IDENTIFYING AND CHARACTERIZING UNMAINTAINED PROJECTS IN GITHUB

JAILTON JUNIOR DE SOUSA COELHO

IDENTIFYING AND CHARACTERIZING UNMAINTAINED PROJECTS IN GITHUB

Tese apresentada ao Programa de Pós--Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Doutor em Ciência da Computação.

Orientador: Marco Túlio de Oliveira Valente

Belo Horizonte

Agosto de 2019

JAILTON JUNIOR DE SOUSA COELHO

IDENTIFYING AND CHARACTERIZING UNMAINTAINED PROJECTS IN GITHUB

Thesis presented to the Graduate Program in Computer Science of the Federal University of Minas Gerais in partial fulfillment of the requirements for the degree of Doctor in Computer Science.

Advisor: Marco Túlio de Oliveira Valente

Belo Horizonte

August 2019

© 2019, Jailton Junior de Sousa Coelho.
Todos os direitos reservados.

Coelho, Jailton Junior de Sousa

C672i Identifying and Characterizing Unmaintained Projects in GitHub / Jailton Junior de Sousa Coelho. — Belo Horizonte, 2019
xxiii, 108 f. : il. ; 29cm

Tese (doutorado) - Universidade Federal de Minas Gerais – Departamento de Ciência da Computação.
Orientador: Marco Túlio de Oliveira Valente

1. Computação – Teses. 2. Engenharia de software – Teses. 3. Software – Manutenção – Teses. 4. Software gratuito – Teses. I. Orientador. II. Título.

CDU 519.6*32(043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Identifying and Characterizing Unmaintained Projects in GitHub

JAILTON JUNIOR DE SOUSA COELHO

Tese defendida e aprovada pela banca examinadora constituída pelos Senhores:

[illegible]

Prof. Marco Túlio de Oliveira Valente - Orientador
Departamento de Ciência da Computação - UFMG

Eduardo Figueiredo

Prof. Eduardo Magno Lages Figueiredo
Departamento de Ciência da Computação - UFMG

[illegible]

Prof. André Cavalcante Hora
Departamento de Ciência da Computação - UFMG

[illegible]

Prof. Ricardo Terra Nunes Bueno Villela
Departamento de Ciência da Computação - UFLA

[illegible]

Prof. Igor Fabio Steinmacher
Campo Mourão - UTFPR

Belo Horizonte, 30 de Agosto de 2019.

"If you want to go fast, go alone. If you want to go far, go together."
(African Proverb)

Acknowledgments

I thank to everyone who helped me scientifically and emotionally along this doctoral degree. Especially, I would like to thank to:

God. I thank God for protecting and blessing me much more than I deserve.

My Parents. A special thanks to my mom Luzinete and my dad Jailton, who gave me the emotional support to get through the obstacles in making this happen.

My Wife. Most importantly, I wish to thank my beloved wife Natália. I would like to express my deepest gratitude for your unconditional love and care. Thank you for being the reason I smile.

My advisor. I would like to express my sincere gratitude to my advisor Prof. Marco Tulio Valente for the lessons, attention, availability, and patience. I could not have imagined having a better advisor and mentor.

ASERG research group. I also thank the members of the ASERG research group for the friendship, talks, jokes, cakes and technical collaboration. I would like to thank especially Luciana Silva for their collaboration on the papers and discussions.

Members of my Ph.D. committee. I thank the remaining members of my Ph.D. committee for reviewing this thesis and your insightful comments: Prof. Igor Steinmacher, Prof. Eduardo Figueiredo, Prof. André Hora, and Prof. Ricardo Terra.

DCC - UFMG. I thank the Department of Computer Science at UFMG for its constant support.

Supporting institutions. I thank the research funding agencies FAPEMIG, CAPES, and CNPq for their financial support.

Resumo

Projetos de código aberto são importantes componentes do desenvolvimento de software moderno. Devido ao surgimento de plataformas inovadoras (como o GitHub e o GitLab) para desenvolver e manter código público, milhares de projetos de código aberto têm sido criados. Consequentemente, um número significativo de projetos também estão enfrentando problemas de manutenção. Para mitigar esse problema, reporta-se nesta tese um conjunto de estudos quantitativos e qualitativos para ajudar desenvolvedores a manterem seus projetos. Primeiro, foi perguntado para proprietários de projetos de código aberto abandonados, as razões que os motivaram a interromper a manutenção de seus sistemas. Como resultado, foi obtida uma lista de nove razões que os motivaram a parar de dar manutenção em seus projetos. Segundo, foi aplicado um questionário com desenvolvedores que recentemente se tornaram importantes contribuidores de projetos GitHub populares. Foram reveladas suas motivações para contributir para esses projetos, as características dos projetos que mais os ajudaram a contributir e as principais barreiras enfrentadas por eles. Os principais resultados desse estudo revelam que os desenvolvedores contribuíam porque eles usavam esses sistemas e precisavam de novas funcionalidades. Os participantes também responderam que a falta de tempo dos líderes dos projetos foi a principal barreira enfrentada por eles. Por último, no terceiro estudo, foi criado um modelo de aprendizado de máquina para identificar projetos GitHub sem manutenção. O modelo foi treinado utilizando um conjunto de métricas de atividades de projeto, como *commits*, *forks*, *issues*, etc. O modelo proposto alcançou uma precisão de 80%, segundo respostas de um questionário com os principais desenvolvedores de 129 projetos GitHub e um *recall* de 96%. Foi mostrado também que o modelo pode ser usado para identificar sistemas sem manutenção, sem a necessidade de esperar por um ano de inatividade, como comumente é feito em outros estudos. Finalmente, foram apresentadas evidências da aplicabilidade desse modelo, investigando seu uso em 2.927 projetos ativos.

Palavras-chave: Manutenção de Software, GitHub, Software de Código Aberto.

Abstract

Open source projects are key components of modern software development. Due to the appearance of novel platforms (e.g., GitHub and GitLab) for developing public code, developers has created thousands of open source projects. As a consequence, a significant number of open source projects is also unmaintained. To tackle this problem, in this thesis, we reported a set of quantitative and qualitative studies to help developers to maintain their open source projects. First, we surveyed the owners of open source projects that are no longer actively maintained, aiming to reveal the reasons for stop the maintenance of their projects. As result, we provide a set of nine reasons that motivated them to abandon their projects. Second, we conducted a survey with developers who recently became core contributors of popular GitHub projects. We reveal their motivations to contribute to these projects, the projects characteristics that mostly helped to contribute, and the barriers faced by them. Our key results show that the surveyed developers contributed to the projects because they are using them and need some improvements. The participants also answered that the lack of time of the project leaders was the principal barrier faced by them. Finally, the project characteristic which mostly helped them to contribute was the existence of a friendly community. Finally, in our third study, we propose a quantitative and data-driven model to identify GitHub projects that are not actively maintained. We train the model using a set of 13 features about project activity (e.g., commits, forks, and issues). The model achieved a precision of 80%, based on the feedback of 129 real open source developers and a recall of 96%. We also showed that the model can be used to identify unmaintained projects early, without having to wait for one year of inactivity, as commonly proposed in the literature. Finally, we defined a metric, called Level of Maintenance Activity (LMA), to assess the risks of projects become unmaintained. We provided evidence on the applicability of this metric, by investigating its usage in 2,927 active projects.

Palavras-chave: Unmaintained Projects, GitHub, Open Source Software.

List of Figures

List of Tables

Contents

Chapter 1

Introduction

In this chapter, we state our problem and motivation (Section 1.1). Next, we discuss the objectives and intended contributions of this thesis (Section 1.2). After that, we describe the research questions addressed in this thesis (Section 1.3). Then, we list our publications (Section 1.4). Finally, we present the outline of this document (Section 1.5).

1.1 Problem and Motivation

Although open source has its origins in the eighties (or even earlier) [Raymond, 1999], the movement is experiencing a renaissance period [Lerner and Tirole, 2002]. One of the key reasons is the appearance of modern platforms for developing and maintaining open source projects [Eghbal, 2016]. The most famous example is GitHub, but other platforms are also relevant, such as Bitbucket[1] and GitLab[2]. These platforms deeply changed the collaboration mechanisms in open source software development [Söderberg, 2015; Kalliamvakou et al., 2015; Vasilescu et al., 2015]. Instead of exchanging e-mails with patches, developers contribute to a project by forking it, working and improving the code locally, and then submitting a pull request to the project's leaders [Jiang et al., 2017].

Today over 80% of the software produced in several applications is composed by open source code and this trend is growing.[3] In a recent investigation conducted by Sonatype[4], they report that downloads of npm packages reached 10 billion per week

[1] *https://bitbucket.org/*

[2] *https://about.gitlab.com/*

[3] *https://www.linuxfoundation.org/blog/chaoss-project-creates-tools-to-analyze-software-development-and-measure-open-source-community-health*

[4] *https://www.sonatype.com/2019ssc*

and 21,448 new open source components are releases per day. Today, open source code is used by government, major software companies, startups, and individuals to build software [Goldman and Gabriel, 2005]. For this reason, open source code can be viewed as the backbone of the digital infrastructure that runs our society [Eghbal, 2016]. Furthermore, over the years, open source also contributed to reduction in the costs of building and deploying software [Qiu et al., 2019]. A large number of open source systems is created and maintained by developers and offered to the users for free. On the other side, organizations often rely on open source code to support their basic software infrastructures, including operating systems, databases, web servers, etc. Finally, most software produced nowadays depends on public libraries and frameworks, which are used for example to abstract out the implementation of code related to security, authentication, user interfaces, execution on mobile devices, etc. For example, in a recent survey—conducted by Black Duck Software—86% of the surveyed organizations report the use of open source in their daily development.[5] Just to mention an example, Instagram—the popular photo-sharing social network—has a special section of its site to acknowledge the importance of public code to the company.[6] In this page, they thank the open source community for their contributions and explicitly list 28 open source libraries and frameworks used by the social network.

More specifically, modern source code hosting platforms (e.g., GitHub[7], Bitbucket[8], and GitLab[9]) are changing the way that developers contribute to Open Source Software (OSS) projects. Due to the facilities brought by these services for developing, maintaining, and sharing code, OSS projects are now facing a high exposure, leading to an increasing number of contributors. This fast growing puts additional strain on the developers who maintain this infrastructure. Most OSS communities are composed by a small number of core developers and a substantial number of peripheral ones [Avelino et al., 2016, 2019; Joblin et al., 2017]. The core developers are those developers who are involved with the OSS project for a long time and who make the contributions that guide the development and evolution of the project [Joblin et al., 2017; Mockus et al., 2002]. Peripheral developers are those who sporadically contribute to the project (e.g., via bug reports or fixing documentation issues) [Pinto et al., 2016; Steinmacher et al., 2016; Setia et al., 2012; Lee et al., 2017].

As a result, developers has created thousands of open source projects. For ex-

[5] *https://pt.slideshare.net/blackducksoftware/you-cant-live-without-open-source-results-from-the-open-source-360-survey*

[6] *https://www.instagram.com/about/legal/libraries*

[7] *https://github.com*

[8] *https://bitbucket.org*

[9] *https://gitlab.com*

ample, today GitHub has more than 31 million developers and 100 million repositories (without excluding forks). In face of this fast growth, the costs of not supporting this infrastructure efficiently are critical [Eghbal, 2016]. For example, a recent study conducted by Avelino et al. [2016] shows that 87 systems (65%) out of 133 most actively used GitHub projects, across programming languages, have only one or two developers responsible to their evolution. Therefore, to keep the growth success of OSS communities, we should provide an adequate infrastructure and support for developers to maintain their projects [Fogel, 2005].

By contrast, **a significant number of open source projects is also becoming unmaintained**. Despite this fact, we have few studies that investigate the reasons that motivate developers to stop the maintenance of their projects [Androutsellis-Theotokis et al., 2011]. We only found similar studies for commercial software projects. For example, by means of a survey with developers and project managers, Cerpa and Verner [2009] study the motivation for discontinuation of 70 commercial software projects. They report that the most common reasons are due to unrealistic delivery dates, underestimated project size, risks not re-assessed through the project, and when the staff is not rewarded for working long hours. However, these findings do not apply to open source projects, which are developed without rigid schedules and requirements, by groups of unpaid developers [Kalliamvakou et al., 2015]. The Standish Group's CHAOS report is another study frequently mentioned by practitioners and consultants [Standish Group, 1994]. The 2007 report mentions that 46% of software projects have cost and schedule problems and that 19% are outright failures. Besides possible methodological problems, as pointed by Jørgensen and Moløkken-Østvold [2006], this report does not target open source. Therefore, **a deep understanding of the reasons for the discontinuation of open source projects and the proposal of metrics about their level of maintenance activity can contribute to the long term sustainability of such projects**.

1.2 Objectives

The main goal of this thesis is to characterize the risks faced by the discontinuation of open source software projects. By identifying projects facing such risks, we intend to reveal the principal reasons that motivate developers to stop the maintenance of their projects. We also plan to propose models and metrics to measure the level of maintenance activity of open source projects.

We propose three specific objectives to achieve this main goal, as described next:

1. Motivated by the lack of studies in the literature, we intend to reveal the reasons that motivate developers to stop the maintenance of popular GitHub projects. By means of a survey with maintainers of these projects, we intent to reveal their motivations to abandon their projects. We also intend to assess the importance of following (or not) a set of best open source maintenance practices, which are widely recommended when hosting projects on GitHub. Finally, we intend to discuss and reveal the principal strategies attempted by the maintainers of open source projects to overcome (without success) the discontinuation of their projects.

2. We also intend to reveal the motivations of recent core developers to contribute to open source projects. By means of a survey with these developers, we aim to reveal (i) the motivations that led recent core developers to contribute to OSS projects, (ii) the project characteristics and practices that helped them in this process, and (iii) the barriers faced by such core developers.

3. Finally, we intend to propose a quantitative and data-driven model to identify unmaintained GitHub projects. By means of this model, we expect to identify unmaintained projects without having to wait for one year of inactivity, as commonly proposed in the literature [Khondhu et al., 2013; Chengalur-Smith and Sidorova, 2003; Valiev et al., 2018]. Furthermore, we intend to provide information about the level of maintenance activity of open source projects.

1.3 Proposed Thesis

In this thesis, we propose to investigate three overarching questions related to the sustainability of open source software projects. We start by investigating the reasons for the discontinuation of modern open source projects, including a discussion about the results of a survey with the maintainers of 104 open source projects which became unmaintained, aiming to reveal the developer's reasons to stop the maintenance of such projects (**Q1**). Next, to complement the principal reasons that motivate developers to stop the maintenance of their projects—such as lack of time (17%) and lack of interest (17%) of the main contributor—we reveal the motivations of core developers to assume a key role in OSS projects. This information is relevant for maintainers that are looking out for new developers to take up ownership of their projects (**Q2**). Finally, we propose a data-driven approach to measure the level of maintenance activity of GitHub projects, i.e., a quantitative metric that reveals how actively a project is being maintained (**Q3**). We argue this metric can help users and developers in two ways. First, by alerting users about the risks of using unmaintained projects. Second, as criteria to select an OSS project to contribute to, i.e., by helping volunteers to choose projects that need help in their maintenance.

Q1. Why do modern open source software projects become unmaintained?

OSS communities have grown in scale and importance. For example, today GitHub has more 100 million repositories and 31 million developers from nearly every country, providing across 1.1 billion contributions.[10] As already discussed in Section 1.1, OSS is an essential and fundamental part of software produced by users and organizations. In fact, it is common nowadays to rely on open source libraries and frameworks when building and evolving proprietary software. However, a significant number of open source projects are also becoming unmaintained or abandoned by their maintainers. Despite this fact, we have very few studies that investigate the developer's motivations to stop the maintenance of such projects. Therefore, with this first question *our goal is to reveal the reasons that motivate developers of modern open source projects to stop the maintenance of their projects.*

Q2. What are the key motivations to contribute to open source projects?

In the previous question (Q1), we investigated the reasons that motivate developers of OSS projects to stop the maintenance of their projects. As result, we found that the third and fourth reasons are the lack of time (17%) and lack of interest (17%) of the main contributor. In this second question, *we intend to investigate an opposite situation, i.e., developers who became core contributors of OSS projects.* We reported the main reasons that led them to contribute, the project characteristics and practices that motivated them to engage and the barriers they faced. Our results show the surveyed developers contributed to the projects because they are using them and were demanding some improvements. The participants also answered that the lack of time of the project leaders was the principal barrier they faced. Finally, the project characteristic that mostly helped them was the existence of a friendly community.

Q3. How to identify unmaintained GitHub projects? How to measure the level of maintenance activity of open source projects?

Currently, *GitHub does not provide precise information about the level of main-*

[10] *https://github.blog/2018-11-08-100m-repos/*

tenance activity of their projects. The lack of this information generates two problems. First, projects are becoming unmaintained by the lack of new contributors. Second, users have to judge by themselves whether a project is under maintenance or not (and therefore whether it is worth to use it) only based on popularity metrics, such as number of stars, forks, and watchers [Meirelles et al., 2010; Borges et al., 2016b; Borges and Valente, 2018].

In order to help on the two aforementioned problems, in this final question *we propose and evaluate a machine learning approach to identify unmaintained GitHub projects and to assess the level of maintenance activity of such projects.* By alerting users about the risks of depending on unmaintained GitHub projects, this metric can motivate contributors to assume the maintenance of projects facing the risks of discontinuation.

1.4 Publications

The work described in this thesis includes material from the following publications:

- Jailton Coelho and Marco Tulio Valente. Why Modern Open Source Projects Fail. In *11th Symposium on The Foundations of Software Engineering (FSE)*, pages 186–196, 2017.

- Jailton Coelho and Marco Tulio Valente and Luciana L. Silva and Andre Hora. Why We Engage in FLOSS: Answers from Core Developers. In *11th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pages 114–121, 2018.

- Jailton Coelho and Marco Tulio Valente and Luciana L. Silva and Emad Shihab. Identifying Unmaintained Projects in GitHub. In *12th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–10, 2018.

1.5 Outline of the Thesis

The studies that comprise the core of this thesis were published in a set of software engineering conferences and workshops. Furthermore, the last study (Chapter 4) has an extension currently submitted to a journal. Therefore, the thesis' chapters preserve most of the structure of these manuscripts in order to facilitate their independent reading. Due to this decision, although all chapters have particular contributions, some redundancy can be found in the sections about procedures and methodologies. We organize the remainder of this work as follows:

- Chapter 2 presents the results of a survey with the maintainers of open source projects that have failed, aiming to reveal the reasons for such failures. We consider that a project has *failed* when the documentation explicitly mentions that it is deprecated or the project is no longer under maintenance according to their owner. We provide a set of nine reasons for the failure of these projects. We also show that there is an important difference between the failed projects and the most popular and active projects on GitHub, in terms of following a set of best open source maintenance practices (e.g., the availability of contribution guidelines, issue template, code of conduct, license, etc.). Particularly, this difference is more important regarding the availability of contribution guidelines and the use of continuous integration. Furthermore, the failed projects have a non-negligible number of opened issues and pull requests. Finally, we describe three strategies attempted by maintainers to overcome the failure of their projects.

- Chapter 3 reports the results of a survey with developers who recently became core contributors of popular GitHub projects. We reveal their motivations to contribute to OSS projects, the project characteristics that mostly helped them in this process, and the barriers faced by these new core developers. We also compare our results with related studies regarding of other kinds of open source contributors, e.g., casual contributors [Pinto et al., 2016; Lee et al., 2017], and newcomers [Steinmacher et al., 2016]. Our results show the surveyed developers contributed to the projects because they are using them and need some improvements. The participants also answered the lack of time of the project leaders was the principal barrier they faced. Finally, the project characteristic that mostly helped them to contributed was the existence of a friendly community.

- Chapter 4 presents a machine learning model to identify GitHub projects that are not actively maintained. We also train machine learning models to compute a metric to express the level of maintenance activity of GitHub projects, based on a set of features about project activity (e.g., commits, forks, issues, etc.). We empirically validated the model with the best performance with the principal developers of 129 GitHub projects. The model achieved a precision of 80% and a recall of 96%. We also showed that the proposed model can identify unmaintained projects early, without having to wait for one year of inactivity, as commonly proposed in the literature. Finally, we proposed a metric, called Level of Maintenance Activity (LMA), to assess the risks of projects become unmaintained. We provide evidences on the applicability of this metric by investigating its usage in 2,927 projects. Finally, we implemented a public Chrome extension to indicate the level of maintenance activity of GitHub projects.

- Chapter 5 concludes this thesis and outlines future work ideas.

Chapter 2

Why Modern Open Source Projects Fail

Open source projects are key elements of the digital infrastructure that runs our society today. Moreover, open source is experiencing a renaissance period, due to the appearance of modern platforms and workflows for developing and maintaining public code. As a result, developers has created thousands of open source projects. As a consequence, a significant number of open source projects is also failing. We consider that a project has failed when the documentation explicitly mentions that it is deprecated or the project is no longer under maintenance according to their owner. To better understand the reasons that lead to the failure of modern open source projects. This chapter describes the results of a survey study with the maintainers of 104 popular GitHub systems that have been deprecated. We provide a set of nine reasons for the failure of open source projects. We also show that some maintenance practices—specially the adoption of contributing guidelines and continuous integration—have an important association with a project failure or success. Finally, we discuss and reveal the principal strategies developers have tried (without success) to overcome the failure of the studied projects.

2.1 Introduction

In this chapter we present an investigation with the maintainers of open source projects that have failed, aiming to reveal the reasons for such failures, the maintenance practices that distinguish failed projects from successful ones, the impact of failures on clients, and the strategies tried by maintainers to overcome the failure of their projects. This investigation addresses the following research questions:

RQ1: Why do open source projects fail? To answer this first RQ, we select 542 popular GitHub projects without any commits in the last year. We complemented this selection with 76 systems whose documentation explicitly mentions that the project is abandoned. We asked the developers of these systems to describe the reasons of the projects' failure. Finally, we categorize their responses into nine major reasons.

RQ2: What is the importance of following a set of best open source maintenance practices? In this second research question, we check whether the failed projects used a set of best open source maintenance practices, including practices to attract users and to automate maintenance tasks (e.g., the availability of contribution guidelines, code of conduct, license, the use of continuous integration, etc.).

RQ3: What is the impact of the project failures? To measure this impact, we counted the number of opened issues and pull requests of the failed projects and also the number of projects that depend on them. The goal is to measure the impact of the studied failures, in terms of affected users, contributors, and client projects.

RQ4: How do developers try to overcome the projects failure? In this last research question, we manually analyze the issues of the failed projects to collect strategies and procedures tried by their maintainers to avoid the failures.

We make the following contributions in this study:

- We provide a list of nine reasons for failures in open source projects. By providing these reasons, using data from real failures, we intend to help developers to assess and control the risks faced by open source projects.

- We reinforce the importance of a set of best open source maintenance practices, by comparing their usage by the failed projects and also by the most and least popular systems in a sample of 5,000 GitHub projects.

- We document three strategies attempted by the maintainers of open source projects to overcome (without success) the failure of their projects.

The remainder of this chapter is organized as follows. Section 2.2 presents the dataset we use to search for failed projects. Section 2.3 to Section 2.6 presents answers to each of the four research questions proposed in the study. Section 2.7 discusses and puts our findings in a wider context. Section 2.8 presents threats to validity; Section 2.9 presents related work; and Section 2.10 concludes this study.

2.2 Dataset

The dataset used in this study was created by first considering the top-5,000 most popular projects on GitHub (on September, 2016). We use the number of stars as a proxy for popularity because it reveals how many people manifested interest or appreciation to the project [Borges et al., 2016b]. We limit the study to 5,000 repositories to focus on the maintenance challenges faced by highly popular projects.

We use two strategies to select systems that are no longer under maintenance in this initial list of 5,000 projects. First, we select 628 repositories (13%) without commits in the last year. As examples, we have Nvie/gitflow[1] (16,392 stars), Mozilla/BrowserQuest[2] (6,702 stars), and Twitter/typeahead.js[3] (3,750 stars). Second, we search in the README[4] of the remaining repositories for terms described on Table 2.1.

Table 2.1: Sentences documenting deprecated projects.

dead project, *deprecated*, *unmaintained*, *no longer being actively maintained*, *no longer maintained*, *no longer under development*, *no longer supported*, *not maintained anymore*, *not under active development*, *is not supported*, *is not maintained*, *is not under development*

We found such terms in the READMEs of 207 projects (4%). We then manually inspected these files to assure that the messages indeed denote inactive projects and to remove false positives. After this inspection, we concluded that 76 repositories (37%) are true positives. As an example, we have Google/gxui[5] whose README has this comment:

Unfortunately due to a shortage of hours in a day, GXUI is no longer maintained.

As an example of false positive, we have Twitter/labella.js.[6] In its README, the following message initially led us to suspect that the project is abandoned:

The API has changed. force.start() and . . . are deprecated.

[1] *https://github.com/nvie/gitflow*

[2] *https://github.com/mozilla/BrowserQuest*

[3] *https://github.com/twitter/typeahead.js*

[4] READMEs are the first file a visitor is presented to when visiting a GitHub repository. They include information on what the project does, why the project is useful, and eventually the project status (if it is active or not).

[5] *https://github.com/google/gxui*

[6] *https://github.com/twitter/labella.js*

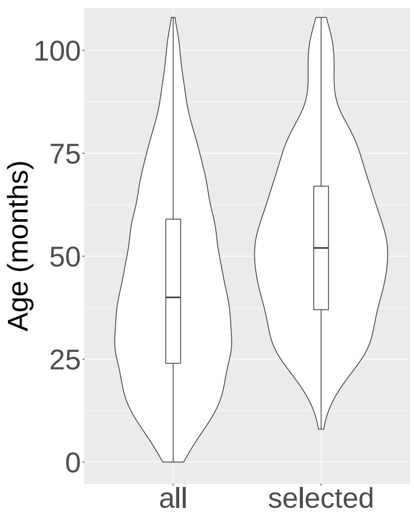
However, in this case, deprecated refers to API elements and not to the project's status. In a final cleaning step, we manually inspected the selected 704 repositores (628 + 76). We removed repositories that are not software projects (51 repositories, e.g., books, tutorials, and awesome lists), repositories whose native language is not English (24 repositories), that were moved to another repository (7 repositories), and that are empty (4 repositories, which received their stars before being cleaned). We ended up with a list of 618 projects (542 projects without commits in the last year and 76 projects with an explicit deprecation message in the README). We only inspected the README because we had to verify manually all of them to discard false positives. Therefore, we do not search explicit deprecation message on commits or issues.

Figure 2.1 shows violin plots with the distribution of age (in months), number of contributors, number of commits, and number of stars of the selected repositories. We provide plots for all 5,000 systems (labeled as *all*) and for the 618 systems (12%) considered in this study (labeled as *selected*). The selected systems are older than the top-5,000 systems (52 vs 40 months, median measures); but they have less contributors (11 vs 23), less commits (137 vs 346), and less stars (2,345 vs 2,538). Indeed, the distributions are statistically different, according to the one-tailed variant of the Mann-Whitney U test (p-value $\leq$ 5%). To show the effect size of this difference, we compute Cliff's delta (or d). We found that the effect is small for age and commits, medium for contributors, and negligible for stars.

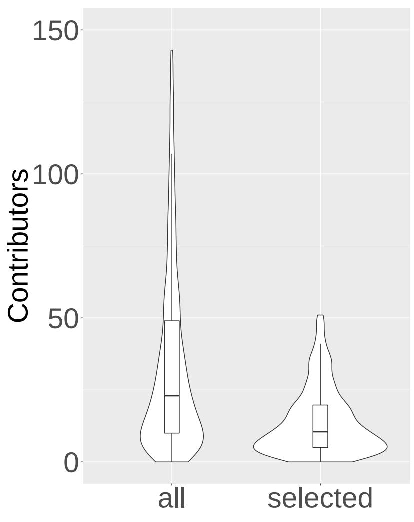
GitHub repositories can be owned by a person (e.g., TORVALDS/LINUX) or by an organization (e.g., MOZILLA/PDF.JS). In our dataset, 170 repositories (28%) are owed by organizations and 448 repositories (72%) by users. JavaScript is the most popular language (219 repositories, 36%), followed by Objective-C (98 repositories, 16%), and Java (75 repositories, 12%). In total, the dataset includes systems spanning 26 programming languages. We manually classified the application domain of the systems in the dataset, as showed in Table 2.2. There is a concentration on libraries and frameworks (502 projects, 81%), which essentially reproduces a concentration also happening in the initial list of 5,000 projects.[7]

Dataset limitations: The proposed dataset is restricted to popular open source projects on GitHub. We acknowledge that there are popular projects in other platforms, like Bitbucket, GitLab or that have their own version control installations. Also, the dataset does not include projects that failed before attracting the attention of developers and users. We consider less important to study such projects since their failures

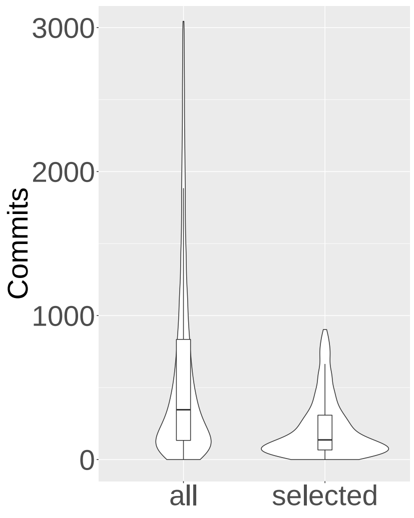
[7]For another research, we classified the domain of the top-5,000 GitHub projects; 59% are libraries and frameworks.



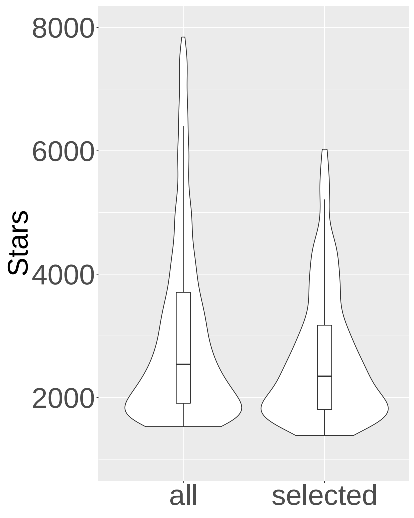
(a) Age



(b) Contributors



(c) Commits



(d) Stars

Figure 2.1: Distribution of the projects by (a) age, (b) contributors, (c) commits, and (d) stars, without outliers.

Table 2.2: Application domain of the selected projects

Application Domain	**Projects**
Libraries and frameworks	502
Application software (e.g., text editors)	63
Software tools (e.g., compilers)	31
System software (e.g., databases)	22

did not have much impact. Instead, we focus on projects that succeeded to attract attention, users, and contributors, but then failed, possibly impairing other projects.

2.3 Why do open source projects fail?

To answer the first research question, we conducted a survey with the developers of open source projects with evidences of no longer being under maintenance.

2.3.1 Survey Design

The survey questionnaire has three open-ended questions: (1) Why did you stop maintaining the project? (2) Did you receive any funding to maintain the project? (3) Do you have plans to reactivate the project? We avoid asking the developers directly about the reasons for the project failures, because this question can lead to multiple interpretations. For example, an abandoned project could have been an outstanding learning experience to its developers. Therefore, they might not consider that it has failed. In Section 2.3.3, we detail the criteria we followed to define that a project has failed based on the answers to the survey questions.

Specifically to the developers of the 542 repositories without commits in the last year, we added a first survey question, asking them to confirm that the projects are no longer being maintained. We also instructed them to only answer the remaining questions if they agree with this fact. We sent the questionnaire to the repositories' owners or to the project's main contributor, in the case of repositories owned by organizations. Using this criterion, we were able to find a public e-mail address of 425 developers on GitHub. However, 9 developers are the owners—or the main contributors—of two or more projects. In this case, we only sent one email to these developers, referring to their first project in number of stars, to avoid a perception of our mails as spam messages.

We sent the questionnaire to 414 developers. After a period of 20 days, we obtained 118 responses and 6 mails returned due to the delivery problems, resulting in a response rate of 29%, which is $118/(414-6)$. To preserve the respondents' anonymity, we use labels D1 to D118 to identify them. Furthermore, when quoting their answers we replace mentions to repositories and owners by *@[Project-Name]* and *@[Project-Owner]*. This is important because some answers include critical comments about developers or organizations.

Finally, for some projects, we found answers to the first survey question ("Why did you stop maintaining the project?") when inspecting their READMEs. This happened with 36 projects, identified by R1 to R36. As an example, we have the following README:

Unfortunately, I haven't been able to find the time that I would like to dedicate to this project. (R6)

Therefore, for the first survey question, we collected 154 answers (118 answers by e-mail and 36 answers from the projects' README). We analyzed these answers using thematic analysis [Cruzes and Dyba, 2011; Silva et al., 2016], a technique for identifying and recording "themes" (i.e., patterns) in textual documents. Thematic analysis involves the following steps: (1) initial reading of the answers, (2) generating a first code for each answer, (3) searching for themes among the proposed codes, (4) reviewing the themes to find opportunities for merging, and (5) defining and naming the final themes. After this, a sequence of meetings was held to resolve conflicts and to assign the final themes (step 5).

2.3.2 Survey Results

This section presents the answers to the survey questions. For the 118 developers of systems with no commits in the last year, the survey included an opening question asking if he/she agrees that the project is no longer under maintenance. A number of 101 developers (86%) confirmed this project condition, as in the following answer:

Yes, I surely have abandoned the project. (D20)

By contrast, 17 developers (14%) did not agree with the project status. For example, two developers mentioned work being performed out of the main GitHub repository:

One current issue that does need to be resolved is that the entire site is served over https, but you wouldn't see that change in the repo. (D18)

It is under maintenance. It's just not a lot of people are using it, and I am working on a new breaking version and thus didn't want to commit on the master branch. (D30)

Next, we present the reasons that emerged after analysing the answers received for the first survey question ("Why did you stop maintaining the project?"). We discuss each reason and give examples of answers associated to them.

Lack of time: According to 27 developers, they do not have free time to maintain the projects, as in the following answers:

It was conceived during extended vacation. When I got back to working I simply didn't have time. Building something like @[Project-Name] requires 5-6 hours of work per day. (D15)

I was the only maintainer and there was a lot of feature requests and I didn't have enough time. (D115)

Lack of interest: 30 developers answered they lost interest on the projects, including when they started to work on other projects or domains, changed jobs, or were fired.[8] As examples, we have:

My interest began to wane; I moved to other projects. (D67)

I'm not working in the CMS space at the moment. (D77)

It became less professionally relevant/interesting. (D80)

I was fired by the company that owns the project. (D65)

Project is completed: 17 developers consider that their projects are finished and do not need more features (just few and sporadic bug fixes). As an example, we have the following answers:

Sometimes, you build something, and sometimes, it's done. Like if you built a building, at some point in time it is finished, it achieved its goals. For @[Project-Name] — it achieved all its goals, and it's done. ... The misconception is that people may mistake an open source project with news. Sometimes there are just no more features to add, no more news — because the project is complete. (D28)

I felt it was done. I think the dominant idea is that you have to constantly update every open source project, but in my opinion, this thing works great and needs no updates for any reason, and won't for many, many years, since it's built on extremely stable APIs (namely git and Unix utilities). (D69)

Usurped by competitor: 30 developers answered they abandoned the project because a stronger competitor appeared in the market, as in the case of these projects:

Google released ActionBarCompat whose goal was the same as @[Project-Name] but maintained by them. (D2)

The project no longer makes sense. Apple has built technical and legal alternatives which I believe are satisfactory. (D71)

[8]Consequently, these developers do not have more time to work on their projects; however, we reserve the lack of time theme to the cases where the developers still have interest on the projects, but not the required time to maintain them.

It's not been maintained for well over half a year and is formally discontinued. There are better alternatives now, such as SearchView and FloatingSearchView. (R42)

Specifically, 12 projects explicitly declare in their READMEs that they are no longer maintained due to the appearance of a strong competitor. In all cases, the update date of the project status as unmaintained occurred after competitor appeared. For example, Node-js-libs/node.io was declared unmaintained four years after its competitor appeared. We also found this statement in its README: *I wrote node.io when node.js was still in its infancy.*

Project is obsolete: According to 21 developers, the projects are not useful anymore, i.e., their features are not more required or applicable.[9] As examples, we have the answers:

This was only meant as a stopgap to support older OSes. As we dropped that, we didn't need it anymore. (D11)

I do not have an app myself anymore using that code. (D36)

I personally have no use for it in my work anymore. (D38)

Project is based on outdated technologies: This reason, mentioned by 16 respondents, refer to discontinuation due to outdated, deprecated or suboptimal technologies, including programming languages, APIs, libraries, frameworks, etc. As examples, we have the following answers:

Due to Apple's abandonment of the Objective-C Garbage Collector which @[Project-Name] relied heavily on, future development of @[Project-Name] is on an indefinite hiatus. (R20)

The core team is now building @[Project-Name] in Dart instead of Ruby, and will no longer be maintaining the Ruby implementation unless a maintainer steps up to help. (R34)

Low maintainability: This reason, as indicated by 7 developers, refers to maintainability problems. As examples, we have:

It is difficult to maintain a browser technology like JavaScript because browsers have very different quirks and implementations. (D28)

[9]The theme does not include projects that are obsolete due to outdated technologies, which have a specific theme.

The project reached an unmaintainable state due to architectural decisions made early in the project's life. (D30)

Conflicts among developers: This reason, indicated by three developers, denotes conflicts among developers or between developers and project owners, as in this answer:

The project was previously an official plugin—so the @[Project-Owner] team worked with me to support it. However, they decided would not longer have the concept of plugins—and they ended the support on their side. (D73)

The remaining reasons include acquisition by a company, which created a private version of the project (two answers), legal problems (two answers), lack of expertise of the principal developer in the technologies used by the project (one answer), and high demand of users, mostly in the form of trivial and meaningless issues (one answer). Finally, in five cases, it was not possible to infer a clear reason after reading the participant's answers. Thus, we classified these cases under an *unclear answer* theme. An example is the following answer: *I am not so sure, but you can probably check the last commit details in GitHub.* (D95)

We also asked the participants a second question: *did you receive any funding to maintain the project?* 82 out of 118 answers (69%) were negative. The positive answers mention funding from the company employing the respondent (12 answers), non-profit organizations (three answers; e.g., European Union), and other private companies (two answers). Finally, we asked a third question: *do you have plans to reactivate the project?* Only 18 participants (15%) answered positively to this question.

2.3.3 Combining the Survey Answers

In our study, we consider that a project has *failed* when at least one of the following conditions hold:

1. The project is no longer under maintenance according to the surveyed developers and they do not have plans to reactivate the project (question #3) and the project is not considered completed (question #1).

2. The project documentation explicitly mentions that it is deprecated (without considering it completed).

Among the considered answers, 76 projects attend condition (1) and 32 projects attend condition (2). The reasons for the failure of these projects are the ones presented in Section 2.3.2, except when the themes are *lack of interest* or *lack of time*. For these themes and when the answer comes from the top-developer of a project owned by an organization, we made a final check on his/her number of commits. We only accepted the reasons suggested by developers that are responsible for at least 50% of the projects' commits. For example, D85 answered he stopped maintaining his project due to a lack of time. The project is owned by an organization and D85—although the top-maintainer of the project—is responsible for 30% of the commits. Therefore, in this case, we assumed that it would be possible to other developers to take over the tasks and issues handled by D85. By applying this exclusion criterion, we removed four projects from the list of projects. The final list, which includes reasons for failures according to relevant top-developers or project owners, has 104 projects. In this study, we call them *failed projects*.

Table 2.3 presents the reasons for the failure of these projects. The most common reasons are project was *usurped by competitor* (27 projects), project is *obsolete* (20 projects), *lack of time* of the main contributor (18 projects), *lack of interest* of the main contributor (18 projects), and project is based on *outdated technologies* (14 projects). It is also important to note that projects can fail due to multiple reasons, which happened in the case of 6 projects. Thus, the sum of the projects in Table 2.3 is 110 (and not 104 projects).[10]

Table 2.3: Why open source projects fail?

Reasons	**Group**	**Projects**
Usurped by competitor	Environment	27 ■
Obsolete	Project	20 ■
Lack of time	Team	18 ■
Lack of interest	Team	18 ■
Outdated technologies	Project	14 ▮
Low maintainability	Project	7 ▮
Conflicts among developers	Team	3 ❘
Legal problems	Environment	2 ❘
Acquisition	Environment	1 ❘

As presented in Table 2.3, we classified the reasons for failures in three groups: (1) reasons related to the development team (including lack of time, lack of interest, and conflicts among developers); (2) reasons related to project characteristics (including project is obsolete, project is based on outdated technologies, and low project

[10]The values in Table 2.3 are not exactly the ones presented in Section 2.3.2 due to the inclusion and exclusion criteria defined in this section.

maintainability); (3) reasons related to the environment where the project and the development team are placed (including usurpation by competition, acquisition by a company, and legal issues).

Summary: Modern open source projects fail due to reasons related to project characteristics (41 projects; e.g., low maintainability), followed by reasons related to the project team (39 projects; e.g., lack of time or interest of the main contributor); and due to environment reasons (30 projects; e.g., project was usurped by a competidor or legal issues).

2.4 What is the importance of open source maintenance practices?

In this second question, we investigate whether the failed projects followed (or not) a set of best open source maintenance practices, which are recommended when hosting projects on GitHub.[11] Section 2.4.1 describes the methodology we followed to answer the research question and Section 2.4.2 presents the results and findings.

2.4.1 Methodology

We analyzed four groups of projects: the 104 projects that have failed, as described in Section 2.3.3 (*Failed*), the top-104 and the bottom-104 projects by number of stars (*Top* and *Bottom*, respectively), and a random sample of 104 projects (*Random*). *Top*, *Bottom*, and *Random* are selected from the initial sample of top-5,000 projects, described in Section 2.2, and after applying the same cleaning steps defined in that section. The rationale is to compare the *Failed* projects with the most popular projects in our dataset, which presumably should follow most practices; and also with the least popular projects and with a random sample of projects.

For each project in the aforementioned groups of projects we collected the following information:[12] (1) presence of a README file (which is the landing page of GitHub repositories); (2) presence of a separate file with the project's license; (3) availability of a dedicated site and URL to promote the project, including examples, documentation, list of principal users, etc.; (4) use of a continuous integration service (we check whether

[11] *https://opensource.guide*

[12] Five of these maintenance practices are explicitly recommended at: *https://help.github.com/articles/helping-people-contribute-to-your-project*

the projects use Travis CI, which is the most popular CI service on GitHub, used by more than 90% of the projects that enable CI, according to a recent study [Hilton et al., 2016]); (5) presence of a specific file with guidelines for repository contributors; (6) presence of an issue template (to instruct developers to write issues according to the repository's guidelines); (7) presence of a specific file with a code of conduct (which is a document that establishes expectations for the behavior of the project's participants [Tourani et al., 2017]); and (8) presence of a pull request template (which is a document to instruct developers to submit pull requests according to the repository's guidelines).

After collecting the data for each project in each group we compared the obtained distributions. First, we analyzed the statistical significance of the difference between the *Top*, *Bottom*, and *Random* groups vs the *Failed* group, by applying the Mann-Whitney test at p-value = 0.05. To show the effect size of the difference, we used Cliff's delta. Following the guidelines of previous work [Grissom and Kim, 2005; Tian et al., 2015b; Linares-Vásquez et al., 2013], we interpreted the effect size as small for $0.147 < d < 0.33$, medium for $0.33 \leq d < 0.474$, and large for $d \geq 0.474$.

2.4.2 Results

Table 2.4 shows the percentage of projects following each practice. Despite the group, the most followed practices are the presence of a README file, the presence of a license file, and the availability of a project home page. For example, for the *Failed* projects the percentage of projects following these practices are 99%, 61%, and 58%, respectively. For the *Top* projects, the same values are 100%, 88%, and 87%, respectively. The least followed practices are issue templates, code of conduct, and pull request templates. We did not find a single project following these practices in the *Failed* group. By contrast, 15%, 13%, and 3% of the *Top* projects have these three kind of documents, respectively. In general, we observe the following order among the groups of projects regarding the adoption of the eight considered practices: *Top* > *Random* > *Failed* ≡ *Bottom*. In other words, there is a relevant adoption of most practices by the *Top* projects. By contrast, the 104 projects that failed are more similar to the *Bottom* projects. This fact is reinforced by the analysis of Cliff's delta coefficient. There is a *large* effect size between the adoption of contributing guidelines by the *Top* (72%) and the *Failed* projects (16%), and a *medium* difference in the case of continuous integration services (68% vs 27%). For licenses, home pages, and issue templates, the difference is *small*. For the remaining practices, the difference is negligible or does not exist in statistical terms. In the case of the *Bottom* projects, there is no statistical difference for the

Table 2.4: Percentage of projects following practices recommended when maintaining GitHub repositories. The effect size reflects the extent of the difference between the repositories in a given group (Top, Bottom, or Random) and the failed projects

Maintaince Practice	Failed	Top	Effect	Bottom	Effect	Random	Effect
README	99	100	-	100	-	100	-
License	61	88	small	60	-	73	-
Home Page	58	87	small	52	-	60	-
Continuous Integration	27	68	medium	41	-	45	small
Contributing	16	72	large	13	-	32	small
Issue Template	0	15	small	2	-	5	-
Code of Conduct	0	13	-	0	-	2	-
Pull Request Template	0	3	-	0	-	0	-

eight considered documents. Finally, for *Random*, there is a *small* difference when we consider the use of continuous integration and contributing guidelines.

Summary: Regarding the adoption of best open source maintenance practices, the failed projects are more similar to the least popular projects than to the most popular ones. Therefore, these practices seem to have an effect on the success or failure of open source projects. The practices with the most relevant effects are contributing guidelines (large), continuous integration (medium), and licences, home pages, and issue templates (small).

2.5 What is the Impact of Failures?

With this third research question, we intend to assess the impact of the failure of the studied projects, both to end-users and to the developers of client systems. First, we present the approach we used to answer the question (Section 2.5.1). Then, we present the results (Section 2.5.2).

2.5.1 Methodology

To answer the question, we collected data on (a) the number of issues and pull requests of the failed projects; and (b) the number of systems that depend on these projects, according to data provided by GitHub and by a popular JavaScript package manager.

2.5.2 Results

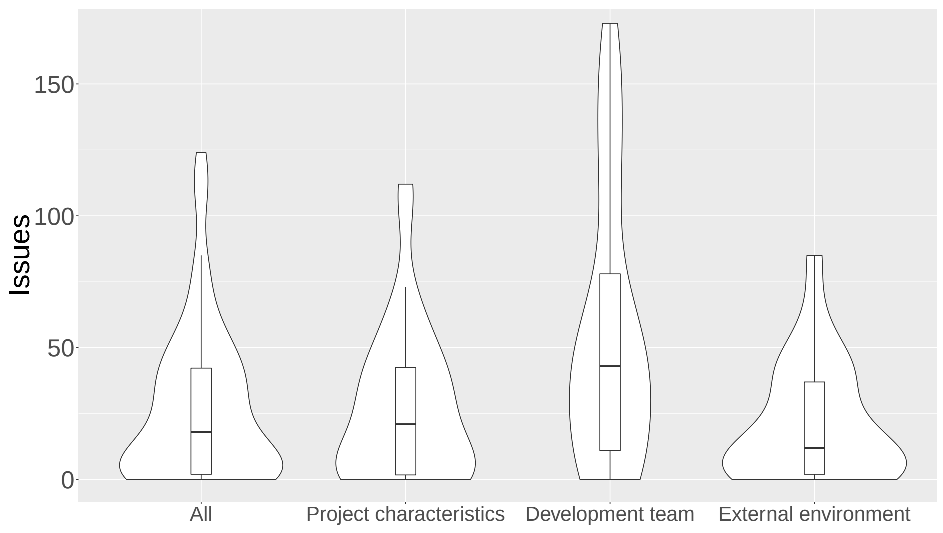
Issues and Pull Requests: We collected the number of opened issues and opened pull requests for each failed project (in the case of issues, we excluded 15 projects that do not use GitHub to handle issues). Our rationale is that one of the negative impacts of an abandoned project is a list of bugs and enhancements (issues) that will not be considered and a list of source code modifications (pull requests) that will not be implemented. Pending issues impact the projects' users, who need to keep using a project with bugs or a frozen set of features, or who will have to migrate to other projects. Pending pull requests contribute to the frustration of the projects' contributors, who will not have their effort appreciated.

Dependencies: We also collected data on projects that depend on the failed projects and that therefore are using unmaintained systems. To collect dependencies data, we first rely on a GitHub service that reports the number of client repositories that depend on a given repository.[13] Unfortunately, this feature is available only to Ruby systems. To cover more projects, we also consider dependency data provided by Npm, a popular package manager for JavaScript. As result, we analysed the dependencies of 38 projects, including 10 Ruby projects and 28 JavaScript ones.

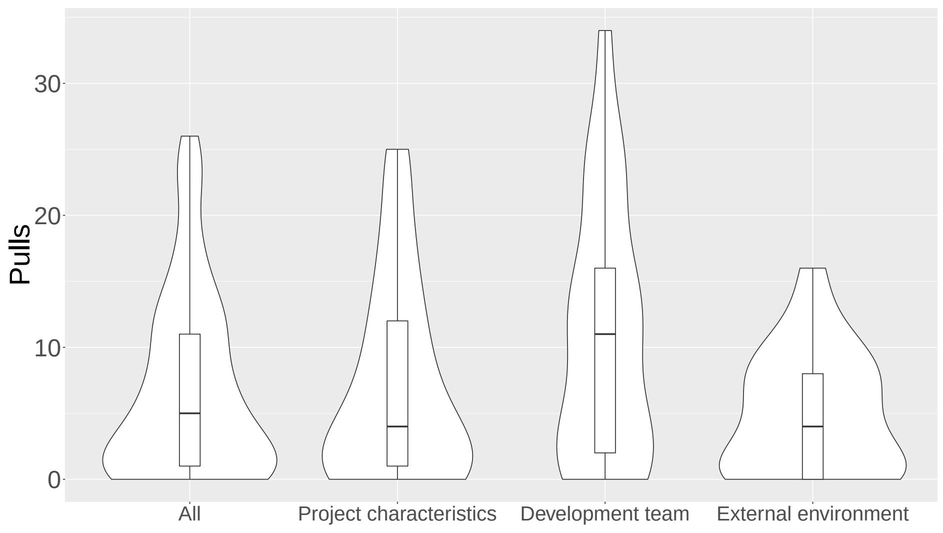
Issues and Pull Requests: Figure 2.2 shows violin plots with the distributions of opened issues and pull requests. Considering the 89 failed projects with issues on GitHub, the median number of opened issues is 18 and the median number of pending pull requests is 5. The top-3 failed projects with the highest number of pending issues have 230, 173, and 160 issues. The top-3 failed projects with the highest number of pending pull requests have 54, 45, and 38 pull requests. The figure also shows the number of opened issues and pull requests grouped by failure reasons. The median number of issues associated to project characteristics, development team, and environment reasons are 21, 43, and 12 issues, respectively. For pull requests, the median measures for the same groups of reasons are 4, 11, and 4 pull requests, respectively. By applying Kruskal-Wallis test to compare multiple samples, we find that these distributions are not different.

Dependencies: A total of 6 (out of 10) Ruby repositories do not have dependent projects. However, we also found projects with 2,460, 270, 36, and 18 dependents. Regarding the JavaScript systems, 10 (out of 28) projects do not have an entry on Npm (although Npm is very popular, systems can use other package managers or do

[13] *https://github.com/blog/2300-visualize-your-project-s-community*



(a) Opened issues



(b) Opened pull requests

Figure 2.2: Distribution of the (a) Opened issues and (b) Opened pull requests, without outliers

not use a package manager at all). A total of 15 projects have five or less dependents and three systems have respectively 158, 37, and 13 dependents.

Summary: The failed projects have 18 opened issues and 5 opened pull requests (median measures). 55% of the Ruby and JavaScript projects have less than five dependents, which suggests that most clients have also abandoned these projects.

2.6 How do developers try to overcome the projects failure?

In this fourth research question, we qualitatively investigate attempts to overcome the failure of the studied projects.

2.6.1 Methodology

We read the 20 most recent opened issues and the 20 most recent closed issues of each of the 104 failed projects (in a total of 1,654 issues). As a result, he collected 32 issues, which the developers question the status of the projects and/or discuss alternatives to restart the development. The issues, which are identified by I1 to I32, cover 32 projects in the list of failed projects. Examples of titles of selected issues are: *Is this project dead?* (I18), *Is this project maintained?* (I1), and *Is development of this ongoing?* (I7). After this first step, we extracted a set of recurrent strategies (or "themes") suggested by developers to overcome the failure of the projects the issues refer to. The proposed themes were validated in a last step.

2.6.2 Results

After analyzing the issues, we found three strategies attempted by owners or collaborators to overcome the unmaintained status of the projects. Next, we describe these strategies.

Moving to an organization account: This strategy, mentioned in five issues, refers to the creation of an organization account with a name similar to the project's name. The hope is that with this kind of account it would be easier to attract new maintainers and to manage permissions. As examples, we have these comments:

Would creating a @[Project-Name] repo in a @[Project-Name] org be something people would want? (I31)

I am totally cool with setting up an org and transferring control... Just let me know what you need. (I3)

Transfer the project to new maintainers: This strategy, discussed for three projects, consists in a complete transfer of the project's maintainership to other developers (but keeping the project's name), as discussed in these issues:

Who want to take over this project will be appreciated. We will watch the project together for a while and I will grant every permission. (I10)

In two projects, a new developer was found and assumed the project, as documented in this issue:

I have started working on @[Project-Name]*.* @[Project-Owner] *transferred the repository to my account.* (I7)

We tracked the activity of the new maintainers, until February, 2017. They did not perform significant contributions to the projects, despite minor commits. In one project, we found the following complaint about the new maintainer:

@[Owner-Name] gave this repo to someone who has never been active on GitHub, so this repo is basically dead. (I11)

Accepting new core developers: In five cases, to overcome the low activity on the repositories, volunteers offered to help with the maintenance, as core developers. For example, we have this issue:

@[Project-Name] Would you be open to adding more collaborators to this repo? (I17)

In all cases, the proposals were not answered or accepted. As an example, we have this project owner, who requested a detailed maintenance plan before accepting the maintainer:

I'd be willing to do this if the collaborators provided a roadmap of what they'd like to accomplish with the library. (I17)

Although it is not exactly an overcome strategy, in 19 cases owners suggested the developers to start collaborating on another project, as in this issue:

I'd suggest you look at @[Project-Name]. It's very active and modern. I'm trying to find time to switch over myself. (I9)

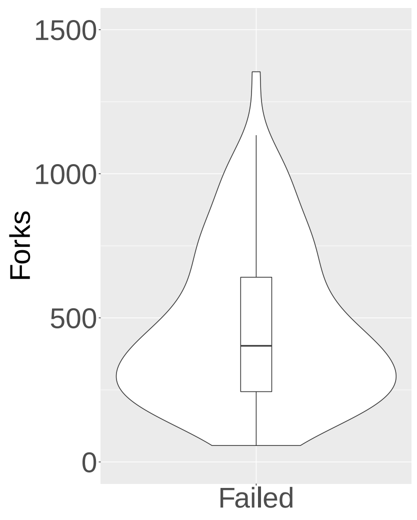
Finally, although the presented strategies were not able to restart the development of the studied projects, they should not be considered as completely failed ones. To illustrate this fact, we selected 348 projects that almost failed in the year before the study (they have five or less commits). 182 projects (52%) indeed failed in the next year (the studied one). However, 35 projects show evidences of recovering (they have more than the first quartile of commits/year in the studied year, i.e., 15 commits). After inspecting the documentation and issues of these 35 projects, we found that 14 projects attracted new core developers (third strategy), two were transferred to new maintainers (second strategy), and two projects moved to an organization account (first strategy).

Summary: Developers attempted three strategies to overcome the failure of their projects: (a) moving to an organization account; (b) transfer the project to new maintainers; and (c) accepting new core developers.

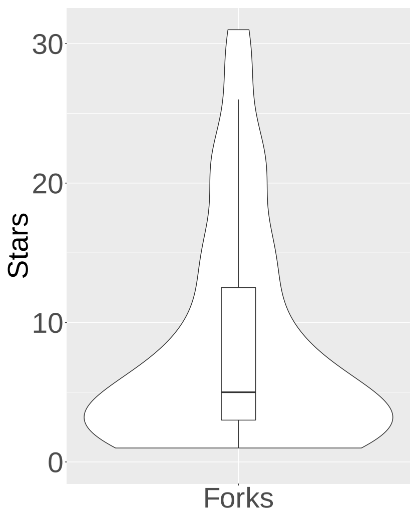
2.6.3 Complementary Investigation: Forks

Forks are used on GitHub to create copies of repositories. The mechanism allows developers to make changes to a project (e.g., fixing bugs or implementing new features) and submit the modified code back to the original repository, by means of pull requests. Alternatively, forks can become independent projects, with their own community of developers. Therefore, forks can be used to overcome the failure of projects, by bootstrapping a new project from the codebase of an abandoned one. For this reason, we decided to complement the investigation of RQ4 with an analysis of the forks of the failed projects.

Figure 2.3a shows the distribution of the number of forks of the failed projects. They usually have a relevant number of forks since it is very simple to fork projects on GitHub. The first, median, and third quartile measures are 244, 400, and 638 forks, respectively. The violin plot in Figure 2.3b aims to reveal the relevance of these forks. For each project, we computed the fork with the highest number of stars. The violin plot shows the distribution of the number of stars of these most successful forks. As we can see, most forks are not popular. They are probably only used to submit pull requests or to create a copy of a repository, for backup purposes [Jiang et al., 2016]. For example, the third quartile measure is 13 stars. However, there are two systems with an outlier behavior. The first one is an audio player, whose fork has 1,080 stars. In our survey, the developer of the original project answered that he abandoned the project due to other interests. However, his code was used to fork a new project, whose README acknowledges that this version *"is a substantial rewrite of the fantastic work done in version 1.0 by [Projet-Owner] and others"*. Besides 1,080 stars, the forked project has 70 contributors (as February, 2017). The second outlier is a dependency injector for Android, whose fork was made by Google and has



(a) Forks



(b) Stars (best fork)

Figure 2.3: Distribution of the (a) number of forks of the studied projects and (b) number of stars of the fork with the highest number of stars, for each studied project, both violin plots without outliers

6,178 stars. The forked project's README mentions that it "is currently in active development, primarily internally at Google, with regular pushes to the open source community".

Summary: Forks are rarely used on GitHub to continue the development of open source projects that have failed.

2.7 Discussion

In this section, we discuss the main findings of our study.

Completed projects and first Law of Software Evolution: An interesting finding of the survey with developers is the category of completed projects (17 systems, 11%), which are considered feature-completed by their developers. They do not plan to evolve these systems, because "*adding more features would just obfuscate the original intent*" (D55) of the projects. Moreover, they also think the projects will not need adaptive maintenance, as in this answer:

I just stopped working on it because what I have works very well, and will continue working very well until Unix stops being the foundation of most Web development, *which basically means until the end of the human race . . . Most projects don't build on similarly solid foundations so they probably need to change more often.* (D56)

Someone can argue that these projects contradict the first Law of Software Evolution, which prescribes that "programs are never completed" [Lehman, 1980]. However, Lehman's Laws only apply to E-type systems, where the "E" stands for evolutionary.[14] In these systems, the environment around the program changes and hence the requirements and the program specification [Herraiz et al., 2013]. Therefore, Lehman opens the possibility to have completed programs, when they target an environment controlled by the developers or that is very stable (e.g., the Unix ecosystem, as mentioned by Developer D56).

Competition in open source markets: The study reveals an important competition between open source projects. The most common reason for project failures is the appearance of a stronger open source competitor (27 projects). Usually, this competitor is the major organization responsible for the ecosystem the project is inserted on, specifically Google (Android ecosystem, 7 projects) and Apple (iOS ecosystem, 5 projects). Therefore, open source developers should be aware of the risks of starting a project that may attract the attention of major players, particularly when the projects have a tight integration and dependency with established platforms, like Android and iOS. Clients should also evaluate the risks of using these "non-official" projects. They should evaluate if it is worth to accept the opportunity costs of delaying the use of a system until it is provided as a built-in service. Alternatively, they can conclude that the costs of delaying the adoption further exceeds the additional benefits of providing earlier a service to end-users. Other competitors mentioned in the survey are D3/d3 (a visualization library for JavaScript) and MVC frameworks, also for JavaScript, such as Facebook/react. For example, one developer mentioned that "*high-end front-end development seems to be moving away from jQuery plugins*" (D18). This result confirms that web development is a competitive domain, where the risks of failures are considerable, even for highly popular projects.

Practical implications: This study provides insights to the definition of lightweight "maturity models" to open source projects. By lightweight, we mean that such models should be less complex and detailed than equivalent models for commercial software projects, like CMMI [Chrissis et al., 2003]. But at least they can prescribe that open source projects should manage and constantly assess the risk factors that emerged from

[14]The first law (Continuing Changing) is as follows: "An E-type system must be continually adapted, else it becomes progressively less satisfactory in use." [Lehman et al., 1997]

our empirical investigation. We shed light on three particular factors: (a) risks associated to development teams (for example, projects than depend on a small number of core developers may fail due to the lack of time or lack of interest of these developers, after a time working in the project); (b) risks associated to the environment the projects are immersed (which seems to be particularly relevant in the case of projects with a tight integration with mobile operating systems or in the case of web libraries and frameworks); (c) risks associated to project characteristics and decisions, such as the use of outdated technologies. Furthermore, we also showed the importance of practices normally recommended to open source development on GitHub. We show that successful projects provide documents like README, contributing guidelines, usage license declarations, and issue templates. They also include a separate home page, to promote the projects among end-users. Finally, we showed evidences on the benefits provided by continuous integration, in terms of automation of tasks like compilation, building, and testing.

2.8 Threats To Validity

The threats to validity of this work are as follows:

External Validity: Threats to external validity were partially discussed when presenting the dataset limitations (Section 2.2). We complement this discussion as follows. First, when investigating the use of continuous integration by the failed, top, bottom, and random projects (RQ2, Section 2.4), we only consider the use of Travis CI. However, Travis is the most popular CI service on GitHub, used by more than 90% of the repositories that enable CI [Hilton et al., 2016]. Second, the investigation of dependent projects (RQ3, Section 2.5) only considered systems implemented in Ruby and JavaScript. For JavaScript, we only analyzed dependency data provided by a single package manager system (Npm).

Internal Validity: The first threat relates to the selection of the survey participants. We surveyed the project owner, in the case of repositories owned by individuals, or the developer with the highest number of commits, in the case of repositories owned by organizations. Although experts on their projects, it is possible that some participants omitted in their answers the real reasons for the project failures. To mitigate this threat, we avoid asking the participants directly about the causes of the project failures. A second threat relates to the themes denoting reasons for project failures (RQ1) and strategies on how to overcome them (RQ4). We acknowledge that the choice of these themes is to some extent subjective. For example, it is possible that different researchers reach a different set of reasons, than the ones proposed in Section 2.3.2. To mitigate this threat, the initial selection of themes in RQ1 was performed independently by the collaborators of this study. After this initial proposal, daily meetings were performed during a whole week to refine and improve the initial selection. A third internal validity threat might appear when interpreting the results of RQ2. In this case, it is important to consider that association does not imply in causation. For example, by just providing contributing guidelines or codes of conduct, a project does not necessarily will succeed.

Construct Validity: A first construct validity threat relates to thresholds and parameters used to define the survey sample. We consider as unmaintained the projects that did not have a single commit in the last year (Section 2.2). We recognize a threat in the selection of this threshold and time frame. However, to mitigate this threat, we included in the survey 36 projects whose README explicitly declares that the project is unmaintained or deprecated. The second threat concerns the data about the maintenance practices used to answer RQ2 (Section 2.4). This data was collected automatically, by means of scripts that rely on regular expressions to match different names and extensions used by the documents of interest (e.g., license.md and license.txt). However, we cannot guarantee that the implemented expressions match all possible variations of file names. Moreover, we did not investigate and check the quality of the retrieved documents. For example, we consider that a project has contributing guidelines when this document exists in the repository and it is not empty.

2.9 Related Work

Capiluppi et al. [2003] analyze 406 projects from FreshMeat (a deprecated open source repository). For each project, they compute a set of measures along four main dimensions: community of developers, community of users, modularity and documentation, and software evolution. They report that most projects (57%) have one or two developers and that only a few (15%) can be considered active, i.e., continuing improving their popularity and number of users and developers. However, they do not investigate the reasons for the project failures. Khondhu et al. [2013] discuss the attributes and characteristics of inactive projects on SourceForge. They report that more than 10,000 projects are inactive (as November, 2012). They also compare the maintainability of inactive projects with other project categories (active and dormant), using the maintainability index (MI) [Oman and Hagemeister, 1992]. They conclude that the majority of inactive systems are abandoned with a similar or increased maintainability, in comparison to their initial status. However, there are serious concerns on using MI as a maintainability predictor [Bijlsma et al., 2012].

Tourani et al. [2017] investigate the role, scope and influence of codes of conduct in open source projects. They report that seven codes are used by most projects, usually aiming to provide a safe and inclusive community, as well as dealing with diversity issues. After surveying the literature on empirical studies aiming to validate Lehman's Laws, Fernandez-Ramil et al. [2008] report that most works conclude that the first law (Continuing Change) applies to mature open source projects. However, in this work we found 17 completed projects, according to their developers. These projects deal with stable requirements and environments and therefore do not need constant updates or modifications.

Ye and Kishida [2003] describe a study to understand what motivates developers to engage in open source development. Using as case study the GIMP project (GNU Image Manipulation Program) they argue that learning is the major driving force that motivates people to get involved in open source projects. Eghbal [2016] reports on the risks and challenges to maintain modern open source projects. She argues that open source plays a key role in the digital infrastructure that sustain our society today. But unlike physical infrastructure, like bridges and roads, open source still lacks a reliable and sustainable source of funding. Avelino et al. [2016] concluded that nearly two-thirds of a sample of 133 popular GitHub projects depend on one or two developers to survive.

Humphrey [2005] presents 12 reasons for project failures, but in the context of commercial software and to justify the adoption of maturity models, like CMMI [Chrissis et al., 2003]. The reasons are presented and explained in the form of questions concerning why large software projects are hard to manage, the kinds of management systems needed, and the actions required to implement such systems. Lavallée and Robillard [2015] weekly observed during ten months the development of software projects in a large telecomunnication company. They show that organization factors, e.g., structure and culture, have a major impact on the success or failure of software projects. However, in our study these factors did not appear with the same importance. For example, only three projects failed due to conflicts among developers. We hypothesise this is due to the decentralized and community-centric characteristics of open source code. Jr et al. [2016] analyse 155 postmortems published on the gaming site Gamasutra.com. They report the best practices and common challenges faced in game development and provide a list of factors that impact project outcomes. For example, they found that the creativity of the development team is often a relevant factor in the success or failure of a game. As a practical recommendation, they mention that projects should practice good risk management techniques. We argue that the failure factors elicited in this study are a start point to include such practices in open source development, i.e., to control the risks we need first to known them.

Recent research on open source has focused on the organization of successful open source projects [Mockus et al., 2002], on how to attract and retain newcomers [Zhou and Mockus, 2015; Steinmacher et al., 2016; Lee et al., 2017; Pinto et al., 2016; Canfora et al., 2012], and on specific features provided by GitHub, such as pull requests [Gousios et al., 2014, 2015, 2016], forks [Jiang et al., 2016], and stars [Borges et al., 2016b,a].

2.10 Conclusion

In this study, we showed that the top-5 most common reasons for the failure of open source projects are: project was usurped by competitor (27 projects), project became functionally obsolete (20 projects), lack of time of the main contributor (18 projects), lack of interest of the main contributor (18 projects), and project based on outdated technologies (14 projects). We also showed that there is an important difference between the failed projects and the most popular and active projects on GitHub, in terms of following best open source maintenance practices. This difference is more important regarding the availability of contribution guidelines and the use of continuous integration. Furthermore, the failed projects have a non-negligible number of opened issues and pull requests. Finally, we described three strategies attempted by maintainers to overcome the failure of their projects.

Chapter 3

Why We Engage in FLOSS: Answers from Developers

The maintenance and evolution of Free/Libre Open Source Software (FLOSS) projects demand the constant attraction of core developers. In this chapter, we report the results of a survey with 52 developers, who recently became core contributors of popular GitHub projects. We reveal their motivations to assume a key role in FLOSS projects (e.g., improving the projects because they are also using it), the project characteristics that most helped in their engagement process (e.g., a friendly community), and the barriers faced by the surveyed core developers (e.g., lack of time of the project leaders). We also compare our results with related studies about others kinds of open source contributors (casual and newcomers).

3.1 Introduction

Free/Libre and Open Source Software (FLOSS) projects have an increasing impact on our daily lives [Kon et al., 2011]. For example, many companies depend nowadays on open source operating systems, databases, and web servers to run their basic operations. Similarly, most commercial software produced today depend on a variety of open source libraries and frameworks. However, there is a growing concern on the long term sustainability of FLOSS projects [Eghbal, 2016; Hata et al., 2015]. For example, in a recent study, Avelino et al. [2016] looked at a sample of 133 popular GitHub projects and concluded that nearly two-thirds depend on just one or two developers to survive. For this reason, FLOSS projects must continuously attract new *core developers* to mitigate the risks of failing.

Core developers are the ones responsible for the design, implementation, and maintenance of the most important features in a project. They are also responsible to manage the project and to plan and drive its evolution [Mockus et al., 2002; Joblin et al., 2017; Robles et al., 2009]. By contrast, peripheral contributors are those who occasionally contribute to the projects, mostly by fixing bugs [Lee et al., 2017; Pinto et al., 2016; Steinmacher et al., 2016]. Usually, core developers represent just a small fraction of the project contributors. For example, d3/d3—a very popular JavaScript visualization library, with over 73K stars on GitHub—has 121 contributors. However, the system is maintained and evolved by just one core developer [Avelino et al., 2016].

Since core developers are the heart and brain of FLOSS projects, we report in this chapter a survey with 52 developers who, in the last year, contributed to popular GitHub systems to the point of becoming core developers in these projects. By surveying these developers, our goal is to reveal their motivations for joining an open source project. We also asked them about the project characteristics that most helped in this process and about the main barriers they faced. The survey results can help FLOSS developers to improve some of the management practices followed in their projects, aiming to possibly expand the base of core developers.

We make the following contributions in this chapter:

- We provide a list of motivations that led recent core developers to contribute to open source projects. We found that 60% of the survey participants contribute because they are also using the projects.

- We reveal a list of projects characteristics and practices that helped recent core developers to contribute a FLOSS project. We found they are most attracted by non-technical characteristics, especially the ones related to a friendly and available FLOSS community.

- We provide a list of the main barriers faced by recent core contributors that led them to contribute. We found that non-technical barriers are the most relevant impediment they face to contribute, as the lack of time of the project leaders.

We organize the remainder of the chapter as follows. Section 3.2 presents the study design, how we selected the studied projects and the heuristic we used to identify core developers. Section 3.3 discusses the main findings of the survey. Section 3.4 presents a segmented analysis of the survey answers. Section 3.5 discuss threats to validity and Section 3.6 presents related work. Section 3.7 presents the main implications of our study, including implications to practitioners and researchers. Finally, Section 3.8 concludes the chapter.

3.2 Study Design

We start by considering the top-5,000 most popular GitHub projects, ranked by number of stars. Stars are similar to *likes* in popular social networks and therefore are a common measure of the popularity of GitHub projects [Borges et al., 2016b]. Then, we apply four strategies to discard projects from this initial selection, as follows:

1. *Non-Software Projects*: We discarded 61 repositories that are not software projects, including books (e.g., Vhf/free-programming-books and Getify/You-Dont-Know-JS) and awesome-lists (e.g., Sindresorhus/awesome). To remove these projects, we relied on their GitHub topics. Specifically, we discarded projects with the topics *book* or *awesome-list*.[1]

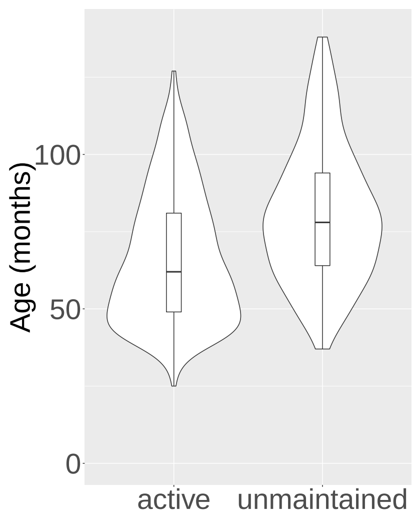
2. *Projects with no lines of code in a set of programming languages*: First, we used the tool AlDanial/cloc[2] to compute the size of the projects, in lines of code (LOC). We configured this tool to only consider code in the top-100 most popular programming languages in the TIOBE list.[3] As a result, we discarded 397 projects, which are implemented in languages like HTML, CSS, and Markdown (i.e., in non-programming languages). For these projects, the size in LOC (counting only source code implemented in major programming languages) is equal to zero. As examples, we removed the following projects: Github/gitignore (which is a collection of textual .gitignore templates), Jlevy/the-art-of-command-line (a selection of notes and tips on using Linux command-line tools), and Necolas/normalize.css (a collection of HTML element and attribute style-normalization).

3. *Inactive projects*: We are interested in projects under active development. Therefore, we discarded 830 repositories without commits in the last six months.

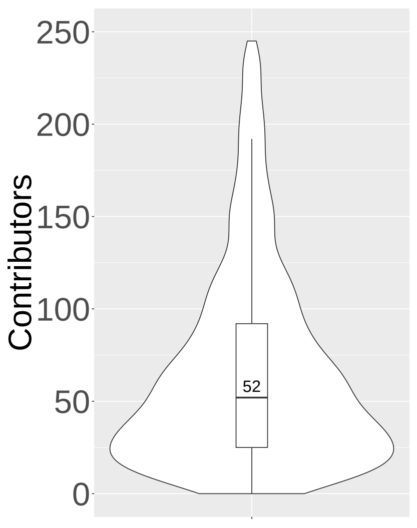
4. *Non-mature projects*: Our central goal is to survey recent core developers of mature FLOSS projects. Particularly, it is important the projects have a minimal age in order to provide enough development time to compute new core developers. For this reason, we discarded 1,450 repositories with less than three years.

We ended up with 2,262 open source systems, including well-known projects, like Facebook/react, Angular/angular, and Rails/rails. Figure 3.1 shows violin plots with the distribution of age (in months), number of contributors, number of commits, and

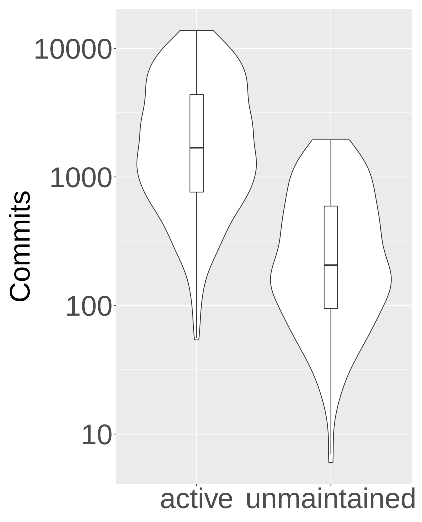
[1]This step represents just a first attempt to remove non-software repositories; step (2) is also used to this purpose.
[2]*https://github.com/AlDanial/cloc*
[3]*https://www.tiobe.com/tiobe-index*



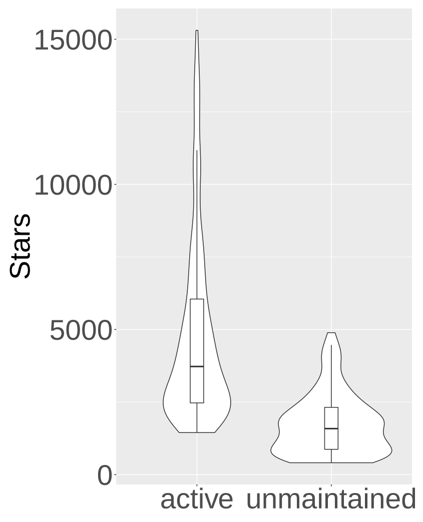
(a) Age



(b) Contributors



(c) Commits



(d) Stars

Figure 3.1: Distribution of the (a) age, (b) contributors, (c) commits, and (d) stars of the selected projects, without outliers.

number of stars of the selected projects, without considering outliers. The median measures are 59 months, 50 contributors, 826 commits, and 3.1K stars. 1,256 projects (55%) are owned by organizations and 1,006 repositories (45%) by individual users. These projects are mainly implemented in JavaScript (696 projects, 31%), followed by Ruby (232 projects, 10%), and Python (230 projects, 10%).

3.2.1 Core Developer Identification

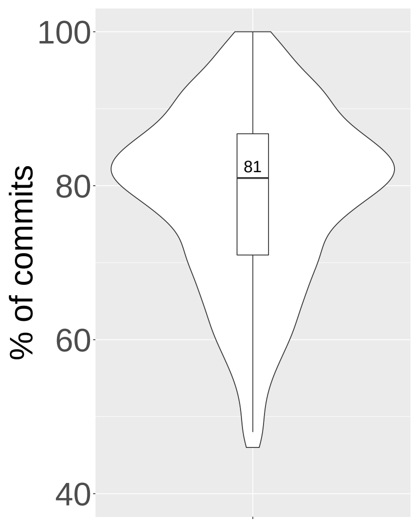
To identify the core developers of each project, we use a Commit-Based Heuristic, which is commonly adopted in other works [Mockus et al., 2002; Robles et al., 2009; Trong and Bieman, 2005; Koch and Schneider, 2002]. This heuristic is centered on the number of commits by the project contributors, which usually follows a heavy-tailed distribution [Mockus et al., 2002; Koch and Schneider, 2002], i.e., a minority of developers accounts for most contributions. According to this heuristic, the core team are those who produce 80% of the overall amount of commits in a project. However, as usually defined, this heuristic accepts developers with few contributions, regarding the total number of commits. For this reason, we customized the heuristic after some initial experiments to require core developers to have at least 5% of the total number of commits; candidates who have fewer commits are excluded. For example, to achieve 80% of the commits in Moment/moment, the core team initially identified by the heuristic consists of 41 contributors. However, 38 contributors have less than 5% of the overall amount of commits. Thus, only three developers are classified as *core* by our customized heuristic. These developers represent 35%, 25% and 7% of the project's commits, respectively. In favor of using this second threshold, the literature reports that even in complex projects, the core team is no bigger than 10-15 developers [Mockus et al., 2002].

Despite the adoption of this second threshold, we can observe in Figure 3.2a that the median percentage of commits by the selected core teams is 81%. Figure 3.2b shows the core team size per project considering the minimal threshold of 5%. We can see that more than half of the selected projects have only one or two core developers. Finally, as presented in Figure 3.2c, the median percentage of commits by the selected core developers is 19%, in contrast to 0.5% using the original strategy.

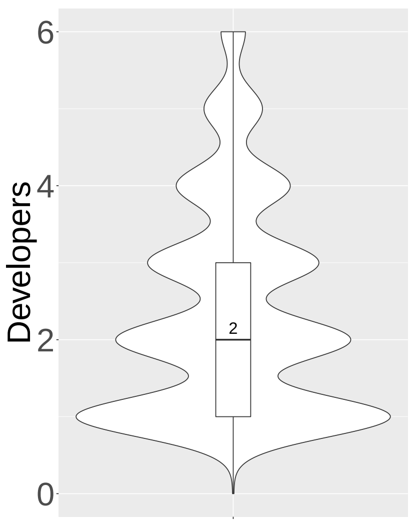
Finally, we follow three steps to select developers who became core contributors in the last year of each project (see an illustration in Figure 3.3): (a) we apply the proposed heuristic on all commits of the project (set A); (b) we remove the last year of commits and recalculate the core team (set B); and (c) the selected set of *core developers* is formed by developers in the set A, but who are not in set B. In other words, this group includes developers who entered in the core team in the last year. We ended up with a list of 380 core developers, distributed over 331 projects.

3.2.2 Survey Design

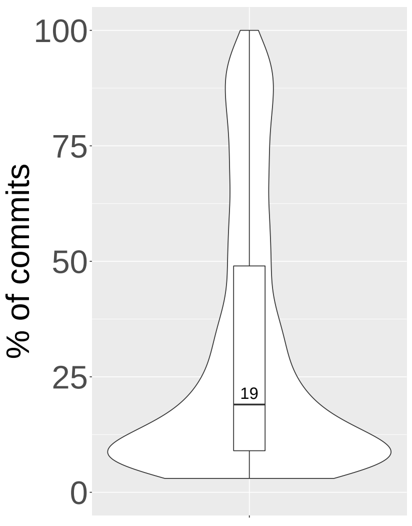
To some extent, our survey can be seen as a firehouse study, i.e., one that is conducted right after the event of interest has happened [Silva et al., 2016; Brito et al., 2018].



(a) Core team contributions



(b) Core team size



(c) Core developer contributions

Figure 3.2: (a) Total percentage of commits by the selected core teams, (b) number of core developers per project, and (c) percentage of commits by the selected core developers. Outliers are omitted in these plots.

Essentially, we surveyed *recent* core developers, to reveal their motivations to engage in FLOSS projects and the main barriers they faced during this process. After removing the core developers who do not have a public email address on GitHub, we obtained a list of 151 potential survey participants. We sent a email to these participants with two parts, as showed in Figure 3.4. First, we include the developer's name and data on his/her percentage of commits in the project. Then, the second part includes three open-ended questions about his/her contributions to the project: (1) What motivated

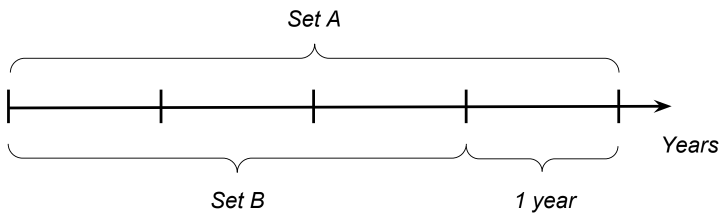


Figure 3.3: Set A= core developers computed considering the complete commit history; Set B= core developers computed considering the commits until the year before the study; *New Core Developers = Set A - Set B*

you to contribute to this project? (2) What project characteristics and practices helped you to contribute? (3) What were the main barriers you faced to contribute?

Dear <Name>,

I found that you have become one of the **main developers** of <Project Name>, with **<x>% of the project's commits**.

Could you please answer three questions about your contributions to this project?

1. What motivated you to contribute to this project?
2. What project characteristics and practices helped you to contribute?
3. What were the main barriers you faced to contribute?

Figure 3.4: Email sent to new core developers.

We received 52 answers (covering distinct projects), which corresponds to a response rate of 34% (and a confidence interval of 11.04 for a confidence level of 95%). Finally, we use Thematic Analysis [Cruzes and Dyba, 2011] to interpret the survey answers. This technique is used for identifying and recording *themes* (i.e., patterns) in textual documents. Thematic Analysis consists of: (1) identifying themes from the answers, (2) reviewing the themes to find patterns for merging, and (3) defining and naming the final themes. The initial theme identification and merge steps were performed independently by the collaborators of this chapter. Then, we had several meetings to resolve conflicts and define the final themes. In the first question, we suggested semantically equivalent themes for 32 answers (62%). These themes were then rephrased and standardized to compose the final theme set. As the remaining 20 answers had divergent themes, they were discussed by the collaborators to reach a consensus. For the last two questions, an initial agreement was reached in 36 (69%)

and 38 (73%) answers, respectively.

3.3 Survey Results

The presentation and discussion of the survey results is organized around the survey questions. To preserve the respondents' anonymity, we use labels D1 to D52 to identify them. Furthermore, when quoting their answers we replace mentions to GitHub repositories, owners, and organizations by *[Project-Name]*, *[Project-Owner]*, and *[Organization-Name]*, respectively. This is important because some answers include sensitive comments about developers or organizations. It is also important to note that a question could have received two or more themes during the thematic analysis process.

3.3.1 Motivations

In the next paragraphs, we present the reasons that emerged for the first survey question (*What motivated you to contribute?*) We discuss each reason and also give examples of answers.

To improve the project because I am using it: According to 31 new core developers, they increased their contributions primarily to fulfill their own needs. As examples, we have the following answers:

I started using it, I ran into minor issues or opportunities to improve, or things that were blocking me from making progress. Since it was an open source project, I was able to contribute improvements and make the project better for my needs, and everyone else's. (D50)

First, I was a very active user of this project at the time. However, I felt that this software could be better. I believed I had enough experience to contribute, so I stepped in. (D15)

I was using [Project-Name] for my startup in our internal dashboards and I needed a couple of features. (D43)

To have a volunteer work: 10 developers answered they contributed to take part in an open source community. As examples, we have the following answers:

I wanted to give back to the community in some small way... (D27)

I'm also in love with the idea of people sharing tools for free in order to help build a better world and promote scientific development and improving people's lives. (D48)

I think that the fact that I was helping a lot of people, immediate feedback, motivated me to contribute more at the difficult time. (D15)

I have interest or expertise on the project domain: According to seven respondents, they were motivated by their interest or expertise on the project domain or programming language. As examples, we have these answers:

I've always had an interest in optimizing things, which I definitely did a lot of in this case. (D06)

I'm well acquainted with the Ruby open source world... (D10)

I am a paid developer: Five new core developers mentioned they were paid to contribute, as in the following answer:

To be honest I'm paid for contributing to [Project-Name]... (D23)

To contribute to a widely used or relevant project: According to four developers, they were motivated by the fact the project is widely used or supported by well-known organizations. As examples, we have the following answer:

Working on a project as large as [Project-Name], and knowing that any work I contribute may be used by thousands of developers, was a pretty good motivator. (D06)

The remaining motivations are as follows: *because I know the maintainer* (3 answers), *to improve my programming skills* (2 answers), *to improve my CV* (2 answers), *because the project has a nice design and implementation* (1 answer), and *to train developers to contribute to FLOSS* (1 answer).

Table 3.1 summarizes the motivations reported by the participants for the first question. Among the 10 motivations mentioned by the developers, only two can be viewed as technical ones (e.g., *because I have interest or expertise on the project domain* and *because the project has a nice design and implementation*). The other motivations are non-technical and related to the interests of the developers or the community and the environment of the project.

Table 3.1: What motivated you to contribute?

Motivations	**Developers**	
To improve the project because I am using it	31	▬
The pleasure of having a volunteer work	10	▮
I have interest or expertise on the project domain	7	▮
I am a paid developer	5	▮
To contribute to a widely used or relevant project	4	▮
Other motivations ($\leq$ three answers each)	9	▮

3.3.2 Project Characteristics and Practices

In the next paragraphs, we present the themes that emerged for the second survey question (*What project characteristics and practices helped you to contribute?*). We describe each reason and also provide examples of answers.

Friendly community: According to 13 developers, they decide to increase their contributions due to the friendly community of project maintainers, who helped with issues and provided detailed feedback when revising pull requests. As examples, we have the following answers:

The main thing that helped me contribute was the friendliness of maintainers and the instructions they've left in the issues they answered. (D48)

The [Project-Name] community gave very detailed feedback during pull requests (sometimes quite strict feedback!) which I found really helpful, and learned a lot about Git in the process. (D27)

Availability of the project leaders: According to 11 developers, the availability of the project leaders helped them to contribute, as in the following example:

In order for people to become contributors, in any kind of open source project, the most important thing is communication and availability of the owner/maintainer. (D07)

Unit tests: According to 9 respondents, the presence of unit tests helped them to increase the number of contributions. As example, we have the following answer:

Unit tests also helped a lot, allowed me to make changes freely with the comfort that I most likely haven't broken anything. (D25)

Documentation: This characteristic, as indicated by eight developers, refers to a

Table 3.2: What project characteristics/practices helped you?

Type	**Characteristics/Practices**	**Developers**	
Technical	Unit tests	9	▮
	Documentation	8	▮
	Well-structured design	4	▏
	Code review	3	▏
	Continuous integration	3	▏
	Programming language	3	▏
	Open source license	3	▏
	Small project	3	▏
	Coding guidelines	2	▏
	Clear code	2	▏
	Contribution guidelines	2	▏
	Other technical characteristics	13	■
Non-Technical	Friendly community	13	■
	Availability of the project leaders	11	■
	Financial support by a company	1	▏
	Open and meritocratic culture	1	▏
	Small number of core developers	1	▏

clear and complete documentation. As examples, we have:

Extended documentation that helped to keep an idea of what it was all about: which things belongs to the project and which do not. (D26)

Documentation for the whole code, especially documentation for setting up development environments of the project, I would really have struggled without that. (D25)

Table 3.2 summarizes the answers for the second question. In addition to the previously mentioned characteristics, we received answers citing *well-structured design* (4 answers), *code review* (3 answers), *continuous integration* (3 answers), *programming language* (3 answers), *open source license* (3 answers), *small project* (3 answers), *coding guidelines* (2 answers), *clear code* (2 answers), *contribution guidelines* (2 answers), *financial support by private company* (1 answer), *large scale tests* (1 answer), and *small number of core developers* (1 answer). In Table 3.2, we also provide a classification of the developers answers in two major groups: technical and non-technical characteristics.

3.3.3 Barriers

In this section, we present and discuss the themes that emerged for the third survey question (*What were the main barriers you faced to contribute?*).

Lack of time of the project leaders: According to eight developers, the main barrier was the absence of the project leaders. As examples, we have the following answers:

Sometimes there were very slow replies to Issues/PRs as there were very few project leaders who could merge them. (D20)

The original developer basically stopped working on it years ago. Many of us were still using the plugin, but bug reports and pull requests built up for years without attention. (D38)

Large and complex project: Seven developers answered that project complexity and size were the main barriers they faced to increase their contributions, as in the example:

The project as a whole is complex and requires specialized knowledge or skill sets that I don't always have. (D45)

Non-clear, complex or buggy codebase: According to five respondents, the main barrier concerns a non-clear, complex, or buggy codebase. As example, we have the following answer:

The code was plagued with race conditions, code smells, bad practices and ugly workarounds. This made it very hard for me to quickly make changes. (D41)

Inappropriate design or architecture: This barrier, mentioned by four respondents, refers to inappropriate design or architecture. As example, we have the following answer:

Less than optimal project structure or release structures. (D07)

Table 3.3 summarizes the responses for the third question. In addition to the previously mentioned barriers, we received answers citing *inexperience of the own contributor* (3 answers), *lack of time of the own contributor* (3 answers), *lack or incompleted documentation* (3 answers), *programming language* (3 answers), *lack of tests* (3 answers), and *conflicts among developers* (3 answers). Furthermore, *English language*, *decisions must be approved by a committee*, *old coding styles*, *hostile attitudes*, *lack of*

Table 3.3: What were the barriers you faced to contribute?

Type	Barriers	Developers	
	Large and complex project	7	▮
	Non-clear, complex or buggy codebase	5	▮
	Inappropriate design or architecture	4	▮
Technical	Lack or incompleted documentation	3	▮
	Programming language	3	▮
	Lack of tests	3	▮
	Other technical barriers	8	▮
	Lack of time of the project leaders	8	▮
	Lack of time of the own contributor	4	▮
	Conflicts among developers	3	▮
Non-Technical	Inexperience of the own contributor	3	▮
	Hostile attitude	1	▮
	Unpaid work	1	▮
	Other non-technical barriers	7	▮

build tools, *project requires specialized knowledge* are other mentioned barriers, all of them with a single answer. Finally, six (11%) participants answered they faced no barriers. As we can see, there is in this case a balance between technical and non-technical barriers, which received 33 and 27 answers, respectively.

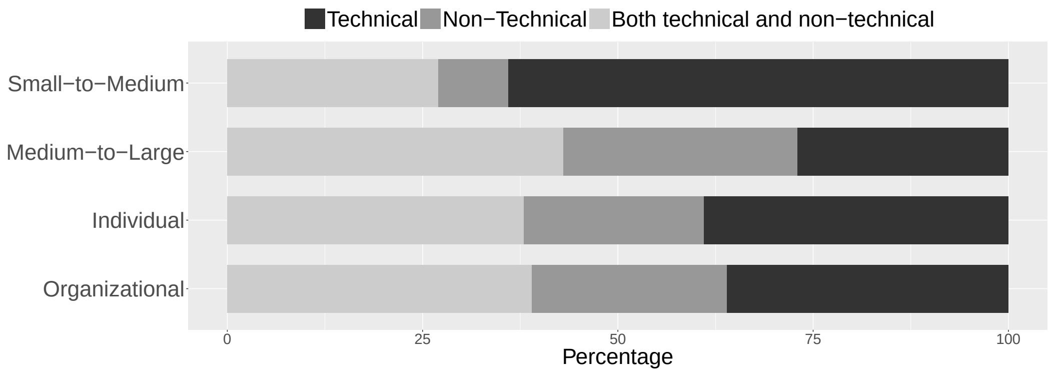
3.4 Analysis by Project Categories

In this section, we provide results grouped by the following categories of projects: *small-to-medium* vs *medium-to-large* projects and *individual* vs *organizational* projects.

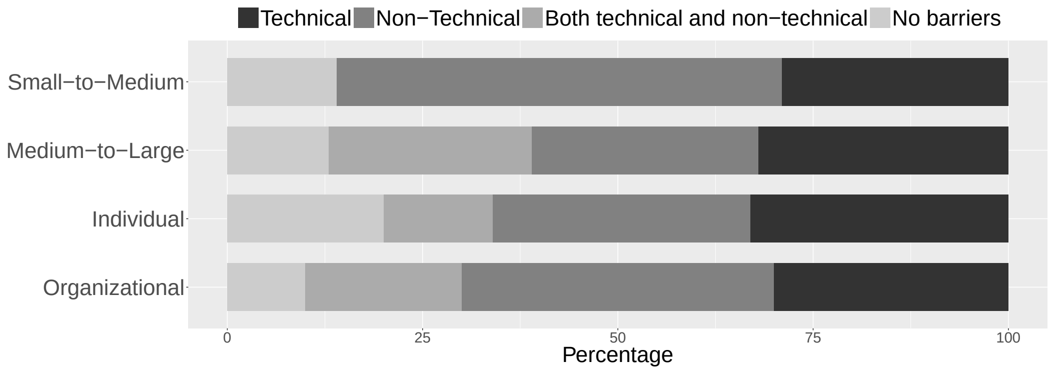
Project Categories: We classify the 52 projects according to their size, considering the distribution of LOC of the 2,262 projects. The projects in the first and second quartiles are classified as *Small-to-Medium* projects (LOC $\leq$ 4,894); the ones in the third and fourth quartiles are named *Medium-to-Large* projects (LOC $>$ 4,894). We ended up with a list of 17 *Small-to-Medium* and 35 *Medium-to-Large* projects.

We also group the 52 projects considering the type of the account on GitHub: 18 projects are developed using individual accounts (e.g., Javan/whenever) and 34 projects using an organizational account (e.g., Google/guava).

Results: Figure 3.5a shows the results for project characteristics and practices. The figure shows the percentage of technical, non-technical, and both technical and non-



(a) Project characteristics and practices



(b) Barriers

Figure 3.5: Results grouped by project categories (Small-to-Medium, Medium-to-Large, Individual, and Organizational)

technical characteristics. For example, developers contributed to individual projects exclusively due to their technical (24%), non-technical (38%), and both technical and non-technical characteristics (38%). According to the results in Figure 3.5a, technical characteristics are the most important factor in small-to-medium projects (63%). By contrast, they are exclusively responsible to the engagement of core developers in only 26.7% of the medium-to-large projects. In these projects, most answers include a combination of technical and non-technical factors (43%). Finally, there is no major difference in the results for individual and organizational projects. For example, technical factors are the only factors responsible by the engagement of core developers in 38.5% of the individual projects and 35.7% of the organizational ones.

Figure 3.5b shows the breakdown results for the barriers faced by the surveyed developers. First, the percentage of projects presenting no barrier ranges from 10% (organizational projects) to 20% (individual projects). The most common barriers in small-to-medium projects are exclusively non-technical ones (57.1%). In other words, in small-to-medium projects, developers are attracted by their technical characteristics, but often face non-technical barriers. As example, we have this answer from a core developer about the technical characteristics and practices of a small-to-medium project:

Proper coding guidelines and documentations do help a lot. (D05)

But he also complained about non-technical barriers:

Not all contributors have a consistent and equivalent share of time to invest in the project. This sometimes stalls the progress . . . (D05)

Regarding medium-to-large projects, we found a balance among technical barriers (32%), non-technical barriers (29%), and both types of barriers (26%). As in the case of project characteristics, there is no major difference in the results for individual and organizational projects.

In summary, we found that core developers engaged in small-to-medium projects mostly due to their technical characteristics (e.g., unit tests), but often face non-technical barriers (e.g., lack of time of the project leaders). In medium-to-large projects, the surveyed core developers increased their contributions due to a combination of both technical and non-technical characteristics; they also faced both technical and non-technical barriers. Finally, we found that there is no major difference between individual and organizational projects, regarding their characteristics and offered barriers.

3.5 Threats To Validity

The threats to validity of this work are as follows:

External Validity: The dataset used in this study is restricted to popular open source projects on GitHub. We acknowledge that there are popular projects in other platforms (e.g., Bitbucket and GitLab) or projects that have their own version control infrastructure.

Internal Validity: This threat relates to the themes denoting the survey answers. We acknowledge that the selection of these themes is to some extent subjective. For example, it is possible that different researchers reach a different set of motivations, practices and barriers, than the ones proposed in Section 3.3. To mitigate this threat, the initial theme selection was performed independently by the collaborators of this study. After this initial proposal, several meetings were performed to refine and improve the initial selection.

Construct Validity: A construct validity threat relates to the commit-based heuristic for core developer identification. However, we decided to use a traditional heuristic to this purpose, widely used in other studies [Trong and Bieman, 2005; Koch and Schneider, 2002; Mockus et al., 2002]. Furthermore, we customized this heuristic to exclude developers with few commits (less than 5% of the total number of commits).

3.6 Related Work

In this section, we first compare our results with related studies which focused on three profiles of open source contributors:

- *Casual Contributors* are those that performed at most one commit to a software project and who do not want to become active project members. Pinto et al. [2016] conduct surveys with (1) casual contributors to understand what motivates them to contribute and (2) with project maintainers to understand how they perceive casual contributions.

- *One-Time Code Contributors (OTC)* are developers who have exactly one accepted patch. OTCs are a subset of the casual contributors. Lee et al. [2017] conduct a survey with OTCs to comprehend their impressions, motivations, and barriers, when contributing to FLOSS.

- *Newcomers* are those contributors who attempted to conclude their first contribution to an open source project. Steinmacher et al. [2016] elicit 58 barriers that may hinder newcomers onboarding to open source projects.

Table 3.4 contrasts our results with the aforementioned studies. The most common motivation for OTCs and casual contributors is *bug fixing* because it can affect their work. In contrast, the most common motivation for core developers engagement, as revealed in our survey, is *improving the project because I am using it.* Therefore, their motivation include not only bug fixing tasks, but also adding new features. Lee et al. [2017] investigate impressions that increase the chances of a potential developer to contribute to a project. The most common positive impression reported by OTCs is the presence of skilled, friendly, and helpful project members. Similarly, we found that core developers are also attracted by a *friendly community* and by the *availability*

Table 3.4: Comparison of our findings with related studies.

Topic	**Our study**	**Lee et al. [2017]**	**Pinto et al. [2016]**	**Steinmacher et al. [2016]**
Contributors	Core developers	One-Time code Contributors	Casual contributors	Newcomers
Motivations	To improve the project because I am using it	To fix bugs	To fix bugs	x
	To the pleasure of having a volunteer work	The desire to give back to the community	To improve documentation	
	Because I have interest or expertise on the project domain	I am a paid developer	To add new features	
Project characteristics	Friendly community	Skilled project members	x	x
	Availability of the project leaders	Friendly project members		
	Unit tests	Helpful project members		
Barriers	Lack of time of the project leaders	Lack of time of the own contributor	Lack of time of the own contributor	Technical barriers
	Large and complex project	Complex submission process	Limited skills or knowledge	Lack of contribution guidelines
	Non-clear, complex or buggy codebase	Complex project	Complex project	Lack of documentation

of the project leaders. However, the third characteristic cited by core developers is the presence of *unit tests*, while on the case of OTCs are helpful project members. The most common barrier faced by OTCs and casual contributors is *lack of time of the own contributor*. By contrast, only three core developers reported this fact as a main impediment.

In a previous work (Chapter 2), we conduct an investigation with maintainers of 104 open source projects that failed to understand the reasons of such failures. The most common reasons are projects that were usurped by competitors (27 projects), obsolete projects (20 projects), lack of time of the main contributors (18 projects), and lack of interest of the main contributors (17 projects). Robles et al. [2014] describe a curated dataset with data from over 2,000 FLOSS contributors. Among the collected data, this dataset includes the contributors motivations for joining FLOSS. However, these answers can be seen as general motivations; in our survey, we decided to ask specific developers (core developers) about their motivations for recently joining well-defined open source projects.

In a recent survey promoted by GitHub with thousands of open source developers, documentation was indicated as a pervasive problem when contributing to open source, according to 93% of the respondents (see *http://opensourcesurvey.org/2017*). However, in our survey, restricted to core developers, documentation is mentioned as barrier by only three participants.

Eghbal [2016] reports on the risks and challenges to maintain open source projects. Ye and Kishida [2003] describe a study to understand what motivates developers to engage in open source development. Other studies on open source have focused on how to attract and retain contributors [Zhou and Mockus, 2015; Canfora et al., 2012; Bosu et al., 2014]. Gousios et al. [2015, 2016, 2014] provide insights on the pull-based development model as implemented in GitHub from the integrator and contributors' perspective. Mirhosseini and Parnin [2017] investigate the use of pull request notifications in GitHub projects. Jiang et al. [2016] examine why and how developers fork repositories on GitHub. They found that developers fork repositories to submit pull requests, fix bugs, add new features, and keep copies. Joblin et al. [2017] categorized core and peripheral developers based on social and technical perspectives.

3.7 Implications

Our study has implications both to practitioners and researchers, as follows:

Implications to Practitioners: First, core contributors should strive to provide an interesting and high-quality software product, which can attract a large base of users. Then, some of these users will decide to improve the product to fulfill their own needs. Finally, they will share the improvements with the project community, which can trigger a new cycle of improvements. Second, two non-technical practices are important to engage core developers in open source projects: nurturing a friendly community and being always available. However, technical factors—specially, the availability of unit tests and documentation—are also important. Third and last, the main barrier faced by new core contributors is also non-technical, the lack of time of project leaders, followed by two technical ones: project complexity and low quality code.

Implications to Researchers: First, open source projects are increasingly important elements of the digital infrastructure that supports our modern societies [Eghbal, 2016]. We also know that these projects depend on a small number of core developers [Mockus et al., 2002; Avelino et al., 2016]. Thus, researchers should continue to investigate strategies to improve open source practices and communities. Particularly, our findings might contribute to current efforts to develop health and analytics models and tools to open source projects, as proposed for example by the CHAOSS[4] and SECOHealth projects [Mens et al., 2017]. Second, we also showed the importance of requiring a minimal percentage of commits when identifying core developers. When this threshold is not applied, the traditional heuristic can select core developers with very few commits, which are included just to reach the total amount of 80% of the commits of a system.

3.8 Conclusion

In this chapter, we reported the results of a survey with 52 developers, who recently became core contributors of popular GitHub projects. We revealed their motivations to assume a key role in FLOSS projects (e.g., improving the projects because they are also using it), the project characteristics that most helped in their engagement process (e.g., a friendly community), and the barriers faced by the surveyed core developers (e.g., lack of time of the project leaders). We also compared our results with related studies about others kinds of open source contributors (casual and newcomers).

[4] *https://chaoss.community/*

Chapter 4

Identifying Unmaintained Projects in GitHub

GitHub hosts an impressive number of high-quality OSS projects. However, selecting "the right tool for the job" is a challenging task because we do not have precise information about those high-quality projects. In this study, we propose a data-driven approach to measure the level of maintenance activity of GitHub projects. Our goal is to alert users about the risks of using unmaintained projects and possibly motivate other developers to assume the maintenance of such projects. We train machine learning models to define a metric to express the level of maintenance activity of GitHub projects. Next, we analyze the historical evolution of 2,927 active projects in the time frame of one year. From 2,927 active projects, 16% become unmaintained in the interval of one year. We also found that Objective-C projects tend to have lower maintenance activity than projects implemented in other languages. Finally, software tools—such as compilers and editors—have the highest maintenance activity over time. A metric about the level of maintenance activity of GitHub projects can help developers to select open source projects.

4.1 Introduction

Open source projects have an increasing relevance in modern software development [Eghbal, 2016], powering applications in practically every domain. In fact, it is common nowadays to rely on open source libraries and frameworks when building and evolving proprietary software. For example, in a recent survey—conducted by Black Duck Software—86% of the surveyed organizations report the use of open source in their daily development.[1] Furthermore, the emergence of world-wide code sharing platforms—such as GitHub—is contributing to transform open source development in a competitive market. Indeed, in a recent survey with maintainers we found that the most common reason for the failure of open source projects is the appearance of a stronger competitor in GitHub (Chapter 2).

However, GitHub does not include objective data about project's maintenance activity. Users can access historical data about commits or repository popularity metrics, like number of stars, forks, and watchers. However, based on the values of these metrics, they should judge by themselves whether a project is under maintenance or not (and therefore whether it is worth to use it). In order to help on this decision, in this study, we propose and evaluate a machine learning approach to measure the level of maintenance activity of GitHub projects. Our goal is to provide a simple and effective metric to alert users about the risks of depending on a given GitHub project. This information can also contribute to attract new maintainers to a project. For example, users of libraries facing the risks of discontinuation can be motivated to assume their maintenance.

Previous work in this area relies on the last commit activity to classify projects as unmaintained or in a similar status. For example, Khondhu et al. [2013] use an one-year inactivity threshold to classify *dormant* projects on SourceForge. The same threshold is used in works by Mens et al. [2014], Izquierdo et al. [2017], and in our previous work about the motivations for the failure of open source projects (Chapter 2). However, in this study, we do not rely on such thresholds when investigating unmaintained projects due to three reasons. First, because setting a threshold to characterize unmaintained projects is not trivial. For example, in the mentioned works, this decision is arbitrary and it is not empirically validated. Second, our intention is to detect unmaintained projects as soon as possible; preferably, without having to wait for one year of inactivity (or another threshold). Third, our definition of unmaintained projects does not require a complete absence of commits during a given period; instead, a project is considered unmaintained even when sporadic and few commits happen in a time interval. Stated otherwise, in our view, unmaintained projects do not necessarily need to be dead, deprecated, or archived.

In this chapter, we first train ten machine learning models to identify unmaintained projects, using as standard metrics provided by GitHub about a project's maintenance activity, e.g., number of commits, forks, issues, and pull requests. Then, we select the model with the best performance and validate it by means of a survey

[1] *https://pt.slideshare.net/blackducksoftware/you-cant-live-without-open-source-results-from-the-open-source-360-survey*

with the owners of projects classified as *unmaintained* and also with a set of deprecated GitHub projects. Particularly, we ask five research questions about properties of this model:

RQ1: What is the precision of the proposed machine learning model according to GitHub developers? The intention was to check precision in the field, by collecting to the feedback provided by the principal developers of popular GitHub projects.

RQ2: What is the recall of the proposed machine learning model when identifying unmaintained projects? Recall is more difficult to compute in the field because it requires the identification of *all* unmaintained projects in GitHub. To circumvent this problem, we compute recall considering only projects that declare in their README they are not under maintenance. To answer this question, we construct a ground truth with projects that are no longer being maintained.

RQ3: How early does the proposed machine learning model identify unmaintained projects? As mentioned, the proposed model does not depend on an inactivity interval to classify a project as unmaintained. Therefore, in this third question, we investigate how early we are able to identify unmaintained projects, e.g., without having to wait for a full year of commit inactivity.

RQ4: How long does a GitHub project survive before become unmaintained? The goal is to investigate the survival probability over time of the projects classified as unmaintained by the proposed machine learning model. Moreover, we analyze the survival probability of these projects under different perspectives (e.g., organizational or individual account, programming language, and application domain).

RQ5: How often unmaintained projects follow best OSS maintenance practices? We investigate whether projects classified as unmaintained follow a set of best open source maintenance practices, recommended by GitHub, such as presence of contributing guidelines, presence of project's license, and use of a continuous integration service.

Our contributions are twofold: (1) we propose a machine learning approach to identify unmaintained (or sporadically maintained) projects in GitHub, which achieved a precision of 80% and a recall of 96% when validated with real open source developers and projects; and (2) we propose a metric to reveal the maintenance activity level of GitHub projects.

This chapter is organized as follows. In Section 4.2, we present and evaluate a machine learning model to identify unmaintained projects. Section 4.3 validates this model with GitHub developers and projects that are documented as deprecated. In Section 4.4, we assess the characteristics of projects classified as unmaintained by our model. Section 4.5 defines and discusses the Level of Maintenance Activity (LMA) metric. Section 4.6 lists threats to validity and Section 4.7 discusses related work. Section 4.8 concludes the chapter and outlines further work.

4.2 Machine Learning Model

In this section, we describe our machine learning approach to identify projects that are no longer under maintenance.

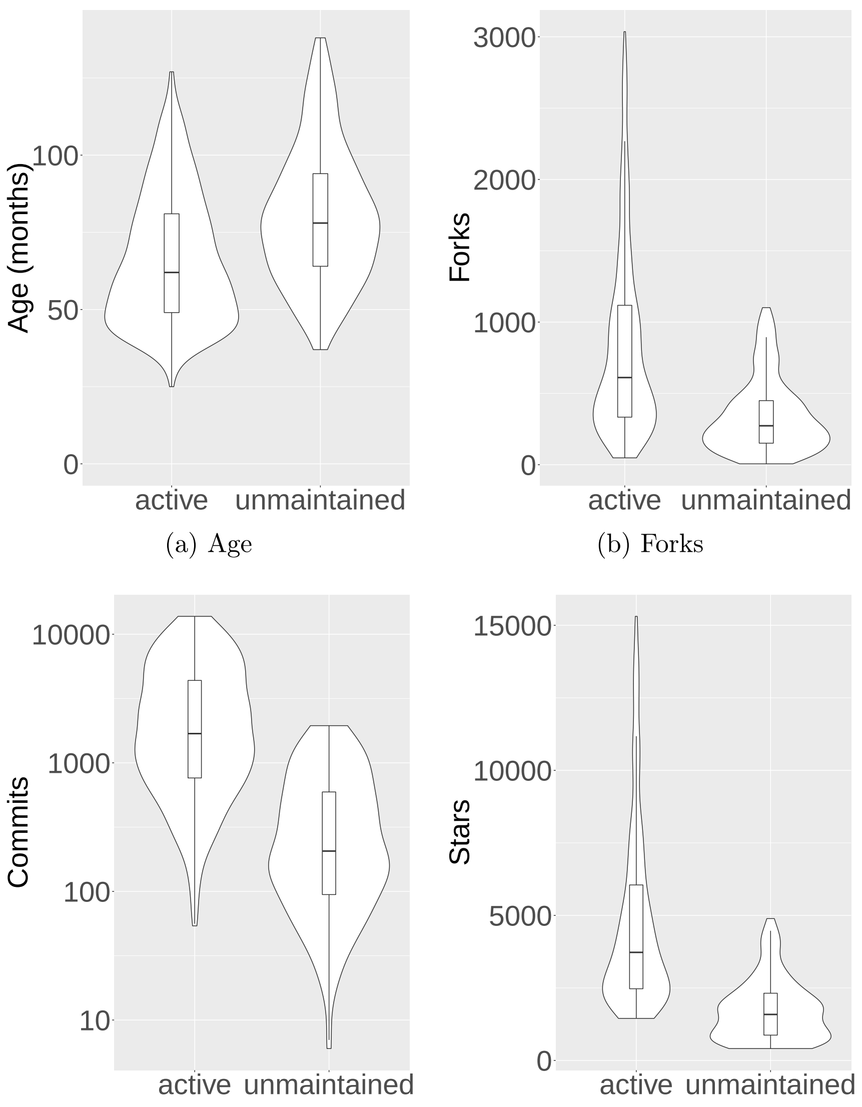
4.2.1 Experimental Design

Dataset. We start with a dataset containing the top-10,000 most starred projects on GitHub (in November, 2017). Stars—GitHub's equivalent for *likes* in other social networks—is a common proxy for the popularity of GitHub projects [Borges et al., 2016b; Borges and Valente, 2018]. Then, we follow three strategies to discard projects from this initial selection. First, we remove 2,810 repositories that have less than two years from the first to the last commit (because we need historical data to compute the features used by the prediction models). Second, we remove 331 projects with null size, measured in lines of code (typically, these projects are implemented in non-programming languages, such as CSS, HTML, etc.). Finally, we remove 74 non-software projects, which are identified by searching for the topics in Table 4.1. We end up with a list of 6,785 projects.

Table 4.1: Topics used to discard projects.

project-based tutorials, *project-based-learning*, *programming tutorials*, *tutorial*, *books*, *awesome-lists*, *example*, *toy-project*, *documentation*

Next, we define two subsets of systems: *active* and *unmaintained*. The active (or under maintenance) group is composed by 754 projects that have at least one release in the last month, including well-known projects, such as FACEBOOK/REACT, D3/D3, and NODEJS/NODE. We adopt this strategy because a release constitutes a fully functional version of a software. Thus, we assume that these projects are active (under maintenance). By contrast, the unmaintained group is composed by 248 projects, including 104 projects that were explicitly declared by their principal developers as unmaintained in our previous work (Chapter 2) and 144 *archived* projects. Archiving is



(a) Age (b) Forks

(c) Commits (d) Stars

Figure 4.1: Distribution of the (a) age, (b) forks, (c) commits, and (d) stars, without outliers.

a feature provided by GitHub that allows developers to explicitly move their projects to a read-only state. In this state, users cannot create issues, pull requests, or comments, but can still fork or star the projects.

Figure 4.1 shows violin plots with the distribution of age (in months), number of forks, number of commits, and number of stars of the selected repositories. We provide plots for the 754 *active* projects and for the 248 *unmaintained* projects. As we can check, unmaintained projects are older than the active ones (78 vs 62 months, median measures); but they have less forks (299 vs 735), less commits (241 vs 2,136), and less stars (1,714 vs 4,078). In our dataset, active projects are composed by 74% of organizational projects and 26% of user projects. By contrast, unmaintained projects consists of 37% and 63% of organizational and user projects, respectively (Figure 4.2).

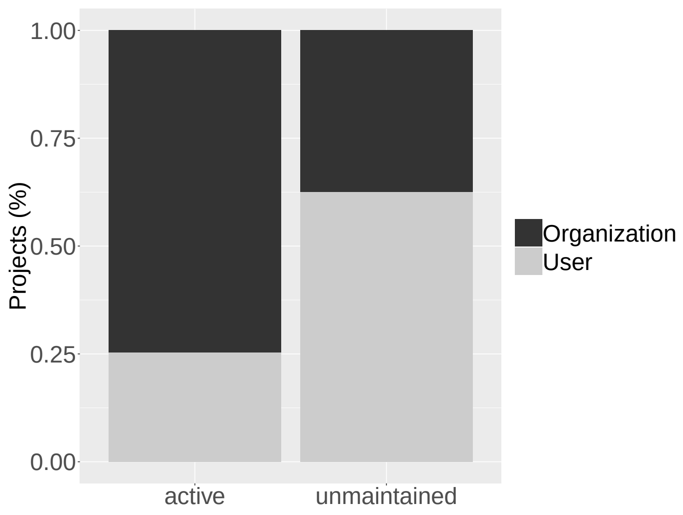


Figure 4.2: Number of projects owned by a person or by an organization.

Features. Our hypothesis is that a machine learning classifier can identify unmaintained projects by considering features about (1) projects, including number of forks, issues, pull requests, and commits; (2) contributors, including number of new and distinct contributors (the rationale is that maintenance activity might increase by attracting new developers); (3) project owners, including number of projects he/she owns and total number of commits in GitHub (the rationale is that maintenance might be affected when project owners have many projects on GitHub). In total, we consider 13 features, as described in Table 4.2. The feature values are collected using GitHub's official API. However, they do not refer to the whole history of a project, but only to the last n months, counting from the last commit; moreover, we collect each feature in intervals of m months. The goal is to derive temporal series of feature values, which can be used by a machine learning algorithm to infer trends in the project evolution, e.g., an increasing number of opened issues or a decreasing number of commits. Figure 4.3 illustrates the feature collection process assuming that n is 24 months and that m is 3 months. In this case, for each feature, we collect 8 data points, i.e., feature values. We call data points all the features calculated for each considered interval.

We experiment with different combinations of n and m; each one is called a scenario, in this chapter. Table 4.3 describes the total number of data points extracted for each scenario. This number ranges from 13 data points (scenario with features

Table 4.2: Features used to identify unmaintained projects.

Dimension	Feature	Description
Project	Forks	Forks created by developers
	Open issues	Issues opened by developers
	Closed issues	Issues closed by developers
	Open pull requests	Pull requests opened by developers
	Closed pull requests	Pull requests closed by developers
	Merged pull requests	Pull requests merged by developers
	Commits	Commits performed by developers
	Max days without commits	Maximum number of consecutive days without commits
	Max contributions by developer	Commits of the developer with most commits
Contributor	New contributors	Contributors who made their first commit
	Distinct contributors	Distinct contributors that committed
Owner	Projects created by the owner	Projects created by a given owner
	Number of commits of the owner	Commits performed by a given owner

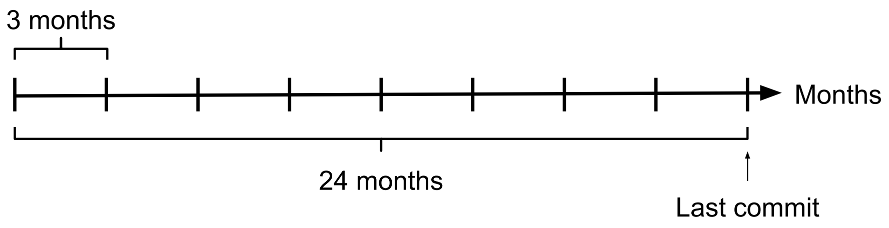


Figure 4.3: Feature collection during 24 months in 3-month intervals.

extracted in a single interval of 6 months) to 104 data points (scenario with features extracted in intervals of 3 months during 24 months, as in Figure 4.3). We limited our

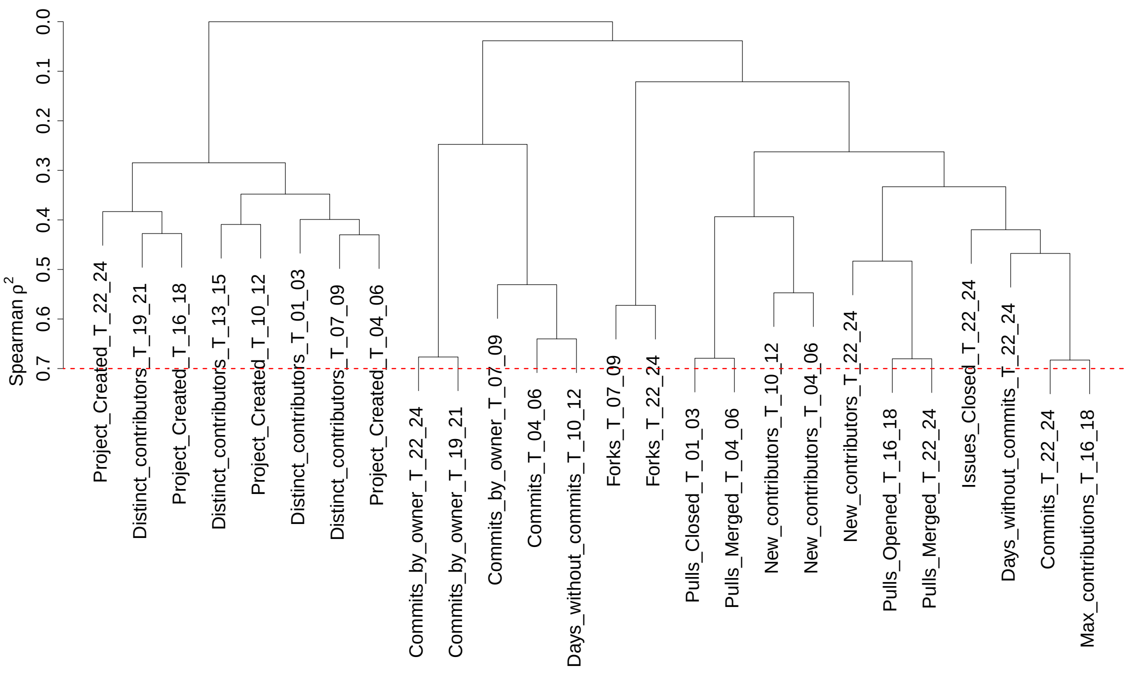


Figure 4.4: Correlation analysis for the 104 data points collected for the features in scenario 8 (24 months, 3-month interval). 78 data points (75%) are removed in this case, due to correlations with other data points, and therefore do not appear in this final clustering.

analysis in 24 months to avoid exclude a considerable number of projects.

Table 4.3: Scenarios used to collect features and train the machine learning models (length and intervals are in months; data points is the total number of data points collected for each scenario).

Scenario	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Length	6	6	12	12	12	18	18	24	24	24
Intervals	3	6	3	6	12	3	6	3	6	12
Data points	26	13	52	26	13	78	39	104	52	26

Correlation Analysis. As usual in machine learning experiments, we remove correlated features, following the process described by Bao et al. [2017]. To this purpose, we use a clustering analysis—as implemented in a R package named *Hmisc*[2]—to derive a hierarchical clustering of correlations among data points (extracted for the features in each scenario). For sub-hierarchies with correlations larger than 0.7, we select only one data point for inclusion in the respective machine learning model, as common in other works [Bao et al., 2017; Tian et al., 2015c]. For example, Figure 4.4

[2]*http://cran.r-project.org/web/packages/Hmisc/index.html*

shows the final hierarchical clustering for the scenario with 24 months, considering a 3-month interval (scenario 8). The analysis in this scenario checks correlations among 104 data points (13 features $\times$ 8 data points per feature). As a result, 78 data points are removed due to correlations with other points and therefore do not appear in the dendogram presented in Figure 4.4. Finally, Table 4.4 shows the total number and percentage of data points removed in each scenario, after correlation analysis. As we can see, the percentage of removed points is relevant, ranging from 43% (scenario 7) to 75% (scenario 8).

Table 4.4: Total number and percentage of data points removed in each scenario, after correlation analysis.

Scenario	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
#	17	6	38	18	7	56	17	78	34	19
%	65	46	73	69	54	72	43	75	65	73

Machine Learning Classifier. We use the data points extracted in each scenario to train and test models for predicting whether a project is unmaintained. In other words, we train and test ten machine learning models, one for each scenario. After that, we select the best model/scenario to continue with the chapter. Particularly, we use the Random Forest algorithm [Breiman, 2001] to train the models because it has several advantages, such as robustness to noise and outliers [Tian et al., 2015c]. In addition, it is adopted in many other software engineering works [Menzies et al., 2013; Peters et al., 2013; Provost and Fawcett, 2001; Hora et al., 2016]. We compare the result of Random Forest with two baselines: baseline #1 (all projects are predicted as unmaintained) and baseline #2 (random predictions). We use the Random Forest implementation provided by *randomForest*'s R package[3] and 5-fold stratified cross validation to evaluate the models effectiveness. In 5-fold cross validation, we randomly divide the dataset into five folds, where four folds are used to train a classifier and the remaining fold is used to test its performance. Specifically, stratified cross validation is a variant, where each fold has approximately the same proportion of each class [Breiman, 2001]. We perform 100 rounds of experiments and report average results.

Evaluation Metrics. When evaluating the projects in the test fold, each project has four possible outcomes: (1) it is truly classified as unmaintained (True Positive); (2) it is classified as unmaintained but it is actually an active project (False Posi-

[3] *https://cran.r-project.org/web/packages/randomForest/*

Table 4.5: Prediction results (mean of 100 iterations, using 5-cross validation); best results are in bold.

Metrics	**0.5 Year**		**1 Year**			**1.5 Years**		**2 Years**		
	3 mos	6 mos	3 mos	6 mos	12 mos	3 mos	6 mos	3 mos	6 mos	12 mos
Accuracy	0.90	0.91	0.91	0.90	0.89	0.91	0.90	**0.92**	0.91	0.90
Precision	0.83	0.87	0.87	0.84	0.82	0.86	0.83	**0.86**	0.85	0.83
Recall	0.78	0.74	0.77	0.75	0.72	0.78	0.76	**0.81**	0.79	0.73
F-measure	0.80	0.79	0.81	0.79	0.77	0.82	0.79	**0.83**	0.82	0.78
Kappa	0.74	0.74	0.76	0.73	0.70	0.76	0.73	**0.78**	0.76	0.71
AUC	0.86	0.85	0.86	0.85	0.83	0.87	0.85	**0.88**	0.87	0.84

tive); (3) it is classified as an active project but it is actually an unmaintained one (False Negative); and (4) it is truly classified as an active project (True Negative). Considering these possible outcomes, we use six metrics to evaluate the performance of a classifier: precision, recall, F-measure, accuracy, AUC (Area Under Curve), and Kappa, which are commonly adopted in machine learning studies [Tian et al., 2015c,a; da Costa et al., 2014; Lamkanfi et al., 2010; Lessmann et al., 2008]. Precision and recall measure the correctness and completeness of the classifier, respectively. F-measure is the harmonic mean of precision and recall. Accuracy measures how many projects are classified correctly over the total number of projects. AUC refers to the area under the Receiver Operating Characteristic (ROC) curve. Finally, kappa evaluates the relationship between the observed accuracy and the expected one [Ramasubramanian and Singh, 2017], which is particularly relevant in imbalanced datasets, as the dataset used to train the machine learning models (754 active projects *vs* 248 unmaintained ones).

4.2.2 Experimental Results

Table 4.5 shows the results for each scenario. As we can see, Random Forest has the best results (in bold) when the features are collected during 2 years, in intervals of 3 months. In this scenario, precision is 86% and recall is 81%, leading to a F-measure of 83%. Kappa is 0.78—usually, kappa values greater than 0.60 are considered quite representative [Landis and Koch, 1977]. Finally, AUC is 0.88, which is an excellent result in the Software Engineering domain [Lessmann et al., 2008; Thung et al., 2012; Tian et al., 2015c]. Table 4.6 compares the results of the best scenario/model with baseline #1 (all projects are predicted as unmaintained) and baseline #2 (random predictions). Despite the baseline under comparison, there are major differences in all evaluation metrics. For example, F-measure is 0.37 (baseline #1) and 0.30 (baseline #2), against 0.83 (proposed model).

Random Forest produces a measure of the importance of the predictor features.

Table 4.6: Comparison of the proposed machine learning model with baseline #1 (all projects are predicted as unmaintained) and baseline #2 (random predictions).

Metrics	**Model**	**Baseline #1**	**Baseline #2**
Accuracy	0.92	0.22	0.49
Precision	0.86	0.22	0.22
Recall	0.81	1.00	0.48
F-measure	0.83	0.37	0.30
Kappa	0.78	0.00	0.01
AUC	0.88	0.50	0.49

Table 4.7 shows the top-5 most important features by Mean Decrease Accuracy (MDA), for the best model. Essentially, MDA measures the increase in prediction error (or reduction in prediction accuracy) after randomly shifting the feature values [Calle and Urrea, 2010; Louppe et al., 2013]. As we can see, the most important feature is the number of commits in the last time interval (i.e., the interval delimited by months 22-24, $T_{22,24}$), followed by the maximal number of days without commits in the same interval and in the interval $T_{10,2}$. As also presented in Table 4.7, the first four features are related to commits; the first feature non-related with commits is the number of issues closed in the first time interval ($T_{1,3}$).

Table 4.7: Top-5 most relevant features, by Mean Decrease Accuracy (MDA).

Feature	**Period**	**MDA**
Commits	$\mathrm{T}_{22,24}$	38.5
Max days without commits	$\mathrm{T}_{22,24}$	28.6
Max days without commits	$\mathrm{T}_{10,12}$	21.9
Max contributions by developer	$\mathrm{T}_{16,18}$	21.1
Closed issues	$\mathrm{T}_{1,3}$	18.0

4.3 Empirical Validation

In this section, we *validate* the proposed machine learning model by means of a survey with the owners of projects classified as *unmaintained* and also with a set of deprecated GitHub projects. Overall, our goal is to strengthen the confidence on the practical value of the model proposed in this work. Particularly, in this section we provide answers to first three research questions about this model:

RQ1: What is the precision according to GitHub developers?

RQ2: What is the recall when identifying deprecated projects?

RQ3: How early does the model identify unmaintained projects?

4.3.1 Methodology

RQ1: To answer RQ1, we conduct a survey with GitHub developers. To select the participants, we first apply the proposed machine learning model in all projects from our dataset that were not used in the model's construction, totaling 5,783 projects (6,785 − 1,002 projects). Then, we select 2,856 projects classified as unmaintained by the proposed model. From this sample, we remove 264 projects whose developers were recently contacted in our previous surveys (Chapter 2 and 3). We make this decision to not bother again these developers, with new emails and questions. Finally, we remove 2,270 projects whose owners do not have a public email address on GitHub. As a result, we obtain a list of 323 survey participants (2,856 − 2,270 − 264). However, before emailing these participants, we inspected the main page of each project on GitHub, to check whether it includes mentions to the project status, in terms of maintenance. We found 21 projects whose documentation states they are no longer maintained, by means of messages like this one:

This project is deprecated. It will not receive any future updates or bug fixes. If you are using it, please migrate to another solution.

Therefore, we do not send emails to the project owners, in such cases; and automatically consider these 21 projects as *unmaintained.*

Survey Period: The survey was performed in the first two weeks of May, 2018. It is important to highlight that the machine learning model was constructed using data collected on November, 2017. Therefore, the *unmaintained* predictions evaluated in the survey refer to this date. We wait five months to ask the developers about the status of their projects because it usually takes some time until developers actually accept the unmaintained condition of their projects. In other words, this section is based on predictions performed and available on November, 2017. However, these predictions are validated five months later, on May, 2018.

Survey Pilot and Questions: Initially, we perform a pilot survey with 75 projects ($\approx$ 25%), randomly selected among the remaining 302 projects (323 − 21 projects). We email the principal developers of these projects with a single open question, asking them to confirm (or not) if their projects are unmaintained. We received 23 answers, which corresponds to a response ratio of 30.6%. Then,we analyzed the received answers to derive a list of recurrent themes. As a result, the following three common maintainability status were identified:[4]

1. **My project is under maintenance and new features are under implementation (6 answers):** As an example, we can mention the following answer:

   [Project-Name] is still maintained. I maintain the infrastructure side of the project myself (e.g., make sure it's compatible with the latest Ruby version, coordinate PRs and issues, emailing list, etc.) while community provides features that are still missing. One such big feature is being developed as we speak and will be the highlight of the next release. (P57)

2. **My project is finished and I only plan to fix important bugs (13 answers):** As an example, we mention the following answers:

   It's just complete, at least for now. I still fix bugs on the rare occasion they are reported. (P10)

   I view it as basically "done". I don't think it needs any new features for the foreseeable future, and I don't see any changes as urgent unless someone discovers a major security vulnerability or something. I will probably continue to make changes to it a couple times per year, but mostly bug fixes. (P68)

3. **My project is deprecated and I do not plan to implement new features or fix bugs (4 answers):** As an example, we can mention the following answer:

   The project is unmaintained and I'll archive it. (P74)

After the pilot study, we proceed with the survey, by emailing the remaining 227 projects. However, instead of asking an open question—as in the pilot—we provide an objective question to the survey participants, about the maintainability status of their projects. In this objective question, we ask the participants to select one (out of three) status identified in the pilot study, plus an *other* option. This fourth option also

[4]Project names are omitted, to preserve the respondent's anonymity; survey participants are identified by means of labels, in the format Pxx, where *xx* is an integer.

includes a text form for the participants detailing their answers, if desired. Essentially, we change to an objective question format to make answering the survey easier, but without limiting the respondents' freedom to provide a different answer from the listed ones. In this final survey, we received 89 answers, representing a response ratio of 39.2%. When considering both phases (pilot and final survey), we sent 302 emails, received 112 answers, representing an overall response ratio of 37.1%. After adding the 21 projects that document their maintainability status, this empirical validation is based on 133 projects.

RQ2: To answer this second question, we construct a ground truth with projects that are no longer being maintained. First, we build a script to download the README (the main page of GitHub's projects) and search for a list of sentences that are commonly used to declare the unmaintained state of GitHub projects. This list is reused from our previous work (Chapter 2), where we study the motivations for the failure of open source projects. It includes 32 sentences; in Table 4.8, we show a partial list, with 15 sentences.

Table 4.8: Sentences documenting unmaintained projects

no longer under development, *no longer supported or updated*,
deprecation notice, *dead project*, *deprecated*, *unmaintained*,
no longer being actively maintained, *not maintained anymore*,
not under active development, *no longer supported*
is not supported, *is not more supported*, *no longer supported*,
no new features should be expected, *isn't maintained anymore*
no longer be shipping any updates, *don't want to maintain*

We searched (in May, 2018) for these sentences in the README of 5,783 projects, which represent all 6,785 projects selected for this work minus 1,002 projects used in Section 4.2. In 451 READMEs (7.8%), we found the mentioned sentences. Then, we carefully inspected each README to confirm the sentences indeed refer to the project's status, in terms of maintenance. In the case of 112 projects ($\approx$ 25%), we confirmed this fact. Therefore, these projects are considered as a ground truth for this investigation. Usually, the unconfirmed cases refer to the deprecation of specific elements, e.g., methods or classes.

RQ3: To answer this third research question, we rely on projects whose unmaintained status, as predicted by the proposed model, is confirmed by the participants of the survey conducted in RQ1. Then, we compute the number of days between November 30, 2018 (when the machine learning model proposed in this study was built) and

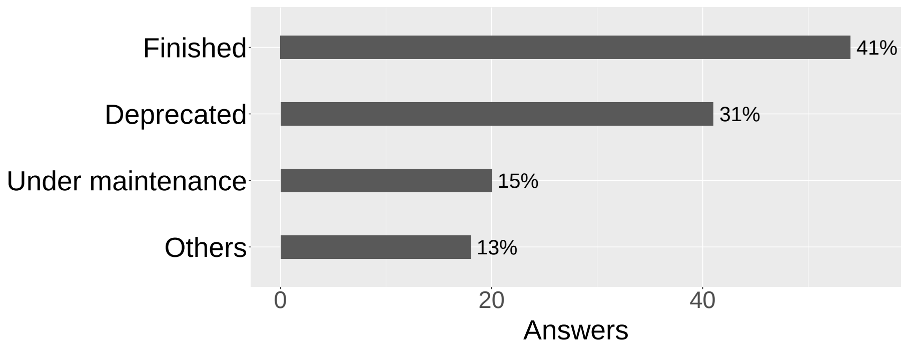


Figure 4.5: Survey answers about projects' status.

the last commit of the mentioned projects. For projects where this interval is less than one year, the proposed model is better than the strategy adopted in previous work [Khondhu et al., 2013; Mens et al., 2014; Izquierdo et al., 2017; Coelho and Valente, 2017], which requires one year of commit inactivity to identify unmaintained projects.

4.3.2 Results

RQ1: Precision according to GitHub developers

Before presenting the precision results, Figure 4.5 shows the survey results, including answers retrieved from the project's documentation, answers received in the pilot and answers received in the final survey. As we can see, the most common status, with 54 answers (41%) refers to *finished* projects, i.e., cases where maintainers see their projects as feature-completed and only plan to resume the maintenance work if relevant bugs are reported.[5] We also received 41 answers (31%) mentioning the projects are deprecated and no further maintenance is planned, including the implementation of new features and bug fixes. Finally, we received 18 answers in the *other* option. In this case, four participants did not describe their answer or provide unclear answers; furthermore, one participant mentioned his project is in a *limbo* state:

The status of [Project-Name] fits into a special category. Some of the tools its based on are either deprecated or not powerful enough for the goal of the project. This is part of the reason what's keeping the project from being "done". I would call this status

[5]In our previous work (Chapter 2), we also identified finished or completed open source projects. However, we argued these projects do not contradict Lehman's Laws [Lehman, 1980] about software evolution because they usually deal with a very stable or controlled environment (whereas Lehman's Laws focus on E-type systems, where E stands for evolutionary).

limbo. *(P24)*

Seven participants answered their projects are *stalled*, as in this answer:

It is under going a rewrite... but has been **stalled** *based on my own priorities (P33)*

To compute precision, we consider as *true positive* answers related to the following status: finished (54 answers), deprecated (41 answers), stalled (7 answers), and limbo (1 answer). The remaining answers are interpreted as *false positives*, including answers mentioning that new features are being implemented (20 answers) and the answers associated to the fourth option (*other* option), but without including a description or with an unclear description (4 answers). By following this criteria, we received 103 true positive answers and 26 false positive ones, resulting in a precision of 80%.[6]

> By validating the proposed model with 127 GitHub developers, we achieve a **precision** of 80%, which is a result very close to the one obtained when building the model (86%).

RQ2: Recall considering deprecated projects

The proposed machine learning model classifies 108 (out of 112) projects of the constructed ground truth as unmaintained, which represents a recall of 96%. This value is significantly greater than the one computed when testing the model in Section 4.2. Probably, this difference is explained by the fact that only projects that are completely unmaintained expose this situation in their READMEs. Therefore, it is easier to detect and identify this condition.

> By validating the proposed model with projects that declare themselves as unmaintained, we achieve a **recall** of 96%.

RQ3: How early can we detect unmaintained projects?

A total of 77 (out of 103) projects classified as true positives by the surveyed developers have commits after November, 2016. Therefore, these projects would not be classified as unmaintained using the strategy followed in the literature, which requires

[6]This computation of precision assumes that finished projects are unmaintained. However, we recognize that the risks of using finished projects might be lower than the ones faced when using deprecated or stalled projects.

one year of commit inactivity. In other words, in November, 2017, the proposed model classified 77 projects as unmaintained, despite the existence of recent commits, with less than one year. Figure 4.6 shows a violin plot with the age of such commits, considering the date of November, 2017. The first, second, and third quartiles are 35, 81, and 195. Interestingly, for two projects the last commit occurred exactly on November, 30, 2018. Despite this fact, the proposed model classified these projects as unmaintained in the same date. If we relied on the standard threshold of one year without commits, these projects would have had to wait one year to receive this classification.

> 75% of the studied projects are classified as unmaintained despite having recent commits, performed in the last year.

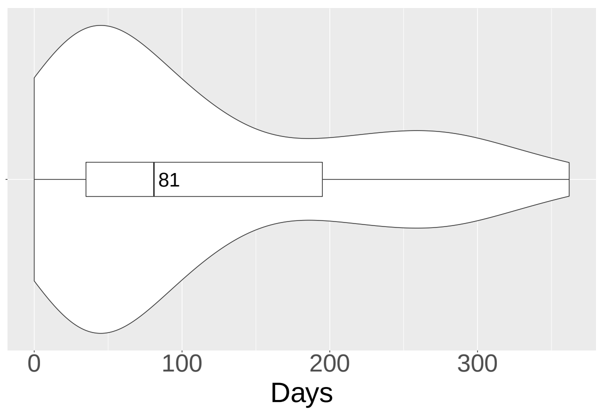


Figure 4.6: Days since last commit for projects classified as unmaintained (considering the date of November, 2017, when the proposed model was computed).

4.4 Characteristics of Unmaintained Projects

In this section, we assess the characteristics of 2,856 projects classified as unmaintained by the proposed model. Although this model is not fully accurate, it showed a precision of 86% in a dataset containing 754 active and 248 unmaintained projects (Section 4.2.2). Moreover, it achieved a precision of 80%, when validated with the developers of 129 GitHub projects (Section 4.3.2). Therefore, this high precision measures—in different contexts—stimulated us to rely on the model to shed light on the characteristics of a large sample of unmaintained GitHub projects.

Particularly, we provide answers to two research questions:

RQ4: How long does a GitHub project survive before become unmaintained? The goal is to provide quantitative data about the lifetime of GitHub projects.

RQ5: How often unmaintained projects follow best OSS maintenance practices? With this question, the goal is to check whether common maintenance practices contribute to extend the lifetime of GitHub projects.

Table 4.9: List of GitHub maintenance practices.

Maintenance Practice	**Description**
License	Presence of project's license (e.g., Apache, GNU, MIT)
Home Page	Availability of a dedicated homepage outside GitHub
Continuous Integration	Use of a continuous integration service (i.e., we check whether the projects use Travis CI, which is the most popular CI service on GitHub [Hilton et al., 2016])
Contributing Guidelines	Presence of contributing guidelines to help external developers make meaningful and useful contributions to the project
Issue Template	Presence of an issue template (to instruct developers to write issues according to the repository's guidelines)
Code of Conduct	Presence of a specific file with a code of conduct, which is a document that establishes expectations for behavior of the project's participants [Tourani et al., 2017]
Pull Request Template	Presence of a pull request template, which is a document to help developers to submit pull requests according to the repositories guidelines
Support File	Presence of a support file to direct contributors to specific support guidelines, such as community forums, FAQ documents, or support channels
First-timer-only issues	Presence of labels recommending issues to newcomers (e.g., *help wanted* or *good first issue*)

4.4.1 Methodology

RQ4: To answer this fourth research question, we apply a survival analysis algorithm on 2,856 projects classified as unmaintained by our model. Survival analysis is a well-known technique, which is used for example on medical sciences to compute the probability of patients to stay alive for a certain number of days [Cox, 2018]. Moreover, survival analysis was successfully used in several software engineering studies [Maldonado et al., 2017; Valiev et al., 2018; Samoladas et al., 2010; Lin et al., 2017]. In this work, we use survival analysis considering the lifetime of GitHub projects. In other words, we analyze the survival probability of a project over time. To determine the lifetime of a GitHub project, we compute the time difference between the first and last commit dates. Then, we use the Kaplan and Meier [1958] non-parametric approximation to compute the survival curve, witch is the most widely used curve for estimating survival probabilities.

RQ5: To answer this last research question, we investigate whether projects classified as unmaintained follow (or not) a set of best open source maintenance practices, which are recommended by GitHub.[7] The rationale of this study is to compare the adoption of these practices between active and unmaintained projects. For each project, we collect the practices described in Table 4.9.[8]

4.4.2 Results

RQ4: How long does a GitHub project survive before become unmaintained?

Figure 4.7 shows the survival plot of 2,856 unmaintained projects, as classified by our model. As we only studied projects with at least 24 months, the survival curve is constant during this initial time frame. By inspecting Figure 4.7, we observe that the likelihood of an unmaintained project surviving for more than 50 months is close to 50%. However, after 84 months (seven years) it declines to less than 10%. In other words, 50% of the unmaintained projects considered in this study moved to this state after 50 months ($\approx$ 4 years) and only 10% remained active for more than 7 years.

We also generate specific survival plots for three projects features: account type, programming language, and application domain. Figure 4.8a shows the survival plots for projects owned by organizations and individual users. As we can see, projects

[7] *https://opensource.guide*

[8] Seven of these maintenance practices are explicitly recommended at: *https://help.github.com/articles/helping-people-contribute-to-your-project*

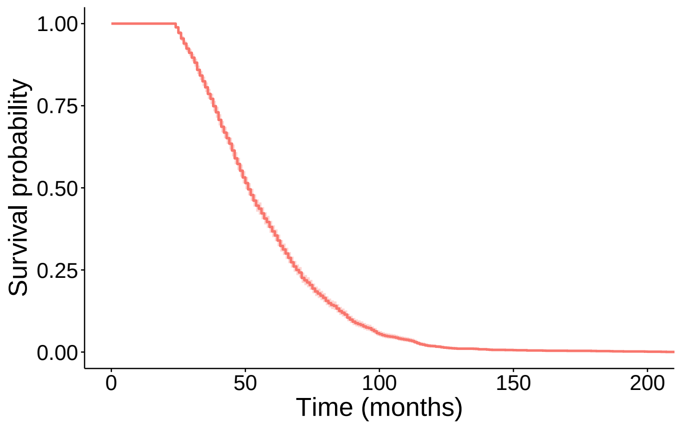
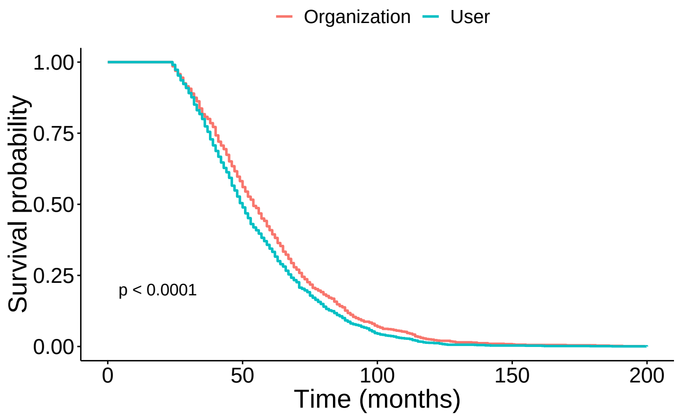


Figure 4.7: Survival probability of GitHub projects classified as unmaintained by classification model.

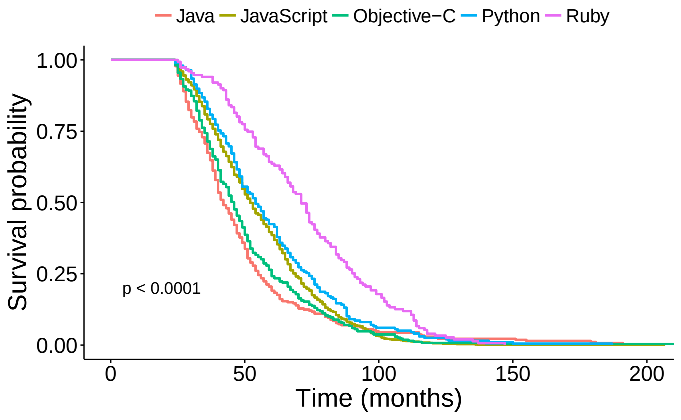
maintained by organizations have slightly greater survival probabilities than projects whose owner is an individual GitHub user. For example, after four years, the survival probabilities are 53% and 60%, for user and organization-owned projects, respectively. The distributions are statistically different, according to the one-tailed variant of the Mann-Whitney U test (p-value ≤ 0.05). However, we compute Cliff's delta (or d) to show the effect size of this difference and we found a negligible effect size.

Figure 4.8b shows the survival probabilities for the top-5 programming languages with more projects in our set of unmaintained projects. By applying Kruskal-Wallis test to compare multiple samples, we found that these distributions are different (p-value ≤ 0.05). The greatest difference happens between projects implemented in Java and Ruby. Particularly, projects implemented in Ruby tend to have higher survival probabilities than in other languages. By contrast, Java projects show lower survival probabilities. For example, after four years, the survival probabilities are 38% and 79%, for Java and Ruby projects, respectively.

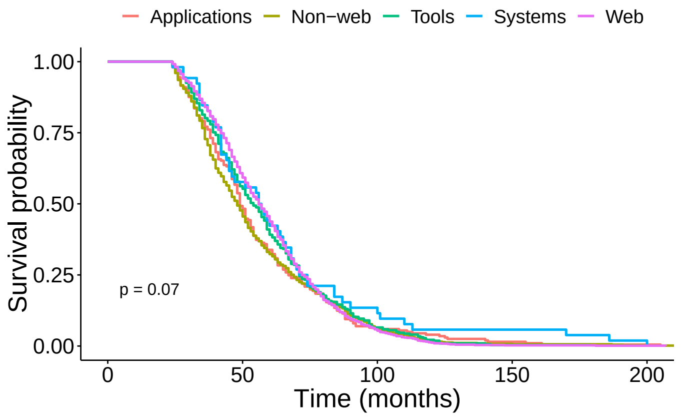
Finally, Figure 4.8c shows survival plots by application domain. To this propose, we manually classified the projects in five major application domains: *Application software*, *Non-web libraries and frameworks*, *Software tools*, *Systems software*, and *Web libraries and frameworks*. We reused these domains from a similar classification performed by Borges et al. [Borges et al., 2016b; Borges and Valente, 2018]. The same domains are used in others studies [Coelho and Valente, 2017; Borges et al., 2016a]. By applying Kruskal-Wallis, the highest statistical difference occurs between *Non-web* and *Web* applications. For example, after four years, the survival probabilities are 50% and 63%, for *Non-web* and *Web* projects, respectively. Therefore, *Web* libraries tend to survive for more time than *Non-web* ones. Finally, we can also observe that *Systems*



(a) Account type



(b) Programming Language



(c) Application Domain

Figure 4.8: Survival analysis by (a) account type, (b) programming language, and (c) application domain.

software have the highest survival probabilities. For example, after 8 years, the survival probabilities is twice than in other domains.

After characterizing the unmaintained projects, we found that there is a negligible difference on the survival probabilities of projects owned by individual and organizational accounts. Moreover, Ruby projects show higher probabilities of survival. Finally, *System software* is the application domain with the highest survival probability.

RQ5: How often unmaintained projects follow best OSS maintenance practices?

In this last RQ, we compare the adoption of a set of well-known maintenance practices between active and unmaintained projects. First, we analyze the statistical significance of the difference in the usage of each practice between these groups of projects, by applying the Mann-Whitney test at p-value = 0.05. To show the effect size, we use Cliff's delta. Following the guidelines of previous work [Grissom and Kim, 2005; Tian et al., 2015b; Linares-Vásquez et al., 2013], we interpret the effect size as small for $0.147 < d < 0.33$, medium for $0.33 \leq d < 0.474$, and large for $d \geq 0.474$.

Table 4.10 shows the percentage of projects following each practice, considering *Active* vs *Unmaintained* projects. We found a *negligible* effect size for all practices, with the exception of continuous integration, contributing guidelines, and first-timers-only issues, when the effect size is *small*.

The results presented reveal a slight difference on the adoption of best open source maintenance practices between active and unmaintained projects. Therefore, we conclude that the main factors that are responsible for moving projects to an unmaintained status are external ones. Indeed, in our previous work (Chapter 2), we surveyed the maintainers of 104 failed open source projects to reveal the reasons for their failures. Our findings show that the top-5 most common reasons are: project was usurped by competitor (26%), project became functionally obsolete (19%), lack of time of the main contributor (17%), lack of interest of the main contributor (17%), and project based on outdated technologies (13%). In other words, the sustainability of GitHub projects is not associated with the adoption of best open source maintenance practices, such as the ones investigated in this RQ.

The maintenance practices with the most relevant effect are the use of continuous integration services, followed by the adoption of contributing guidelines, and the presence of labels recommending issues to newcomers, but with small effect size. For the remaining practices, the difference is negligible or does not exist in statistical terms.

Table 4.10: Percentage of projects following recommended practices when maintaining GitHub repositories. The effect size reflects the extent of the difference between the unmaintained and active projects.

Maintenance Practice	**Active**	**Unmaintained**	d	**Effect**
License	0.83	0.73	0.10	negligible
Home Page	0.65	0.51	0.14	negligible
Continuous Integration	0.71	0.45	0.26	small
Contributing Guidelines	0.44	0.20	0.24	small
Issue Template	0.08	0.02	0.06	negligible
Code of Conduct	0.13	0.03	0.10	negligible
Pull Request Template	0.03	0.00	0.03	negligible
Support File	0.01	0.01	0.00	negligible
First-timers-only issues	0.53	0.31	0.22	small

4.5 Level of Maintenance Activity

In this section, we define a metric to express the *level of maintenance activity* of GitHub projects, i.e., a metric that reveals how often a project is being maintained. The goal is to alert users about projects that although classified as active by the proposed model are indeed close to an unmaintained status.

4.5.1 Definition

The proposed machine learning model—generated by Random Forest—consists of multiple decision trees built randomly. Each tree in the ensemble determines a prediction to a target instance and the most voted class is considered as the final output. One possible prediction type of the Random Forest is the matrix of class probabilities. This matrix represents the proportion of the trees' votes. For example, projects predicted as *active* have probability p ranging from 0.5 to 1.0. If $p = 0.5$, the project is very similar to an unmaintained project; by contrast, $p = 1.0$ means the project is actively maintained. Using these probabilities, we define the *level of maintenance activity (LMA)* of a GitHub project as follows:

$$LMA = 2 * (p - 0.5) * 100$$

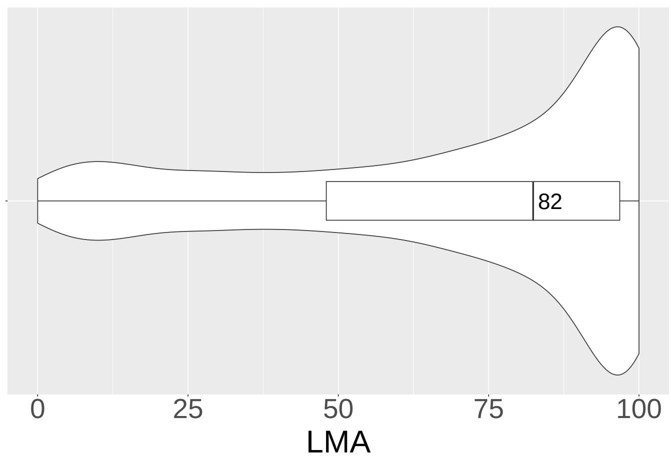


Figure 4.9: Level of maintenance activity (LMA).

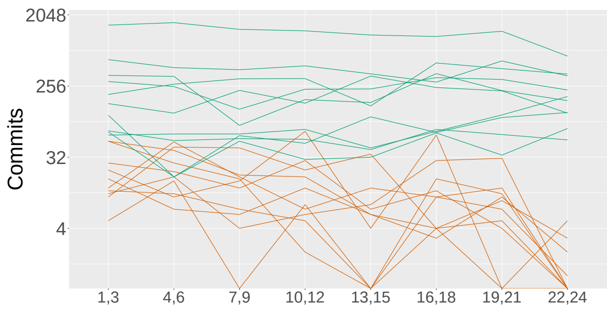
This equation simply converts the probabilities p computed by Random Forest to a range from 0 to 100; LMA equals to 0 means the project is very close to an unmaintained classification (since $p = 0.5$); and LMA equals to 100 denotes a project that is actively maintained (since $p = 1.0$).

4.5.2 Results

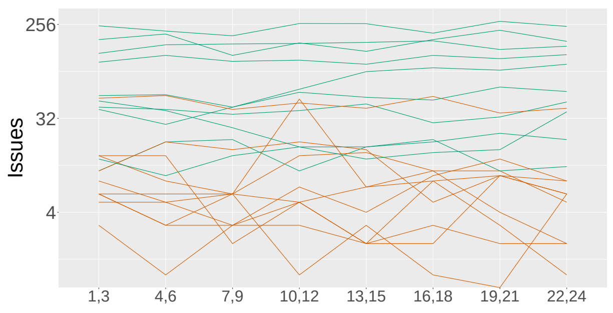
Figure 4.9 shows the LMA values for each project predicted as *active* (2,927 projects, after excluding the projects used to train and test the proposed model, in Section 4.2). The first, second, and third quartiles are 48, 82, and 97, respectively. In other words, most studied projects are under constant maintenance (median 82). Indeed, 171 projects (5.8%) have a maximal LMA, equal to 100. This list includes well-known and popular projects such as TWBS/BOOTSTRAP, METEOR/METEOR, RAILS/RAILS, WEBPACK/WEBPACK, and ELASTIC/ELASTICSEARCH.

Figure 4.10 compares a random sample of 10 projects with LMA equals to 100 (actively maintained, therefore) with ten projects with the lowest LMA ($0 \leq \text{LMA} \leq 0.4$). These projects are compared using number of commits (Figure 4.10a), number of issues (Figure 4.10b), number of pull requests (Figure 4.10c), and number of forks (Figure 4.10d), in the last 24 months. Each line represents the project's metric values. The figures reveal major differences among the projects, regarding these metrics. Usually, the projects with high LMA present high values for the four considered metrics (commits, issues, pull requests, and forks), when compared with projects with low LMA. In other words, the figures suggest that LMA plays an aggregator role of maintenance activity over time.

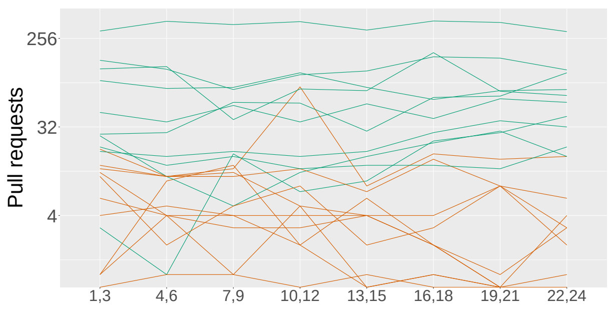
Figure 4.11 shows scatter plots correlating LMA and number of stars, contribu-



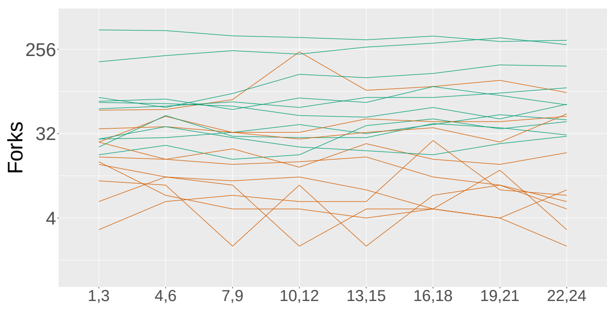
(a) Commits



(b) Issues



(c) Pull requests



(d) Forks

Figure 4.10: Comparing number of commits, issues, pull request, and forks over time of ten projects with maximal LMA (green lines) and ten projects with the lowest LMA in our dataset (red lines). Metrics are collected in intervals of 3 months (x-axis).

tors, core contributors, and size (in LOC) of projects classified as *active*. To identify core contributors, we use a common heuristic described in the literature: core contributors are the ones responsible together for at least 80% of the commits in a project [Koch and Schneider, 2002; Mockus et al., 2002; Robles et al., 2009]. To measure the size of the projects, in lines of code, we used the tool ALDANIAL/CLOC[9], considering only the programming languages in the TIOBE list.[10] We also compute Spearman's rank correlation test for each figure. The correlation of stars and core contributors is very weak ($\rho = 0.10$ and $\rho = 0.15$, respectively); with size, the correlation is weak ($\rho = 0.38$); and with contributors, it is moderate ($\rho = 0.44$); all p-values are less than 0.01. Therefore, it is common to have highly popular projects, by number of stars, presenting both low and high LMA values. For example, one project has 50,034 stars, but LMA = 8. A similar effect happens with size. For example, one project has ≈2 MLOC, but LMA = 10.8. The highest correlation is observed with contributors, i.e., projects with more contributors tend to have higher levels of maintenance activity.

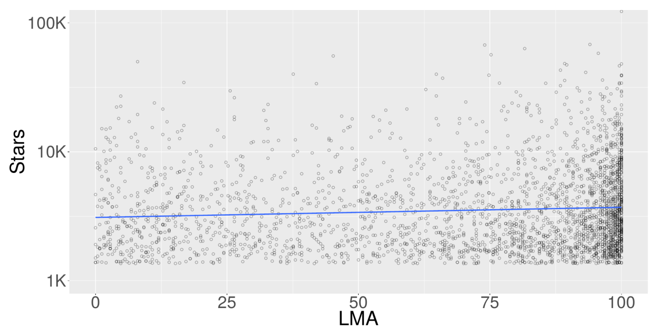
4.5.3 Validation with False Negative Projects

In Section 4.3, we found four projects that although declared by their developers as *unmaintained* are predicted by the proposed machine learning model as *active*. Therefore, these projects are considered false negatives, when computing recall. Two of such projects have a very low LMA: NICKLOCKWOOD/IRATE (LMA = 2) and GORANGAJIC/REACT-ICONS (LMA = 12). Therefore, although predicted as *active*, these projects are similar to projects classified as *unmaintained*, as suggested by their low LMA. A second project has an intermediate LMA value: SPOTIFY/HUBFRAMEWORK (LMA = 39.2). Finally, one project HOMEBREW/HOMEBREW-PHP has a high LMA value (LMA = 99.2). However, this project was migrated to another repository, when facing continuous maintenance. In other words, in this case, the GitHub repository was deprecated, but not the project; therefore, HOMEBREW/HOMEBREW-PHP is a false, false negative (or a true negative).

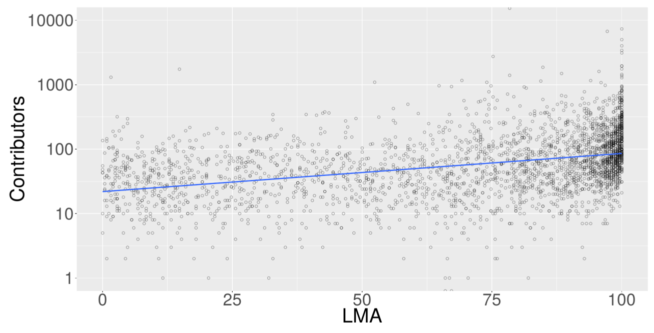
4.5.4 Historical Analysis

In this section, we analyze the historical evolution of 2,927 active projects, as classified by our model in November, 2017. To build a trend line, we compute new LMA values for these projects in November, 2018, i.e., after one year. Our goal is to study how often projects become unmaintained and whether the LMA values change significantly over time. Particularly, we compute LMA values in intervals of 3 months during the period of analysis, i.e., February 2018, May 2018, August 2018, and November 2018. We also

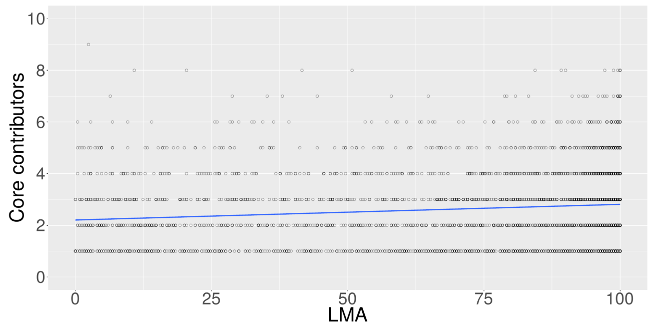
[9] *https://github.com/AlDanial/cloc*
[10] *https://www.tiobe.com/tiobe-index*



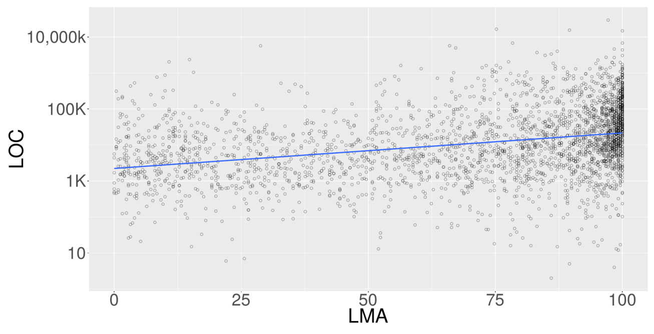
(a) LMA vs Stars ($\rho = 0.10$)



(b) LMA vs Contributors ($\rho = 0.44$)



(c) LMA vs Core contributors ($\rho = 0.15$)



(d) LMA vs LOC ($\rho = 0.38$)

Figure 4.11: Correlating LMA with (a) stars, (b) contributors, (c) core contributors, and (d) size. Spearman's ρ is also presented.

evaluate LMA values under two perspectives: programming language and application domain. Figure 4.12 shows the total number of projects classified as *unmaintained* in each time interval. As we can see, the number of unmaintained projects is increasing over time, moving from 117 projects (4%) in February 2018 to 468 projects (16%) in November 2018.

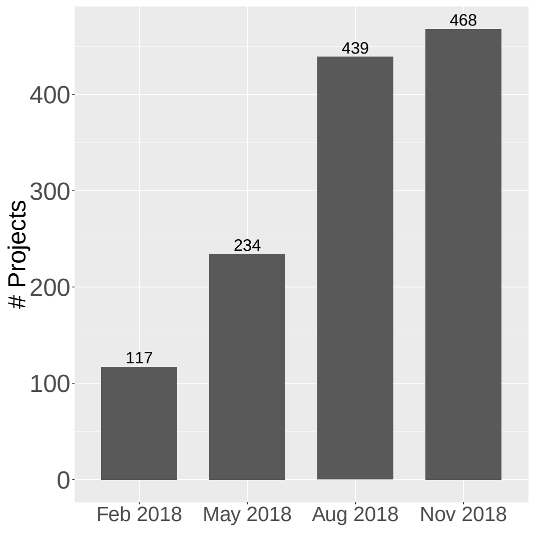


Figure 4.12: Number of new unmaintained projects over time.

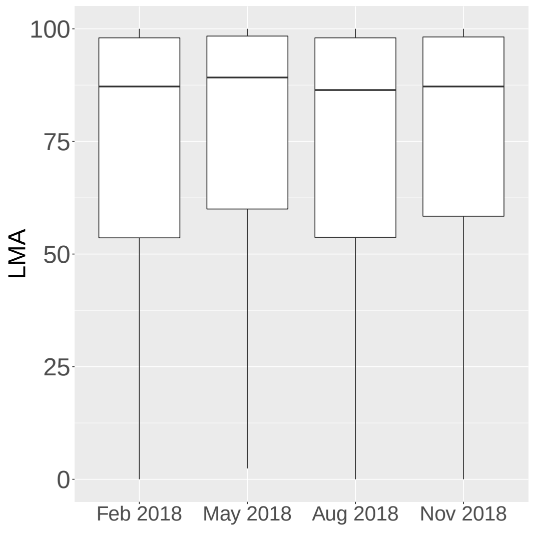


Figure 4.13: LMA values over time.

Figure 4.13 shows the distribution of the LMA values on each period. The median values are 87.2, 89.2, 86.4, and 87.2, respectively. By applying Kruskal-Wallis to compare multiple samples, we found that these distributions are statistically different, but the difference is *negligible* by Cliff's delta. Although there is an increasing number

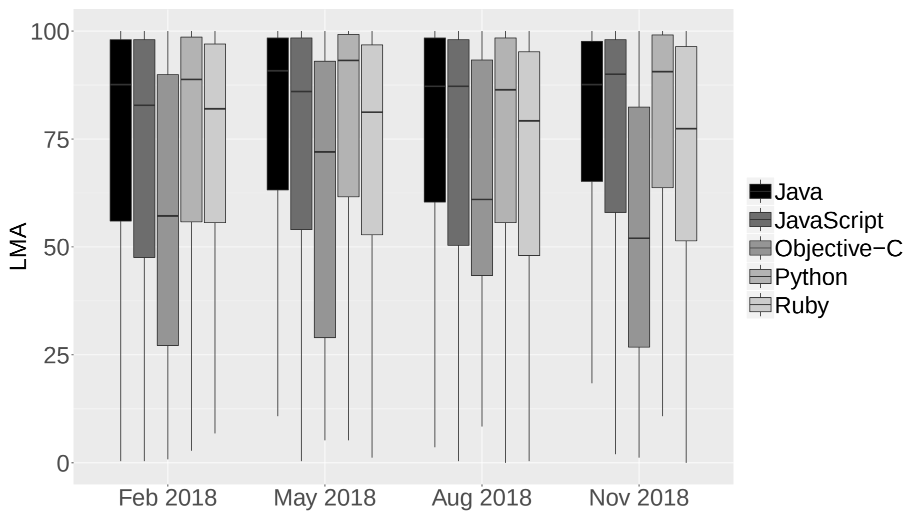


Figure 4.14: Historical LMA values by programming language.

of unmaintained projects, there is also a significant number of projects with maximal LMA values. In each interval, we found 215, 230, 208, and 197 projects with maximal LMA values, respectively. Some popular projects—such as RAILS/RAILS, MATPLOTLIB/MATPLOTLIB, and NUMPY/NUMPY—have maximal LMA in all considered time frames.

Figure 4.14 shows the LMA values by programming language over the studied 3-month time intervals. We consider only the top-5 languages by number of projects, which are JavaScript (700), Python (360), Java (259), Ruby (216), and Objective-C (179). For all languages, the LMA values remain stable throughout the studied intervals, with the exception of Objective-C. These projects increased their LMA values from 57.2 in February 2018 to 72.0 in May 2018, but then decreased to 61.0 in August 2018 and 52.0 in November 2018.

Finally, Figure 4.15 shows the historical LMA values by application domain. We use the same domains from the survival analysis (Section 4.4). We found no significant differences between the distributions in the analyzed time frames. However, *Software Tools* have higher LMA values in all considered time intervals, which median measures 92.8, 94.4, 92.8, and 91.2, respectively.

From 2,927 active projects in November 2017, 468 projects (16%) moved to an unmaintained state in the time interval of one year. We also found that Objective-C projects have lower LMA values than projects implemented in other programming languages. Finally, *Software Tools* have the highest LMA values over time.

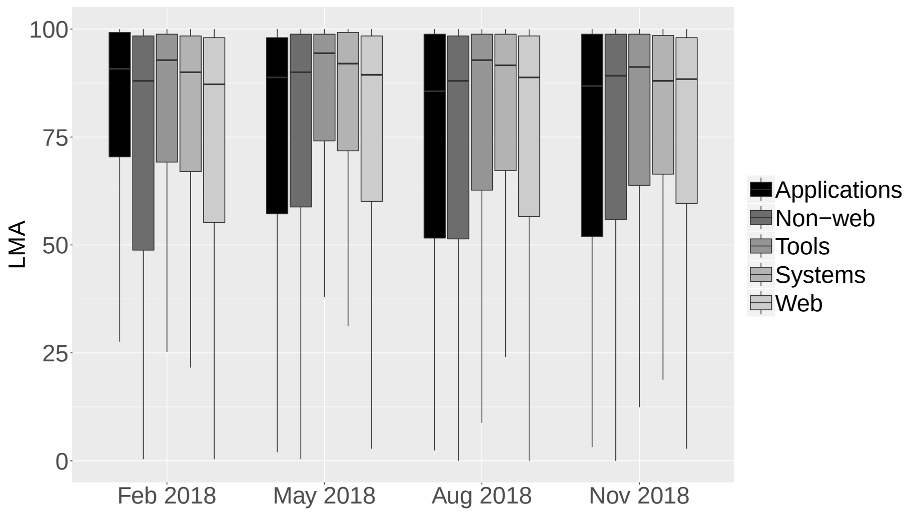
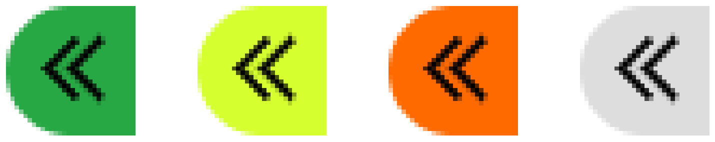


Figure 4.15: Historical LMA values by application domain.

4.5.5 Chrome Extension

We implemented a Chrome extension called *isMaintained* to indicate whether a GitHub project is actively maintained or not.[11] This extension is publicly available at the Chrome Store.[12] It only installs a small icon on the right side of a repository's page. This icon's color is used to inform the maintenance level of a project. The projects classified as *unmaintained* have a red icon. On the other hand, the level of maintenance activity for active projects can be high, fair, or borderline. The projects with LMA values in the fourth quartile of LMA values are labeled as high (green icon); projects in the second and third quartiles are labeled as fair (yellow icon); and projects in the first quartile are labeled as borderline (orange icon). Finally, the remaining repositories in our dataset (e.g., books, tutorials, awesome-lists, etc.) receive a gray icon. Table 4.11 shows the levels of maintenance activity and their respectively color used to classify the projects. Figure 4.16 shows an example for each icon used to represent the project's LMA.



(a) Green (b) Yellow (c) Orange (d) Gray

Figure 4.16: Example of the icons (a) green, (b) yellow, (c) orange, and (d) gray.

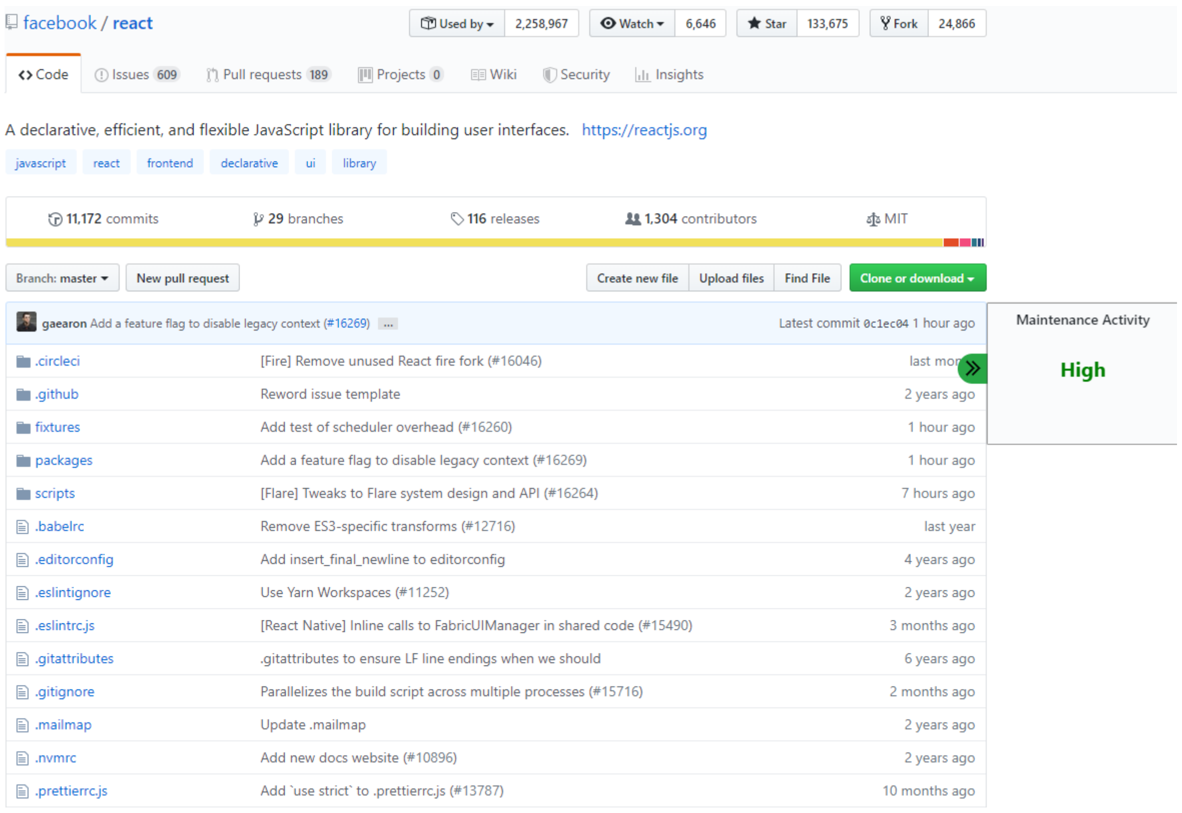
[11]This Chrome extension was implemented by Luciano Otoni Milen—he is a undergraduate student at UFMG—under my orientation.

[12]*https://chrome.google.com/webstore/search/ismaintained*

Table 4.11: Levels of maintenance activity as considered by the implemented Chrome extension.

Level of activity	**Color**	**LMA Quartile**
High	green	4th
Fair	yellow	2nd and 3rd
Borderline	orange	1st
Unmaintained	red	-
Not analysed	grey	-

Figure 4.17 shows an example of a GitHub page using the proposed Chrome extension. In this example, we show the level of activity for FACEBOOK/REACT with a green icon (high maintenance activity).


facebook / react — Used by 2,258,967 · Watch 6,646 · Star 133,675 · Fork 24,866

Code · Issues 609 · Pull requests 189 · Projects 0 · Wiki · Security · Insights

A declarative, efficient, and flexible JavaScript library for building user interfaces. https://reactjs.org

javascript · react · frontend · declarative · ui · library

11,172 commits · 29 branches · 116 releases · 1,304 contributors · MIT

Branch: master · New pull request · Create new file · Upload files · Find File · Clone or download

gaearon Add a feature flag to disable legacy context (#16269) … Latest commit 0c1ec04 1 hour ago

```
.circleci          [Fire] Remove unused React fire fork (#16046)                     last mon
.github            Reword issue template                                             2 years ago
fixtures           Add test of scheduler overhead (#16260)                           1 hour ago
packages           Add a feature flag to disable legacy context (#16269)             1 hour ago
scripts            [Flare] Tweaks to Flare system design and API (#16264)            7 hours ago
.babelrc           Remove ES3-specific transforms (#12716)                           last year
.editorconfig      Add insert_final_newline to editorconfig                          4 years ago
.eslintignore      Use Yarn Workspaces (#11252)                                      2 years ago
.eslintrc.js       [React Native] Inline calls to FabricUIManager in shared code (#15490)   3 months ago
.gitattributes     .gitattributes to ensure LF line endings when we should           6 years ago
.gitignore         Parallelizes the build script across multiple processes (#15716)  2 months ago
.mailmap           Update .mailmap                                                   2 years ago
.nvmrc             Add new docs website (#10896)                                     2 years ago
.prettierrc.js     Add `use strict` to .prettierrc.js (#13787)                       10 months ago
```

Maintenance Activity: High

Figure 4.17: Example of a GitHub page using the LMA plugin (on the right side).

How it works: isMaintained relies on the proposed machine learning model (Chapter 4) to show the maintenance status of a GitHub project. First, we use a Java script to create a dataset with the top-10,000 most popular projects on GitHub. Second, we remove those repositories that have less than two years between the first and the last commit. Third, we define two subsets of systems: *active* and *unmaintained*. The active group is composed by those projects that have at least one release in the last month. By contrast, the unmaintained group is composed by those projects that were explicitly declared by their principal developers as unmaintained in our previous work (Chapter 2) and *archived* projects. Then, we calculate the temporal series for each feature considered in our previous study (Chapter 4) and remove correlated features. Fourth, we use the Random Forest algorithm [Breiman, 2001] as implemented by a R script to train the model and apply it in all projects from our dataset that were not used in the model's construction. Finally, the status for each project is stored in a Firebase database.[13] Firebase is built on Google infrastructure and have functionalities such as analytics, databases, messaging and crash reporting. Thus, when a user visits the main page of GitHub's projects their status is searched on the Firebase dataset.

4.6 Threats to Validity

The threats to validity of this work are described as follows:

External Validity: Our work examines open source projects on GitHub. We recognize that there are popular projects in other platforms (e.g., Bitbucket, SourceForge, and GitLab) or projects that have their own version control installations. Thus, our findings may not generalize to other open source or commercial systems. A second threat relates to the features we have considered. By adding other features, we may improve the prediction of unmaintained projects; however, given our high prediction performance, we feel confident that our features are effective. Also, some of the features we use may not be available in other projects, however, most of our features are available in most code control repositories/ecosystems. In the future, we intend to investigate additional projects and consider more features.

Internal Validity: The first threat relates to the selection of the survey participants. We surveyed the project owner, in the case of repositories owned by individuals, or the developer with the highest number of commits, in the case of repositories owned by organizations. We believe the developers who replied to our survey are the most relevant given their level of activity in the project. It is also possible that most missing answers are from developers of unmaintained projects. As with any human activity, the derived themes may be subject to bias and different researchers might reach different observations. However, to mitigate this threat, a first choice of themes was conducted in parallel by others researchers. Also, they attended several meetings

[13] *https://firebase.google.com*

during a whole week to improve the initial selected themes. A third threat relates to the parameters used to perform our experiment. We set the number of trees to 100 to train our classifier. To attenuate the bias of our results, we run 5-fold cross-validation and use the average performance for 100 rounds. A forth threat is related to how the accuracy of our machine learning approach was evaluated. We relied on developer replies about their projects to evaluate the performance of our machine learning classifier. In some cases, the developer replies (or developers who did not reply) may impact our results. That said, our survey had a response rate of 37.1%, which is very high for a software engineering study, giving us confidence in the reported performance results.

Construct Validity: A first threat relates to the definition of active projects. We consider as active projects those with at least one release in the last month (Section 4.2). We acknowledge a threat in the definition of the time frame. To mitigate this threat, we inspected each selected project to look for deprecated projects (21 projects declare they are no longer being maintained) and we conduct a survey with 112 developers to confirm our findings. A second threat is related to the projects we studied. Our dataset is composed of the most starred projects (and additional filtering). Although the starred projects may not be representative of all open source projects, we did carefully select such projects to ensure that our study is conducted on real (and not toy) projects. Finally, a third threat is related to the number of forks of the projects used in the machine learning model. We only considered the creation date of the fork. Therefore, abandoned forks may have been considered.

4.7 Related Work

Machine Learning. Recently, the application of machine learning in software engineering contexts has gained much attention. Several researchers have used machine learning to accurately predict defects (e.g., Peters et al. [2013]), improve issue integration (e.g., Alencar da Costa et al. [2014]), enhance software maintenance (e.g., Gousios et al. [2014]), and examine developer turnover (e.g., Bao et al. [2017]). For example, Gousios et al. [2014] investigate the use of machine learning to predict whether a pull request will be merged. They extract 12 features organized into three dimensions: pull request, project, and developer. They conduct their study using six algorithms (Logistic Regression, Naive Bayes, Decision Trees, AdaBoost with Decision Trees, and Random Forest). Bao et al. [2017] build a model to predict developer turnover, i.e., whether a developer will leave the company after a period of time. They collect several features based on developers monthly report from two companies. We evaluate the performance of five classifiers (KNN, Naive Bayes, SVM, Decision Trees, and Random Forest). In both studies, Random Forest outperforms the results of other algorithms. In another study, Martin et al. [2016] train a Bayesian model to support app developers on causal impact analysis of releases. They mine time-series data about Google Play app over a period of 12 months and survey developers of significant releases to check their results. Tian et al. [2015c] use Random Forest to predict whether an app will be high-rated. They extract 28 factors from eight dimensions, such as app size and library quality. Their findings show that external factors (e.g., number of promotional images) are the most influential factors. Our study also uses machine learning techniques, however, our main goal is to detect projects that are not going to be actively maintained. Moreover, our study extracts project, contributor and owner features that we input to the machine learning models.

Open source projects maintainability. In previous work (Chapter 2), we survey maintainers of 104 failed open source projects to understand the rationale for such failures. Their findings revealed that projects fail due to reasons associated with project properties (e.g., low maintainability), project team (e.g., lack of time of the main contributor), and to environment reasons (e.g., project was usurped by a competitor). Later, we report results of a survey with 52 developers who recently became core contributors on popular GitHub projects (Chapter 3). Our results show the developer's motivations to assume an important role in FLOSS projects (e.g., to improve the projects because they use them), the project characteristics (e.g., a friendly community), and the obstacles they faced (e.g., lack of time of the project leaders).

Also related is the work by Yamashita et al. [2014], which adapts two population migration metrics in the context of open source projects. Their analysis enables the detection of projects that may become obsolete. Khondhu et al. [2013] report that more than 10,000 projects are inactive on SourceForge. They use the maintainability index (MI) [Oman and Hagemeister, 1992] to compare the maintainability between inactive projects and projects with different statuses (active and dormant). Their results reveal that the majority of inactive systems are abandoned with a similar or increased maintainability, when compared to their initial status. Nonetheless, there are critical concerns on using MI as a predictor of maintainability [Bijlsma et al., 2012]. Eghbal [2016] reports risks and challenges to maintain modern open source projects. She argues that open source plays a key role in the digital infrastructure of our society today. Opposed to physical infrastructure (e.g., bridges and roads), open source projects still lack a reliable and sustainable source of funding.

Liu et al. [2018] present a learning-to-rank model to recommend open source projects for developers. Rastogi et al. [2018] investigate 70,000+ pull requests from 17 countries to model the relationship between the geographical location of developers and pull request acceptance decision. Steinmacher et al. [2018] conducted surveys with quasi-contributors to understand their perceptions for pull-request non-acceptance. Their results show that non-acceptance discourage developers to submit new pull-requests. Barcomb et al. [2019] show five factors that affect retention of episodic volunteers in FLOSS communities. Other recent research on open source has focused on the organization of successful open source projects [Mockus et al., 2002] and on how to attract and retain contributors [Zhou and Mockus, 2015; Steinmacher et al., 2016; Lee et al., 2017; Pinto et al., 2016; Canfora et al., 2012].

Survival analysis. Survival analysis was first used in the medical domain and then applied to other domains including software engineering. For example, Maldonado et al. [2017] use survival analysis to determine how long self-admitted technical debt lives in a project before it is actually removed. Lin et al. [2017] applied survival analysis on five open source projects to understand the impact of several factors on developers leaving a project. Valiev et al. [2018] use survival analysis on a large set of PyPI projects hosted on GitHub. Samoladas et al. [2010] proposed a framework for assessing the survival probability of a FLOSS project and evaluate the benefits of adding more committers in a project. Businge et al. [2012] investigate the survival of 467 third-party Eclipse plug-ins. Different from this works, we use survival analysis to reveal the survivability probability of a large scale of open source projects under different perspectives (e.g., organizational or individual account, programming language, and application domain).

4.8 Conclusion

In this chapter, we proposed a machine learning model to identify unmaintained GitHub projects and to measure the level of maintenance activity (LMA) of active GitHub projects. By our definition, the *unmaintained* status includes three types of projects: finished projects, deprecated projects, and stalled projects. We validated the proposed model with the principal developers of 127 projects, achieving a precision of 80% (RQ1). Then, we used the model with 112 deprecated projects—as explicitly mentioned in their GitHub page. In this case, we achieved a recall of 96% (RQ2). We also showed that the proposed model can identify unmaintained projects early, without having to wait for one year of inactivity, as commonly proposed in the literature (RQ3). We assessed the survival probability of unmaintained projects under three perspectives: organizational or individual account, programming language, and application domain (RQ4). We found a negligible difference on the survival probabilities of projects owned by individual and organizational accounts. Moreover, Ruby projects have higher probabilities of survival. Regarding the analysis by application domain, we found that *System Software* is the domain with the highest survival probability. Finally, we investigate whether unmaintained projects follow (or not) a set of best open source maintenance practices (RQ5). Our results show that the practices with the highest effect are continuous integration, followed by the adoption of contributing guidelines, and the presence of labels to recommend issues to newcomers. However, the effect size is *small*, which suggests that external factors are the ones responsible to turn projects unmaintained.

Finally, we defined a metric, called Level of Maintenance Activity (LMA), to assess the risks of projects become unmaintained. We provided evidence on the applicability of this metric by investigating its usage in 2,927 projects classified as active in our dataset. We evaluate the LMA of these projects in the time frame of one year under two different perspectives: programming language and application domain. We found that 16% become unmaintained over this time. We also reported that Objective-C projects have lower LMA values than projects implemented in other languages. Software tools have the highest LMA values over time. Finally, we implemented a public Chrome extension called *isMaintained* to show the level of maintenance activity of a GitHub project. This extension is publicly available at: *https://chrome.google.com/webstore/search/ismaintained*.

The dataset used in this chapter is available at: *https://zenodo.org/record/1313637*.

Chapter 5

Conclusion

In this chapter, we present our closing points and arguments. In Section 5.1, we revisit each question introduced in Chapter 1. In Section 5.2, we review our main contributions. In Section 5.3, we discuss our main results. Finally, in Section 5.4, we outline possible ideas for future work.

5.1 Summary

This thesis explored, through a set of quantitative and qualitative studies, a list with the main reasons for the discontinuation of modern open source software projects. Through the thesis we proposed a machine learning model, conducted empirical investigations, and surveyed systems developers. This study was guided by the following questions:

Q1. Why do modern open source software projects become unmaintained?

To address this question, in Chapter 2, we conducted a survey with the maintainers of 104 GitHub projects that are no longer under maintenance. We revealed a set of nine reasons that motivated them to stop the maintenance of their projects. We also showed that there is a relevant difference between unmaintained and active projects, in terms of following best open source maintenance practices. Furthermore, we showed that unmaintained projects have a non-negligible number of opened issues and pull requests. Finally, we described three strategies attempted by maintainers to overcome the unmaintained state of their projects. The main contributions of this study are: *(i)* a list with nine reasons that led modern open source software projects to become unmaintained; *(ii)* a comparison of the usage of a set of best maintenance practices by unmaintained projects and also by the most and least popular systems in a sample of 5,000 GitHub projects; and *(iii)* we documented three strategies attempted by the maintainers of open source projects to overcome (without success) the unmaintained state of their projects.

Q2. What are the key motivations to contribute to open source projects?

To address this question, in Chapter 3, we reported reasons that led recent core developers to contribute to open source projects. We also revealed the most common project characteristics and practices that motivated them to contribute and the barriers they faced. In summary, our contributions are: *(i)* a list with motivations that led recent core developers of open source projects to engage in such projects; *(ii)* a list with technical and non-technical project characteristics that helped recent core developers to contribute; and *(iii)* a list with the main barriers that recent core developers faced when contributing to open source projects.

Q3. How to identify unmaintained GitHub projects? How to measure the level of maintenance activity of open source projects?

To address these questions, in Chapter 4, we proposed a machine learning model to measure the level of maintenance activity of GitHub projects, based on a set of features about project activity. We validated the model calculating its precision and recall. By means of a survey with developers of 129 GitHub projects, the model achieving a precision of 80%. By validating the proposed model with projects that declare themselves as unmaintained, we achieved a recall of 96%. We also demonstrated that the proposed model can identify unmaintained projects early, without having to wait for one year of inactivity, as commonly proposed in the literature. We defined a metric, called Level of Maintenance Activity (LMA), to measure the level of maintenance activity of GitHub projects. We provided evidence on the applicability of this metric by investigating its usage in 2,927 active projects in the time frame of one year. Finally, we implemented a public Chrome extension called *isMaintained* to show the level of maintenance activity of a GitHub project. In summary, our contributions are: *(i)* a machine learning approach to identify unmaintained projects in GitHub; *(ii)* a metric to reveal the maintenance activity level of GitHub projects; and *(iii)* a public Chrome extension to inform the maintenance level of a particular GitHub project.

5.2 Contributions

We summarize our contributions as follows:

- In Chapter 2, we provide a list of nine reasons of developer's motivations to stop the maintenance of their open source projects. The top-5 most common reasons are: project was usurped by competitor (27 projects), project became functionally obsolete (20 projects), lack of time of the main contributor (18 projects), lack of interest of the main contributor (18 projects), and project based on outdated technologies (14 projects).

- In Chapter 2, we also reinforce the importance of a set of best open source maintenance practices, by comparing their usage by the unmaintained projects and also by the most and least popular systems in a sample of 5,000 GitHub projects. The difference is more important regarding the availability of contribution guidelines and the use of continuous integration.

- In Chapter 2, we also document three strategies attempted by the maintainers of open source projects to overcome (without success) the failure of their projects. We found that 14 projects attracted new core developers (third strategy), two were transferred to new maintainers (second strategy), and other two projects were moved to an organization account (first strategy).

- In Chapter 3, we provide a list of motivations that led recent core developers to contribute to open source projects. We found that 60% of such developers decided to contribute because they use the projects.

- In Chapter 3, we also reveal a list of projects characteristics and practices that helped recent core developers to contribute to open source projects. We found they are most attracted by non-technical characteristics, especially the ones related to a friendly and available open source community.

- In Chapter 3, we also provide a list of the main barriers faced by recent core developers to contribute to open source projects. We found that non-technical barriers are the most relevant impediment they face to contribute (e.g., the lack of time of the project leaders).

- In Chapter 4, we propose a machine learning approach to identify unmaintained (or sporadically maintained) projects in GitHub, which achieved a precision of 80% and a recall of 96% when validated with real open source developers and projects.

- In Chapter 4, we also propose a metric, called Level of Maintenance Activity (LMA), to assess the risks of projects become unmaintained. We provided evidence on the applicability of this metric by investigating its usage in 2,927 projects.

5.3 Key Findings and Discussion

In this section, we discuss and put our findings in perspective.

Reasons to stop the maintenance of open source projects: In Chapter 2, we report a list of nine reasons of open source developers to stop the maintenance of their projects. The study reveals an important competition between open source projects. The most common reason for projects become unmaintained is the appearance of a stronger open source competitor (27 projects). Usually, this competitor is a major organization responsible for the ecosystem where the project is inserted in, specifically Google (Android ecosystem, 7 projects) and Apple (iOS ecosystem, 5 projects). Therefore, open source developers should be aware of the risks of starting to built a project that may attract the attention of major players, particularly when the projects have a tight integration and dependency with established platforms, such as Android and iOS.

The third and fourth reasons are the lack of time and lack of interest of the main maintainer. These reasons reinforce the importance of projects providing continuous feature enhancements to attract new contributors, mitigating the risks of become unmaintained [Ye and Kishida, 2003; Pinto et al., 2016; Steinmacher et al., 2016]. For example, clients of libraries facing the risks of discontinuation can be motivated to assume the project's maintenance.

Importance of following a set of best open source maintenance practices: In Chapter 2, we also showed that there is an important difference between the unmaintained projects and the most popular and active projects on GitHub, in terms of following best open source maintenance practices. However, in this case, it is important to consider that association does not imply in causation. For example, by just providing contributing guidelines or license, a project does not necessarily will succeed. However, we showed that successful projects include almost all documents, guidelines, and templates. Finally, we showed evidences on the benefits provided by continuous integration, in terms of automation of tasks like compilation, building, and testing. In this context, we emphasize the importance of adopting these practices by maintainers of open source projects.

Motivations and barriers faced by recent core developers: In Chapter 3, we reported the results of a survey with 52 developers, who recently became core contributors of popular GitHub projects. We reveal their motivations to contribute to an open source project (e.g., improving the projects because they are also using it), the project characteristics that most helped to contribute (e.g., a friendly community), and the barriers faced by them (e.g., lack of time of the project leaders). This knowledge may help maintainers of open source projects to improve the base of contributors of their projects. For example, our results show that maintainers should be whenever possible friendly and available with external contributors. Finally, our findings might contribute to current efforts targeting the development of health and analytics models and tools to open source projects [Jansen, 2014; Head, 2016; Steinmacher et al., 2016; Mens et al., 2017].

Assessing the risks of projects become unmaintained: In Chapter 4, we define a metric to express the *level of maintenance activity* of GitHub projects, i.e., a metric that reveals how often a project is being maintained. We analyze the historical evolution of 2,927 active projects, as classified by our model. Therefore, although developers has created thousands of open source projects (e.g., today GitHub has more than 100 million repositories), we show a considerable percentage of projects become unmaintained in the time frame of one year. Therefore, the proposed LMA metric may help users on selecting libraries that are not facing discontinuation risks. On the other hand, this information can also contribute to attract new maintainers to the project. For example, clients of libraries facing the risks of discontinuation can be motivated to assume their maintenance.

5.4 Future Work

Improvement of our dataset. To address **Q1**, **Q2**, and **Q3**, we used datasets created with open source projects, collected from GitHub. Although we selected a large sample of important and representative projects, our findings cannot be generalized to other ecosystems. Therefore, future research should investigate to which extent our results are applicable to other ecosystems (e.g., Bitbucket, GitLab, Android, etc.).

Propose strategies to prevent the discontinuation of OSS projects. We also recommend investigation on proactive strategies to prevent the discontinuation of projects, for example by identifying and recommending new maintainers with the required expertise to work in projects under threats of being deprecated.

Propose guidelines and practices for maintaining OSS projects: We showed that there is an important difference between the unmaintained projects and the most popular and active projects on GitHub, in terms of use of contributing guidelines, continuous integration, use of licenses, availability of home pages, and issue templates (Chapter 2). As future work, we plan to explore these findings to propose and validate a set of guidelines and practices for maintaining open source projects and attracting new contributors. We also plan to publish these guidelines in a public web site to facilitate their verification and adoption by practitioners.

Improvement of our machine learning model. To address **Q3**, we consider a total of 13 metrics provided by GitHub about a project's maintenance activity, e.g., number of commits, forks, issues, and pull requests. Future work may attempt to improve the performance of the proposed machine learning model by adding other features (e.g., number of dependents[1], average time to close an issue, average time to close a pull request, etc.).

Improvement of our Chrome extension. The goal of this improvement is to add visualizations, actionable dashboards, and analytics for extension. The idea is to automate the collection and continuous tracking of the proposed metrics. We expect that this tool could facilitate the analysis and visualization of the level of maintenance activity of GitHub projects, aiming to assess and mitigate the risks of discontinuation.

LMA for unmaintained projects. The goal of this work would be improve the LMA metric to also consider unmaintained projects.

Study of GitHub projects development process. We plan to compare the projects that become abandoned by development process. We expect to reveal whether the projects that become abandoned followed or not a given software development process (e.g., agile, waterfall, or spiral model).

Exploring the motivation to abandon a open source project. We plan to investigate other motivations for developers abandon a project, e.g., they might have decided to work in a different project.

Build models using other machine leaning algorithms. We plan to create other machine learning models using other algorithms, such as Support Vector Machine

[1] *https://help.github.com/en/articles/listing-the-projects-that-depend-on-a-repository*

(SVM) [Cortes and Vapnik, 1995], XGBoost [Chen and Guestrin, 2016], and Naive Bayes [Patil et al., 2013].

Analyze feature completed projects. We plan to build a machine learning model to identify feature completed projects. These projects deal with stable requirements and environments and therefore do not need constant updates or modifications. Therefore, we plan to remove these projects from our unmaintained list and label them as completed. We also plan to consider the usage of source code features to improve the model.

Bibliography

Alencar da Costa, D., Abebe, S. L., McIntosh, S., Kulesza, U., and Hassan, A. E. (2014). An Empirical Study of Delays in the Integration of Addressed Issues. In *International Conference on Software Maintenance and Evolution (ICSME)*, pages 281--290.

Androutsellis-Theotokis, S., Spinellis, D., Kechagia, M., Gousios, G., et al. (2011). Open source software: A survey from 10,000 feet. *Foundations and Trends in Technology, Information and Operations Management*, 4(3–4):187--347.

Avelino, G., Passos, L., Hora, A., and Valente, M. T. (2016). A novel approach for estimating truck factors. In *24th International Conference on Program Comprehension (ICPC)*, pages 1--10.

Avelino, G., Passos, L., Hora, A., and Valente, M. T. (2019). Measuring and analyzing code authorship in 1+118 open source projects. *Science of Computer Programming*, 176(1):14--32.

Bao, L., Xing, Z., Xia, X., Lo, D., and Li, S. (2017). Who will leave the company?: a large-scale industry study of developer turnover by mining monthly work report. In *14th International Conference on Mining Software Repositories (MSR)*, pages 170--181.

Barcomb, A., Stol, K.-J., Riehle, D., and Fitzgerald, B. (2019). Why do episodic volunteers stay in floss communities? In *41st International Conference on Software Engineering (ICSE)*, pages 1--12.

Bijlsma, D., Ferreira, M. A., Luijten, B., and Visser, J. (2012). Faster issue resolution with higher technical quality of software. *Software Quality Journal*, 20(2):265--285.

Borges, H., Hora, A., and Valente, M. T. (2016a). Predicting the popularity of GitHub repositories. In *12th International Conference on Predictive Models and Data Analytics in Software Engineering (PROMISE)*, pages 1--10.

Borges, H., Hora, A., and Valente, M. T. (2016b). Understanding the factors that impact the popularity of GitHub repositories. In *32nd IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 334--344.

Borges, H. and Valente, M. T. (2018). What's in a github star? understanding repository starring practices in a social coding platform. *Journal of Systems and Software*, 146:112--129.

Bosu, A., Carver, J., Guadagno, R., Bassett, B., McCallum, D., and Hochstein, L. (2014). Peer impressions in open source organizations: a survey. *Journal of Systems and Software*, 94(1):4--15.

Breiman, L. (2001). Random forests. *Machine learning*, 45(1):5--32.

Brito, A., Xavier, L., Hora, A., and Valente, M. T. (2018). Why and how Java developers break apis. In *25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 1--11.

Businge, J., Serebrenik, A., and van den Brand, M. (2012). Survival of eclipse third-party plug-ins. In *28th International Conference on Software Maintenance (ICSM)*, pages 368--377.

Calle, M. L. and Urrea, V. (2010). Letter to the editor: stability of random forest importance measures. *Briefings in bioinformatics*, 12(1):86--89.

Canfora, G., Penta, M. D., Oliveto, R., and Panichella, S. (2012). Who is going to mentor newcomers in open source projects? In *20th International Symposium on the Foundations of Software Engineering (FSE)*, pages 44–54.

Capiluppi, A., Lago, P., and Morisio, M. (2003). Characteristics of open source projects. In *7th European Conference on Software Maintenance and Reengineering (CSMR)*, pages 317--330.

Cerpa, N. and Verner, J. M. (2009). Why did your project fail? *Communications of the ACM*, 52(12):130--134.

Chen, T. and Guestrin, C. (2016). Xgboost: A scalable tree boosting system. In *22nd international conference on knowledge discovery and data mining (KDD)*, pages 785--794.

Chengalur-Smith, S. and Sidorova, A. (2003). Survival of open-source projects: A population ecology perspective. In *24th International Conference on Information Systems (ICIS)*, pages 1--7.

Chrissis, M. B., Konrad, M., and Shrum, S. (2003). *CMMI guidlines for process integration and product improvement*. Addison Wesley.

Coelho, J. and Valente, M. T. (2017). Why modern open source projects fail. In *11th Symposium on The Foundations of Software Engineering (FSE)*, pages 186--196.

Cortes, C. and Vapnik, V. (1995). Support-vector networks. *Machine learning*, 20(3):273--297.

Cox, D. R. (2018). *Analysis of survival data*. Routledge.

Cruzes, D. S. and Dyba, T. (2011). Recommended steps for thematic synthesis in software engineering. In *5th International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 275--284.

da Costa, D. A., Abebe, S. L., McIntosh, S., Kulesza, U., and Hassan, A. E. (2014). An empirical study of delays in the integration of addressed issues. In *30th International Conference on Software Maintenance and Evolution (ICSME)*, pages 281--290.

Eghbal, N. (2016). Roads and bridges: The unseen labor behind our digital infrastructure. Technical report, Ford Foundation.

Fernandez-Ramil, J., Lozano, A., Wermelinger, M., and Capiluppi, A. (2008). *Empirical Studies of Open Source Evolution*, pages 263--288. Springer.

Fogel, K. (2005). *Producing open source software: How to run a successful free software project*. " O'Reilly Media, Inc.".

Goldman, R. and Gabriel, R. P. (2005). *Innovation happens elsewhere: Open source as business strategy*. Morgan Kaufmann.

Gousios, G., Pinzger, M., and van Deursen, A. (2014). An exploratory study of the pull-based software development model. In *36th International Conference on Software Engineering (ICSE)*, pages 345--355.

Gousios, G., Storey, M.-A., and Bacchelli, A. (2016). Work practices and challenges in pull-based development: The contributor's perspective. In *38th International Conference on Software Engineering (ICSE)*, pages 285--296.

Gousios, G., Zaidman, A., Storey, M.-A., and Deursen, A. V. (2015). Work practices and challenges in pull-based development: the integrator's perspective. In *37th International Conference on Software Engineering (ICSE)*, pages 358--368.

Grissom, R. J. and Kim, J. J. (2005). *Effect sizes for research: A broad practical approach.* Lawrence Erlbaum.

Hata, H., Todo, T., Onoue, S., and Matsumoto, K. (2015). Characteristics of sustainable oss projects: A theoretical and empirical study. In *8th International Workshop on Cooperative and Human Aspects of Software Engineering (CHASE)*, pages 15--21.

Head, A. (2016). Social health cues developers use when choosing open source packages. In *24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*, pages 1133--1135.

Herraiz, I., Rodriguez, D., Robles, G., and Gonzalez-Barahona, J. M. (2013). The evolution of the laws of software evolution: A discussion based on a systematic literature review. *ACM Computing Surveys*, 46(2):1--28.

Hilton, M., Tunnell, T., Huang, K., Marinov, D., and Dig, D. (2016). Usage, costs, and benefits of continuous integration in open-source projects. In *31st International Conference on Automated Software Engineering (ASE)*, pages 426--437.

Hora, A., Valente, M. T., Robbes, R., and Anquetil, N. (2016). When should internal interfaces be promoted to public? In *24th International Symposium on the Foundations of Software Engineering (FSE)*, pages 280--291.

Humphrey, W. S. (2005). Why big software projects fail: The 12 key questions. *Crosstalk, The Journal of Defense Software Engineering*, 3(18):25--29.

Izquierdo, J. L. C., Cosentino, V., and Cabot, J. (2017). An empirical study on the maturity of the eclipse modeling ecosystem. In *20th International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 292--302.

Jansen, S. (2014). Measuring the health of open source software ecosystems: Beyond the scope of project health. *Information and Software Technology*, 56(11):1508--1519.

Jiang, J., Lo, D., He, J., Xia, X., Kochhar, P. S., and Zhang, L. (2016). Why and how developers fork what from whom in GitHub. *Empirical Software Engineering*, 22(1):547--578.

Jiang, J., Lo, D., He, J., Xia, X., Kochhar, P. S., and Zhang, L. (2017). Why and how developers fork what from whom in GitHub. *Empirical Software Engineering*, 22(1):547--578.

Joblin, M., Apel, S., Hunsen, C., and Mauerer, W. (2017). Classifying developers into core and peripheral: An empirical study on count and network metrics. In *39th International Conference on Software Engineering (ICSE)*, pages 164--174.

Jørgensen, M. and Moløkken-Østvold, K. (2006). How large are software cost overruns? a review of the 1994 CHAOS report. *Information and Software Technology*, 48(4):297--301.

Jr, M. W., Sathiyanarayanan, P., Nagappan, M., Zimmermann, T., and Bird, C. (2016). What went right and what went wrong: an analysis of 155 postmortems from game development. In *38th International Conference on Software Engineering Companion (ICSE)*, pages 280--289.

Kalliamvakou, E., Damian, D., Blincoe, K., Singer, L., and German, D. M. (2015). Open source-style collaborative development practices in commercial projects using GitHub. In *37th International Conference on Software Engineering (ICSE)*, pages 574--585.

Kaplan, E. L. and Meier, P. (1958). Nonparametric estimation from incomplete observations. *Journal of the American statistical association*, 53(282):457--481.

Khondhu, J., Capiluppi, A., and Stol, K.-J. (2013). Is it all lost? a study of inactive open source projects. In *9th International Conference on Open Source Systems (OSS)*, pages 61--79.

Koch, S. and Schneider, G. (2002). Effort, co-operation and co-ordination in an open source software project: GNOME. *Information Systems Journal*, 12(1):27--42.

Kon, F., Meirelles, P., Lago, N., Terceiro, A., Chavez, C., and Mendonça, M. (2011). Free and open source software development and research: Opportunities for software engineering. In *25th Brazilian Symposium on Software Engineering (SBES)*, pages 82--91.

Lamkanfi, A., Demeyer, S., Giger, E., and Goethals, B. (2010). Predicting the severity of a reported bug. In *7th IEEE Working Conference on Mining Software Repositories (MSR)*, pages 1--10.

Landis, J. R. and Koch, G. G. (1977). The Measurement of Observer Agreement for Categorical Data. *Biometrics*, 33(1):159--174.

Lavallée, M. and Robillard, P. N. (2015). Why good developers write bad code: An observational case study of the impacts of organizational factors on software quality. In *37th International Conference on Software Engineering (ICSE)*, pages 677--687.

Lee, A., Carver, J. C., and Bosu, A. (2017). Understanding the impressions, motivations, and barriers of one time code contributors to FLOSS projects: A survey. In *39th International Conference on Software Engineering (ICSE)*, pages 187--197.

Lehman, M. M. (1980). Programs, life cycles, and laws of software evolution. *IEEE*, 68(9):1060--1076.

Lehman, M. M., Ramil, J. F., Wernick, P. D., Perry, D. E., and Turski, W. M. (1997). Metrics and laws of software evolution the nineties view. In *4th International Software Metrics Symposium Proceedings (METRICS)*, pages 20--32.

Lerner, J. and Tirole, J. (2002). Some simple economics of open source. *The journal of industrial economics*, 50(2):197--234.

Lessmann, S., Baesens, B., Mues, C., and Pietsch, S. (2008). Benchmarking classification models for software defect prediction: A proposed framework and novel findings. *IEEE Transactions on Software Engineering*, 34(4):485--496.

Lin, B., Robles, G., and Serebrenik, A. (2017). Developer turnover in global, industrial open source projects: Insights from applying survival analysis. In *12th International Conference on Global Software Engineering (ICGSE)*, pages 66--75.

Linares-Vásquez, M., Bavota, G., Bernal-Cárdenas, C., Penta, M. D., Oliveto, R., and Poshyvanyk, D. (2013). API change and fault proneness: a threat to the success of Android apps. In *9th International Symposium on the Foundations of Software Engineering (FSE)*, pages 477--487.

Liu, C., Yang, D., Zhang, X., Ray, B., and Rahman, M. M. (2018). Recommending GitHub projects for developer onboarding. *IEEE Access*, 6(1):52082--52094.

Louppe, G., Wehenkel, L., Sutera, A., and Geurts, P. (2013). Understanding variable importances in forests of randomized trees. In *26th Advances in Neural Information Processing Systems (NIPS)*, pages 431--439.

Maldonado, E., Abdalkareem, R., Shihab, E., and Serebrenik, A. (2017). An empirical study on the removal of self-admitted technical debt. In *33rd International Conference on Software Maintenance and Evolution (ICSME)*, pages 238--248.

Martin, W., Sarro, F., and Harman, M. (2016). Causal impact analysis for app releases in google play. In *24th International Symposium on Foundations of Software Engineering (FSE)*, pages 435--446.

Meirelles, P., Jr, C. S., Miranda, J., Kon, F., Terceiro, A., and Chavez, C. (2010). A study of the relationships between source code metrics and attractiveness in free software projects. In *24th Brazilian Symposium on Software Engineering (SBES)*, pages 11--20.

Mens, T., Adams, B., and Marsan, J. (2017). Towards an interdisciplinary, socio-technical analysis of software ecosystems health. In *16th Belgian-Netherlands Software Evolution Symposium (BENEVOL)*, pages 7--9.

Mens, T., Goeminne, M., Raja, U., and Serebrenik, A. (2014). Survivability of software projects in gnome–a replication study. In *7th International Seminar Series on Advanced Techniques & Tools for Software Evolution (SATToSE)*, pages 79 – 82.

Menzies, T., Butcher, A., Cok, D., Marcus, A., Layman, L., Shull, F., Turhan, B., and Zimmermann, T. (2013). Local versus global lessons for defect prediction and effort estimation. *IEEE Transactions on software engineering*, 39(6):822--834.

Mirhosseini, S. and Parnin, C. (2017). Can automated pull requests encourage software developers to upgrade out-of-date dependencies? In *32nd International Conference on Automated Software Engineering (ASE)*, pages 84--94.

Mockus, A., Fielding, R. T., and Herbsleb, J. D. (2002). Two case studies of open source software development: Apache and Mozilla. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 11(3):309--346.

Oman, P. and Hagemeister, J. (1992). Metrics for assessing a software system's maintainability. In *8th International Conference on Software Maintenance (ICSM)*, pages 337--344.

Patil, T. R., Sherekar, S., et al. (2013). Performance analysis of naive bayes and j48 classification algorithm for data classification. *International journal of computer science and applications*, 6(2):256--261.

Peters, F., Menzies, T., and Marcus, A. (2013). Better cross company defect prediction. In *10th Working Conference on Mining Software Repositories (MSR)*, pages 409--418.

Pinto, G., Steinmacher, I., and Gerosa, M. A. (2016). More common than you think: An in-depth study of casual contributors. In *23th International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 112--123.

Provost, F. and Fawcett, T. (2001). Robust classification for imprecise environments. *Machine learning*, 42(3):203--231.

Qiu, H. S., Nolte, A., Brown, A., Serebrenik, A., and Vasilescu, B. (2019). Going farther together: The impact of social capital on sustained participation in open source. In *41st International Conference on Software Engineering (ICSE)*, pages 688--699.

Ramasubramanian, K. and Singh, A. (2017). *Machine Learning Model Evaluation*. Apress.

Rastogi, A., Nagappan, N., Gousios, G., and van der Hoek, A. (2018). Relationship between geographical location and evaluation of developer contributions in GitHub. In *12th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, page 22.

Raymond, E. (1999). The cathedral and the bazaar. *Knowledge, Technology & Policy*, 12(3):23--49.

Robles, G., Gonzalez-Barahona, J. M., and Herraiz, I. (2009). Evolution of the core team of developers in libre software projects. In *6th International Working Conference on Mining Software Repositories (MSR)*, pages 167--170.

Robles, G., Reina, L. A., Serebrenik, A., Vasilescu, B., and González-Barahona, J. M. (2014). Floss 2013: A survey dataset about free software contributors: challenges for curating, sharing, and combining. In *11th Working Conference on Mining Software Repositories (MSR)*, pages 396--399.

Samoladas, I., Angelis, L., and Stamelos, I. (2010). Survival analysis on the duration of open source projects. *Information and Software Technology*, 52(9):902--922.

Setia, P., Rajagopalan, B., Sambamurthy, V., and Calantone, R. (2012). How peripheral developers contribute to open-source software development. *Information Systems Research*, 23(1):144--163.

Silva, D., Tsantalis, N., and Valente, M. T. (2016). Why we refactor? confessions of GitHub contributors. In *24th International Symposium on the Foundations of Software Engineering (FSE)*, pages 858--870.

Söderberg, J. (2015). *Hacking capitalism: The free and open source software movement.* Routledge.

Standish Group (1994). CHAOS: Project failure and success report report. Technical report, MISSING.

Steinmacher, I., Conte, T. U., Treude, C., and Gerosa, M. A. (2016). Overcoming open source project entry barriers with a portal for newcomers. In *38th International Conference on Software Engineering (ICSE)*, pages 273--284.

Steinmacher, I., Pinto, G., Wiese, I. S., and Gerosa, M. A. (2018). Almost there: A study on quasi-contributors in open-source software projects. In *40th International Conference on Software Engineering (ICSE)*, pages 256--266.

Thung, F., Lo, D., and Jiang, L. (2012). Automatic defect categorization. In *19th Working Conference on Reverse Engineering (WCRE)*, pages 205--214.

Tian, Y., Lo, D., Xia, X., and Sun, C. (2015a). Automated prediction of bug report priority using multi-factor analysis. *Empirical Software Engineering*, 20(5):1354--1383.

Tian, Y., Nagappan, M., Lo, D., and Hassan, A. E. (2015b). What are the characteristics of high-rated apps? a case study on free Android applications. In *31st International Conference on Software Maintenance and Evolution (ICSME)*, pages 301--310.

Tian, Y., Nagappan, M., Lo, D., and Hassan, A. E. (2015c). What are the characteristics of high-rated apps? a case study on free Android applications. In *30th International Conference on Software Maintenance and Evolution (ICSME)*, pages 301--310.

Tourani, P., Adams, B., and Serebrenik, A. (2017). Code of conduct in open source projects. In *24th International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 24--33.

Trong, T. T. and Bieman, J. M. (2005). The FreeBSD project: A replication case study of open source development. *Transactions on Software Engineering*, 31(6):481--494.

Valiev, M., Vasilescu, B., and Herbsleb, J. (2018). Ecosystem-level determinants of sustained activity in open-source projects: a case study of the pypi ecosystem. In *26th Symposium on the Foundations of Software Engineering (FSE)*, pages 644--655.

Vasilescu, B., Yu, Y., Wang, H., Devanbu, P., and Filkov, V. (2015). Quality and productivity outcomes relating to continuous integration in GitHub. In *10th Joint Meeting on Foundations of Software Engineering (ESEC)*, pages 805--816.

Yamashita, K., McIntosh, S., Kamei, Y., and Ubayashi, N. (2014). Magnet or sticky? an oss project-by-project typology. In *11th working conference on mining software repositories (MSR)*, pages 344--347.

Ye, Y. and Kishida, K. (2003). Toward an understanding of the motivation open source software developers. In *25th International Conference on Software Engineering (ICSE)*, pages 419--429.

Zhou, M. and Mockus, A. (2015). Who will stay in the FLOSS community? modeling participant's initial behavior. *IEEE Transactions on Software Engineering*, 41(1):82--99.