflow`, those who reap the benefits are the same who consume effort, *i.e.*, software users. In IaaS world, however, it's infrastructure providers who reap the benefits in the form of revenue, while users request effort directly from project maintainers. This situation does not allow project maintainers to effectively negotiate for resource contributions or set barriers. Quoting Eliot Horowitz, a co-founder of `MongoDB` project, "*Unfortunately, once an open-source project becomes interesting, it is too easy for cloud vendors who have not developed the software to capture all of the value while contributing*

[3] https://github.com/tensorflow/tensorflow/blob/master/ISSUES.md
[4] https://numpy.org/gethelp/
[5] https://requests.readthedocs.io/en/master/community/support/

little back to the community." In October 2018, `MongoDB` prominently introduced changes to the project's license, imposing additional constraints on commercial users, targeting cloud providers in particular.[6] A similar license change was implemented by another open-source project, `Redis`, in February 2019.[7] Other high profile projects, such as `ElasticSearch`, `Confluent`, `CocroachDB`, `Timescale`, etc., followed their example since then. It is not clear, however, in what context and for what projects this model could work.

Looking beyond these examples, can we generalize strategies to make a playbook for project maintainers to reduce effort demand, cost, and, perhaps, get better at converting the demand to supply? The above can serve as anecdotal examples of strategies applied in different circumstances. Different project context might call for different strategies to maintain effort balance. Even in the cases above, effectiveness of these measures was called into question as Amazon, one of the biggest cloud providers, simply started its own fork of `ElasticSearch` only few months after its license change[8]. Perhaps, it would be better to have signals of mismatch between effort supply and demand and sources of effort requests to inform development of effort management strategies to fit project context.

To this end, we report on a mixed-methods case study of 26 open-source projects part of the PyPI Python ecosystem, combining data mined from the projects' GitHub repositories and interviews of project maintainers and issue reporters. We study who the effort appropriators are and we investigate potential signals that can be used to distinguish between types of appropriators and their project adoption status — two types of information that could be useful to maintainers when deciding how to prioritize requests. Specifically, we develop and evaluate a set of signals to identify the context in which reported issues were discovered, including commercial, personal, and academic usage, and new versus long-time users. We also look into how such context was used by maintainers to prioritize how they spend their effort, if at all. Among our findings, we highlight that existing practices used by maintainers to prioritize effort demand make only limited use of such differentiation. We find that there are likely no strong signals that can be used to identify issue context, but there are many either weak or sparse signals that can be used jointly. Our findings can be used to inform project decisions for better sustainability in a situation of scarce resources.

3.3 Related work

Our study builds on a substantial body of literature on supply of and demand for contributor effort in open source. In this section we review research on the factors that attract effort to open source projects, factors that create demand for effort, and evidence for the prevalence of mismatches between supply and demand.

3.3.1 Developer Participation: Supply of Effort

Previously, a "private-collective" model of innovation was proposed [73] to explain individual participation in terms of the private returns individuals receive, including solving their own problems and contributing the solution to the community for use. Later, the definition of what private problems developers try to solve has been substantially expanded to include objectives such as learning or even building reputation to signal their skills to potential employers. Intrinsic developer motivation was also expanded with the value of enjoyment and gift culture benefits [9]. It was shown that technical changes that reduce the costs of participation result in an enlarged pool of contributors [78]. The private-collective model also helps to anticipate

[6] https://hub.packtpub.com/mongodb-switches-to-server-side-public-license-sspl-to-preven
[7] https://redislabs.com/blog/redis-labs-modules-license-changes/
[8] https://aws.amazon.com/blogs/opensource/keeping-open-source-open-open-distro-for-elast

how developers respond to the possibility of monetary rewards. If their private motivation is intrinsic or extrinsic, they are more amenable to monetary rewards, and less so for community motivation [92]. Adherence to open source ideology also impacts effectiveness, attracting more participants, but slowing task completion [145].

The body of research on motivating volunteers or companies to invest effort in OSS suggests that the factors that drive contribution are not related in any obvious way to how widely used a project is or how much need for maintenance it has. Factors that were found to influence where volunteers spend their effort [93, 104, 137] include using new and exciting technologies, providing opportunities to learn and creating a portfolio of work to impress prospective employers. Companies support projects when they align with the interests of their business [4, 59, 141, 164], *e.g.*, adding features to an open-source project that will increase the sales of their complementary proprietary product. These factors do not necessarily influence volunteers or companies to focus their contributions on projects that need their effort the most.

While we do not (and can not) directly measure all types of motivation to provide effort revealed by the literature, the design of the measures we collect attempts to approximate most of these dimensions through the identification of academic/commercial/hobby/community background of issue reporters and hypothesising (and observing) their likely behaviors.

3.3.2 Demand for Development Effort

Software maintenance is modifying software "to correct faults, improve performance or other attributes, or adapt to a changed environment." [77]. Open source software, like any software, requires maintenance and new features to remain relevant. This creates demand for developer effort, with severe consequences: unfilled demand would render projects obsolete until the necessary improvements are made. The questions of demand for tasks such as issue fixes/new features, coordination, mentorship, bug triage, testing, and documentation have been investigated indirectly. For example, it was documented how the core team in Apache explicitly prioritizes bug fixes by the size of user population and how that, in turn, increases the resolution times for issues affecting few users [115]. It was noted [63] that new core team members typically join when they bring expertise that is currently lacking, *i.e.*, fulfilling missing demand. The projects in an ecosystem typically need coordinators [128] who ensure that the individual projects have features needed for an ecosystem-wide release. Work on managing open-source requirements [140, 167] shows how demand is discovered, analyzed, prioritized, and validated within discussions and issue requests. Popular projects need help triaging user-reported issues [5] and triagers save tremendous amount of developer effort [168]. The lack of developers with extensive expertise was associated with lower productivity and quality in recently offshored projects [112, 113]. Many ways to predict demand have been investigated. For example, historic relationships among changes in a system were used to predict where future changes (demand) will occur [68]. Other examples include research on how demand may be inferred from the extent of changes in upstream or interdependent (when a combination of projects makes a meaningful downstream product) projects [14, 128].

It is also the case that not all usage of open-source (and, thus, demand for maintenance effort) is observable on social coding platforms. Kalliamvakou et al [82] conducted a survey and a series of interviews to explore development practices on GITHUB. Out of 240 survey participants, only 67 (28%) indicated they adopted GITHUB to contribute to open-source. Further interviews with 30 users randomly selected from those indicated they participate in collaborative projects, with 24 (80%) reporting they are using GITHUB primarily for job-related purposes. This estimate can be used to calibrate our measures of private contributions (that count only contributions to private GITHUB projects).

In this paper we primarily focus on the demand generated by direct and indirect users and by classifying

these users into community, commercial, academic, and hobby users.

3.3.3 Match and Mismatch Between Effort Supply and Demand

Participants in open source ecosystems, both volunteers and firms, are generally free to make contributions wherever they want. These individual decisions bring about an emergent allocation of effort across the projects in an ecosystem. Other than the actions of individual participants, there is generally no mechanism to force, or even to nudge, participants to apply effort where there is the greatest need. This is one major difference between peer production communities and commercial firms participating in markets, where the forces of supply and demand determine price, a very strong signal guiding the allocation of resources [39].

There are many examples of mismatch between effort supply and demand in related domains outside of open-source. A study of another peer production community, Wikipedia, examined the alignment of article quality with popularity, and found strong evidence of mismatch, meaning, for example, that two billion page views per month were for articles in high demand but with relatively low quality, while there are a large number of very high quality articles that are rarely accessed [162]. There is also evidence from the way nonprofits allocate volunteer time that it is unrelated to objectively measured community needs, but based rather on volunteer preferences and members' and leaders' perceptions of the severity of problems [103].

The study of requirements management points out the difficulties of discovering, articulating, and implementing features even when the development effort is plentiful [167]. The lack of development effort has been documented in the highly publicised in the Heartbleed bug [48], but we are not aware of a systematic investigation of cases, causes, or consequences of such undersupply or how to recognize and address it. Regarding the question of supply and demand we conducted a survey to better understand how maintainers prioritize which issues they will address.

Closer to our study, there have been a few investigations of how issue reports are generated in open-source. In a study of the connection between usage of issue trackers by open-source projects and project success it was found that 33% of developers reported issues in their projects, and 42% of users did not contribute to the code base [8]. More recently, a study of patterns of pull requests and issues in the NPM ecosystem found that "Users contribute and demand effort primarily from packages that they depend on directly with only a tiny fraction of contributions and demand going to transitive dependencies" [45]. We also find relatively few issue reports that involve transitive dependencies in our sample. Another study [90] of power users in the Mozilla issue tracker revealed that most of 150K users in the study reported issues not resulting in code change that devolved into technical support, duplicates, or narrow feature requests. Reports that did lead to a code changes were reported by a comparably small group of experienced, frequent reporters. Analysis of Windows Vista and 7 issue trackers found that bugs reported by people with better reputation or located on the same team are more likely to be fixed [66]; the same study also reported that most critical bugs are fixed anyway. In our sample we identify "established users" and find that only a small fraction of issue reporters in our sample qualify as established users, as also reported elsewhere [90].

To the best of our knowledge, the question of classifying and quantification of sources of effort demand (*i.e.*, reported issues) remains relatively unexplored. Understanding sources of effort demand and their relative size could help to inform resource decisions in OSS projects and help improving their sustainability.

3.4 Research questions

As discussed above, in OSS projects, a necessary condition of long term project sustainability is ability to match the support requests, bugs, and new features (*i.e.*, the effort demand) with appropriate contributions

(*i.e.*, the effort supply). In case there is not enough supply of effort, maintainers risk to burn out and projects risk their sustainability [109]. Possible strategies of resolving this include prioritization of demand and attracting new resources. Such prioritization and request for resources require additional information on where the demand originates, *e.g.*, who benefits the most from the project and who possesses the necessary resources.

To select the best effort management strategy we need to understand the factors that projects might want to take into account, *i.e.*, the problem space, first. We might expect that the structure of effort demand in a feature complete projects [26] is quite different from projects that did not release their first stable version yet. Projects providing low-level API probably will have quite different audience, and, consequently, different requests for effort, than projects intended for end users. The full range of signals to be used for better effort management strategy selection, however, is unclear and requires further research.

The first step to developing sustainable effort management practices is to understand what factors should be taken into account by project maintainers. We could see how maintainers limited effort appropriation by restricting certain type of effort appropriation (*e.g.*, support questions) or deployed license restrictions on certain types of users, depending on the project context. However, this evidence is mostly anecdotal and might not reflect the full range of possible project contexts. To better understand the problem space, *i.e.*, factors affecting the choice of management strategies, we need to to look beyond these examples. Systematization of maintainer goals, strategies they use, and factors leading to the selection of these strategies, might help other maintainers to select the best fit strategy and inform future models to develop better strategies. Thus,

RQ1. What are the effort management strategies currently used by maintainers in OSS projects, and how they relate to the project context?

Whether on the project or per-request level, maintainers need to know where the demand comes from to develop effective effort management strategies. While the full answer might be a bit more complicated, as the first approximation we can characterize appropriators by how much resources they can potentially contribute to the project. We could see already how in the MongoDB case, the deployed license policy targeted a specific set of users in their consumption chain, IaaS vendors. Notably, this change did not affect other users, even customers of those IaaS vendors actually using these products. While MongoDB and Redis still get some resources from smaller appropriators in form of direct code contributions and paid support, IaaS vendors, such as Amazon, have a lot more resources, and thus can potentially contribute more. So, the first dimension along which we can differentiate effort demand is the amount of resources at appropriators' disposal.

A mere access to resources, however, does not fully characterize effort demand. Description of demand sources is a bit more nuanced than simply understanding how much resources are at appropriator's disposal, *e.g.*, by deriving their commercial affiliation as it was done in effort supply studies[24, 135]. For effective negotiation, maintainers seeking to attract resources to their project also need to know the appropriator's degree of engagement, *i.e.*, how costly it would be for the requester if their effort demand request is not satisfied. For example, a problem in Cobol, a legacy programming language still used in many financial systems, might be a lot more critical to users of such systems, comparing to a problem discovered in a new tool not so deeply entrenched in their infrastructure; in this case, maintainers of Cobol-related projects have better leverage on effort appropriators. Besides simple negotiation for resources, established users might be more familiar with software internals, making them more suitable for potential recruitment as project contributors. We might expect other differences in effort appropriation patterns between users and non-users, such as higher share of support questions from newcomers and more bugs reported by advanced users. Thus, we can expect the degree of engagement to be another important characteristic of effort demand.

Yet this is not the full range of factors effort demand can be characterized along, at the very least we can expect it to have multiple components. With this in mind, we need to explore,

RQ2. What are the sources of effort demand in OSS projects?

Whether on the project or request level, in order for participants in OSS, be they volunteers, corporations, foundations, funding agencies, philanthropists, or crowdfunders, to take action towards aligning the provision of the resource — maintenance effort — with its appropriation, they require accurate, timely, and actionable information about status of supply and demand of effort. Yet, despite the high level of transparency on social coding platforms like GITHUB [37] and the multitude of *signals* available as indicators of developer expertise and commitment [37, 105, 150], project attractiveness [16, 54, 133], and software quality [130, 148], there are still basic questions about the patterns of effort demand that remain hard to address.

To develop general understanding of effort demand sources, both for research and practical purposes, we need to be able to differentiate them at scale, using available signals. Besides simply listing potential signals, it's important to know their predictive power and practical usefulness. On the effort supply side, multiple strategies were previously used to identify commercial contributors, including daytime contribution patterns[25] and email addresses[24, 135]. However, many of these signals are sparse or not available in effort demand. While git commits are usually stamped with the author's email address, reported issues are not. Often reporter profiles don't contain enough data to derive their daily activity patterns. So, even a strong signal might have limited practical application for practical purposes due to its sparsity.

Another issue with available signals is their interpretation. Signals might be non-linear (*e.g.*, number of days since reporter's first issue), categorical (*e.g.*, reporter affiliation with the project on GITHUB), or even open text (*e.g.*, reporter's bio). Some signals are multidimensional in nature, *e.g.*, contribution patterns in work-time vs. off-time. Interpretation of these signals might be more complicated than running a large array of data through a regression, and requires qualitative investigation. Thus,

RQ3. What signals can be used to differentiate effort demand by sources?

3.5 Methodology

To explore existing effort management practices and their interaction with project context, we interviewed maintainers of 14 OSS Python projects hosted on GITHUB. We then conducted another series of interviews with issue reporters (*i.e.*, effort appropriators) in another 13 projects, to determine the source of these issues. The demographics of both groups of projects can be found in Table 3.1. Finally, using data obtained from the interviews as the ground truth, we collected a range of signals from issues and reporter profiles, and evaluated their predictive power to differentiate sources of effort demand.

In this study, we used two groups of projects, sampled from Python Package Index (PyPI), the official repository of Python packages. Python ecosystem was chosen due to a combination of language diversity and popularity. It is the second most popular programming language on GITHUB[9]. However, unlike JavaScript, the most popular language on GITHUB, which is heavily skewed to web development, Python projects belong to many different domains, including desktop software, Machine Learning applicationd, security, etc. Such diversity is especially important to get a broader view on existing practices, which might be domain-specific.

The first group of 35 projects was made of packages hosted on GITHUB, having at least 200 reported issues in the year of 2019, all while having at most five contributors in the same time period.[10] It is

[9] https://octoverse.github.com/

[10] list of packages, along with their repositories obtained from https://libraries.io/data. Issue and

expected that a limited supply of effort in these projects, combined with high demand, is more likely to force maintainers to develop effort management practices and make them more aware of the factors taken into account in development of such practices. Thus, this group of projects is important to understand the relation between effort supply and project context factors.

Even in this study, we observed several projects using bots to automatically close unattended issues, which potentially means the number of closed issues becomes less reflective of project team's productivity. While duplicated and invalid issues that were closed without much effort before, it was relatively easy to discard these issues from analysis using issue status or labels. Bots, however, close unattended issues without assigning labels or correct status, and occasionally can close "normal" issues waiting for an action from either maintainers or the reporter. Introduction of such a bot in standard GITHUB actions[11] in August 2019 will likely result in wider adoption of this practice on GITHUB and might undermine validity of previously developed sustainability models.

The second group of 13 projects was selected at random using stratified sampling, controlling for the number of package users[12]. These projects were used to complement the first group, which, while representing the target audience of effort management studies, makes a relatively small part of the whole ecosystem and could miss some important cases. In addition, besides understanding how to deal with high effort demand, we might want to look into the circumstances preceding it, to understand the degree of leverage and feasibility of early interventions. So, we might get extra insight into the evolution of effort demand patterns by comparing high demand projects to "regular" ones.

Besides simple comparison of effort demand patterns in packages of different popularity, we might use this to further diversify our sample to get insight into factors that might lead to different effort demand patterns. Since most published packages have only few users (over 90% of all packages on PyPI have less than 100 users, including their developers), simple random sampling would result in mostly trivial projects. At the same time, sampling only popular packages would also result in a biased sample. For example, we might expect end user projects, such as desktop programs, to have relatively "few" users, as their usage will not generate import statements in software code. Likely, projects in the early stage of their lifecycle might not have enough users - just yet - while understanding their patterns of effort demand might be important for early stage interventions. Finally, we observed a class of infrastructural projects, placed so high in the dependency chain only few people use them directly. For example, Python package `warehouse`, implementing the backend of Python Package Index, is almost never used directly, even though the whole Python ecosystem depends on it. Thus, this sample of randomly selected projects is instrumental to cover a wider variety of environmental factors that might affect effort demand patterns. Sampling stratification by usage allows to get a better spectrum of project types and their roles in the dependency chain.

A side observation from this study is that effort demand is not limited only to reported issues, and at least some parts of it are not reflected on GITHUB. Even before, some projects on GITHUB used external issue trackers (*e.g.*, `django`), and others handled version control outside of GITHUB, using it only as a code mirror (as, *e.g.*, Linux kernel). What is new in this observation, is that even projects using GITHUB as the primary code hosting and issue tracker might have substantial amounts of ongoing activity untraceable on GITHUB. As one of the maintainers stated, "*I've just done an hour of project work and none of it was coding or in GitHub issues. As I mentioned, any analysis that focuses on GitHub metrics alone will miss a great deal of important work that happens outside GitHub (grant writing, outreach, conferences, sprint planning, GSoC/GSoD administration and review, "helpdesk" support...)*". These invisible activities

contribution statistics were retrieved using GITHUB API.

[11] https://github.com/actions/stale

[12] usage information was obtained by counting unique developers authoring commits including package import statements, extracted from World of Code dataset[101]

(coordination, fundraising etc.) are more common in most popular projects, while almost non-existent in smaller ones. Besides the mere fact of activities key to project sustainability being unobservable, it might impose an extra challenge on project sampling in OSS sustainability research, as practices used in most important projects are different from general population.

3.5.1 RQ1: existing management practices and their relation to project context

We solicited email interviews from most active contributors in the first group of projects, to understand their effort prioritization practices and context factors contributing in their choice. Interviews consisted of two open-ended questions, followed up by clarification questions when necessary:

- Would you describe the project as challenged or sustainable, and why?
- As a project maintainer, how do you decide what issue to work on?

The first question (sustainability status and factors) is intended as a prompt to understand factors of project context affecting project sustainability, also triangulating on sustainability framing as the balance of effort supply and demand. The question was intentionally broadly worded to avoid framing sustainability in any particular terms.

The second question is intended to understand practices of effort demand prioritization. Looking at these practices can be especially useful to understand the key dimensions along which effort demand can be characterized.

We received 14 responses total. In some cases, respondents did not answer the follow up questions so the details level of these responses varied from a couple paragraphs to a full scale interview. These answers were coded using grounded theory approach and, whenever possible, related to known project characteristics, such as project type, place in the dependency chain, etc. Overall, we came up with twenty codes grouped into five categories for project context factors and ten codes grouped into five categories for effort management strategies.

3.5.2 RQ2: sources of effort demand in OSS projects

Similar to **RQ1**, we solicited email interviews from issue reporters. We sampled up to 30 issues reported by non-contributors from each project, limiting to one issue per unique reporter. The sampling was limited to issues reported at most three years ago, since origin context of older issues is harder to reconstruct for reporters. We obtained 182 issues from randomly selected projects first, as those were expected to be more representative for the demand part of effort balancing. After that, we continued to sample issues from high effort demand projects until reaching saturation with two processed projects, sampling 242 issues total. Issue reporters were requested to give an online interview with two questions, followed up when necessary:

- In what context did you discover this issue?
- When, (and whether) you adopted this library?

The first question (the context of discovery) is open ended, prompting respondents to describe the source of the issue. From the negotiations standpoint, maintainers might be interested to understand issue sources to determine how much resources are at the reporter's disposal. However, reporters might not be interested to disclose, so this question was intentionally put in broader terms. The goal of such this

intentional ambiguity was to understand supplemental factors to issue demand generation, such as role of project's place in the dependency chain, user experience, etc.

The second question (usage status) aims to find out the degree of reporters' dependency on the project. This information can be used by maintainers to understand leverage over issue reporter to negotiate resource contribution to the project on per-request level. On the project level, aggregated information about usage status can also be used to understand project's lifecycle stage. While this question could be derived from the issue context description, pilot interviews showed respondents often omit this aspect, so it was added as an explicit question.

We were able to identify emails of 185 reporters out of 242, asking them for an email interview. Out of remaining 57 reporters, 36 were requested to answer the same questions via an online form, also prompting to leave an email address for follow-up questions. Due to low response rate (only five valid responses), after several privacy complaints we stopped soliciting answers from these users. Overall, we received 64 responses of different detail level in total from 221 reporters, 49 from randomly sampled projects and 15 from projects with high effort demand. The responses were coded using grounded theory approach, and related to project characteristics and visible characteristics of effort demand, *e.g.*, such as issue type. Overall, we came up with 16 codes grouped into four categories for issue origin context and eight codes grouped into four categories for reporter's usage status.

3.5.3 RQ3: signals to differentiate effort demand by sources

We collected a range of observable signals from issue reports and reporter profiles and analyzed their power to predict the issues origin context and reporter's usage status, using coded responses from issue reporters as the ground truth. The signals' power is measured using Cohen's D[27]. Intuitively, this metric shows how much difference between groups compares to natural deviation within groups. D values of 0.2 indicate small effect size, 0.5 - medium, 0.8 - large. We used the following signals:

Weekly activity patterns were previously used to identify commercial contributors[25, 135]. The idea is that paid programmers tend to work almost exclusively 9AM to 5PM, Monday through Friday. We cannot state the same about non-commercial activity, which can be done at any time. Thus, we can assume activity outside of working hours and on weekend to be non-commercial; however, work time activity can originate from either commercial or non-commercial environment. In practice, only weekdays are easily observable on GITHUB since often reporter's timezone is unknown. Timezone information on GITHUB is only available in commit metadata, which either requires extra analysis, or not available at all for users without public commit activity.

In this study, we test two assumptions. First, issues reported on weekends are assumed to be non-commercial. We use a binary variable, indicating whether the issue was reported on a Saturday or Sunday. The second assumption is that people with higher share of weekday activity are more likely to be commercial developers. We use ratio of the number of weekday contributions in the year preceding the issue report to the number of weekend contributions as a signal.

Level of private activity might be used to detect "commercial" users. Until January 2019, private repositories were only available on paid and educational plans. Even after GITHUB enabled private repositories on their free plan, newly repositories are created public by default. Since commercial code is usually kept in private, and OSS projects are public by their nature, we might expect users with predominantly commercial activity to have higher amount of contributions in private repositories. On GITHUB, we can collect this information from user's timeline, which includes the number of private contributions by default.

We used the ratio of contributions in private repositories in the year preceding issue reporting to the

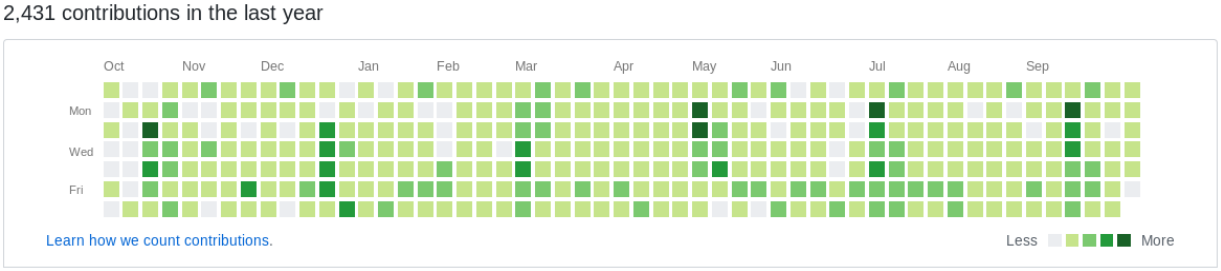

Figure 3.2: Example of a GITHUB user activity timeline

number of public contributions. This signal is expected to generate false negatives more often than false positives, commercial development can happen outside of GITHUB. Another reason for false positives is that the display of private activity can be turned off in user profile (on by default), rendering users even with heavy activity in private repositories as completely public.

Ratio of code contributions, similar to activity in private repositories, might be used to detect "commercial" users. Following the same logic, code contributions (*i.e.*, effort supply) in "commercial" repositories are likely to remain unobserved. Issue reports in related OSS dependency projects, however, will be observable. Thus, we might expect lower share of code contributions in "commercial" users' activity, comparing to non-commercial ones.

We used share of commits in user's activity in the year preceding the issue reporting as a signal. This data was obtained from GITHUB timeline and already accounts for non-public contributions.

Job profile match can be used as a defensive criteria of whether this effort demand does not come from a job-related environment. Intuitively, issue reported by a user working in a completely different domain is unlikely to be work-related. The reverse, however, might not be true, *e.g.*, users can have personal projects in a job-related domain. Also, we need to be careful interpreting this signal as not all job profiles are commercial, *e.g.*, there are users working for non-profit organizations, OSS foundations etc.

We defined job profile match as "a person can plausibly use the focal project for job-related purposes in any of the occupied positions they were working at the time of report". These matches were manually coded by a hired Software Engineering professional, who was given pairs of GITHUB issue reports and reporters' LINKEDIN profiles. Such encoding is inherently subjective, but also reflects the expected quality of matching that can be done by project maintainers in practice.

LINKEDIN profiles for this signal were mined using Google search for a combination of the reporter's name, GITHUB username, company name and location. Only definitive matches were considered, *e.g.*, when LINKEDIN profile was linked to to the GITHUB profile, or both profiles had matching profile picture, etc. Weakening this condition could yield more profiles, but would inevitably result in false profile matches.

Public usage, *i.e.*, explicit use of a library in reporter's code, might be used to determine the degree of engagement with the project. Intuitively, a user introducing the same library in multiple repositories over the last few years is likely familiar with the library and has some degree of dependency on it. Not publicly using the library, however, does not indicate non-usage, since it can happen in private repositories or outside of GITHUB.

We use presence of a corresponding import statement in any of the reporter's repositories as a signal. To differentiate recent adopters from established users, we only considered import statements introduced at least a month prior to the issue reporting. We also did not consider import statements in forked repositories without any commits commits from the issue reporter, as they might not serve as a reliable evidence of personal familiarity with the library.

This signal is not easily observable per say, as some users have hundreds of repositories with sometimes

thousands of commits. However, extraction of this signal can be automated with tools. For the purpose of this study, we created such a tool and shared it as a part of the replication package.

3.6 Results

The list of projects used in this study, alongside with their demographics, can be found in Table 3.1.

3.6.1 RQ1, effort management practices and their relation to project context

We found that interpretation of project context factors was fairly similar across all projects. Factors that brought more resources to the project, such as external contributors, bigger core team, or financial support from a company, were considered positive, while factors limiting effort supply or increasing effort demand, such as limiting project team to a single person or heightened user support workload, were considered negative.

Likely, effort prioritization strategies appear to fairly consistent across all projects, and being relatively uninfluenced by project context. For example, all maintainers tend to prioritize requests with high impact on users, *e.g.*, security vulnerabilities, prioritize bugs over feature requests over questions, and give higher priority to internal issues in company-maintained projects.

Project context factors

Sustainability factors named by project maintainers can be split into two groups, factors affecting effort supply and factors affecting effort demand. On the supply side, we have company support, size of the core team and external contributions. On the demand side, we have project state and the amount of support requests.

Company support was stated as a major project sustainability factor by multiple maintainers, sometimes (project P4) even as the only one. Multiple projects in our sample were actually controlled by companies (P3, P4, P9 - see Table 3.1). In P1 and P7, maintainers were allowed to spend some of the company paid time on OSS projects. In P11, an external company provided a project with a small grant to pay external contributors.

In the cases above, the involvement of commercial companies played mostly positive role. However, in other cases, it had somewhat adverse effects. Maintainer of P9 reported that company control over project imposed a substantial overhead on project development. Contributions, both internal and external, had to be accepted by designated officials, which were often inaccessible. In addition to that, some potential users were wary of the project ownership by a company due to potential vendor lock. In P12, multiple contributors were paid by employers to implement new features, but then were immediately moved to higher priority tasks, leaving no time to make these features public.

Company involvement was observed to have both positive and negative effects. Companies provide resources, but also tie these resources to internal processes and business interests. Many projects would not even be possible without supporting companies. However, in extreme cases the overhead imposed by company processes and excessive prioritization of internal work items might outweigh the benefits.

Size and involvement of the core team was, perhaps, the second influential factor of project sustainability. In this study, we observed that in many, even large and company-supported projects, the majority

Table 3.1: List of projects used for issue sampling

Project	Size, commits	Age, years	Public users	Scope description	Maintained by	Audience
Projects facing high effort demand						
P1	1200	1	700	REST API framework	volunteer	end programmers
P2	4K	2	1K	GUI framework	volunteer	end programmers
P3	500	3	1500	ML library	company	end programmers
P4	300	3	700	data processing library	company	end programmers
P5	600	4	NA	CLI content scraper	1.5 volunteers	end users
P6	800	3	200	content scraping library	volunteer	end programmers
P7	18K	8	16K	plotting library	community	end programmers
P8	1.5K	6	NA	CLI VCS tool	volunteer	end users
P9	2K	5	NA	scalability tool	company	end programmers
P10	2K	5	16K	chat bot API abstraction	community	end programmers
P11	400	2	NA	CLI content scraper	volunteer	end users
P12	5K	15	NA	CLI network utility	community	end users
P13	7K	11	11K	ML toolkit	scholar	end programmers
P14	1K	7	800	image reader binding	community	programmers
Projects sampled at random, weighted by usage						
P15	70	1	20	hardware interface	company	hobbyists
P16	60	3	2K	monitoring tool binding	volunteer	end programmers
P17	18K	3	15K	A compiler tool	community	programmers
P18	29K	4	17K	scientific computations library	(mostly academic) community	research programmers
P19	300	3	9K	graphics library binding	company	programmers
P20	40	6	4K	filesystem helper class	volunteer	programmers
P21	30	1	350	parallel execution primitive	volunteer	programmers
P22	500	5	45K	Web framework	company	end programmers
P23	200	7	24K	text processing library	volunteer	programmers
P24	70	2	1500	network analysis library	company	end programmers
P25	8K	5	3K	distributed processing framework	company	programmers
P26	10K	5	400K	image processing library	community	programmers

of code contributions were made by a single person. Some of maintainers in these projects explicitly recognized this as a threat to project sustainability (P2, P6), while others (P11) stated their project is sustainable. In both cases, however, maintainers stated one or the other by comparing their available capacity against the amount of requests they are dealing with. Maintainers of P2 and P6 faced a decline in their availability (*e.g.*, due to a recent job transition in P6), while P11 had the reverse situation (recently got supported by an external company).

External contributions were referred several times in a positive key as a sustainability factor (P3, P5, P11). The effect of external contributions (*i.e.*, coming from outside of core team) was described as lowering the bus factor (P5), and was simply named among positive indicators of project sustainability in two other cases. In our sample, external contributors typically produced less than 10% of project commits. From the practical standpoint, it might mean that external contributors are more important as a recruitment pool to draw new core members, rather than a substantial source of effort supply.

On the opposite side, the maintainer of P2 deliberately rejected all external code contributions to assert full control over the code. The original intention was to eventually monetize the project, but among other reasons the maintainer indicated it was also to save time on code reviews. At the moment, the project was characterized by the maintainer as not sustainable.

Project state, *i.e.*, the place of project in its lifecycle, was named as a sustainability factor in several projects. Feature complete projects, *i.e.*, those already implemented all planned features (P3, P8, P14), do not deal with the same amount of workload as projects under active development. Feature completeness is a known phenomenon, being one of the common reasons for projects to stop their development [26]. Maintainers in these projects reported dealing with manageable amount of requests, most of which were due to edge use cases or external factors - "*The rate of actual bugs has been steadily decreasing so maintenance costs are not too high; our greatest maintenance cost comes from changes in upstream packages*". A similar statementwas made by a maintainer of a project with relatively small code base (P5).

On the opposite side of the spectrum, we have a project under active development (P12). This project has large user base, and has to be compatible with multiple platforms and dependency versions, including outdated ones. With extra overhead to maintain compatibility and substantial technical debt, this project is characterized as challenged by its maintainer.

What seems particularly interesting about P12 is that, besides being under active development, it might never become feature complete. This project is constantly expanded to support new scenarios, and essentially has unlimited scope. This situation is by far not unique, even in this study we observed several projects that can be extended indefinitely. For example, P18, an academic project, is constantly adapting to new directions of research and cannot possibly be completed. Similarly, P7 also can be extended with new features almost indefinitely.

User support practices, in particular maintainers' willingness to answer support questions, rather than offloading them to community, was mentioned as a sustainability factors by several maintainers. In many projects, a substantial portion, and often the majority, of effort demand comes from questions and issues in users' code (P1, P3, P5, P6, P8, P9, P10, P13, P14). Maintainers reported using different practices to cope with these issues, including encouraging community member to respond to these issues (P1), treating them as lower priority items (P3, P5, P6) – "*going through our questions has become a kind of procrastination and something I like to do when sitting in a train or similar*", utilizing issue templates (P1), or, when resources allow, simply answer them (P3, P8, P10, P13, P14).

Multiple maintainers complained about high number of low quality issue reports among this class of issues, *e.g.*, "*Among opened issues about half of them I would call garbage, they have absolutely irrelevant questions and it is possible to find the answer either by doing GH issue search or by googling for a couple of minutes*" (P9). It was also reported that such issues, even seemingly unattended, draw project resources:

"*Around a quarter of all issues are considered invalid... These issues do not get further attention, but the effort needed to fairly identify these issues as such is a significant portion of the time I spend*" (P5).

Effort demand management strategies

User impact is, perhaps, the most common important factor used by maintainers to prioritize effort demand (P2, P3, P5, P10, P11, P12, P14). Even when not stated explicitly, high impact requests are often prioritized by other means, *e.g.*, by putting bugs over other types of reported issues. User impact is articulated in multiple ways, *e.g.*, by giving higher priority to requests that don't have a workaround (P2, P3), affect more users (P2, P3, P11, P12), or impose a security threat (P12). Prioritization of high impact requests seems to be a common approach, not induced by any specific project context factors.

Cost, *i.e.*, effort required to fix a bug or implement a feature, is often considered in requests prioritization (P2, P3, P5, P10, P12). For example, requests coming with a pull request and thus having lower cost to implement, are more likely to be satisfied (P5); in other cases, maintainers use their own estimate of what it will take to implement the requested feature or fix a bug.

Own interests of the maintainer were named the most important criteria for request prioritization by maintainers of P1 and P13 — "*Whatever strikes my fancy*" (P13). Similarly, in two out of three projects controlled by companies in our sample (P4 and P9), internal issues had the highest priority. In other projects maintained by community and volunteers were factored into request prioritization, even though not being the decisive criteria (P2, P5, P8, P11).

Issue type was used by for effort prioritization at some degree by nearly all projects. Examples include giving higher priority to bugs over other request types (P1, P5, P6, P8, P9, P10), or giving lower priority to questions (P5, P6, P10). The ranking order, bugs over feature requests over questions, was consistent across all projects.

The most noticeable difference in prioritization strategies was observed in treating questions. Some projects making sure to answer all questions (P3, P10, P14) — "*I make a point to be sure that every question gets an answer, though sometimes it might take me a few weeks*" (P3), while others can be less committed to do so (P5, P7) – "*As a maintainer, unfortunately, there is little incentive to answer a too trivial question, since with asking such a question, the author demonstrates little technical understanding and as such is considered unlikely to ever contribute to the project*".

Quality of request was reported to be used in prioritization in several projects (P5, P7, P12). Besides naturally being unable to address invalid or incomprehensible requests, at least occasionally some requests are ignored because of their quality – "*it's worth mentioning that some issues are addressed by explicitly choosing a path of no action*" (P7).

3.6.2 RQ2 sources of effort demand

In our sample, over half of all issues (33 out of 64) originated from commercial, usually work-related, context. By the reporters usage status, almost every other issue (30 out of 64) was reported by an established user of the software, as opposed to adopters or ad-hoc users. We did not find any significant relation between issue origin context and their reporter's usage status except a small class of non-users coming from OSS context. The statistics of issues origin contexts and reporters' usage status can be found in Table 3.2.

We found that reporter's usage status is correlated with the issue type (as, *e.g.*, bug, feature request or question). New adopters and one-time users (more about usage statuses in Section 3.6.2) were more likely

Table 3.2: Issue origin context by reporter's usage status

	Issue origin context					
Reporter's usage status	OSS-related	academic	commercial	personal	other	total
non-user	4	0	0	0	0	4
one-time user	0	2	2	3	1	8
adopter	0	1	13	4	1	19
established user	1	3	17	8	1	30
did not disclose	0	0	1	2	0	3
Issue type						
bug	5	2	7	5	0	19
feature request	0	0	7	3	1	11
error in user code	0	2	4	5	0	10
question	0	1	7	2	1	11
other	0	1	8	3	1	13
total	5	6	33	17	3	64

to ask questions, while established users and a special class of non-users were more likely to report bugs. The statistics of reporters usage status relation to issue types can be found in Table 3.3.

Sources of issues by context

Commercial. Over a half of respondents, 33 out of 64, reported their issues as originating from work-related environment or personal projects that were intended for potential commercialization. One of the interviewed maintainers expressed confidence that his project is actively used by corporations, while others did not emphasize this aspect. Since none of the observed issues could be clearly identified as commercial from its content, the degree of maintainers' awareness of commercial involvement in their

Table 3.3: Issue types by reporter's usage status

	Reporter's usage status					
Issue type	non-user	one-time user	adopter	established user	did not disclose	total
bug	4	1	2	12	0	19
feature request	0	2	4	5	0	11
error in user code	0	1	2	6	1	13
question	0	3	6	1	1	11
other	0	1	5	6	1	13

projects is unclear.

The structure of "commercial" effort demand in our sample seems to be quite different from other issues. Most environment-related issues, such as compatibility issues with OS, hardware configuration, or dependency versions, originated from commercial context. This is somewhat intuitive, since commercial players do have access to more diverse infrastructure than individuals. A potential implication is that if, compared to individuals, commercial players use legacy software for longer, their share of effort demand is expected to increase the older the project gets.

Another observation common for commercial projects is that issues originate farther from issue reporters. While individuals might experience issues first hand, corporate software developers discover issues on the customer side, remote servers, in a dependency of a dependency, a legacy project etc. Often, this "distance" from the issue source affects the extent of control the reporter has over the issue, limiting their ability to deploy a workaround (*e.g.*, a dependency version cannot be changed as it has negative effects on the system). The implications of this observation is that issues originating from commercial context more often come from niche use cases, often considered as low impact by maintainers, and are less likely to have workarounds (which usually means higher priority). These cases might be exploited by challenged projects as a perfect point to negotiate with effort consumers.

In four cases out of seven "commercial" issues related to package adoption, such as installation issues and starter support questions, issues were reported by people in entry level positions (three interns and one fresh engineer), working on exploratory projects. While the observed sample of such issues is small, a potential implication is that adoption-related issues, even coming from commercial users, might not be the best point for resource negotiation for project maintainers.

Personal or hobby-related. The second biggest source of issues, 17 out of 64, were personal projects. All these respondents discovered issues in end-programmer projects, interacting with these projects directly, as opposed to a dependency of a dependency, or merely using the software without any programming involved.

Most respondents in this category seem to be experienced programmers engaged in software development for fun, *e.g.*, running a web page for a local fire brigade or a personal website, adding functions to a smart home system, etc. Few respondents also reported discovering issues while proactively learning new skills to advance their professional careers: "It was only for self-guided growth honestly.Which could be a advantage for me in the future when I apply to jobs". While in neither of these scenarios such users have high stakes in the project or can contribute substantial resources, they might play the role of early adopters for new software and could be instrumental for its further adoption in the industry.

Academic. In our sample, six out of 61 issues originated from academic context. These issues can be further split into two categories, three coming from research-related projects and three originating from class environment. All issues originating from class environment were usage or installation questions, and none of these reporters continued using the library after finishing their class project. Issues originating from research environment were all non-trivial bug reports reported by long-term users. In some sense, research-related issues were similar to "commercial" ones, being triggered by platform changes and more complex usage scenarios.

OSS-related. A distinct small class of issues was reported by maintainers OSS projects. While it is relatively rare in our sample (five out of 64), reporters of these issues tend to report hundreds, sometimes thousands, of issues per year in many different projects. One of these issues was reported by a developer from a dependent feature-rich software project, and four others were reported by maintainers of three different Linux distributions. In all of these cases, the reported issues were discovered by automatic tests and were caused by compatibility with other dependencies or platform updates.

Reporters of such issues might be of a special interest to project maintainers. While they might be

less likely to contribute code or other resources, issues reported by them have very high multiplicative impact, as they effectively serve as proxies for a much bigger user base. Also, they proactively test software with updates that are not yet widely deployed: "*We often upgrade system dependencies much earlier than project developers do, as part of "unstable" or "staging" repositories... New versions of dependencies often require some changes to the project source code, for example adapting to renamed functions. When we discover these, we submit issues to add this compatibility*". At the same time, the mere presence of a package in repositories of a Linux distribution might both indicate package recognition and create additional effort demand due to bigger user base.

Sources of issues by usage

We found that usage status is represented by a spectrum of possible stages of adoption, rather than being a binary classification into users and new adopters. Common notion of library adoption assumes conscious choice, direct usage, and intention to use the library in the future. Contrary, we observed people using software without even realizing it, *e.g.*, through a dependency or supporting infrastructure. People are involuntary assigned to projects, and sometimes work on one-time tasks far from their main field, limiting potential future reuse. In this study, we build our analysis around four main usage statuses, namely: established users, new adopters, one-time users, and non-users.

Non-users. As previously mentioned, four respondents in our sample represented a special case of users, Linux distributions' maintainers. These users report bugs discovered by automatic tests, without ever using this software or planning to do so (more details in Section 3.6.2). Effectively, this special class of *non-users* serves as a proxy to a much bigger userbase. Even though this group of users is fairly small, due to their distinct role in the software ecosystem we recognize them as a distinct separate class.

Established users. Almost half of respondents (30 people) started using the software a while ago (usually, years ago), and by the time of reporting the issue were reasonably familiar with it. Twelve of these respondents did not use the software directly and were exposed to it through a dependency in a legacy or a third party project, or were assigned to work on a project that used this software, which shows that software adoption is not always voluntary.

Established users reported bugs more often (40% of their requests) than new adopters and one time users (10% and 13%, respectively). Intuitively, established users are more familiar with the software and thus they are less likely to ask questions. They are, however, more likely to engage in advanced use cases, which might be less documented, and run into less obvious problems in their code or discover bugs in less used parts of the project. Also, as more active users of the software, established users are more likely to be first to discover newly introduced bugs.

Adopters. Nineteen respondents reported their issues either shortly after starting to use the software, or while being in the process of learning it for future adoption. Besides users representing a "normal" adoption cycle, *i.e.*, seeking for projects providing desired functionality, attempting to use them and possibly starting to use it long-term, two respondents indicated that they were aware of the available solution long time ago and discovered issues the first time they actually tried to use it. While these users are no different from regular adopters from the resource standpoint, it indicates existence of "latent users", who are already determined to adopt the software but did not have a chance to do it yet.

Among adopters, we also observed some variation of commitment to continue using the library, with an extreme case of one-time users. We observed several "just-in-case" adopters, who are learning how to use the software as self-guided learning, without a specific application in mind. For example, one of these users was primarily working with Python web frameworks and wanted to learn more about asynchronous

Table 3.4: Signals to differentiate between issue sources

Signal	Issue source	Count	Average	σ	Cohen's D	p
Job profile match	commercial	25	0.60	0.50	0.50	0.05
	others	12	0.25	0.45		
Reported on weekend	personal or OSS	22	0.18	0.39	0.26	0.08
	others	42	0.05	0.22		
Share of weekday activity	commercial	33	0.85	0.24	0.02	0.86
	others	31	0.84	0.16		
Share of private activity	commercial	33	0.15	0.32	0.24	0.07
	others	31	0.04	0.12		
Share of commit activity	commercial	33	0.41	0.39	0.31	0.04
	others	31	0.60	0.31		
Public adoption	established users	27	0.47	0.51	0.42	0.02
	new and one time users	30	0.19	0.40		

alternatives in to strengthen their professional profile, rather than to fulfill an immediate need in an existing project.

New adopters in our sample asked more questions, comparing to established users. This difference is intuitively explained with their lack of familiarity with project features and documentation. In our sample, we observed somewhat higher number of feature requests coming from new adopters. In all four cases, these were experienced developers evaluating projects to fit their environments, asking maintainers to implement missing features to match their needs.

One time users. Finally, a group of eight reporters discovered their issues while working with the software on a one-time basis, without long term plans to adopt it. Three of these reporters came from academic environment, where usage of these particular software was required in the class project. Four remaining users were reasonably experienced programmers, working on side projects not aligned with their primary domain. One of them was working on a proof-of-concept project, and was looking for a standalone implementation of a function that was present in a bigger framework they were normally using. In this situation, the library implementing this function was a temporary replacement, not a long-term solution the user or the company would normally use.

One time users are similar to new adopters in their degree of familiarity with the project, just not intending to continue using the software. Just as among new adopters, the dominant type of requests filed by one time users, were questions.

3.6.3 RQ3, Signals to differentiate issues by source

Users reporting issues discovered commercial context are more likely to have job profile that can plausibly involve using the software the issues is reported to, comparing to all other users. The size of the difference

between these groups is indicative of medium power of this signal[27], and the difference is statistically significant at $\alpha = 5\%$ level. This signal is somewhat sparse, as we were able to identify LinkedIn handles for only 34 out of 64 respondents, or 58%. With relatively high cost of mining LinkedIn handles and sparsity issues, this signal alone, despite the highest discriminative power, has only limited practical application.

Issues originating from personal projects and OSS context are more often reported on weekends comparing to all other issues (18% vs. 5%). However, this difference is not sufficiently high to identify issue sources in practice. Another observation here is even personal project and OSS issues are more likely to be reported on weekdays comparing to random chance (82% vs. 71%).

Reporters of "commercial" issues tend to have almost exactly the same share of weekday activity and only slightly higher share of private activity in their GITHUB profiles, comparing to all other users. This goes in line with the previous observation that issues originating from personal and OSS projects also tend to be reported on weekdays.

Reporters of "commercial" issues on average tend to have slightly higher share of private contributions. Further check revealed that only eight issue reporters in our sample had more than 10% of private contributions in the last yea. Six of them (75%) reported "commercial" issues, comparing to a random chance of 54% (see Section 3.6.2). So, the issue is not that this signal lacks discriminative power, but that it is too sparse.

Reporters of "commercial" issues have lower share of commit contributions on average, comparing to reporters of other issues. However, there are both "non-commercial" appropriators with mostly issues activity, like Linux distribution maintainers described in Section 3.6.2, and "commercial" users having a lot of public code activity. Overall, the discriminative power of this signal is small.

We found that less than a half of established users used their software publicly. At the same time, almost every fifth adopter or a one-time user publicly used the software they reported issues to. From the interview responses, we learned following explanations for non-public usage:

- issue originated in proprietary code, often in a work-related context.
- many personal projects are not shared publicly, either being stored in private repositories or not being uploaded to code hosting platforms at all.
- end-user software, such as command line utilities and desktop programs, is not imported the same way as libraries. Usage of such software is practically untraceable.
- the issue was triggered by an indirect dependency, *e.g.*, dependency of a dependency

Public usage by in adopters and one-time user repositories (5 cases in our sample) came from adoption attempts and one-time projects that were shared on GITHUB. Overall, public usage has medium predictive power to differentiate between established users from adopters and one-time users, with Cohen's D of 0.42.

3.7 Discussion and implications

In this study, we conducted a series of interviews with maintainers of projects facing high effort demand, interviewed 64 issue reporters about sources of their issues, and evaluated a set of observable signals for their ability to differentiate reported issues by sources.

3.7.1 RQ1, effort management practices and their relation to project context

In this study, we observed that maintainers in multiple projects with high effort demand use similar prioritization strategies and have consistent interpretation of sustainability factors in their projects. We also observed prevalence of questions in effort demand structure of these projects, which was also consistent across projects. The most interesting finding, however, come out from comparison of projects facing high effort demand to randomly sampled popular projects. It seems that vast majority of projects with high effort demand combine popularity with a specific role in the dependency chain.

Many sustainability factors named by maintainers and effort demand prioritization strategies were common across sampled projects. For example, requests with higher user impact were almost unconditionally given higher priority, and, among other issues, bugs were prioritized over feature requests, with questions typically being the lowest priority. On the effort demand side, the pattern also seems to be common across projects. Questions seem to be the most prolific type of requests, making a substantial article of effort demand in many projects in our sample, sometimes the majority of them (P1, P6). As the lowest priority items, questions were the first type of effort demand to be cut off if the project was low on resources.

Maintainers' decision whether to address all requests or skip low priority ones seems to be dictated by a simple equation of whether they have enough effort supply to match the demand. Projects capable of addressing all requests are often feature complete (P3, P14), implement a rather small scope (P10, P14), or have fewer users, *i.e.*, have relatively lower effort demand. In addition, some of them have bigger core team (P3, P10), have active external contributors, or supported by a company (P3), *i.e.*, also have more effort supply. On the opposite side, projects skipping some of the requests have fewer or no reinforcing factors, but have more users. Potentially, it could mean that adjusting project practices to reduce cost of questions, *e.g.*, by offloading them to community as it was done in `numpy` and `scipy`, might be a cost-effective sustainability improvement measure in challenged projects.

The question central to this study, however, is where does this effort demand comes from. Looking at the projects with high number of reported issues - this is how projects were sampled for this part of the study - we can see these high numbers are in a large part created by an influx of questions. What makes these project so special and creates so many questions, is a combination of their place in the dependency chain and high popularity. Vast majority of these projects is either end user software, *i.e.*, intended to be directly run by users, or end programmer libraries, *i.e.*, used by programmers in projects that are not intended for reuse, such as web sites, ML projects etc.

One explanation for higher share of questions in end user and end programmer projects is their audience. Intuitively, projects higher up in the dependency chain implement smaller lower level "building bricks" to be reused by other programmers. Projects farther down the dependency chain tend to implement higher abstraction levels and have bigger scope. At the very end of the chain, we have end user software, which directly performs its functions, rather than being reused in other programs. The farther down the chain we go, the less programming experience typically is required, with end user software often being used by non-programmers. End programmer software, *i.e.*, libraries intended to be used in projects catering to end users, is often among the first things new programmers learn, *e.g.*, a web framework like `django` or a data manipulation library like `pandas`. This does not mean there are no experts in end user software or end programmer projects, but these projects are more likely to be used by inexperienced developers, which have questions more often.

By the same reason, end user and end programmer software is likely to have more users. Software farther down the dependency chain implements higher level abstractions, and thus might have lower the entry barriers for programmers and end users. At the same time, it often delivers more end user value than low level primitives implemented by libraries at the top of the dependency chain. So, not every end user or

end programmer software has a lot of users, but they might have bigger target audience and thus are more likely to have big user base than low-level libraries.

3.7.2 RQ2, sources of effort demand

In this study, we observed strong relation between reporter's usage status and the type of issues they reported, with established users reporting dominantly bugs, and one-time users and adopters asking more questions. We did not observe significant correlation between issue origin context (as, *e.g.*, commercial environment or personal project) and reporters' usage status. We also did not find a noticeable relation between issue origin and the request type (as, *e.g.*, bug, feature request or question).

While in our sample most issues originated from commercial context and were dominantly discovered by established users of the software, this might not be descriptive of the general effort demand distribution. As we noted before, GITHUB users with empty profiles generate a substantial portion of effort demand. At the same time, the response rate among these users was very low, making them sort of a "dark matter", a massive, yet unobservable, portion of the GITHUB population.

Most of support questions and "beginner issues", like installation related problems, are reported by people with empty profiles. At the same time, many "advanced" issues like non-trivial bug reports and requests for advanced features are reported by people with decorated profiles. A potential explanation can be users' tenure, so that experienced programmers are less likely to look for support and also tend to have stronger profiles to appeal to recruiters, while inexperienced programmers and new GITHUB users are less likely to build appealing public image, while more likely to ask questions. This phenomenon was not the major point of this study, so at this point we can only report it and accept that our sample is potentially biased towards more experienced programmers.

3.7.3 RQ3, signals to differentiate effort demand

After evaluating a set of signals, we did not find a single one that could decisively differentiate effort demand requests by their origin context or reporter's usage status, but we found there is a set of medium strength signals that can be used jointly. One of the inherent challenges in differentiating issue sources is the large amount of users with empty profiles. For these users, we can only use signals coming from the issue report itself, but cannot derive reporter's affiliation, position and usage status, which might be critical for resources negotiation.

Another challenge to issue differentiation is interpretation of the signals. Many signals are non-linear in nature, making them less fit for traditional classification models like logistic regression. However, many more reporters, regardless of their background, have small share of private activity here and there, thus making means of these distributions very close. In this situation, humans will easily identify the few "commercial" reporters, while linear models will likely put the decision boundary somewhere between the two means, rather than discarding low confidence predictions. With this example in mind, signals to differentiate between issue sources should not be merely extracted, but engineered, taking such interpretation into account.

3.7.4 Implications

Questions, as we observed, make a substantial portion of effort demand in some projects. They are especially common in projects closer to the end of the dependency chain, serving to end users or end programmers, sometimes potentially enough to challenge those projects' sustainability. Popular end

programmer projects, such as `numpy` and `scipy` offloaded them to community by moving questions to STACK OVERFLOW, which seems to be an effective measure to conserve maintainers' effort.

However, moving support to STACK OVERFLOW is not an option for end user projects, as it handles exclusively programming-related questions. While it might not be a problem for StackExchange, the network STACK OVERFLOW belongs to, to create another site dedicated to end user support, a broader problem is that there might not be an established expert end user community to use as an effort supply there. Vocalizing the need for such platform and creation of such community is perhaps the first step to improve sustainability of such projects.

In the second part of the study, we explored issue sources in terms of their origin context and reporter's usage status. We discovered that established users dominantly report bugs (as opposed to questions and other types of issues), which are typically given the highest priority among other issue types by maintainers. This finding challenges our initial assumption that for the purpose of resource negotiation maintainers will have better leverage over established users, as more dependent on their projects.

We also found that commercial users more often experience issues by running software in uncommon environments, *e.g.*, a combination of a specific OS and dependency versions. Maintainers reported to give less priority to such niche cases as they only affect a handful of users. The implication of these observations is that perhaps the ideal point for resource negotiation might be niche bugs and feature requests, rather than any requests reported by established users representing large organizations.

In the third part, we found that there are no signals that can decisively differentiate issues by their sources, but there is a set of medium strength signals that can be used jointly. However, there are some inherent challenges to using these signals. First, they aren't free to obtain, *e.g.*, counting the amount of private contributions in a GITHUB user's profile is not trivial. Second, these signals aren't always available – there are a lot of users with empty profiles and absolutely no activity except this reported issue. Finally, these signals need interpretation – we need to use a tool, or at least a bit of human judgement, to come to a conclusion about the source of the issue.

A much easier way, however, would be to simply ask issue reporters. In fact, in one of the studied projects an issue template included a question about reporter's experience with the library. With enough projects doing this, it might be a perfect opportunity to mine user status at scale to evaluate more signals. There are still issues with self-reporting, *e.g.*, users might not be willing to disclose their commercial affiliation if they know maintainers might request some resources in exchange for addressing their request. Doing the reverse, however, *i.e.*, creating an "express lane" for requests from sponsoring organizations, might actually encourage users to self-report.

Finally, a rather philosophical question raised by this study is, whether Open Source Software can be interpreted as a *public good*? There is definitely a lot of similarity – just as a public good, OSS is non-excludable, and non-rivalrous, *i.e.*, anyone can access and one person using it does not prevent others from using, somewhat like public roads. However, as we observed in this study, while the software itself can be copied almost indefinitely, the maintainers effort is fundamentally *subtractive* [123], *i.e.*, resource units used by one party are not available to others.

As open-source practitioners have started to argue,[13] maintenance effort for open-source infrastructure should be more naturally thought of as a *common pool resource* (CPR) [122] than a public good, in that the effort spent by maintainers addressing any particular request, fixing bugs, writing documentation, or adding new features is rivalrous. A classical example of a CPR is a shared grazing area [131] — it is rivalrous in the sense that overuse by one consumer reduces the ability of others to consume the resource; if the pasture is subject to excessive grazing, it may erode and yield less benefit to its users. In our case, since the supply

[13] See blog posts by C. Titus Brown http://ivory.idyll.org/blog/2018-oss-framework-cpr.html and Nadia Eghbal https://nadiaeghbal.com/tragedy-of-the-commons

of maintenance effort is inherently limited (*e.g.*, by the number of qualified contributors and the number of hours in a day), effort consumed by one consumer (*e.g.*, addressing a particular request generated by some user) may similarly reduce the ability of open-source contributors to address other maintenance tasks.

The CPR nature of maintenance effort in open-source increases the likelihood that selfish but rational "tragedy of the commons" [67] behavior would occur, where all with access to a jointly held resource would over-appropriate and under-provision (*i.e.*, insufficiently contribute back), eventually destroying the resource. As shown in Figure 3.1, there are signs of such behavior occurring — most appropriators of maintenance effort (*i.e.*, issue reporters) in the open-source Python PyPI projects do not participate in the provision of the resource by making project contributions (*i.e.*, commits). In our case, similarly to excessive grazing, it can be expected that over-consumption of maintenance effort may negatively affect the sustainability of individual projects (*e.g.*, through maintainer burnout) and, by extension, of the open-source ecosystem as a whole.

Yet when viewed through the Elinor Ostrom's theoretical framework of CPR [122], a substantial body of research points the way toward understanding on how to achieve sustainability. Ostrom undertook a review of field studies and found communities that had in some cases sustained CPRs for years, decades, or even generations. From this analysis, she induced eight design principles that characterized communities that successfully managed CPRs [122]. Among these eight, three seem most directly and immediately related to sustaining open-source maintenance effort: (1) *boundaries*, *i.e.*, distinction between those who are regarded as having the right to appropriate maintenance effort, *e.g.*, whose pull requests and issues should be attended to quickly, and those whose attempts to appropriate effort are considered less legitimate; (2) *congruence between provision and appropriation*, *i.e.*, rules and norms about what entities are expected or required to contribute effort must match the effort needed by the project under the existing conditions of use and change; and (3) *monitoring*, *i.e.*, actively audit the CPR conditions and behavior of appropriators.

Even in these study, we could see some project maintainers applying these principles. For example, adoption of restrictive licenses such as SSPL by MongoDB and other projects can be viewed as a straightforward case of establishing boundaries to whom can and cannot appropriate maintainers' effort. In our study, we could observe establishment of much softer boundaries in company-run projects (P4, P9), prioritizing internal requests above all others. In the same vein, maintainers' own interests (*i.e.*, needs of those who provide resources) have higher priority than needs of those not contributing to the effort pool, which can be viewed as a mean of alignment provision and appropriation of effort as a CPR.

Evidently, maintainers also pay a great deal of attention to the state of project resource sustainability, whereas the three main sustainability factors named by maintainers (company support, size and involvement of the core team, and amount of external contributions) can be directly translated to the size and state of supplied effort, in a similar way that as CPRs are monitored by their maintainers.

The observations made in this study suggest that CPR theory, perhaps with some adjustments, can be at least at some extent applied to OSS. This might help to reuse an existing body of research to develop new sustainability practices and improve the existing ones.

3.8 Conclusions

We conducted a study of issue sources in OSS projects, using a combination of survey and data mining. We evaluated signals that can be used to differentiate between types of appropriators and their project adoption status. We also surveyed OSS project maintainers about current ways of effort demand prioritization.

Perhaps, the biggest finding of this study is that popular projects taking a specific role in the dependency chain, namely end user and end programmer software, are specifically prone to be overwhelmed with effort demand, especially in form of questions. While end programmer projects can offload questions to

community on other platforms, such as STACK OVERFLOW, there is no equivalent support platform for end user projects

Another finding of this study is that established users dominantly report bugs, while new adopters and one time users mostly ask questions. This might allow to infer reporter's usage status, which is usually not visible, from the type of issue they report.

We observed a special type of *non-users*, which report bugs discovered by autotests. In our sample, four of five such users were Linux distribution maintainers, serving as an effective proxy for a much larger user base.

Finally, we theorized about possible ways current effort management practices can be improved to increase sustainability of OSS projects. The main idea in this part is that, perhaps, Open Source software should not be treated as a public good, but rather as a Common Pool Resource due to the subtractive nature of maintenance effort.

3.9 Threats to validity

Just as any other qualitative study on a limited sample, this work has very limited statistical power in representing effort demand distribution. It is conceived as a case study to generate hypotheses, and will require quantitative analysis on a much larger scale to confirm them. Two particular challenges of this study are possible self-selection bias and possible differences in practices of the studied software ecosystem.

We observed that GITHUB users with empty profiles generate a substantial portion of effort demand. The response rate among these users was was three times lower comparing to other users, which might indicate we are dealing with self-selection bias. In this case, interview responses will not be representative of the general population. We tried to avoid interpreting descriptive demographics and limited our analysis only to interaction between different factors, which is less susceptible to such bias. With development of more robust signals to differentiate issue sources, we might be able to overcome this limitation in the future.

This study was performed on PyPI projects, and might generalize to other ecosystems. While no factors inherent to PyPI only were reported, it is plausible that practices in other ecosystems might differ to the point to challenge the conclusions of this study. To generalize beyond Python, this study needs to be reproduced on other ecosystems.

For now, we note these issues as potential threats to validity and leave them for future work.

Chapter 4

Estimating dependency factors

4.1 Abstract

There are many applications of Machine Learning in Software Engineering (SE) where it is difficult to engineer salient features, but possible to collect large amounts of data and resort to Deep Learning (DL) instead. However, researchers too often blindly reuse approaches developed in other domains, such as Natural Language Processing (NLP), without accounting for the specifics of SE tasks. Finding similar and competing libraries, or matching developers and projects based on the libraries they use is particularly hard, since libraries do not follow the same patterns as words in a sentence.

In this work we focus on the problem of generating vector representations ("embeddings") for software libraries that display desirable properties given a task. This is an underexplored area in SE research with many promising applications. Given four representative tasks with fundamentally different characteristics, we systematically explore how architectural decisions in a range of DL-based and traditional models producing embeddings for software libraries affect the usefulness of the resulting embeddings across the four tasks.

We found that models tailored specifically for library embedding outperform those traditionally used for Natural Language Processing and code-level tasks. In particular, deep neural models trained with constrained embedding norm achieve superior performance in similarity based tasks. In some other cases, we observed that simple heuristics achieve performance comparable to neural models, suggesting these task do not require DL tools.

4.2 Introduction

In the modern world, programmers rarely write programs from scratch. Instead, they actively reuse existing libraries. Modern programming languages usually facilitate such reuse by providing access to public repositories of software libraries, such as Python Package Index (PyPI) in Python or Node Package Manager (NPM) repositories in JavaScript. Some of these libraries are niche and small, while others might stretch across multiple packages and can be further extended by programmers, creating sub-ecosystems of packages often used together, aka "*stacks*". Often, hiring requirements for Software Engineers include familiarity with such "stacks", *e.g.*, `react`, `vue` or `angular` in JavaScript, or `django` in Python.

Matching software libraries, developer profiles and software projects is non-trivial, and we could really use help of statistical analysis and Machine Learning. However, here we face a problems: we are working with sparse, high dimensional input and potentially unbound vocabulary, which are hard to deal with. In

this work, we will focus on application of Machine Learning models transforming these sparse inputs into smaller, real valued vectors - embeddings - for software libraries, developer profiles, and software projects.

Historically, researchers in SE derived embeddings, be it code fragments [3], variable or method names [85], or software libraries [146], by reusing approaches previously developed for NLP tasks. Many programming languages are reasonably similar to English language, so these approaches often produce decent results. Modeling software libraries, however, is different, as library use and adoption follow different patterns than words in a sentence.

Unlike words or code tokens (*i.e.*, variable or method names), which can be used multiple times in one statement, libraries can be adopted only once. In this sense, recommending libraries is more similar to recommending movies, rather than completing a sentence. In addition, library names are rarely semantic, *i.e.*, we cannot make any assumptions about relation between libraries with similar names, while in NLP we can easily tell words "scholar" and "scolarship" are related. In natural language, word order matters - and thus, we have a family of NLP models using "attention" mechanism. Library imports, however, are order-agnostic. Thus, one might expect NLP models reused without accounting for these differences to perform suboptimally in SE tasks.

In addition to differences in input data patterns, applications of library embeddings in SE also have different underlying assumptions of similarity. For example, mining competing projects is quite different from finding synonyms or antonyms. While use of antonyms in the same sentence is quite common, using two competing stacks in the same project is, at the very least, unusual. At the same time, analogies mining, reflecting real world relations in word pairs, such as finding a capital by country name, despite being the workhorse of NLP tasks have fairly limited application in SE domain. To achieve better performance, architecture of embedding models should be optimized to account for input patterns and primary usage scenarios of the produced embeddings.

To select a better matching embedding model, it is important to understand what properties of the model will match the task better. Even within NLP domain, models trained on the same data but based on different assumptions often show different performance in different tasks. For example, word embeddings produced by topic modeling (LDA, LSA) are rarely used for analogies mining, while skip-gram models are rarely used for topic modeling. In this work, we are exploring the effects of architectural decisions made in embedding models for software libraries on their performance in different SE tasks.

We found that autoencoders, a family of deep neural models not commonly used in NLP or code-related tasks, outperform other embedding models in similarity based tasks, *e.g.*, finding competing or complementary libraries. Contrary to our expectations, embeddings trained on software projects performed at least comparable to those trained on developer profiles. This allows to avoid relatively expensive developers profile mining and achieve the same result with simple project sampling. Finally, in some tasks we could see that naive models performing almost as well or even slightly better than embedding models. This suggests that these tasks do not require sophisticated DL tools and can be accomplished with simple heuristics.

The contributions of this work are: (1) methodological recommendations on embedding model choice for different Software Engineering tasks (2) pretrained embedding models used in this study, ready to be used for practical applications (3) a set of benchmarks to measure performance of new software libraries embedding models.

4.3 Background and Related Work

Discrete entities represented by words in natural language, source code tokens in a programming languages, or package names in import statements defining relationships among packages, make it difficult to define

topology that is often needed in various prediction and recommendation tasks. What should be the next token, who should I hire as a developer, which library best fits my needs? All of these questions require some notion of distance or similarity that, generally, is elusive in the discrete world. To solve this issue the natural language processing community developed ways to represent such discrete entities as real-valued vectors that can be easily compared and otherwise manipulated. This mapping from the discrete space of terms to a linear space represented by vectors is referred to as embedding as the discrete entity is "embedded" or represented in the vector space.

For example, while the natural language is split into words, programming source code may be split into *tokens*, representing a statement, an identifier, or their parts [85]. The full set of possible tokens makes a *vocabulary*, assigning each token a unique numeric identifier. For example, training a model from the code statement:

```
print("Hola!"),
```

we might end up learning a vocabulary:

{"`print`", "`(`", "`"`", "`Hola!`", "`)`"}.

Using this vocabulary, a naive vector representation of any token would be $X = (w_1, \ldots, w_V)$, where V is the vocabulary size and w_i is an indicator variable showing whether i-th entry is present in our sample. In other words, a naive encoding would use one dimension for each word in the vocabulary. Since the number of words is five, the dimensionality of the space would also be five with "`print`" corresponding to vector $(1, 0, 0, 0, 0)$, "`(`" to $(0, 1, 0, 0, 0)$, and so on. In our example, the complete input sequence will be:

```
[(1,0,0,0,0), (0,1,0,0,0), (0,0,1,0,0),
 (0,1,0,0,0), (0,0,1,0,0), (1,0,0,0,0)].
```

In practice, this input will be extremely sparse, meaning that at each step the sequence will contain only one positive number and a lot of zeroes. For example, we found over 100K top-level package names across a large collection of open source Python projects. That means that to encode an import statement in a Python file we would need a vector with one non-zero value and with over 100K zeroes.

Dealing with so sparse and high-dimensional data is rather impractical. Fortunately, many alternative low-dimensional embeddings can be constructed. While any vector representation of discrete quantities is an embedding, not all embeddings are useful. Vast literature exists on how to produce embeddings that are effective for certain tasks. Typically, the key requirement is that "similar" embedded entities also have similar representations. Since the representations are real vectors, a variety of distance functions can be conveniently computed for any pair. For example, cosine distance (dot product of normalized vectors) is one of the most commonly used measures and it represents the cosine of the angle between two vectors.

In this work we decompose the design space of software development tasks where computing distances between libraries, projects, and developers could be useful, and discuss how these task requirements map into the different approaches to construct embeddings that are likely to be effective for the task. In particular, we focus on neural embeddings, which have become standard in NLP. Before exploring in detail the design space for these embeddings, we start with an overview of the main approaches to constructing them and discuss the software engineering related work using them.

4.3.1 Traditional vs Neural Embeddings

Over the years, researchers proposed multiple dimensionality reduction methods to produce embeddings, including Primary Component Analysis (PCA), Latent Semantic Indexing (LSI), Latent Dirichlet Allocation (LDA), Skip-gram, autoencoders, etc. These ways to reduce dimensionality were developed having different objectives in mind and thus serve them better than others. For example, PCA finds a linear projection of

input data maximizing the variance in the projected space. LSI makes entities used in similar context close in the low dimensional space. LDA was designed to decompose document-term matrix into a smaller set of topics, where each term contributes to probability of document belonging to certain topics.

One fundamental difference between approaches to constructing embeddings is whether they take distance between inputs into account or not. In NLP, the intuition is that far-away tokens may be less relevant than nearby ones, therefore a model may perform better if it only pays attention to words that are close in the text. Typical examples are the neural embedding models based on a shallow-window approach to reading in their (tokenized) inputs (*e.g.*, Skip-gram [108], Continuous Bag of Words, CBoW [107]). For example, with a window of size three, the sentence "I think, therefore I am" is first transformed into snapshots: "I think therefore", "think therefore I", "therefore I am". Each snapshot is then transformed into a set of all word pair combinations, so the snapshot "therefore I am" will produce the pairs "(therefore, I)", "(therefore, am)" and "(I, am)". The finite and typically short width of the sliding window ensures that only neighboring tokens are considered.

Shallow-window models are based on a single-layer neural network with linear activation on the hidden layer [107, 108]. The output of such a neural network is effectively a function of the dot product of weights from the hidden and the output layers, which allows one to interpret the weights corresponding to input units as embeddings.

Deep models, such as autoencoders are also based on neural networks. As Figure 4.3 illustrates, the high-level architecture of these model classes is fairly similar. In contrast, instead of taking as input pairs from a sliding window over the text, autoencoders take the full input as a bag of words [72]. Similarly, models based on explicit factorization such as LSI [95], feed the whole text into the model, ignoring distance between tokens.

The primary software artifact represented by prior work is source code. Many high level programming languages were modeled after natural languages; sometimes, statements in these languages can be read as a plain English. It was even demonstrated that source code has even higher regularity than natural languages [71]. Not surprisingly, models previously developed for NLP were successfully applied to a wide range of code-related tasks, such as code generation [65, 84], bug detection [129], establishing program equivalence [69, 120], etc [20, 83].

Historically, deep models were mostly used for dimensionality reduction given their inability to incorporate distance between words in text. Instead, shallow-window models were typically preferred in NLP. However, as we argue in this paper, shallow-window models aren't necessarily the obvious choice for all library embeddings.

Pretty much all embedding models ultimately perform reduced rank matrix factorization. For example, LSI is performing factorization $M = U\Sigma V$, where M is a $W \times D$ document term frequency matrix, U is a $W \times T$ word-topic matrix, Σ is a $T \times T$ diagonal topic importance matrix, V is a $T \times D$ document topics distribution, W is the vocabulary size, D is the number of documents and T is the number of topics. In this example, we can use U entries for word embeddings, and V entries for document embeddings. Such factorization can be performed explicitly through singular value decomposition or by optimizing a certain objective (SVD-L, Glove [125]), or using stochastic optimization methods based on neural networks (Skip-gram, Continuous Bag of Words (CBoW), Doc2vec [97, 107, 108]).

An important advantage of neural methods is that while explicit factorization requires full retraining with updated vocabulary, it has been shown that neural methods can incorporate new vocabulary entries by simply running few training steps on data including new entries [85]. If we can show both types of embedding models (*i.e.*, performing explicit factorization and neural embeddings) show similar performance, neural embeddings might be preferred for their ability to incorporate new vocabulary entries.

4.3.2 Related Work on Embeddings in Software Engineering

Library embeddings. The first work on library embeddings, Import2vec, was published in 2019[146]. Authors used Skip-gram model to train embeddings for Python and JavaScript software libraries, based on a set of popular packages from GitHub. This work provides some reasoning about library popularity and how to handle common and rare ("surprising") libraries in a library recommendation task. However, this work only provided limited analysis of practical tasks embeddings could be applied to, did not consider other embedding models, and used only limited benchmarks to measure the embeddings quality.

Dey et al[46] applied various embedding models (LSI, Doc2vec) to produce embeddings of developer profiles for expertise mining. This work demonstrates the quality of embeddings by showing a positive relation between alignment of developer profile and technology stack in embedding space with self-reported expertise in those stacks. Along with other practical applications, this work also demonstrates how alignment of developer profile and a project in embeddings space can be used to predict the chance of pull request acceptance.

Modes of similarity. There is an existing body of work on different modes of similarity in NLP, differentiating between "similarity", "relatedness", "association", etc.[2, 89] This corresponds well to multiple notions of libraries similarity we observe in this study, namely semantic similarity, statistical similarity, competition etc.

Wainakh et al evaluated performance of several source code embedding models using a new benchmark of variable identifiers [160]. This study also looks into multiple modes of similarity, using human-annotated data as the ground truth.

Library adoption. A study of competing projects in Java was conducted by Decan et al[40]. This is a case study of three competing Java libraries coexisting in software projects rather than replacing each other. While this did not address the aspect of technical dependency between these libraries (*i.e.*, for technical reasons, library A could not be used without library B, and B could not be used without C), its conclusions fully apply to complementary projects. Using the three libraries case, this work explores the order and evolution of library adoption, and, among other things, shows that less popular libraries, providing higher level abstractions, are adopted later in the project lifecycle.

Source code embeddings. Hoang et al used Hierarchical Attention Networks to produce embeddings of changesets in software projects [74]. They demonstrated how this bidirectional RNN can be used to improve log message generation, defect prediction, and patch identification. Another bidirectional RNN, developed by Hu et al and producing contextual code embeddings, was demonstrated to achieve state-of-the-art results in code search task [76].

4.4 Tasks

Many practical applications involve finding best matches based on software packages. The notion of "best match", however, varies depending on the task nature. This section introduces four representative tasks, which we will use throughout our study, that illustrate this variability in applications. Later, in Section 4.5, we discuss how the fundamental differences between these tasks lead to different properties of embeddings and different design decisions.

To reason about different notions of similarity, we also need to understand nuances of similarity properties, required by different tasks. For example, to offer "often bought together"-style recommendations, we might want to identify libraries often used in a given context, which implies a statistical[1] notion of

[1]The NLP literature sometimes refers to this notion as "relatedness" or "paradigmatic association" [84, 89].

Table 4.1: Overview of the example tasks in our study and their main characteristics.

Task	Input(s)	Output(s)	Ordered inputs	Notion of similarity	Symmetry
Discover competing libraries	• Given library • Set of candidate libraries	Ranked list of candidate libraries	No	Semantic	Yes
Discover complementary libraries	• Given library • Set of candidate libraries	Ranked list of candidate libraries	No	Statistical	No
Recommend a new library to a developer	• Ordered list of existing libraries • Set of candidate libraries	Ranked list of candidate libraries	Yes	Gradual shift from statistical to semantic	–
Match a developer to a project	• Set of existing project libraries • Set of developer profiles	Ranked list of developer profiles	No	Semantic	–

similarity. Note that this type of similarity is not commutative $A \sim B = B \sim A$, as, for example, using the Python library `django-annoying` strongly implies using `django` as well, while the reverse is often not true. Similarly, using the plotting library `matplotlib` implies using `numpy` (`matplotlib` uses `numpy`'s data format), but often `numpy` is used with other visualization libraries or without any at all.

In another example, if we are looking to offer a replacement library, we need to find a library with similar or exactly the same functionality. This functional similarity is not based on statistics and is more of a semantic nature. This notion of similarity is nearly symmetric, with only rare exceptions.

Understanding such properties allows us to inform model choices in a principled way. For example, given two embedding models, one producing symmetric embeddings (*i.e.*, the same embedding is used to compare A to B and vice versa), and one asymmetric (so, producing two sets of embeddings per vocabulary item), we can assume that the former one has only limited capacity in tasks involving statistical similarity.

Table 4.1 shows where in this design space each task resides.

4.4.1 Discover Competing Libraries

Modern software projects actively reuse public open-source libraries, often through package manager repositories, *e.g.*, PyPI for Python and NPM for Javascript. These libraries are often maintained by the community and released without any warranties. As a result, libraries completely abandoned by the community or lacking capacity to timely address discovered issues can pose a security threat for software projects relying on them. Therefore, projects dealing with such risky dependencies might need to identify replacements offering the same functionality. Such information on competing libraries can also be of interest to individual developers exploring potential alternatives for adoption, or to researchers studying how competition within an open source ecosystem impacts the ecosystem's sustainability, just to name a few.

With these applications in mind, we define library A as *competing* with B if A operates in the same domain and can completely replace B in a software project. For example, the Python web frameworks `flask` and `bottle` seem interchangeable. In contrast, the Python data manipulation libraries `pandas` and `dask` do not — even though both provide similar functionality, `dask` is intended for distributed processing while `pandas` performs in-memory manipulations; these libraries operate in different "marketing niches".

Manual identification of competing projects requires quite a bit of expertise and a lot of manual effort.

However, with the right embeddings, we could replace this with a simple comparison of vectors, allowing one to explore a large number of candidates efficiently (potentially, all available packages) for functional fitness.

Thus, one possible formulation of this task is: given a focal library A and a set of possible candidates (*i.e.*, all other libraries), return a ranked list of top-k candidates most likely to be competitors of A.

4.4.2 Discover Complementary Libraries

"Technology stacks" are sets of compatible projects forming around popular libraries, offering to extend their functionality. These extension libraries sometimes form extended sub-ecosystems within the package structure, stretching over thousands of packages. Developers might be interested in identifying and reusing such satellite libraries [11] revolving around a major framework, to avoid reimplementing existing functionality. Such information on complementary libraries can also be of interest to library maintainers and researchers, since larger communities of interdependent libraries may be more sustainable in the long run [153]. Finally, library maintainers might pay special attention to complementary libraries to avoid breaking shared interfaces [14].

We define library B as *complementary* to A if B extends the functionality of A, while still focusing on the same application. For example, `flask-admin` is complementary to the `flask` web application framework as it extends `flask` with a reusable admin interface that can be added to an existing data model. Similarly, `pandas` extends the functionality of `numpy` arrays by adding the ability to mix column types and to use symbolic indexes. In contrast, the library for astronomic calculations `astropy`, while offering similar functionality through `astropy.table`, is focusing on astronomy and rarely used for data manipulation. Thus, under our definition `astropy` is not complementary to `numpy`, but `pandas` is.

The task formulation is similar to the one above: given a focal library A and a set of possible candidates, return a list of top-k candidates most likely to be complementary to A.

A naive approach to discovery of complementary packages would be to maintain statistics of packages co-usage. However, even in the simplest case, under an independence assumption, it is a fairly large array of data with a size of $O(N^2)$, where N is the number of available packages. Maintaining a set of embeddings allows one to reduce the amount of stored data to $O(N)$, and can also take more complex relations between libraries into account.

4.4.3 Recommend a Next Library

Libraries and frameworks are used throughout most programmers' code. Moreover, while programmers of past decades might have been able to learn a single framework, *e.g.*, Java's SDK, and be set for years, today, new libraries and frameworks appear continuously. Thus, programmers at all levels, from novices to experts, must continuously spend significant time learning new APIs [116]. In this sense, programmers may be interested in what trajectories they could follow to learn new libraries more effectively, given what they already know. Researchers in education and programming languages might be interested in modeling and understanding such trajectory data as well, to design better curricula or more natural, relatable abstractions. Finally, tech writers could use such information to tailor documentation and draw analogies.

The subtle, yet important difference in these applications is that while identification of competing or complementary libraries is static, recommending a next library to a developer to learn needs to take the history and tenure of that developer into account. For example, we might expect a data scientist to move from libraries operating entirely in-memory to more advanced distributed, yet niche, solutions (*e.g.*, from `scikit-learn` to `tfx`), but not to start directly with the latter.

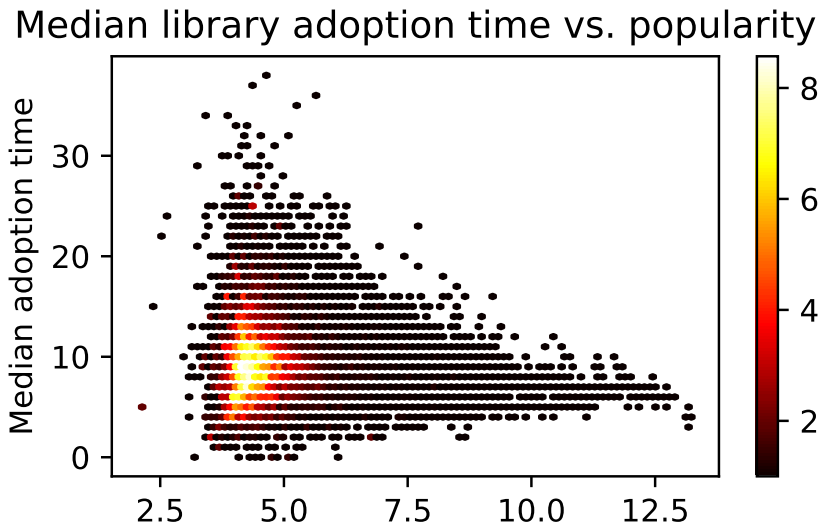

Figure 4.1: Median time of library adoption by developers.
Horizontal axis shows library popularity as a log number of users,
vertical axis shows median adoption time in months, color indicates number of libraries.

Library adoptions by projects are similar. An established project is unlikely to fully change its technology stack, but it may well adopt specialized software to meet more advanced requirements, such as multi-site deployment or advanced logging, depending on what major components of its stack it has adopted most recently.

In both cases, the libraries used early in one's career or in a project's lifecycle are less relevant and thus should contribute less or not contribute to the recommendation for a next library at all. At the same time, *non-use* of a library might also carry important information, *e.g.*, indicating a technical constraint in a project. Figures 4.1 and 4.2 illustrate well that library adoption is time-dependent in practice: in our dataset (detail in Section 4.6.1) popular libraries tend to be adopted early in project lifecycle or developer career, while less popular, niche libraries, have much higher variation in adoption times.

We define a developer's (or project's) *profile* at time t as the time-ordered list of libraries used up until t. Then, assuming a goal of modeling developer (or project) library use trajectories, we can formulate the task as: given a developer's profile at time t and a set of possible candidates (*i.e.*, all other libraries not part of that profile), return a ranked list of top-k candidates most likely to be used next, at time $t + 1$.

4.4.4 Match Developers to Projects

Recruitment of software developers remains an important challenge both in open source and in industry. However, identification of developers matching exactly the project context can be similar to searching for a needle in a haystack. Out of all possible domains and technology stacks, ideally, we need to find people who are familiar with the exact tools, libraries, frameworks, *etc.* we use. Often direct match is not even possible, *e.g.*, if the libraries we use are new and there just isn't enough of a usage history. In such cases, we might want to make an approximate match. For example, the two Python web frameworks `flask` and `bottle` have a high degree of similarity, therefore hiring a `flask` developer to a `bottle` project is likely an acceptable match. However, experience with `flask` may not help as much in a machine learning project using `tensorflow`.

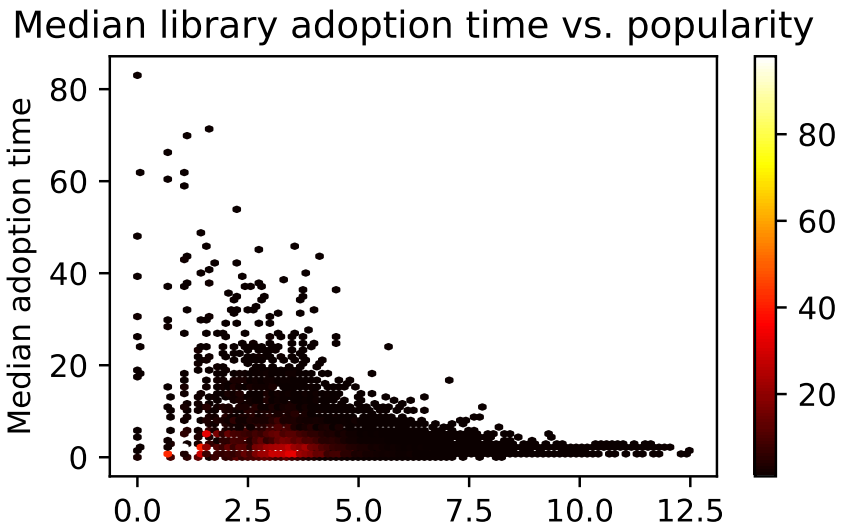

Figure 4.2: Median time of library adoption by projects.
Popular libraries tend to be adopted earlier in project's lifecycle,
while less popular ones can be adopted at any point.

Ignoring, for simplicity, the temporal dimension,[2] one possible task formulation is: given the set of libraries currently in use in a given project, and a set of candidate developers (each with their set of known libraries), return a ranked list of top-k candidates whose past experience best matches the project.

4.5 Design Space

As discussed in Section 4.4, we have two critical variations that may influence the design decisions on constructing the embeddings: (1) the need to model the dynamics and (2) the nature of similarity. Furthermore, since we are borrowing the methods for embeddings from other domains and since, unfortunately, the design decisions involved in the construction of embeddings are often implicit, it is important to consider how, if at all, the borrowed methods might apply for our selected software engineering tasks.

Vector Norm: Fixed vs Unconstrained. First, a key step in solutions addressing all tasks is measuring how different two embedding vectors are. In the simplest case, we can use cosine similarity to measure the collinearity of two embedding vectors:

$$S(i,j) = \frac{w_i \cdot w_j}{|w_i||w_j|},$$

where $S(i,j)$ denotes the cosine similarity of entries i and j, w_i and w_j are their corresponding embedding vectors, and $|w_i|, |w_j|$ are embedding norms.

However, neural embedding models, such as Skip-gram and Doc2vec, will optimize for embedding dot product:

$$P(j) = \sigma(w_i \cdot w_j) = S(i,j)|w_i||w_j|,$$

[2]In practice one would also consider time, as recent experience with some library is likely more important than past experience with it.

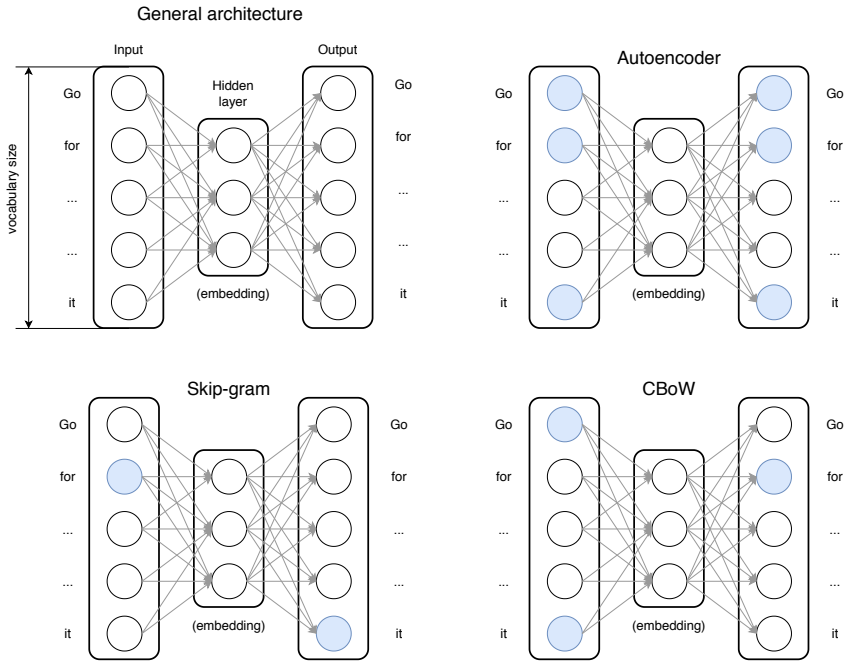

Figure 4.3: Architecture of shallow-window neural embedding models. The blue nodes indicate active inputs and outputs in different architectures for a text window "Go for it".

where σ is the activation function used by the neural network (more about this in Section 4.6.3). Without special measures, this allows the neural network to increase vector norm instead of cosine similarity, which will lead to suboptimal embedding performance. For example, if the word "cruel" was used in a text corpus 1000 times with the word "king" and only 100 times with "tyrant", without such penalty "cruel" will be closely associated with "king", even if "king" is only "cruel" sometimes while "tyrant" is always "cruel".

While this might be a desirable property in prediction tasks, such as our next library recommendation, as discussed above, it is likely harmful for tasks relying on semantic properties of embeddings, such as our competing library discovery.

There are multiple approaches to penalize frequent entries and avoid this behavior, specific to the different classes of models. Shallow window models in NLP use subsampling to even out the frequency of words in the training sample [107]. Models using explicit factorization use a sublinear transformation of the word frequency matrix, *i.e.*, applying some function that grows slower than linear to word frequencies. Common choices for this transformation include binarization (effectively used by Skip-gram), log (Glove embeddings), and normalization (TFIDF). Finally, similar effects can be achieved by autoencoders by imposing additional constraints on vector norms, *e.g.*, by restricting them to unit length.

Therefore, based on this discussion one can expect that:

H$_6$. *Embedding models with fixed embedding norm perform better in tasks based on semantic similarity than unconstrained models.*

Models Based on Recurrent vs Deep Neural Networks. As discussed above, in tasks such as measuring the semantic similarity between libraries, there is no notion of sequence. However, in others, such as library adoption, the order of "tokens" does matter. An intuitive example from NLP is two phrases, "drink milk, not beer" and "drink beer, not milk", which contain the same words, but due to the differences in order have opposite meaning. Models producing embeddings for an ordered sequence of tokens, such as

LSTM and GRU, are usually based on Recurrent Neural Networks (RNNs) [21, 75] and are dominant in code-related tasks where order obviously matters [20]. While incorporating order seems like a reasonable improvement, these models are not applicable to tasks involving out-of-context comparison, *e.g.*, measuring various modes of similarity between tokens. We might expect though that use of RNN-based models can boost performance on tasks taking sequences as input. Given our four tasks, one can hypothesize that:

H$_7$. *Sequential embedding models outperform those based on shallow window or bag-of-words approaches in library adoption prediction tasks.*

Training Data: Developer Profiles vs Projects. It has been shown in NLP that words with opposite meanings are often close in embedding space due to a similar usage context [89]. So, both "hot" and "cold" are used to describe the same objects, thus making them almost as interchangeable as synonyms. Often this is also true for software libraries, *e.g.*, Python ML frameworks like `tensorflow` and `pytorch` both maintain data-level compatibility with `numpy` arrays, and are often used in combination with the same tools for data preprocessing (*e.g.*, image scaling). In other cases, however, competing projects form different "stacks", forming distinct ecosystems of tools used in conjunction with these libraries. For example, two Python web frameworks, `flask` and `django` both have a plethora of plugins to extend their features, but these plugins are not interchangeable between frameworks. In the absence of shared context, and presumably never used simultaneously in the same project, such competing libraries might appear very distant in embedding space, despite their semantic similarity (*e.g.*, in the pioneering work on library embeddings, `django` and `flask` appear to be pretty distant from each other in Figure 12 in the original paper [146]). The same projects, however, are more likely to be used by the same developers. An experienced Python web developer is likely to be familiar with both frameworks; the same way Machine Learning engineers are likely to be fluent with both `tensorflow` and `pytorch`. Therefore, by training embedding models on developer profiles rather than software projects, we might expect competing projects to have higher similarity:

H$_8$. *Embedding models trained on developer profiles render competing projects more similar than those trained on software projects.*

4.6 Methodology

To test our hypotheses, we assemble a novel set of benchmarks using data mined from the World of Code (WoC) [101] infrastructure, implement models from each class discussed above, and compare them across our example tasks. This section describes how we collected data, built models, and evaluated them on the different tasks. We released all code artifacts used to obtain and preprocess input data, train models, and run benchmarks on GitHub.[3]

4.6.1 Input Data

We used WoC [101] to get training data for all our embedding models. This dataset contains raw Git objects (*i.e.*, commits, directory structure, and file content) from nearly all projects ever published on GitHub, BitBucket, GitLab and other code hosting platforms. On top of raw Git objects, this dataset also offers a lot of metadata, including import statements parsed from Python, JavaScript, Perl, Java, C/C++, etc. We limited models to use Python import statements in this study to get smaller and more interpretable models. Python was chosen as a diverse language with clearly identified libraries, meaning that Python import

[3]Shared anonymously at https://github.com/si3nv/ICSE2021

statements relate to software libraries (unlike file-based imports in C and Java), and its focus is not heavily skewed to just one application domain (*e.g.*, as JavaScript is skewed to web development).

To build a vocabulary of valid namespaces, we used several filtering steps. First, to filter out import statements referring local files rather than software libraries, we only considered top-level namespaces of published packages, *i.e.*, from an import statement "`import matplotlib.pyplot as plt`" we only consider the `matplotlib` part[4]. We further eliminated namespaces used by less than 100 developers, as there is not enough usage context to train a meaningful embeddings of these packages. This step substantially reduced our vocabulary, since the majority of packages are not used by anybody except their owners. Besides that, we found that over 20% of packages are invalid (*i.e.*, have inappropriate package structure or impossible to import) or do not provide any namespaces (console scripts). As a result, we obtained 13265 vocabulary entities, less than 10% the number of published Python packages.

Using this vocabulary, we created a timeline of library adoption by developers and projects. WoC provides a list of project URL, Git commit ID, author ID, timestamp, and libraries used in the commit. We aggregated these data into a timeline of library adoptions (*i.e.*, the first time the library was used) by developers and software projects. To account for small inconsistencies in adoptions order, introduced by merging of different branches, we aggregated adoptions by calendar month. At the end for each developer and project we got a sequence of $(month, adoptions)$ pairs.

In line with the common practices, we removed outliers caused by shared developer IDs (*e.g.*, `root@localhost`), bots and single-use IDs produced by tools like svn2Git, by omitting the 0.5% of developer records with the highest and the lowest numbers of commits. For the same reason, we eliminated 0.5% of projects with the highest activity. We also removed all activity before 2008, as it predates most package repositories and is likely to be irrelevant for modern development. Finally, we eliminated records having only one record of library adoptions.

The resulting project records were split into training, validation and test sets in a 80-10-10 proportion. The training set was later used to train embedding models, and the validation set was used to monitor training progress. The test set was further narrowed down to 100,000 project records to make a model performance benchmark. Developer records were split in a similar way, except that we took an extra step to make sure developers contributing to the projects in the test set were not included in the training data.

4.6.2 Benchmarks

We started by analyzing the suitability for our tasks of the two benchmarks, analogical reasoning and library prediction, released together with Import2vec [146], the pioneering prior work on software library embeddings. As we describe next, we found neither of those benchmarks to be fully applicable to our study, and compiled a new set instead.

First, the Import2vec *analogical reasoning* benchmark consists of fourteen two-pair records. It is built by analogy to NLP benchmarks, testing ability of a model to answer questions of a form: "A is to B as C to ??". Often embedding properties allow for vector operations in embedding space to answer such questions, by finding the closest token D such that $embed(D) \approx embed(A) - embed(B) + embed(C)$. In NLP, this was showcased by identifying capitals given a country name (Figure 2, [108]). The same mechanics were argued to be applicable to discover the same static files packaged for use by different JavaScript frameworks. While the property measured by this benchmark might be desirable for models

[4]The term *namespace* here refers to the name used to import a library, as opposed to its name in the package repository. For example, in import statements a Machine Learning package "scikit-learn" is referred using namespace "sklearn".

optimized for semantic similarity, the use of this technique is limited to a few dozen libraries. Given the limited size, the results may not generalize well to more realistic practical applications.

Second, the Import2vec *prediction* benchmark uses average embedding of software project libraries to reconstruct library names, by predicting most used libraries in N closest neighbor projects in embedding space. Our preliminary experiments suggest that this design might be overly sensitive to the benchmark size. With enough projects, the probability of finding an exact equivalent project (*i.e.*, the same libraries) becomes high enough that even a random model shows decent performance; indeed, tested on 100,000 Python projects, we obtained 88% accuracy with a random model. While Import2vec scored higher (92%), as expected, we concluded that this benchmark may not show enough discriminative power to test well the quality of the many embedding models we compare.

Instead, we propose the following three new benchmarks:

Library similarity benchmark. To measure ability of embedding models to enforce different modes of similarity (**H6**), we collected 100 pairs of competing Python libraries, 100 complementary, and 100 orthogonal (*i.e.*, not directly related) library pairs. Projects pairs were sampled from a community-maintained list of Python libraries, organized by topics[5] [166]. Library pairs from the same category were checked to match the definition of competition and complementarity; orthogonal pairs were sampled from unrelated categories. While we tried to be very specific in our definitions, there is certainly some gray area. For example, a built-in template engine used by web framework `django` can be used to replace templating library `jinja2`, but also `jinja2` is often used together with `django` to improve its performance. To measure the quality of selected relations, the resulting list was blindly re-coded by an independent researcher. The resulting Cohen's kappa agreement score of 0.81 indicates strong, yet not exactly perfect, agreement [106].

The intended measure in this benchmark is Cohen's D [27] of angles between pairs embedding vectors in different relations. Intuitively, we want to see how much a relation between library pair affects the alignment of their embedding vectors. It won't be enough to just use alignment of library pairs in a certain relation because a trivially bad model can assign the same embedding vector to all tokens. A good embedding model, on the other hand, will produce weakly aligned vectors for orthogonal libraries, and much better aligned vectors of library pairs carrying some mode of similarity. For a fair comparison of models we suggest comparing angles between embedding vectors rather than their cosine similarity. Otherwise, due to non-linearity of cosine function, models with stronger or weaker enforcement of negative relations (which results in some degree of alignment even for unrelated term vectors) are compared on different parts of cosine curve, thus artificially reducing or inflating variation of vector alignments in different groups. So, the metric we use in this benchmark is expressed as:

$$d = \frac{\bar{A_C} - \bar{A_O}}{s},$$

where $\bar{A_C}$ is the mean angle between competing or complementary library embeddings, $\bar{A_O}$ is the mean angle between embedding vectors of orthogonal libraries, and s is the pooled standard deviation of two samples.

Library adoption by developers and projects. We can expect that order of adoptions carries important information in this task. The order of library adoption in this task can be used to test **H7**, expecting RNN-based models to outperform order-unaware models in predicting the next adopted library.

To measure model performance in this benchmark, we use top-k accuracy, where k is a small number, *e.g.*, $1 \ldots 5$. In this task, there are often multiple reasonable library recommendations to adopt. Offering a

[5] https://awesome-python.com/

short list of libraries, rather than making a one-shot prediction, makes sense both when recommending libraries to developers and projects. Besides that, often developers and projects demonstrate "bursts" of adoptions, starting to use multiple libraries in a short period of time. This could happen with complementary libraries that are best used together, or when libraries are adopted as a part of a "technology stack". We calculate the top-k accuracy as a percentage of adoptions of at least one of the top-k predicted libraries.

Developer recruitment by software projects. Assuming developers joining a software project use the same, or similar libraries as the project itself, we can use this to test **H6**.

Using WoC data, we sampled 100,000 projects with at least two contributors from the adoption timelines test set. Then, we selected one developer at random, except the first contributor, as a positive example. We recorded libraries used by the project before the selected developer's first contribution (positive developer profile), and libraries used by the developer before this moment. As a negative example, we chose a random Python developer using the same number of libraries. As a result, we obtained a set of records *(project libraries, positive profile, negative profile)*.

We propose to use Cohen's d of angles between embedding vectors as a performance metric of model's ability to differentiate between who join a projects comparing to a random developer.

4.6.3 Embedding models

To test our hypotheses, we compare a set of autoencoder models, an LSI model, and an RNN-based model. We also use a pretrained shallow window Skip-gram model (Import2vec) and a task-dependent naive baseline (explained in Section 4.7). All models were trained to produce embedding vectors of length 100 to make a fair comparison. For each model, we briefly describe its architecture and main assumptions.

Neural models.

We use a previously published model based on Skip-gram architecture, Import2vec, as a shallow window model. The architecture of shallow window models is explained in Section 4.3.1. These models are intended to produce only token-level (*i.e.*, library) embeddings. To produce developer or software project embeddigns, we used averaging of token vectors, as suggested by NLP literature [111, 165].

Contrasting to shallow window models, deep neural models take full set of tokens as an input, ignoring their order. It might not seem like a problem with software libraries, where order of import statements does not matter, but it actually makes subsampling technique, used by shallow window models to penalize common entries, inapplicable. To deal with common entries, deep neural models can be trained using explicit restrictions on embedding vectors, *e.g.*, by projecting layer weights onto an n-dimensional sphere after each training step, where n is the embedding dimensionality. While there are efficient greedy ways to train deep autoencoders[7], for better interpretability and result comparison in this study we limit model depth to a single hidden layer with linear activation. This allows us to use weights from input and output layers as library embeddings, while also getting project or developer representation from the content of hidden layer.

In this study, we train and compare four autoencoder models, using all combinations of training on software project and developer profiles (to test **H8**), with and without embedding norm constraint (to test **H6**).

Models based on explicit matrix factorization. Similar to deep neural models, models using explicit matrix factorization do not take word order or distance between tokens in input text into account. This limits their application for word-level text analysis in NLP; historically, these models were used for text-level tasks, such as topic modeling and information retrieval[10, 95]. Working with sets of libraries used by developers and projects, we are free of such limitations. However, it is not clear how performance of models using explicit factorization will compare to neural models.

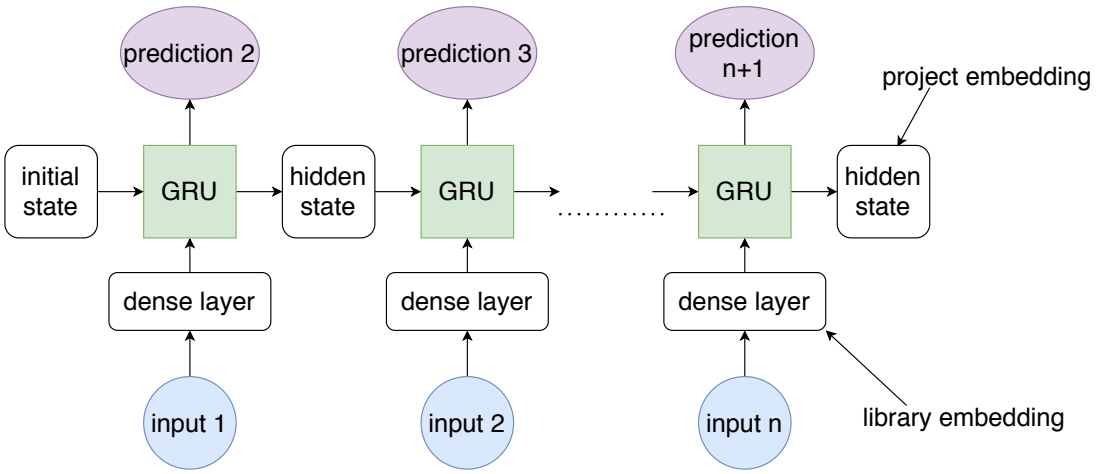

Figure 4.4: Architecture of the recurrent neural network.

We chose Latent Semantic Indexing [43] as a well studied model with known properties. This model is based on the assumption that words having similar meaning are used in a similar context, *i.e.*, surrounded by the same words. It uses Singular Values Decomposition (SVD) on document term matrix to produce reduced rank matrix factorization $M = U\Sigma V$ (more details in Section 4.3.1), where document is a short passage of text under analysis. In this model, we can use entries of matrix U as library embeddings, while also deriving new document (*i.e.*, developer or project) from known M, U, Σ. It is expected that deep autoencoder model will perform at least comparable to LSI.

To test **H8**, in this study we use two instances of LSI model, trained on developer profiles and software projects.

RNN-based models. Recurrent Neural network models, such as Long Short Term Memory (LSTM) and Gated Recurrent Unit (GRU) are behind state-of-the-art models in code-level tasks[69, 83, 84, 85]. These neural networks maintains a hidden state, which is updated on new input and used to produce predictions. This architecture allows to take order of input terms into account, and can track more distant connections between terms in the input data than models with a fixed-size input window[21, 75]. This allows to use RNN models to test **H7**.

In this study we use RNN-based Gated Recurrent Unit (GRU) model. This type of RNN shows performance comparable to LSTM [22, 85], while being more computationally efficient. In addition to GRU, we added an extra trainable embedding layer on top of one-hot input vector, to be used in library similarity benchmark. The overall architecture of this model is shown on Figure 4.4.

4.7 Results

Library similarity benchmark. The results of similarity benchmark are presented in Table 4.2. The naive baseline is based on normalized Levenshtein distance [171], a share of the original string that needs to be edited to get the target. This is based on assumption that related projects might have similar name, *e.g.*, as `django` and `django-annoying`.

In this benchmark, deep neural models trained with embedding norm constraints performed the best in detecting both competing and complementary projects. This supports **H6**, that constraining embedding vectors norm in deep models improves performance in similarity based tasks. The fact that these models

Table 4.2: Difference in similarity of project pairs, unrelated project vs. competing or complementary projects.

Cell values represent Cohen's d. All values are significant at $p < 0.0001$ level.

Model	competing vs orthogonal	complementary vs orthogonal
Autoencoder_dev_norm	0.66	**1.12**
Autoencoder_dev	0.26	0.88
Autoencoder_proj_norm	**0.75**	1.05
Autoencoder_proj	0.17	0.71
RNN_proj	0.41	0.37
LSI_proj	0.64	0.88
LSI_dev	0.57	0.76
Import2vec	0.46	0.49
Naive	0.12	0.55

Table 4.3: Accuracy of project library adoption predictions.

topN metrics indicate accuracy counting a hit if any of top N predictions was actually adopted.

Model	top1	top3	top5
Autoencoder_proj_norm	4.0%	9.3%	12.5%
Autoencoder_proj	3.6%	7.9%	11.0%
Autoencoder_dev_norm	3.1%	7.0%	9.9%
Autoencoder_dev	3.8%	7.6%	9.7%
RNN_proj	**18.6%**	**27.3%**	**36.5%**
LSI_proj	3.1%	6.4%	8.6%
LSI_dev	3.7%	7.6%	10%
Import2vec	9.8%	20.7%	30.4%
Naive baseline	16%	27%	32%

also performed the best to find complementary libraries suggests that, perhaps, this task is more based on semantic similarity, rather than statistical one.

Pretrained shallow window model, Import2vec, performed worse than autoencoders trained with norm constraint and model using explicit factorization. This suggests that models taking full input, rather than using shallow window approach, perform better on software libraries embeddings.

We can also see that properly trained neural models, with their ability to dynamically accommodate for new vocabulary entities, perform better than models using explicit factorization (LSI). This suggests that indeed, neural models might be the best fit for library embeddings.

Contrary to **H8**, models trained on project data, rather than developer profiles, consistently show better performance in competition detection, Possible explanations for this effect are: (1) many competing projects maintain certain level of compatibility and often are used with the same libraries (*e.g.*, as both `tensorflow` and `pytorch` are often used with arrays provided by `numpy`), and (2) besides bringing competing projects together, developer profiles also join many unrelated projects. This suggests that for the purpose of training library embedding models, it makes more sense to sample software projects than perform relatively expensive mining of developer profiles.

Table 4.4: Accuracy of developer library adoption predictions.

topN metrics indicate accuracy counting a hit if any of top N predictions was actually adopted.

Model	top1	top3	top5
Autoencoder_proj_norm	5.4%	12.4%	17.5%
Autoencoder_proj	4.7%	10.7%	14.9%
Autoencoder_dev_norm	4.4%	9.8%	13.8%
Autoencoder_dev	4.1%	8.3%	11.0%
RNN_proj	**18.9%**	**27.7%**	**36.9%**
LSI_proj	3.6%	7.8%	10.7%
LSI_dev	4.3%	9.3%	12.5%
Import2vec	9.6%	22.2%	30.4%
Naive baseline	17.0%	28.6%	33.6%

Table 4.5: Difference in similarity of developers joining project vs random developer.

Cells represent Cohen's d of similarity values. All p-values are indistinguishable from zero.

Model	Cohen's d
Autoencoder_dev_norm	**0.48**
Autoencoder_dev	0.25
Autoencoder_proj_norm	**0.48**
Autoencoder_proj	0.26
LSI_proj	0.23
LSI_dev	0.35
Import2vec	0.26
Naive baseline	**0.50**

Library adoption benchmark. The results of library adoption benchmarks are presented in Tables 4.3 (library adoption by projects) and 4.4 (by developers). The naive baseline always predicts the most common libraries, excluding those already in use by the project or developer.

We can see that indeed, the best performance is demonstrated by the RNN-based model. This technically supports **H7**, since RNN-based models do outperform non-sequential models in adoption tasks. However, RNN-based model has only marginal advantage over the naive baseline. A possible explanation for this is that most library adoptions happen early in project lifecycle or developer career, and involve mostly popular libraries. This suggests that many library adoptions are, in some sense, trivial, and do not require sophisticated machinery to aid developer choices. For practical purposes, future work should focus more on "non-trivial adoptions", by training and testing embedding models on adoptions made later in the project or developer timeline.

Developers recruitment benchmark. The results of developers recruitment benchmark are presented in Table 4.4. The naive baseline is based on simply counting share of project libraries previously used by the developer.

In this benchmark, models based on autoencoders trained with embedding norm constraint outperformed other embedding models. However, a slightly better result is achieved by the naive model. This is likely to happen due to the fact that developer embeddings represent their whole expertise, with extra knowledge effectively decreasing alignment with project embedding, while in real life extra expertise is not harmful

and might be in fact an advantage. One possible solution in this case is to use different similarity function than cosine distance; however, we leave design of such similarity measure for future work.

4.8 Conclusions and future work

In this work, we evaluated multiple designs of embedding models on benchmarks representing four different tasks in Software Engineering. We compared a pretrained model Import2vec, based on popular in NLP shallow window approach, autoencoders trained on various input data with and without embedding norm constraint, an RNN-based model, and a model using explicit matrix factorization. Model performance was evaluated in identification of competing and complementary libraries, recommending next library for adoption, and matching developers to projects.

We found that deep autoencoders trained with embedding norm constraint outperformed all other embedding models in similarity based tasks, namely in detection of competing and complementary projects, and matching developers to projects. This family of models is not traditionally used in NLP or code tasks, but seems to suit well to software libraries domain.

We found that contrary to our expectations, embeddings trained on software projects performed at least comparable to those trained on developer profiles. This has major implications for training of future models, since sampling software projects is a lot easier than collecting full developer profiles.

In many tasks we could see that a naive model performed almost as well, or even slightly better than embedding models. In recommending next library task, predicting the most common but not yet used libraries strategy was only slightly worse than RNN-based solution. This suggests that often even a simple tool is enough, and, perhaps, that we should direct future effort on predicting less trivial library adoptions.

In developer-project matching, the naive model, simply counting how many of project libraries developer used before, slightly outperformed the top scoring embedding model. We attribute this to insufficiency of traditional cosine similarity used in this test, to perform such matching. We leave development of such measure for the future work.

Things that were left out of this study due to lack of resources, include: finding optimum embedding dimensionality; exploring applicability of bi-directional RNNs; possible suboptimal model parameters and architecture; non-deterministic nature of stochastic training algorithms.

Bibliography

[1] Rabe Abdalkareem, Olivier Nourry, Sultan Wehaibi, Suhaib Mujahid, and Emad Shihab. Why do developers use trivial packages? an empirical case study on npm. In *Proc. Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 385–395. ACM, 2017. 2.2, 2.3

[2] Eneko Agirre, Enrique Alfonseca, Keith Hall, Jana Kravalova, Marius Pasca, and Aitor Soroa. A study on similarity and relatedness using distributional and wordnet-based approaches. 2009. 4.3.2

[3] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):40, 2019. 4.2

[4] Morten Andersen-Gott, Gheorghita Ghinea, and Bendik Bygstad. Why do commercial companies contribute to open source software? *International Journal of Information Management*, 32(2): 106–117, 2012. 3.3.1

[5] John Anvik, Lyndon Hiew, and Gail C Murphy. Who should fix this bug? In *Proceedings of the 28th international conference on Software engineering*, pages 361–370. ACM, 2006. 3.3.2

[6] Guilherme Avelino, Leonardo Passos, Andre Hora, and Marco Tulio Valente. A novel approach for estimating truck factors. In *Proc. International Conference on Program Comprehension (ICPC)*, pages 1–10. IEEE, 2016. 2.3

[7] Yoshua Bengio, Pascal Lamblin, Dan Popovici, and Hugo Larochelle. Greedy layer-wise training of deep networks. In *Advances in neural information processing systems*, pages 153–160, 2007. 4.6.3

[8] Tegawendé F Bissyandé, David Lo, Lingxiao Jiang, Laurent Réveillere, Jacques Klein, and Yves Le Traon. Got issues? who cares about it? a large scale investigation of issue trackers from github. In *2013 IEEE 24th international symposium on software reliability engineering (ISSRE)*, pages 188–197. IEEE, 2013. 3.3.3

[9] Jürgen Bitzer, Wolfram Schrettl, and Philipp JH Schröder. Intrinsic motivation in open source software development. *Journal of comparative economics*, 35(1):160–169, 2007. 3.3.1

[10] David M Blei, Andrew Y Ng, and Michael I Jordan. Latent dirichlet allocation. *Journal of machine Learning research*, 3(Jan):993–1022, 2003. 4.6.3

[11] Kelly Blincoe, Francis Harrison, and Daniela Damian. Ecosystems in GitHub and a method for ecosystem identification using reference coupling. In *Proc. International Conference on Mining Software Repositories (MSR)*, pages 202–207. IEEE, 2015. 2.3, 4.4.2

[12] Bradley C Boehmke and Benjamin T Hazen. The future of supply chain information systems: The open source ecosystem. *Global Journal of Flexible Systems Management*, 18(2):163–168, 2017. 2.3

[13] Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. How to break an API: cost negotiation and community values in three software ecosystems. In *Proc. International*

Symposium on Foundations of Software Engineering (FSE), pages 109–120. ACM, 2016. 2.2, 2.3, 2.5.1

[14] Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. How to break an api: cost negotiation and community values in three software ecosystems. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 109–120, 2016. 3.3.2, 4.4.2

[15] Terry Bollinger et al. Use of free and open-source software (foss) in the us department of defense. *Mitre Corporation Rept.# MP*, 2:W0000101, 2003. 1

[16] Hudson Borges and Marco Tulio Valente. What's in a github star? understanding repository starring practices in a social coding platform. *Journal of Systems and Software*, 146:112–129, 2018. 2

[17] Jan Bosch. From software product lines to software ecosystems. In *Proc. International Software Product Line Conference (SPLC)*, pages 111–119, 2009. 2.3

[18] Andrea Capiluppi, Patricia Lago, and Maurizio Morisio. Characteristics of open source projects. In *Proc. European Conference on Software Maintenance and Reengineering (CSMR)*, pages 317–327. IEEE, 2003. 2.2, 2.3

[19] Casey Casalnuovo, Bogdan Vasilescu, Premkumar Devanbu, and Vladimir Filkov. Developer onboarding in GitHub: the role of prior social links and language experience. In *Proc. Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 817–828. ACM, 2015. 2.3

[20] Zimin Chen and Martin Monperrus. A literature study of embeddings on source code. *arXiv preprint arXiv:1904.03061*, 2019. 4.3.1, 6

[21] Kyunghyun Cho, Bart Van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014. 6, 4.6.3

[22] Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014. 4.6.3

[23] Malgorzata Ciesielska and Ann Westenholz. Dilemmas within commercial involvement in open source software. *Journal of Organizational Change Management*, 29(3):344–360, 2016. 4

[24] Maëlick Claes, Mika Mäntylä, Miikka Kuutila, and Umar Farooq. Towards identifying paid open source developers-a case study with mozilla developers. *arXiv preprint arXiv:1804.02153*, 2018. 1, 2

[25] Maëlick Claes, Mika V Mäntylä, Miikka Kuutila, and Bram Adams. Do programmers work at night or during the weekend? In *Proceedings of the 40th International Conference on Software Engineering*, pages 705–715, 2018. 2, 3.5.3

[26] Jailton Coelho and Marco Tulio Valente. Why modern open source projects fail. In *Proc. Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 186–196. ACM, 2017. 2.2, 2.3, 2.4.2, 2.7, 2.9, 3.4, 3.6.1

[27] Jacob Cohen. A power primer. *Psychological bulletin*, 112(1):155, 1992. 3.5.3, 3.6.3, 4.6.2

[28] Jacob Cohen, Patricia Cohen, Stephen G West, and Leona S Aiken. *Applied multiple regression/correlation analysis for the behavioral sciences*. Routledge, 2013. 2.4.3

[29] Stefano Comino, Fabio M Manenti, and Maria Laura Parisi. From planning to mature: On the success of open source projects. *Research Policy*, 36(10):1575–1586, 2007. 2.2, 2.3

[30] Eleni Constantinou and Tom Mens. An empirical comparison of developer retention in the RubyGems and npm software ecosystems. *Innovations in Systems and Software Engineering*, 13(2-3):101–115, 2017. 1, 2.4.3

[31] Eleni Constantinou and Tom Mens. Socio-technical evolution of the ruby ecosystem in github. In *2017 IEEE 24th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 34–44. IEEE, 2017. 1

[32] Valerio Cosentino, Javier Luis Cánovas Izquierdo, and Jordi Cabot. Assessing the bus factor of git repositories. In *Proc. International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 499–503. IEEE, 2015. 2.3

[33] David Roxbee Cox and David Oakes. *Analysis of survival data*, volume 21. CRC Press, 1984. 2.2, 2.4.3

[34] John W Creswell and David J Creswell. *Research design: Qualitative, quantitative, and mixed methods approaches*. Sage publications, third edition, 2017. 2.4

[35] Kevin Crowston, James Howison, and Hala Annabi. Information systems success in free and open source software development: Theory and measures. *Software Process: Improvement and Practice*, 11(2):123–148, 2006. 2.3

[36] Kevin Crowston, Kangning Wei, James Howison, and Andrea Wiggins. Free/Libre open-source software development: What we know and what we do not know. *ACM Computing Surveys (CSUR)*, 44(2):7, 2012. 2.3

[37] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. Social coding in GitHub: transparency and collaboration in an open software repository. In *Proc. ACM 2012 Conference on Computer Supported Cooperative Work (CSCW)*, pages 1277–1286. ACM, 2012. 2.7, 2

[38] Carlo Daffara. Estimating the economic contribution of open source software to the European economy. In *The First Openforum Academy Conference Proceedings*, 2012. 2.2

[39] Jean-Michel Dalle, Paul A David, et al. The allocation of software development resources in 'open source'production mode. *SIEPR-Project NOSTRA Working Paper,(15th February)[Accepted for publication in Joe Feller, Brian Fitzgerald, Scott Hissam, Karim Lakhani, eds., Making Sense of the Bazaar, forthcoming from MIT Press in 2004]*, 2003. 3.3.3

[40] Alexandre Decan, Mathieu Goeminne, and Tom Mens. On the interaction of relational database access technologies in open source java projects. *arXiv preprint arXiv:1701.00416*, 2017. 1, 4.3.2

[41] Alexandre Decan, Tom Mens, and Philippe Grosjean. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering*, 2018. 2.2, 2.3

[42] Alexandre Decan, Tom Mens, and Philippe Grosjean. An empirical comparison of dependency network evolution in seven software packaging ecosystems. *Empirical Software Engineering*, 24(1): 381–416, 2019. 1

[43] Scott Deerwester, Susan T Dumais, George W Furnas, Thomas K Landauer, and Richard Harshman. Indexing by latent semantic analysis. *Journal of the American society for information science*, 41 (6):391–407, 1990. 4.6.3

[44] Bert J Dempsey, Debra Weiss, Paul Jones, and Jane Greenberg. Who is an open source software developer? *Communications of the ACM*, 45(2):67–72, 2002. 4

[45] Tapajit Dey, Yuxing Ma, and Audris Mockus. Patterns of effort contribution and demand and user

classification based on participation patterns in npm ecosystem. In *15th International Conference on Predictive Models and Data Analytics in Software Engineering*. ACM, 2019. 3.3.3

[46] Tapajit Dey, Andrey Karnauch, and Audris Mockus. Representation of developer expertise in open source software. *arXiv preprint arXiv:2005.10176*, 2020. 4.3.2

[47] Nicolas Ducheneaut. Socialization in an open source software community: A socio-technical analysis. *Computer Supported Cooperative Work (CSCW)*, 14(4):323–368, 2005. 2.3

[48] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, et al. The matter of heartbleed. In *Proceedings of the 2014 conference on internet measurement conference*, pages 475–488. ACM, 2014. 2.2, 3.3.3

[49] Steve Easterbrook, Janice Singer, Margaret-Anne Storey, and Daniela Damian. Selecting empirical methods for soft ware engineering research. In *Guide to advanced empirical software engineering*, pages 285–311. Springer, 2008. 2.4

[50] Nadia Eghbal. *Roads and bridges: The unseen labor behind our digital infrastructure*. Ford Foundation, 2016. 1, 2.2, 3.2

[51] Stephen G Eick, Todd L Graves, Alan F Karr, J Steve Marron, and Audris Mockus. Does code decay? assessing the evidence from change management data. *IEEE Transactions on Software Engineering*, 27(1):1–12, 2001. 3.2

[52] Matthieu Foucault, Marc Palyart, Xavier Blanc, Gail C Murphy, and Jean-Rémy Falleri. Impact of developer turnover on quality in open-source software. In *Proc. Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 829–841. ACM, 2015. 2.2

[53] Oscar Franco-Bedoya, David Ameller, Dolors Costal, and Xavier Franch. Open source software ecosystems: A systematic mapping. *Information and Software Technology*, 91:160–185, 2017. 2.3

[54] Felipe Fronchetti, Igor Wiese, Gustavo Pinto, and Igor Steinmacher. What attracts newcomers to onboard on oss projects? tl;dr: Popularity. In *IFIP International Conference on Open Source Systems (OSS)*, pages 91–103. Springer, 2019. 2

[55] Andrew Gelman and Jennifer Hill. *Data analysis using regression and multilevel/hierarchical models*. Cambridge university press, 2006. 2.4.3

[56] Daniel German, Gregorio Robles, Germán Poo-Caamaño, Xin Yang, Hajimu Iida, and Katsuro Inoue. " was my contribution fairly reviewed?" a framework to study the perception of fairness in modern code reviews. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 523–534. IEEE, 2018. 3.2

[57] Mohammad Gharehyazie, Daryl Posnett, Bogdan Vasilescu, and Vladimir Filkov. Developer initiation and social interactions in OSS: A case study of the Apache Software Foundation. *Empirical Software Engineering*, 20(5):1318–1353, 2015. 2.3

[58] Mathieu Goeminne and Tom Mens. Evidence for the pareto principle in open source software activity. In *Proc. International Workshop on Model-Driven Software Migration (MDSM)*, page 74, 2011. 2.3

[59] Jesus M Gonzalez-Barahona and Gregorio Robles. Trends in free, libre, open source software communities: from volunteers to companies. *it–Information Technology it–Information Technology*, 55(5):173–180, 2013. 3.3.1

[60] Georgios Gousios, Martin Pinzger, and Arie van Deursen. An exploratory study of the pull-based software development model. In *Proc. International Conference on Software Engineering (ICSE)*,

pages 345–355. ACM, 2014. 2.3

[61] Georgios Gousios, Margaret-Anne Storey, and Alberto Bacchelli. Work practices and challenges in pull-based development: the contributor's perspective. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*, pages 285–296. IEEE, 2016. 3.2

[62] Patricia M Grambsch and Terry M Therneau. Proportional hazards tests and diagnostics based on weighted residuals. *Biometrika*, 81(3):515–526, 1994. 2.4.3

[63] Simon Grand, Georg Von Krogh, Dorothy Leonard, and Walter Swap. Resource allocation beyond firm boundaries: A multi-level model for open source innovation. *Long Range Planning*, 37(6): 591–610, 2004. 3.3.2

[64] Shane Greenstein and Frank Nagle. Digital dark matter and the economic contribution of Apache. *Research Policy*, 43(4):623–631, 2014. 2.2

[65] Xiaodong Gu, Hongyu Zhang, Dongmei Zhang, and Sunghun Kim. Deep api learning. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 631–642, 2016. 4.3.1

[66] Philip J Guo, Thomas Zimmermann, Nachiappan Nagappan, and Brendan Murphy. Characterizing and predicting which bugs get fixed: an empirical study of microsoft windows. In *Proceedings of the 32Nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 495–504. ACM, 2010. 3.3.3

[67] Garrett Hardin. The tragedy of the commons. *science*, 162(3859):1243–1248, 1968. 3.7.4

[68] Ahmed E Hassan and Richard C Holt. Predicting change propagation in software systems. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.*, pages 284–293. IEEE, 2004. 3.3.2

[69] Jordan Henkel, Shuvendu K Lahiri, Ben Liblit, and Thomas Reps. Code vectors: Understanding programs through embedded abstracted symbolic traces. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 163–174, 2018. 4.3.1, 4.6.3

[70] Guido Hertel, Sven Niedner, and Stefanie Herrmann. Motivation of software developers in Open Source projects: an internet-based survey of contributors to the Linux kernel. *Research policy*, 32 (7):1159–1177, 2003. 4

[71] Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. On the naturalness of software. *Communications of the ACM*, 59(5):122–131, 2016. 4.3.1

[72] Geoffrey E Hinton and Ruslan R Salakhutdinov. Reducing the dimensionality of data with neural networks. *science*, 313(5786):504–507, 2006. 4.3.1

[73] Eric von Hippel and Georg von Krogh. Open source software and the "private-collective" innovation model: Issues for organization science. *Organization science*, 14(2):209–223, 2003. 3.3.1

[74] Thong Hoang, Hong Jin Kang, Julia Lawall, and David Lo. Cc2vec: Distributed representations of code changes. In *International Conference on Software Engineering (ICSE)*. ACM, 2020. 4.3.2

[75] Sepp Hochreiter and Jürgen Schmidhuber. Lstm can solve hard long time lag problems. In *Advances in neural information processing systems*, pages 473–479, 1997. 6, 4.6.3

[76] Gang Hu, Min Peng, Yihan Zhang, Qianqian Xie, and Mengting Yuan. Neural joint attention code search over structure embeddings for software q&a sites. *Journal of Systems and Software*, page 110773, 2020. 4.3.2

[77] IEEE. Ieee standard glossary of software engineering terminology, 1993. 3.3.2

[78] Mazhar Islam, Jacob Miller, and Haemin Dennis Park. But what will it cost me? how do private costs of participation affect open source software projects? *Research Policy*, 46(6):1062–1070, 2017. 3.3.1

[79] Slinger Jansen, Anthony Finkelstein, and Sjaak Brinkkemper. A sense of community: A research agenda for software ecosystems. In *Proc. International Conference on Software Engineering (ICSE) - Companion*, pages 187–190. IEEE, 2009. 2.3

[80] Stephen P Jenkins. Survival analysis. *Unpublished manuscript, Institute for Social and Economic Research, University of Essex, Colchester, UK*, 2005. 2.4.3

[81] Corey Jergensen, Anita Sarma, and Patrick Wagstrom. The onion patch: migration in open source ecosystems. In *Proc. Joint Meeting on Foundations of Software Engineering (ESEC/FSE)*, pages 70–80. ACM, 2011. 2.3

[82] Eirini Kalliamvakou, Daniela Damian, Kelly Blincoe, Leif Singer, and Daniel M German. Open source-style collaborative development practices in commercial projects using github. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*, pages 574–585. IEEE Press, 2015. 3.3.2

[83] Hong Jin Kang, Tegawendé F Bissyandé, and David Lo. Assessing the generalizability of code2vec token embeddings. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1–12. IEEE, 2019. 4.3.1, 4.6.3

[84] Rafael-Michael Karampatsis and Charles Sutton. Maybe deep neural networks are the best choice for modeling source code. *arXiv preprint arXiv:1903.05734*, 2019. 4.3.1, 1, 4.6.3

[85] Rafael-Michael Karampatsis, Hlib Babii, Romain Robbes, Charles Sutton, and Andrea Janes. Big code!= big vocabulary: Open-vocabulary models for source code. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*, 2020. 4.2, 4.3, 4.3.1, 4.6.3, 4.6.3

[86] Leo Katz. A new status index derived from sociometric analysis. *Psychometrika*, 18(1):39–43, 1953. 2.4.2

[87] Jymit Khondhu, Andrea Capiluppi, and Klaas-Jan Stol. Is it all lost? a study of inactive open source projects. In *Proc. IFIP International Conference on Open Source Systems*, pages 61–79. Springer, 2013. 2.2, 2.3

[88] Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. Structure and evolution of package dependency networks. In *Proc. International Conference on Mining Software Repositories (MSR)*, pages 102–112. IEEE, 2017. 2.2, 2.3

[89] Tomáš Kliegr and Ondřej Zamazal. Antonyms are similar: Towards paradigmatic association approach to rating similarity in simlex-999 and wordsim-353. *Data & Knowledge Engineering*, 115: 174–193, 2018. 4.3.2, 1, 7

[90] Andrew J Ko and Parmit K Chilana. How power users help and hinder open bug reporting. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1665–1674. ACM, 2010. 3.3.3

[91] Stefan Koch and Georg Schneider. Effort, co-operation and co-ordination in an open source software project: GNOME. *Information Systems Journal*, 12(1):27–42, 2002. 2.3

[92] Sandeep Krishnamurthy, Shaosong Ou, and Arvind K Tripathi. Acceptance of monetary rewards in open source software development. *Research Policy*, 43(4):632–644, 2014. 3.3.1

[93] Karim R Lakhani and Robert G Wolf. Why hackers do what they do: Understanding motivation and effort in free/open source software projects. 2003. 3.2, 3.3.1

[94] Karim R. Lakhani and Robert G. Wolf. *Why Hackers Do What They Do: Understanding Motivation and Effort in Free/Open Source Software Projects*. 2005. 4

[95] Thomas K Landauer and Susan T Dumais. A solution to plato's problem: The latent semantic analysis theory of acquisition, induction, and representation of knowledge. *Psychological review*, 104(2):211, 1997. 4.3.1, 4.6.3

[96] Meir M Lehman. Programs, life cycles, and laws of software evolution. *Proceedings of the IEEE*, 68(9):1060–1076, 1980. 3.2

[97] Omer Levy and Yoav Goldberg. Neural word embedding as implicit matrix factorization. In *Advances in neural information processing systems*, pages 2177–2185, 2014. 4.3.1

[98] Bin Lin, Gregorio Robles, and Alexander Serebrenik. Developer turnover in global, industrial open source projects: Insights from applying survival analysis. In *Proc. International Conference on Global Software Engineering (ICGSE)*, pages 66–75. IEEE, 2017. 2.4.3

[99] Mircea Lungu, Romain Robbes, and Michele Lanza. Recovering inter-project dependencies in software ecosystems. In *Proc. International Conference on Automated Software Engineering (ASE)*, pages 309–312. ACM, 2010. 2.3

[100] Mircea F Lungu. *Reverse engineering software ecosystems*. PhD thesis, Università della Svizzera italiana, 2009. 2.3

[101] Yuxing Ma, Chris Bogart, Sadika Amreen, Russell Zaretzki, and Audris Mockus. World of code: an infrastructure for mining the universe of open source vcs data. In *Proceedings of the 16th International Conference on Mining Software Repositories*, pages 143–154. IEEE Press, 2019. 12, 4.6, 4.6.1

[102] Konstantinos Manikas and Klaus Marius Hansen. Software ecosystems–a systematic literature review. *Journal of Systems and Software*, 86(5):1294–1306, 2013. 2.3

[103] William T Markham, Margaret A Johnson, and Charles M Bonjean. Nonprofit decision making and resource allocation: The importance of membership preferences, community needs, and interorganizational ties. *Nonprofit and Voluntary Sector Quarterly*, 28(2):152–184, 1999. 3.3.3

[104] Jennifer Marlow and Laura Dabbish. Activity traces and signals in software developer recruitment and hiring. In *Proc. 2013 Conference on Computer Supported Cooperative Work (CSCW)*, pages 145–156. ACM, 2013. 3.3.1

[105] Jennifer Marlow, Laura Dabbish, and Jim Herbsleb. Impression formation in online peer production: activity traces and personal profiles in GitHub. In *Proc. 2013 Conference on Computer Supported Cooperative Work (CSCW)*, pages 117–128. ACM, 2013. 2.7, 2

[106] Mary L McHugh. Interrater reliability: the kappa statistic. *Biochemia medica: Biochemia medica*, 22(3):276–282, 2012. 4.6.2

[107] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781*, 2013. 4.3.1, 4.5

[108] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. Distributed representations of words and phrases and their compositionality. In *Advances in neural information processing systems*, pages 3111–3119, 2013. 4.3.1, 4.6.2

[109] Courtney Miller, David Gray Widder, Christian Kästner, and Bogdan Vasilescu. Why do people give

up flossing? a study of contributor disengagement in open source. In *IFIP International Conference on Open Source Systems*, pages 116–129. Springer, 2019. 3.2, 3.4

[110] Rupert G Miller Jr. *Survival analysis*, volume 66. John Wiley & Sons, 2011. 2.2, 2.4.3

[111] Jeff Mitchell and Mirella Lapata. Composition in distributional models of semantics. *Cognitive science*, 34(8):1388–1429, 2010. 4.6.3

[112] Audris Mockus. Succession: Measuring transfer of code and developer productivity. In *Proceedings of the 31st International Conference on Software Engineering*, pages 67–77. IEEE Computer Society, 2009. 3.3.2

[113] Audris Mockus. Organizational volatility and its effects on software defects. In *Proceedings of the eighteenth ACM SIGSOFT international symposium on Foundations of software engineering*, pages 117–126. ACM, 2010. 3.3.2

[114] Audris Mockus, Roy T Fielding, and James D Herbsleb. Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(3):309–346, 2002. 2.3, 2.4.2

[115] Audris Mockus, David M Weiss, and Ping Zhang. Understanding and predicting effort in software projects. In *25th International Conference on Software Engineering, 2003. Proceedings.*, pages 274–284. IEEE, 2003. 3.3.2

[116] Brad A Myers and Jeffrey Stylos. Improving api usability. *Communications of the ACM*, 59(6): 62–69, 2016. 4.4.3

[117] Kumiyo Nakakoji, Yasuhiro Yamamoto, Yoshiyuki Nishinaka, Kouichi Kishida, and Yunwen Ye. Evolution patterns of open-source software systems and communities. In *Proc. International Workshop on Principles of Software Evolution (IWPSE)*, pages 76–85. ACM, 2002. 2.3

[118] Mathieu Nassif and Martin P Robillard. Revisiting turnover-induced knowledge loss in software projects. In *Proc. International Conference on Software Maintenance and Evolution (ICSME)*, pages 261–272. IEEE, 2017. 2.2

[119] Matus Nemec, Dusan Klinec, Petr Svenda, Peter Sekan, and Vashek Matyas. Measuring popularity of cryptographic libraries in internet-wide scans. In *Proceedings of the 33rd Annual Computer Security Applications Conference*, ACSAC 2017, pages 162–175. ACM, 2017. ISBN 978-1-4503-5345-8. doi: 10.1145/3134600.3134612. URL http://doi.acm.org/10.1145/3134600.3134612. 1

[120] Trong Duc Nguyen, Anh Tuan Nguyen, Hung Dang Phan, and Tien N Nguyen. Exploring api embedding for api usages and applications. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*, pages 438–449. IEEE, 2017. 4.3.1

[121] Felipe Ortega and Daniel Izquierdo-Cortazar. Survival analysis in open development projects. In *Proc. ICSE Workshop on Emerging Trends in Free/Libre/Open Source Software Research and Development*, pages 7–12. IEEE, 2009. 2.4.3

[122] Elinor Ostrom. *Governing the commons: The evolution of institutions for collective action*. Cambridge university press, 1990. 3.2, 3.7.4

[123] Elinor Ostrom. Beyond markets and states: polycentric governance of complex economic systems. *American Economic Review*, 100(3):641–72, 2010. 3.7.4

[124] Jagdish K Patel, CH Kapadia, Donald Bruce Owen, and JK Patel. Handbook of statistical distributions. Technical report, M. Dekker New York, 1976. 2.4.3

[125] Jeffrey Pennington, Richard Socher, and Christopher D Manning. Glove: Global vectors for word representation. In *Proceedings of the 2014 conference on empirical methods in natural language processing (EMNLP)*, pages 1532–1543, 2014. 4.3.1

[126] Yasset Perez-Riverol, Laurent Gatto, Rui Wang, Timo Sachsenberg, Julian Uszkoreit, Felipe da Veiga Leprevost, Christian Fufezan, Tobias Ternent, Stephen J Eglen, Daniel S Katz, et al. Ten simple rules for taking advantage of git and github, 2016. 3.2

[127] Raphael Pham, Leif Singer, Olga Liskin, Fernando Figueira Filho, and Kurt Schneider. Creating a shared understanding of testing culture on a social coding site. In *Proc. International Conference on Software Engineering (ICSE)*, pages 112–121. IEEE, 2013. 2.3

[128] Germán Poo-Caamaño. *Release management in free and open source software ecosystems*. PhD thesis, 2016. 3.3.2

[129] Michael Pradel and Koushik Sen. Deep learning to find bugs. *TU Darmstadt, Department of Computer Science*, 2017. 4.3.1

[130] Gede Artha Azriadi Prana, Christoph Treude, Ferdian Thung, Thushari Atapattu, and David Lo. Categorizing the content of github readme files. *Empirical Software Engineering*, 24(3):1296–1327, 2019. 2

[131] Sebastian Prediger, Björn Vollan, and Markus Frölich. The impact of culture and ecology on cooperation in a common-pool resource experiment. *Ecological Economics*, 70(9):1599–1608, 2011. 3.7.4

[132] Huilian Sophie Qiu, Alexander Nolte, Anita Brown, A. Serebrenik, and Bogdan Vasilescu. Going farther together: the impact of social capital on sustained participation in open source. In *International Conference on Software Engineering*, United States, 12 2018. IEEE Computer Society. 2.2

[133] Huilian Sophie Qiu, Yucen Lily Li, Susmita Padala, Anita Sarma, and Bogdan Vasilescu. The signals that potential contributors look for when choosing open-source projects. In *Proc. ACM Conference on Computer-Supported Cooperative Work and Social Computing (CSCW)*. ACM, 2019. 2

[134] Eric Raymond. The cathedral and the bazaar. *Knowledge, Technology & Policy*, 12(3):23–49, 1999. 2.3

[135] Dirk Riehle, Philipp Riemer, Carsten Kolassa, and Michael Schmidt. Paid vs. volunteer work in open source. In *2014 47th Hawaii International Conference on System Sciences*, pages 3286–3295. IEEE, 2014. 1, 2, 3.5.3

[136] Peter C Rigby, Yue Cai Zhu, Samuel M Donadelli, and Audris Mockus. Quantifying and mitigating turnover-induced knowledge loss: case studies of Chrome and a project at Avaya. In *Proc. International Conference on Software Engineering (ICSE)*, pages 1006–1016. ACM, 2016. 2.2, 2.3

[137] Jeffrey A Roberts, Il-Horn Hann, and Sandra A Slaughter. Understanding the motivations, participation, and performance of open source software developers: A longitudinal study of the apache projects. *Management Science*, 52(7):984–999, 2006. 3.3.1

[138] Gregorio Robles and Jesus M Gonzalez-Barahona. Contributor turnover in libre software projects. In *Proc. IFIP International Conference on Open Source Systems*, pages 273–286. Springer, 2006. 2.2

[139] Ioannis Samoladas, Lefteris Angelis, and Ioannis Stamelos. Survival analysis on the duration of open source projects. *Information and Software Technology*, 52(9):902–922, 2010. 2.3

[140] Walt Scacchi. Is open source software development faster, better, and cheaper than software engineering. In *2nd Workshop on Open Source Software Engineering, Orlando, Florida*, 2002. 3.3.2

[141] Mario Schaarschmidt, Gianfranco Walsh, and Harald FO von Kortzfleisch. How do firms influence open source software communities? a framework and empirical analysis of different governance modes. *Information and Organization*, 25(2):99–114, 2015. 3.3.1

[142] Charles Schweik, Bob English, Qimti Paienjton, and Sandy Haire. Success and abandonment in open source commons: Selected findings from an empirical study of sourceforge.net projects. In *Proc. Workshop on Building Sustainable Open Source Communities (OSCOMM)*, 2010. 2.3

[143] Param Vir Singh, Yong Tan, and Vijay Mookerjee. Network effects: The influence of structural capital on open source project success. *MIS Quarterly*, pages 813–829, 2011. 1, 2.3, 2.4.2

[144] Igor Steinmacher, Gustavo Pinto, Igor Scaliante Wiese, and Marco Aurélio Gerosa. Almost there: A study on quasi-contributors in open-source software projects. In *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*, pages 256–266. IEEE, 2018. 3.2

[145] Katherine J Stewart and Sanjay Gosain. The impact of ideology on effectiveness in open source software development teams. *Mis Quarterly*, pages 291–314, 2006. 3.3.1

[146] Bart Theeten, Frederik Vandeputte, and Tom Van Cutsem. Import2vec learning embeddings for software libraries. In *Proceedings of the 16th International Conference on Mining Software Repositories*, pages 18–28. IEEE Press, 2019. 4.2, 4.3.2, 7, 4.6.2

[147] Terry Therneau, Cindy Crowson, and Elizabeth Atkinson. Using time dependent covariates and time dependent coefficients in the Cox model. *Survival Vignettes*, 2017. 2.4.3

[148] Asher Trockman, Shurui Zhou, Christian Kästner, and Bogdan Vasilescu. Adding sparkle to social coding: an empirical study of repository badges in the npm ecosystem. In *Proc. International Conference on Software Engineering (ICSE)*, pages 511–522. ACM, 2018. 2

[149] Asher Trockman, Shurui Zhou, Christian Kästner, and Bogdan Vasilescu. Adding sparkle to social coding: an empirical study of repository badges in the npm ecosystem. In *Proc. International Conference on Software Engineering (ICSE)*, pages 511–522. ACM, 2018. 2.5.2, 2.7

[150] Jason Tsay, Laura Dabbish, and James Herbsleb. Influence of social and technical factors for evaluating contribution in github. In *Proc. International Conference on Software Engineering (ICSE)*, pages 356–366. ACM, 2014. 2

[151] Jason Tsay, Laura Dabbish, and James Herbsleb. Let's talk about it: evaluating contributions through discussion in GitHub. In *Proc. International Symposium on Foundations of Software Engineering (FSE)*, pages 144–154. ACM, 2014. 2.3, 3.2

[152] Qiang Tu et al. Evolution in open source software: A case study. In *Proc. International Conference on Software Maintenance and Evolution (ICSME)*, pages 131–142. IEEE, 2000. 2.3

[153] Marat Valiev, Bogdan Vasilescu, and James Herbsleb. Ecosystem-level determinants of sustained activity in open-source projects: A case study of the pypi ecosystem. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 644–655. ACM, 2018. 4.4.2

[154] Bogdan Vasilescu, Alexander Serebrenik, and Mark GJ van den Brand. The Babel of software development: Linguistic diversity in Open Source. In *Proc. International Conference on Social Informatics (SocInfo)*, pages 391–404. Springer, 2013. 2.3

[155] Bogdan Vasilescu, Alexander Serebrenik, Mathieu Goeminne, and Tom Mens. On the variation and

specialisation of workload—a case study of the Gnome ecosystem community. *Empirical Software Engineering*, 19(4):955–1008, 2014. 2.3

[156] Bogdan Vasilescu, Vladimir Filkov, and Alexander Serebrenik. Perceptions of diversity on GitHub: A user survey. In *Proc. International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pages 50–56. IEEE, 2015. 4

[157] Bogdan Vasilescu, Kelly Blincoe, Qi Xuan, Casey Casalnuovo, Daniela Damian, Premkumar Devanbu, and Vladimir Filkov. The sky is not the limit: multitasking across GitHub projects. In *Proc. International Conference on Software Engineering (ICSE)*, pages 994–1005. IEEE, 2016. 2.3

[158] Georg Von Krogh, Sebastian Spaeth, and Karim R Lakhani. Community, joining, and specialization in open source software innovation: a case study. *Research Policy*, 32(7):1217–1241, 2003. 2.3

[159] Georg Von Krogh, Stefan Haefliger, Sebastian Spaeth, and Martin W Wallin. Carrots and rainbows: Motivation and social practice in open source software development. *MIS quarterly*, pages 649–676, 2012. 3.2

[160] Yaza Wainakh, Moiz Rauf, and Michael Pradel. Evaluating semantic representations of source code. *arXiv preprint arXiv:1910.05177*, 2019. 4.3.2

[161] Jing Wang. Survival factors for Free Open Source Software projects: A multi-stage perspective. *European Management Journal*, 30(4):352–371, 2012. 2.3, 2.4.2

[162] Morten Warncke-Wang, Vivek Ranjan, Loren Terveen, and Brent Hecht. Misalignment between supply and demand of quality content in peer production communities. In *Ninth International AAAI Conference on Web and Social Media*, 2015. 3.3.3

[163] Michael Wedel, Uwe Jensen, and Peter Göhner. Mining software code repositories and bug databases using survival analysis models. In *Proc. International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 282–284. ACM, 2008. 2.4.3

[164] Joel West and Scott Gallagher. Challenges of open innovation: the paradox of firm investment in open-source software. *R&D Management*, 36(3):319–331, 2006. 3.3.1

[165] John Wieting, Mohit Bansal, Kevin Gimpel, and Karen Livescu. Towards universal paraphrastic sentence embeddings. *arXiv preprint arXiv:1511.08198*, 2015. 4.6.3

[166] Yu Wu, Jessica Kropczynski, Raquel Prates, and John M Carroll. Understanding how github supports curation repositories. *Future Internet*, 10(3):29, 2018. 4.6.2

[167] Xuan Xiao, Aron Lindberg, Sean Hansen, and Kalle Lyytinen. "computing" requirements for open source software: A distributed cognitive approach. *Journal of the Association for Information Systems*, 19(12):1217–1252, 2018. 3.3.2, 3.3.3

[168] Jialiang Xie, Minghui Zhou, and Audris Mockus. Impact of triage: a study of mozilla and gnome. In *2013 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 247–250. IEEE, 2013. 3.3.2

[169] Kazuhiro Yamashita, Shane McIntosh, Yasutaka Kamei, Ahmed E Hassan, and Naoyasu Ubayashi. Revisiting the applicability of the pareto principle to core development teams in open source software projects. In *Proc. International Workshop on Principles of Software Evolution (IWPSE)*, pages 46–55. ACM, 2015. 2.3

[170] Yiqing Yu, Alexander Benlian, and Thomas Hess. An empirical study of volunteer members' perceived turnover in open source software projects. In *Proc. Hawaii International Conference on System Science (HICSS)*, pages 3396–3405. IEEE, 2012. 2.2

[171] Li Yujian and Liu Bo. A normalized levenshtein distance metric. *IEEE transactions on pattern analysis and machine intelligence*, 29(6):1091–1095, 2007. 4.7

[172] Yang Zhang, Yue Yu, Huaimin Wang, Bogdan Vasilescu, and Vladimir Filkov. Within-ecosystem issue linking: a large-scale study of rails. In *Proceedings of the 7th International Workshop on Software Mining*, pages 12–19, 2018. 3.2

[173] Minghui Zhou and Audris Mockus. What make long term contributors: Willingness and opportunity in OSS community. In *Proc. International Conference on Software Engineering (ICSE)*, pages 518–528. IEEE, 2012. 2.3

[174] Minghui Zhou, Audris Mockus, Xiujuan Ma, Lu Zhang, and Hong Mei. Inflow and retention in OSS communities with commercial involvement: A case study of three hybrid projects. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 25(2):13, 2016. 4, 3.2