DISSERTATIONES
INFORMATICAE
UNIVERSITATIS
TARTUENSIS
2

RIIVO KIKAS

Analysis of Issue and Dependency Management in Open-Source Software Projects

DISSERTATIONES INFORMATICAE UNIVERSITATIS TARTUENSIS

2

DISSERTATIONES INFORMATICAE UNIVERSITATIS TARTUENSIS

2

RIIVO KIKAS

Analysis of Issue and Dependency Management in Open-Source Software Projects

UNIVERSITY OF TARTU
Press

Institute of Computer Science, Faculty of Science and Technology, University of Tartu, Estonia.

Dissertation has been accepted for the commencement of the degree of Doctor of Philosophy (PhD) in informatics on October 15, 2018 by the Council of the Institute of Computer Science, University of Tartu.

Supervisors

Prof. Marlon Dumas
University of Tartu
Estonia

Prof. Dietmar Pfahl
University of Tartu
Estonia

Opponents

Prof. Tom Mens
University of Mons
Belgium

Assoc. Prof. Andy Zaidman
Delft University of Technology
The Netherlands

The public defense will take place on December 11, 2018 at 10:15 am in J. Liivi 2-405.

The publication of this dissertation was financed by the Institute of Computer Science, University of Tartu.

ISSN 2613-5906
ISBN 978-9949-77-901-7 (print)
ISBN 978-9949-77-902-4 (PDF)

University of Tartu Press
`http://www.tyk.ee/`

ABSTRACT

Modern open-source software development is being carried out on public platforms such as GitHub. The growing number of open-source projects and their users raises new challenges in how to navigate community contributions and requests. Users can contribute feature requests or bug reports to projects, increasing the workload for developers. Meanwhile, developers try to make their life easier by reusing third-party code in their software, sometimes without understanding the possible side-effects. This thesis deals with two problems in open-source software development: analyzing issue lifetime and understanding the structure of software package dependency networks.

Issue repositories are used to keep track of bugs, development tasks, and feature requests in software development projects. For open-source projects, anyone can submit a new issue report, which can lead to situations where more issues are created than can be effectively handled by the project members. This raises the question of how issues are treated as the capacity of the project members is exceeded. In this thesis, we study the temporal dynamics of issue reports based on a sample of 4,000 open-source projects. We specifically analyze how the rate of issue creation, the number of pending issues, and their average lifetime evolve over the course of time. The results show that more issues are opened shortly after the creation of a project repository and that the number of pending issues increases inexorably due to forgotten (unclosed) issues. The average issue lifetime (for issues that do get closed) is relatively stable over time.

Methods for predicting issue lifetime can help software project managers to prioritize issues and allocate resources accordingly. We explore ways to incorporate different types of data into issue lifetime models, such as comments and developer activity. In this thesis, we develop a machine learning-based method, applied at different points in an issue's lifetime, to determine whether or not the issue will close within a given calendric period. Our method combines static, dynamic and contextual features. The results show that dynamic and contextual features complement the predictive power of static ones, particularly for long-term predictions.

Another problem studied in this thesis is how software dependencies are used. Software developers often include third-party open-source software packages in their projects as a dependency to minimize redundant effort. The included dependencies can also have their own dependencies. A complex network of dependency relationships exists among open-source software packages. This thesis analyzes the dependency network structure and evolution of the JavaScript, Ruby, and Rust ecosystems. We propose a method that can be generalized for other ecosystems to measure their growth and evolution. The results reveal significant differences across language ecosystems. They also indicate that the number of transitive dependencies for JavaScript has grown 60% over the last observed year, suggesting that developers should look more carefully into their dependencies to understand

what exactly is included. This study also reveals that the vulnerability to the removal of the most popular package is increasing, yet most other packages have a decreasing impact on vulnerability.

CONTENTS

1. INTRODUCTION

Computers and software are running essential services for our every-day life. A modern car contains of about hundred million lines of code [34] to control the engine. Moreover, modern medical diagnostics is done with the aid of computers controlled by software. These few examples illustrate how software has been introduced into different domains to support existing complex activities and technologies. However, these software systems need to be first developed and later maintained indefinitely [92]. During the development process, teams need to make a variety of decisions such as which features to include in the next release or which library to use to speed up the development.

Analyzing data generated during the software development process can support developers in decision-making. Software developers leave traces of their activities, such as code change commits in version control systems, lists of tasks worked on, bugs fixed in issue tracking system, communication data in the form of email, and chat logs. These traces can be used to learn models about activities in software projects. A model can be used in decision-making for future activities. For example, bug reports stored in software projects' issue trackers contain textual description and an assessment of their severity from a developer. This information can be used to build a model that estimates bug report severity based on the presence of certain words in bug report descriptions. Using this model to predict the severity level for all new incoming bug reports can help developers resolve important issues, such as security issues, only if in the past security issues were ranked as severe, and the model has learned this capability.

The process of analyzing data from the software development process and making decisions based on the findings has been called software analytics [31]. Software analytics has already been put into practice in large enterprises such as Microsoft [44] and Mozilla [16], where internal tools have been developed to support decision-making in software projects based on past data. The goal of this thesis is to contribute to enhancing the body of existing methods in the area of software analytics to support decision-making by putting forward and testing new methods in the areas of issue and dependency management.

This thesis studies two different problems in the domain of software analytics. The first part of this thesis introduces a method to analyze the life-cycle of issue reports in open-source projects by applying machine learning techniques. We develop an issue lifetime prediction model that can help estimate the time it will take to resolve an issue. This is relevant to different stakeholders in a software project such as developers, project managers, or users requesting a change.

The second part of this thesis deals with analyzing dependency usage in open-source projects. Software projects reuse libraries to reduce duplicate code. However, little is known about how often developers update their dependencies. We study dependency management in open-source projects and introduce methods and metrics for continuously monitoring dependencies.

1.1. Problem area

The problems studied in this thesis, namely issue lifetime analysis and dependency management practices, are part of every modern software development project. This thesis focuses on open-source software projects hosted on GitHub, a social coding platform. GitHub hosts code repositories for millions of projects and provides functionality for issue management and code review (in the form of pull request review). It has become the largest platform for open-source development and collaboration, hosting projects ranging from desktop applications, machine learning libraries, to mobile application development frameworks.

Centralizing development on GitHub has lowered barriers for community contributions [45]. Besides project members, issue reports are also contributed by end-users. Raising the visibility of possible projects and developer reputation [45] can encourage developers to reuse projects and libraries that are being developed and maintained by other developers. This thesis deals with understanding the issue dynamics and evaluating library-level dependencies in open-source projects.

GitHub can be considered as an ecosystem for software development. Lungu [96] has defined software ecosystems as *a collection of software systems, which are developed and co-evolve in the same environment*. The two problems studied in this thesis both revolve around GitHub. We study issue handling in GitHub projects. The dependency analysis is concerned with packages and projects that are also being developed on GitHub. Although there are no explicit links in mining issue reports and dependency management, the implicit relationship can be illustrated by following example. When analyzing issue lifetime distributions for projects we found there are projects where issues are left open for long periods. If these open issue reports are actually bug reports about possible vulnerabilities, it could affect other projects using this project.

In the next section we explain the context of the problems studied in this thesis in more detail.

1.1.1. Issue dynamics

Issue trackers have become essential collaboration instruments in modern software development projects [21]. They are used for registering and tracking new feature requests, development tasks, and bugs. In closed-source projects, usage of issue trackers is generally restricted and sometimes codified, so that new issues can only be opened by development team members, managers and a reduced set of stakeholders, and they may need to comply with established norms and minimum requirements [21].

On the other hand, in open-source GitHub projects, it is common practice for everyone to open new issues in the issue tracker of a project with basically no requirements placed on the content and quality of new issues [24, 42]. This practice can lead to a potentially large and continuous inflow of issues exceeding the project's development team capacity, including low-quality issues or issues that are only marginally relevant to the project. Anvik et al. found in 2005 that the Mozilla repository was receiving more than 300 issues per day and that this was too much for the team to handle [7]. As the inflow of issues exceeds the capacity of the project members, it is natural to conjecture that not all issues are effectively handled, and are either closed without resolution or are implicitly ignored.

Understanding issue dynamics can be beneficial for different stakeholders. Project owners can have an overview of the issue-resolving speed and current team capacity. Users interested in having issues resolved can have an overview of the team performance or estimated speed. In a wider perspective, it can give overall health of the projects – if issues are not being resolved or the issue resolution pace has slowed down, then the project might not be sustainable in the long term. Dabbish et al. [45] also found from interviewing GitHub users that a large number of open pull-requests can be a sign of trouble in handling community contributions in the project. We hypothesize that same applies for issues, considering that pull requests are a solution to some problems and should be easier to resolve than other issues.

1.1.2. Issue lifetime prediction

When analyzing issue dynamics, we noticed a pattern that in many projects, the number of open issues shows an increasing trend. This observation leads us to the problem of predicting when an issue will be closed. Knowing when an issue will be closed is important from two viewpoints. First, it has been found that timeliness is an important determinant of contributor engagement and community contribution acceptance in GitHub [62]. If there is high uncertainty regarding the time frame in which the development team will address a given issue, the stakeholder who submitted it might be discouraged from making further contributions or even from using the software product. Having an estimate of issue closing time can help to reduce this uncertainty and provide greater transparency to all stakeholders. Second, an estimate of issue-closing time provides core team members with a basis to prioritize their efforts and plan their contributions. In this respect, a recent study of long-lived bugs in different projects [118] observes that over 90% of such bugs impact user experience and that automatic prioritization and assignment can minimize the impact of such bugs on end users. It is also observed that in some cases, bugs can be resolved earlier thanks to automatic prioritization and assignment.

In this thesis, we address the problem of predicting, at a given time point during an issue's lifetime, whether or not the issue in question will close after a given time horizon, e.g. predicting whether an issue that has been open for one week will remain open one month after its creation. The general problem of issue (or bug) lifetime prediction has received significant attention in the research literature. The focus of this study differs from previous work in four respects. First, the bulk of previous work has focused on analyzing a small number of hand-picked projects. In contrast, this thesis studies this prediction problem based on a large sample of projects hosted in GitHub. Second, most previous work has focused on exploiting static features, i.e. characteristics extracted for a given snapshot of an issue – typically issue creation time. In contrast, the present study combines static features available at issue creation time with dynamic features, i.e. features that evolve throughout an issue's lifetime. Third, previous approaches have focused on predicting issue lifetime based on characteristics of the issue itself. In contrast, the present study combines characteristics of the issue itself with contextual information, such as the overall state of the project or recent development activity in the project. Finally, most previous studies have not employed temporal splits to construct prediction models. In other words, models are trained on future data and then evaluated on past data. In this study, we construct models predictively using strict temporal splits such that predictions are always made based only on past data, which reflects how such predictive models would be used in practice.

1.1.3. Dependency analysis

Open-source software development has resulted in an abundance of freely available software packages (libraries) that can be used as building blocks for new projects. Usage of existing libraries can increase velocity and reduce the cost of software projects [103]. Thung et al. [124] found by manually examining 1,008 projects on GitHub that 93.3% of them use third-party libraries, with an average of 28 third-party libraries per project. However, introducing third-party libraries makes a project dependent on them. Dependencies need to be kept up-to-date to prevent exposure to vulnerabilities and bugs [41]. At the same time, bugs can also originate from transitive dependencies [71]. Developers might not have an overview of all the transitive dependencies as they did not include them themselves. Updating dependencies also entails risks, as new versions may break existing functionality or API correctness [115].

In March 2016, a single JavaScript package, *left-pad* was removed from the central JavaScript package repository `npm`. The removal also caused issues for projects that depended on it indirectly through transitive dependencies [91]. The *left-pad* incident illustrates the hidden risks of relying on publicly available packages. A problem with a single package can propagate through multiple levels of dependencies.

Over the years, a number of studies have addressed the question of how to develop maintainable software and how to cope with software evolution challenges [100, 127]. On the other hand, dependency management practices have received little attention, despite being a crucial part of almost all software projects. A study of the JavaScript package ecosystem [136] revealed that dependency requirement specifications using semantic versioning with flexible version constraints (e.g., the latest version) are widely used. This practice often leads to a new version of dependency being used implicitly every time a project is built. Another study of Maven packages [115] revealed that the semantic versioning scheme is not always used properly and breaking changes are also introduced in minor version releases. Implicit updates combined with non-conforming API changes can introduce unexpected behavior or software defects. Considering the *left-pad* incident and the lack of studies on dependency management, we seek to enhance the understanding of the state of dependency update practices and the structure of dependency networks.

Data available from package repositories and GitHub repositories enable us to study the package ecosystems of different programming languages. Having access both to packages that are published in a central repository and applications using these can give us an idea of how often dependencies are updated and the state of the dependency ecosystem.

In this thesis, we take a novel network-based approach for studying dependency networks of JavaScript, Ruby, and Rust. We use data from package repositories and a subset of GitHub projects. We compose a network of projects based on dependency relations to understand how the dependency network evolves and how susceptible it is to different types of attacks, such as removal of a random project. We show that dependency networks of popular languages such as JavaScript and Ruby are growing and have at least one single package whose removal can affect more than 30% of projects in the ecosystem.

1.2. Problem statement

For the problems studied in this thesis, we have formulated three groups of research questions (RQ) to guide our research. Each research question consists of several sub-questions.

RQ1: What are the dynamics of issue lifetime in open-source projects?

- RQ1.1: What is the issue arrival rate and how does it change over time?
- RQ1.2: How do opened and pending issue numbers evolve over time?
- RQ1.3: What is the average issue lifetime and how does it change over time?

RQ2: How can we estimate the time period required for an issue to be closed?

- RQ2.1: What level of accuracy is achieved by classification models trained to predict issue lifetime at different calendric time points in an issue's lifetime and for different calendric periods (one day, one week, one month, one quarter, one semester and one year) using both static and dynamic features of an issue as well as contextual features?
- RQ2.2: What features are most important when predicting issue lifetime?

RQ3: What are the characteristics of open-source package ecosystems?

- RQ3.1: What are the static characteristics of package dependency networks?
- RQ3.2: How do package dependency networks evolve?

- RQ3.3: How vulnerable are package dependency networks to the removal of a random project?

The research question RQ1 and its sub-questions were proposed to study issue dynamics in GitHub projects, which can be beneficial for different stakeholders. Project maintainers and users interested in having issues resolved can have an overview of the issue-resolving speed. Studying the proposed research questions should give an overview of issue resolving dynamics in GitHub and indicate how fast issues are dealt with in open-source projects.

The research question RQ2 was proposed to study if it is possible to build automated methods for estimating issue resolution time and how accurate they can be. Such automated methods can bring transparency and can help stakeholders prioritize their work. To bring more insight into which features are important for issue lifetime prediction, we proposed RQ2.2 to study what features are relevant in the prediction models.

The research question RQ3 and its sub-questions were proposed to study the dependency management practices and dependency network structure for popular programming languages. These questions help to guide our research to bring visibility into how package ecosystems are growing and if there are important packages in the ecosystems. Understanding the structure and the dependence on central packages gives insights how vulnerable ecosystems are to specific attacks, such as removal of a package and spreading of security issues through dependencies.

1.3. Research approach

The research method in this thesis is data-driven. We collected GitHub project data using GHTorrent [60]. We then conducted empirical studies to understand issue lifetime and dependency management. We also developed machine learning models for issue lifetime prediction. For issue lifetime prediction, we analyzed existing approaches and found possible shortcomings that would not enable us to apply these models to real-world settings. To validate the constructed machine learning models we use standard techniques in the field, chiefly a strict split between training and testing data, cross-validation inside the training set for deriving new features, and we use well-accepted measures of accuracy.

To ensure reproducibility, we made available all the source code required to perform the experiments, with instructions for its use. One of the studies reported in the thesis has been successfully reproduced by a third party [101] as reported later in the thesis.

1.4. Contributions of the thesis

The main contributions of this thesis are the following:

- We develop an approach for analyzing and quantifying issue accumulation in open-source projects.
- We show that a fraction of issues opened in open-source projects remain open for long periods of time.
- We develop a machine-learning-based method for temporal prediction of issue lifetime in GitHub projects.
- We show that training models over different observation periods can give better estimates. In addition, we find that different sets of features are important for different prediction tasks.
- We develop a method for analyzing package ecosystems based on network analysis.
- We show that in `npm` package ecosystem the number of transitive dependencies is increasing and the vulnerability to the removal of a single package is generally decreasing.

1.5. Thesis organization

This introduction has provided context for the thesis, introduced the methods used, and describes the contributions of the thesis.

Chapter 2 reviews state-of-the-art technologies related to this thesis and introduces wider concepts. In Chapter 3 we give a brief overview of the machine learning and network analysis used in the thesis.

Chapter 4 presents an analysis of issue lifetime dynamics in GitHub. The contents of this chapter have been published earlier [84]. The author is responsible for study design, data analysis, interpretation, and writing the first version of the manuscript.

In Chapter 5 we develop an algorithm for predicting issue closing time in GitHub projects. This chapter is based on findings that have been published [85]. The author participated in study design, data analysis, interpretation and writing the first draft of the manuscript.

Chapter 6 analyses dependencies and package ecosystems. The contents of this chapter have been published [86]. The author developed the idea jointly with co-authors, carried out data collection, data analysis and wrote the first version of the manuscript.

The concluding remarks, discussion, and possible future research directions are outlined in Chapter 7.

2. STATE OF THE ART

In the introduction we discussed analyzing data generated during the software development process and why it is important. To deliver software analytics, we can use data stored in software repositories to find insights about the development process. This chapter introduces the mining software repositories concept, one of the approaches used in software analytics. We then cover state-of-the-art technologies for the research questions studied in the thesis, namely issue analysis and dependency management.

2.1. Mining software repositories

The field of mining software repositories (MSR) deals with extracting useful information from software artifacts [68] generated during the development process. Software developers leave traces of their actions and the tools they use. An essential part of a software project repository is the source code version control system (VCS), which stores and tracks changes made to the source and merges changes from multiple edits into a single file.

In 1975 Rochkind [116] introduced the first VCS named Source Code Control System and demonstrated cases where software configuration management was applied to system development. Later in the 1980s, several new systems appeared, such as RCS in 1982 [126] and followed by Concurrent Version System (CVS). The aim of early version control systems was to help developers build and maintain evolving software systems. In 1980 Lehman used the term software evolution [92] and stated that software needs to be maintained over time to add value.

In the late 1990s, it was discovered that software repositories contained potentially valuable data and multiple studies appeared to study evolving software systems and developer behavior. Earlier studies using source code repositories were carried out on the basis of private code [53], but with the emergence of open-source software, research on open repositories took off [102]. Around the same time Bugzilla [30] was introduced for issue and bug management. SourceForge was one of the first publicly hosted source code repositories for open-source projects. In 2005, the git version control system was introduced. GitHub was launched in 2008, being the first social coding platform to provide an easier mechanism by submitting a pull request instead of e-mailing a patch (a difference of two revisions). The sudden vast amount and variety of data available from open-source projects made it possible to study from historical evolution [145] and to track team dynamics and collaboration in open-source software [23].

In addition to data and traces generated by developer actions and tools, mining software repositories also incorporates other data, such as runtime logs, or data obtained through static analysis such as call graphs of a program or abstract syntax trees of the source code. Different data sources consist of different data types such as sequences, graphs, trees, and text [69, 140]. Next, we list some examples of different data sources and how they can be useful in a software engineering context.

- **Source code**. Commits to the repository and history of source code changes enabling the study of which files were changed together to notify developers in case they forgot to change a file [145] or to discover error patterns [94]. Source code auto-completion [73] leverages frequent sequential patterns found in the code.
- **Issue data.** Issue tracking systems record development tasks, such as new feature requests, bugs, and maintenance tasks. Also, they store state changes for an issue, such as when an issue was entered, updated, commented on, and resolved. This enables us to study the time that bugs spend in each state in the issue tracker [109], automatically triage bugs to the corresponding developer based on past patterns [8], detect duplicate bugs [132], and prioritize [125]. Combining issue tracking data with source code revisions we can predict and detect defects on source code [46, 144].
- **Requirements documentation.** Information retrieval techniques [98] have been used to link source code and software documentation, to support program comprehension and reverse engineer legacy systems.
- **Mailing list.** Developers coordinate and communicate through e-mail lists. These e-mail messages can be used to understand the developer team structure and who is knowledgeable about which part of module [23], or act as documentation for the code when linked together with source code repository [13].
- **Code review data.** Source code review is part of modern development process [12]. By analyzing existing code review data, we can automatically suggest knowledgeable reviewers in large software systems [123].
- **Log data.** Log records generated during software runtime can be useful for software performance analysis when coupled with source code, such as mining stack traces for performance [67] or helping developers to find performance issues when load-testing [79].
- **Question-answering sites.** Sites such as stackoverflow.com enable people to ask technical questions about software development. Question and answer data has been used for multiple purposes, such as to study what problems developers face in the context of energy-aware software development [110]. Stackoverflow data can be incorporated directly into developer tooling to help with documentation [111] or automatically generate comments for source code [137].
- **Application store data.** Mobile application stores contain developer-supported descriptions, reviews and feedback submitted by users. Malicious applications can be detected by comparing API usage and application descriptions to find descriptions that do not describe the APIs used by

the application [59]. The reviews can be used as a source to understand what users do not like [83] or to discover new requirements from user feedback [66].

- **Development environment interaction data.** Recording user interactions in the integrated development environment (IDE) can reveal how developers test their applications and navigate in the source code [18], and reveal how developers use their development environment [5].
- **Build data.** Continuous integration has been adopted in open-source software [19] for automating the building, testing and releasing processes. The build history dataset can reveal the most common failure patterns [20].

These data sources and previous studies have already been turned into everyday tools that developers use such as recommending pull request reviewers in GitHub [76] and the StackMine tool at Microsoft to analyze performance issues using stack traces [67]. Despite the variety of data sources and data types available, the goal of mining software repositories is to support developers in decision-making, build new tools to improve developer productivity and software quality, and automate tasks.

2.1.1. GitHub

GitHub is an online platform for hosting git source code repositories. It offers a web interface for repositories, an issue tracker, and a mechanism for contributing to other people's repositories using pull requests. People can fork a repository, which means cloning the code base into their repository while maintaining link to the original repository. After making changes in their repository, changes can be contributed back to the upstream repository (the original repository forked) by submitting a pull request. The pull request contains changes made in the developer repository. Members of the original team can review the code and discuss changes. The pull-based contribution model can be seen as lowering the contribution barrier. Prior to pull-based development, contributions typically were sent using patch files of code change differences to a mailing list [102,135]. The maintainers then discussed and applied the patches to the repository. Forking a repository also leaves a trace visible to the original repository owner. Dabbish et al. [45] found that the awareness of developers forking and modifying code is beneficial to maintainers. Maintainers can learn this way about user needs. They can also solicit pull requests or monitor usage of their software to prevent breaking changes. The same study revealed that the social nature of GitHub enables developers to follow interesting projects and developers, thus learning about new technologies or development practices faster.

GitHub, originally started in 2008, has been gaining popularity in recent years, with usages from single-person projects to major companies such as Google and Microsoft using GitHub for hosting their open-source projects. Already in January 2014, GitHub hosted 10.4 million repositories [81].

2.1.2. GHTorrent

GHTorrent [60] is a data collection effort to collect public repositories, commits, issues, and pull-requests from GitHub. It monitors the publicly available data stream offered by GitHub and enhances it by additionally fetching data using GitHub's official API. As a result, it stores the metadata of all the repositories, such as commits, issues, issue comments with text, and language used in projects. Data collected by GHTorrent has been used in multiple studies. However, as the whole dataset is large and contains a variety of projects, different studies have employed different sampling techniques [39]. The biggest limitation of using GitHub projects is the lack of ability to automatically filter out irrelevant projects (student homework assignments, repositories cloned manually and not via forking, etc.). This problem has been recently addressed by Munaiah et al. [104] who developed a tool to classify GitHub repositories into categories and aid in selecting suitable repositories for a study. GHTorrent data comes without any quality guarantees. GHTorrent service description states that minor inconsistencies and holes in data collection may exist. In this work we use a sample based approach to select projects for our studies and by selecting a large sample we can reduce the effect of missing data.

All studies in this thesis use GHTorrent either to collect the data (issue lifetime analysis in Chapter 4 and issue lifetime prediction in Chapter 5) or select candidate repositories and collect additional data from external sources (dependency management analysis in Chapter 6).

2.2. Issue management in software projects

Task and issue management are central to every software project for planning and tracking work to be done. Issue tracking systems are tools for organizing all the issues, tasks, and bugs in a software project. Some of the most popular issue trackers are Bugzilla [30] and JIRA [80]. Issue trackers are not specifically tailored to project types. All the previously mentioned tools are in use in private closed-source projects as well as in open-source projects. Although this thesis studies issue management in the context of open-source projects hosted on GitHub, the problem is relevant also in closed-source projects [65]. However, issue handling can be different in various kinds of projects due to other differences besides issue tracking tools, such as resources, goals, and processes [63].

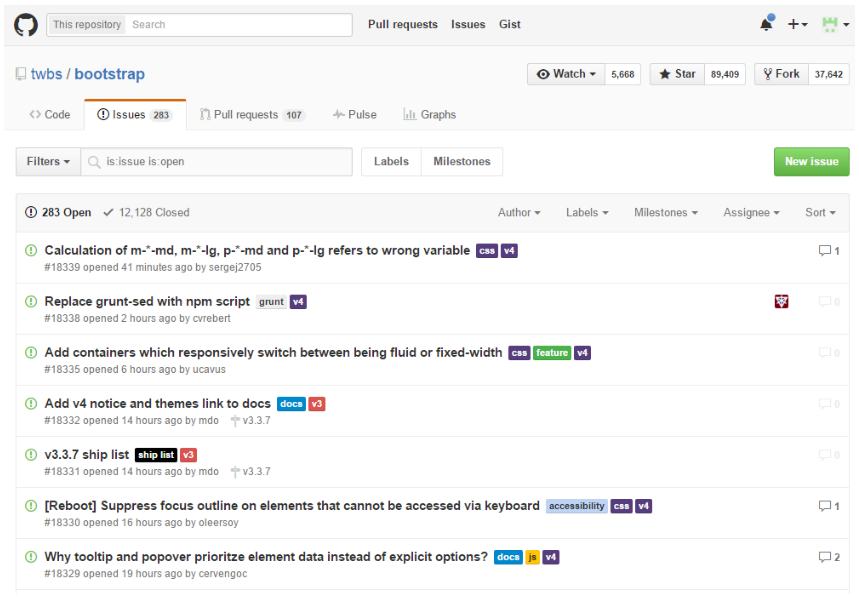
GitHub Issue tracker offers a subset of features compared to more powerful tools such as JIRA and Bugzilla. Each issue report needs a title and a description field at minimum to be filled out to describe the issue. The title and the description are entered as natural language text. An issue report can also have extra metadata, such as assignee, label, and milestone. Issue reports in GitHub lack customizable fields that are common in JIRA, e.g., it is not possible to enter time estimate, story points, priority or create other domain-specific fields. There is no designated field for issue type (e.g., bug, feature request, user story). Hence, in this work we deal with issues in general, and we do not distinguish between bug reports, new development task, maintenance task or non-code related activities such as documentation update tasks that are stored in the issue tracker. Missing issue type information can make a difference in analysis results, as the effort required to resolve an issue might depend on the issue type [11]. For example, simple one line code fixes can be done faster than implementing a new feature. However, even if the issue type information is present, it might not always be accurate. Herzig et al. [72] found that 33.8% of bug reports were misclassified by manually examining more than 7000 issues from five different projects. Even though JIRA and Bugzilla might provide better quality datasets with more fine grained metadata, this thesis focuses on GitHub. The reason is that there is less prior work dealing with GitHub issues and it is worthwhile to investigate how to build models specifically for GitHub, as it is a popular platform used for hosting millions of projects.

Figure 1 shows how an issue listing for a project looks on GitHub. A typical workflow in GitHub is an issue is entered, people discuss it and if code change is required, a pull request is created in relation to the issue. Issues do not have a customizable life cycle in terms of different states – an issue can be open, closed, or locked for everybody except team members. After closing, it can also be reopened. But it lacks states like being in review or being triaged. The issue listing view typically sorts issues in chronological order. Baysal et al. [17] argued that issue trackers are essential tools to bring awareness about what is going on in a project. In addition they found that developers value the temporal proximity of issues which is why recent issues appear first.

2.2.1. Issue lifetime analysis

Open-source projects can receive vast amounts of bug reports and feature requests. For example Mozilla issue trackers received on average over 175 issues per day over an 11-year period from 2002 to 2013 [17]. Reviewing, triaging, and resolving the issues in a timely manner is relevant to end users. Several previous studies have focused on issue lifetime mining. In this subsection we review relevant work to the RQ1 proposed in Section 1.2: *What are the dynamics of issue lifetime in open-source projects.*

Issue report mining has received much attention from the research community since the early 2000s [102]. Several previous studies have focused explicitly on the analysis of issues in GitHub datasets. Closest to our research is a study by Bissyande et al. [24] which gives a basic overview of issue tracker usage in 100,000 Github projects. Their work explores how many issues are tracked on average, labels and tags usage, who enters issues (developer or not), issue tracker usage and project success (number of watchers), and user community size and issue fix time. Compared to our work, they do not analyze the evolution of pending issues



283 Open ✓ 12,128 Closed

Calculation of m-*-md, m-*-lg, p-*-md and p-*-lg refers to wrong variable css v4 1
#18339 opened 41 minutes ago by sergej2705

Replace grunt-sed with npm script grunt v4 0
#18338 opened 2 hours ago by cvrebert

Add containers which responsively switch between being fluid or fixed-width css feature v4 0
#18335 opened 6 hours ago by ucavus

Add v4 notice and themes link to docs docs v3 0
#18332 opened 14 hours ago by mdo v3.3.7

v3.3.7 ship list ship list v3 0
#18331 opened 14 hours ago by mdo v3.3.7

[Reboot] Suppress focus outline on elements that cannot be accessed via keyboard accessibility css v4 1
#18330 opened 16 hours ago by oleersoy

Why tooltip and popover prioritze element data instead of explicit options? docs js v4 2
#18329 opened 19 hours ago by cervengoc

Figure 1. Issues of Bootstrap project in GitHub, captured in November 2015. The project has 283 open issues, while 12,128 have been closed. Labels (in colored blobs) are used to categorize the work to be done based on the technical component, release or type of issue.

and their lifetimes. Cabot et al. [32] studied how tagging is used in GitHub issue trackers. Their results showed that only small sets of projects use labels, and usage of labels correlates with a higher number of closed issues. Related to this, Izquierdo et al. [77] presented a tool demo to explore issue label usage in projects.

The question of issue lifetime has been studied from different perspectives both in open-source and closed-source projects. Marks et al. [99] studied issue lifetimes in Mozilla and Eclipse. They found that 46% of bug reports in the Mozilla project and 76% of bug reports in the Eclipse project are closed within three months of their creation. Grammel et al. [63] studied community involvement in closed source IBM Jazz projects. Their findings suggest that community-created issues can be valuable, but they are handled differently than those created by project members. The average issue lifetime for community-created issues is 39 days, whereas for team issues it is 5.9 days.

Ko and Chilana [87] studied why some issues in the Mozilla tracker are left open for long periods. Their findings suggest that issues resulting in fixes or code changes are proposed by a “group of experienced, frequent reporters.” They concluded that open source projects benefit most from this group of experts.

Garousi [56] studied three open-source projects with a focus on issue-creation and resolution times. Their findings showed that bugs and critical issues are handled faster than other issue types. For the jEdit and DrPython projects, the fractions of issues that are closed within the first day is 23% and 42% percent, respectively. Garousi also concluded that more bugs are submitted at the beginning of a project's lifetime. In addition, he showed that issues pile up over time and then are closed in batches.

Luijten et al. [95] study issue handling in GNOME issue tracker. To study the speed of issue resolution, they aggregate issue resolution times into risk profiles. They construct a risk profile by assigning each issue into a category based on the resolution time. They define four categories, i.e., an issue is resolved in 28 days (low risk), 70 days (moderate risk), 182 days (high risk) or more than 182 days (very high risk). A product will get a rating based on the fraction of issues in each category. Luijten et al. note that risk profiles are useful for comparing different periods of the history of a project. They also note that the GNOME backlog is constantly growing, with more issues entered than resolved. They also find evidence of bulk clean-up actions, where multiple high risk issues are closed during the same time, that have been resolved before but not closed.

Compared to these previous studies on issue lifetime, we employ a larger volume of projects and we consider not only issue lifetime, but also arrival rate and number of pending issues. The latter variable ("number of pending issues") was studied separately by Kenmei et al. [82], who used time-series modeling to analyze how the number of opened issues changed over time.

Other studies have considered how the arrival rate of new issues in a project and their resolution time can be used to plan future work [82] and to predict the lifetime of pending issues [58, 134]. These latter studies are orthogonal to this thesis.

2.2.2. Issue lifetime prediction

Projects with a continuous inflow of issues would benefit from tooling that would help to automatically estimate the properties of incoming issues, such as lifetime. The RQ2 studied in this thesis tries to answer *How can we estimate the time period required for issue to be closed*? Next we will review related work dealing with predicting issue resolution time and works related to predicting some issue attributes.

Weiss et al. [134] predicted issue resolution time for the JBoss project. Their approach enabled early prediction by finding a set of textually similar issues for a newly entered issue and using this set to predict closing time. Their estimated resolution times deviated on average by 7 hours from the actual resolution time, and half of the estimations are in the +/-50% range of the original issue lifetime. This study showed the feasibility of predicting issues based on models trained for one specific project, where the project in question has a large set of issues that share common patterns.

Similarly, Giger et al. [58] predicted bug fix times for Mozilla, Eclipse, and Gnome projects. Their approach consisted of extracting features for each bug report and training a decision tree model to predict whether the fix time will be lower or higher than the project median fix time. They also experimented with dynamic features calculated at different points during a bug report lifetime, such as the number of comments an issue has received and the number of actions performed on an issue. They concluded that the use of dynamic features improves accuracy. Their reported Area Under the ROC Curve (AUC) scores fell in the range [0.65–0.83]. A shortcoming of their work is that it does not apply temporal splits to separate training and testing data – hence the models may use data "from the future" to predict issue lifetime at a particular point in time.

Panjer [109] predicted resolution time for Eclipse bugs using a set of static features and a machine learning approach. He divided the bug lifetime distribution into seven ranges and uses these as classes. Experimenting with different classifiers, his approach can predict around 30% of issues correctly. Cross-validation is used for evaluation of the classifiers, but no temporal split is used to segregate training and test data. No dynamic features are used.

Francis & Williams [55] studied the prevalence of long-living bugs in an open-source Apache HTTP server and a closed-source private project. They trained a decision tree model to predict whether an issue will be closed by the time that 85%, 90%, and 95% of issues are closed. Their model achieved F-scores in the range [0.63–0.95] for the closed-source project and [0.21–0.59] for the Apache open-source project, suggesting that accurate issue lifetime prediction is more difficult in the context of open-source projects.

Besides issue lifetime prediction, there are other related lines of research that seek to predict some attribute or future action on an issue report. Examples are predicting whether an issue will be fixed [65], delayed [35], or reopened [121, 138, 139], as well as estimating its priority [125], assignment, or category [6, 64, 143]. In general, these studies use the same general approach: extract features from issue reports; train a machine learning model; and evaluate the model. A similar framework has also been used for predicting when contributed code patch or pull request will be accepted [61, 78, 142].

Table 1 summarizes the above review of related work in terms of six attributes that delimit the scope of the present study. Specifically, for each referenced study, the table indicates: (i) if the study relies on a large dataset – where "large" is defined as encompassing more than 6 projects (cf. column LD); (ii) whether or not the study relies on dynamic features in addition to static ones (column DF); (iii) whether or not the study in question relies on contextual features about the surrounding project in addition to features extracted from individual features (column CF); (iv) whether or not the study applies temporal splits to separate training data from testing data (column PT); (v) whether the constructed models can be used for predicting lifetime at issue creation time and during an issue's lifetime (column MA);[1] and (vi) whether the study addresses the problem of issue lifetime

[1]This criterion is included because some previous studies calculate features "as of the closing time" of each issue – e.g. they calculate the total number of comments received by an issue throughout its lifetime. Such studies are useful for post-mortem analysis of issue closing time, but not for predictive purposes.

Table 1. Related work for issue lifetime prediction.

Paper	LD	DF	CF	PT	MA	TYPE
Weiss et al. [134]				✓	✓	IL
Giger et al. [58]		✓			✓	IL
Panjer [109]						IL
Francis and Williams [55]					✓	IL
Assar et al. [10]				✓	✓	IL
Guo et al. [65]			✓		✓	IL
Marks et al. [99]			✓			IL
Gousios et al. [61]	✓		✓			CL
Jiang et al. [78]						CL
Ye et al. [142]	✓		✓			CL
Tian et al. [125]			✓	✓	✓	IA
Choetkiertikul et al. [35]			✓	✓		IA
Antoniol et al. [6]					✓	IA
Xia et al. [139]			✓			IA
Shihab et al. [121]			✓			IA
Guo et al. [64]			✓		✓	IA
Our approach	✓	✓	✓	✓	✓	IL

prediction (IL), acceptance time of a contribution (CL), or other issue attributes (IA) – cf. column TYPE.

We note that only Giger et al. [58] relied on dynamic features and made issue lifetime predictions at different time points during the issue's lifetime. Also, only the studies of Ye et al. [142] and Gousios et al. [61] rely on a large dataset when building their models; however this latter study is exploratory rather than intended to construct and test predictive models, and focusing on the problem of contribution acceptance prediction (pull requests). The use of contextual features has been considered in several previous studies, but not in conjunction with dynamic features. A few related studies use predictive splitting and are designed to be used at issue creation time.

In summary, the scope of the research presented in this thesis is that it uses static, dynamic and contextual features on a large issue dataset to construct issue lifetime models predictively (i.e., all predictions are strictly based on data available at the time the prediction is made).

2.2.3. Beyond issue lifetime prediction

The total time it takes to resolve an issue can comprise multiple factors, such as the complexity of the task to implement the issue (for technical tasks or user stories) and uncertainty concerning what needs to be done (debugging and fixing a bug).

In agile software teams, an iterative approach is used, where in a development sprint, typically lasting one to two weeks, software features and releases are delivered. User stories that reflect user requirements, are planned to be completed in each iteration. During planning, the relative complexity of each story is estimated using story points [38]. Story points typically take values from a Fibonacci sequence of $1, 2, 3, 5, 8, 13$ and the estimated values should reflect the relative complexity of the story when compared with other tasks. Story point estimate also incorporates an effort and risk assessment needed to implement the task. Developers participating in the estimation process have to come up with an agreement during a planning poker session [130]. The aim is not have a precise estimate, but to be able to compare tasks. Porru et al. [112] developed a method to estimate story points for JIRA task. They claim that automated estimation can be more reliable and reproducible and is not subject to personal biases or pressures from other stakeholders [9]. Their approach is similar to other works dealing with issue attribute prediction (IA, Section 2.2.2) using classifiers based on features extracted from issue reports. Compared to issue lifetime prediction, story point prediction can be more challenging as the story points in training data are estimates themselves, whereas issue lifetime is an explicit measure derived from issue state changes.

The goal of agile development is to deliver user stories during sprints. Therefore the more natural task is to predict which stories will be complete or how many story points will be completed during a sprint. Choetkiertikul et al. [36] developed a method for estimating delivery capability of an iteration in terms of predicting how many story points the sprint will deliver. They combine sprint-level features (for example number of participants) and aggregate issue-level features (mentions of words in all issues assigned to the sprint). The approach is fundamentally different compared to single level issue lifetime prediction as it takes into account all issues in the sprint. Whereas single issue-level estimate might cause changes because of ongoing conflicting work, the sprint level prediction can take this into account.

2.3. Package dependency management

To facilitate software reuse, programmers release their shared code as packages, so that common code can be shared between projects. Different programming languages have their own dependency management software and they typically distribute source code packaged with metadata about the package. The package manager takes care of the inclusion of the package in the project and code can be invoked as in any other code in the project. If a project A reuses code from a package P, we say the project is depending on the package or has a dependency. Next we review related work for research question RQ3 *What are the characteristics of open-source package ecosystems*. The related work can be grouped into three areas: dependency network analysis, dependency management processes and practices in software projects, and vulnerability spreading through dependencies.

2.3.1. Dependency networks

Dependency relations between projects form a network of dependencies. Network-based analysis of programming language dependency networks has emerged recently. The first large-scale analysis of the `npm` ecosystem was carried out by Wittern et al. [136]. Their analysis concludes that JavaScript is a striving ecosystem because of its frequent releases of new and existing packages. They use GitHub applications only to study version numbering practices and state that there is a prevalence of flexible (not exact) version number specifications. They conclude that usage of flexible version constraints should result in the immediate adoption of a new release.

Decan et al. [47] analyzed topologies of `npm`, `PyPI`, and `CRAN` and found that there are differences across ecosystems, e.g., the `PyPI` is less interconnected than `npm`. They stated that analysis results are not generalizable from one ecosystem to another. Their follow-up work [48] focusing on dependency version specification usage analysis, points out that current tools and versioning schemes can introduce co-installability issues. When a package specifies its dependency with a strict version or with a maximal version to use, it can prevent the package being installed if there are multiple versions of the dependency in the dependency chain with version constraints that do not match. They note that the prevalence of specifying versions with maximal constraint is increasing, thus possibly increasing the likelihood of co-installability issues.

German et al. [57] studied packages in the R ecosystem. They found that most packages do not have any dependencies, but popular ones are more likely to have them. They also found that growth of the ecosystem comes from user-submitted packages, and it takes a longer time to build a community around user-submitted packages than around core contributed packages. Another analysis of the R ecosystem [49] studies on dependency resolution in R packages finds that lack of dependency constraints in package descriptions and backward incompatible changes often break dependencies. As community contributed packages are hosted on GitHub, there is no way to resolve dependencies among GitHub packages, and therefore a small number of GitHub packages cannot be automatically installed.

Bogart et al. [25] interviewed seven maintainers of R and `npm` packages to understand how dependencies are maintained. They found that developers are not aware of the stability of packages in the ecosystems and make changes on ad-hoc principles. In a follow-up work [26], they found that `npm`, `CRAN` and Eclipse ecosystems differ substantially in their practices about resolving API breaking conflicts and expectations toward change.

2.3.2. Dependency management

It is common for projects to have external dependencies. The dependencies need to be maintained by upgrading them if a new version becomes available or a critical bug issue is fixed. Each project and ecosystem might have its own preferred process for dealing with dependency updates. A study of the dependency management process in Apache projects [15] found that if the number of projects in the ecosystem grows linearly, the dependencies among them grow exponentially. Bavota et al. [14] also found that new releases often do not contain updates to their dependencies. Dependencies are updated only if major new features or bug fixes are released for the dependencies. Kula et al. [90] measured latency to adopt new versions among a sample of Java projects that use Maven. They concluded that over time the maintainers become more trusting and update faster, although no reason is known for this behavior. Cox et al. [41] measured dependency freshness in 75 different closed-source projects of 30 different vendors. Their findings indicate that projects with low dependency freshness are more than four times likelier to include a security vulnerability.

Besides programming language ecosystems, previous research has focused on the Debian package ecosystem, how to resolve strong dependencies in it, and how to improve the planning of dependency changes [1,37,50,51].

2.3.3. Vulnerabilities

Dependency networks enable propagation of bugs and security vulnerabilities. If a package contains a security vulnerability, projects and packages that depend on it can become susceptible when they execute vulnerable code branching. At the same time, there might be projects that do not depend directly on the vulnerable package, but inherit the vulnerability through a chain of dependencies.

Hejderup [71] studied vulnerability spreading across `npm` packages. He used information about known vulnerabilities, tracked how long it takes for projects to update from a vulnerable version and showed that vulnerabilities can affect projects through dependencies. He also observed that some projects have a discussion in the issue tracker about vulnerable dependencies that need updating. Through qualitative analysis, he found that developers were not aware of the vulnerabilities and the risk of breaking functionality is what holds back blindly updating vulnerabilities.

Cadariu et al. [33] proposed a tool to track known vulnerabilities in Java projects. They conducted a case study on private Dutch enterprise projects and found that 54 out of 75 projects use at least 1 (and up to 7) vulnerable dependency.

2.3.4. Synthesis of related work for dependency management

The research questions RQ3.1, RQ3.2, and RQ3.3 proposed in this thesis have received attention in the context of existing research. There are similarities within the existing research, but none of them fully covers the scope and problem studied in this thesis. Wittern et al. [136] and Decan et al. looked at the network topologies for `npm` (JavaScript), `PyPI` (Python) and `CRAN` (R). Compared to [136], our work considers the network analysis in more detail and includes applications in the network analysis step. Compared to [47, 48], we also focus on the network evolution and outline a more accurate dependency network model. Hejdreup [71] studied vulnerability spreading among `npm` projects. Our work analyses the whole ecosystem and includes evolution analysis to study if such vulnerabilities will become less or more likely over time.

2.4. Summary

In this chapter, we introduced the related context for our thesis, reviewed the state-of-the-art technologies, and covered related problems studied in the thesis. We also identified the shortcomings of previous work that guided the research in this thesis.

3. BACKGROUND

To analyze software repository data, we have applied different data analysis methods. For issue lifetime analysis, we used statistical machine learning to predict whether an issue will be closed within a predefined time period. To model dependency relationships in software projects, we applied graph theory and network analysis. In this chapter, we give a brief overview of the data analysis methods used in the thesis.

3.1. Machine learning and classification

Computers need to be explicitly programmed to perform a task by following a sequence of predefined operations, also known as an algorithm. For some complex problems, it may be hard to come up with the algorithm or there may be no known algorithm for the task. In this scenario, we can use some existing data to learn the algorithm [4]. Machine learning enables learning from data and the outcome of the learning process, a model, can be used as an algorithm or incorporated into the algorithm.

Consider an example of issue lifetime prediction. We have collected a dataset of issues from the GitHub issue tracker and now want to understand whether issue textual contents can be used to estimate how long the issue will be open. We count all the unique words in the dataset and assign them unique numbers starting from zero. We can represent an i-th issue in the dataset as a vector $\mathbf{x} = (x_{i1}; x_{i2}; \ldots; x_{im})$ of length m, where the element x_{ij} in the vector denotes that word j is present if it is equal to 1. If the word is missing, the value is 0. This approach is also known as one-hot encoding [106]. For each issue, we also know its lifetime, i.e., the number of days it was open, denoted by y_i. Formally, we can represent the dataset $\mathscr{D}$ as

$$\mathscr{D} = \{(\mathbf{x_i} \in \mathbb{R}^m, y_i \in \mathbb{R})\}$$

where $\mathbf{x}$ is a vector of features and y is a real valued scalar.

To predict issue lifetime, we want to learn a function f that corresponds to $f(\mathbf{x_i}) \approx y_i$. This learning process is called supervised learning as we are using already known y values to learn the mapping. In machine learning, learning is an optimization problem – learning the function f that minimizes some error

$$\min \sum_{i=1}^{n} L(f(\mathbf{x}_i), y_i)$$

for all elements in dataset $\mathscr{D}$, where L is an objective function, such as squared error, $L(y, \hat{y}) = y^2 - \hat{y}^2$ where $\hat{y}$ is the estimated value. The learning problem where the target variable y comes from a continuous distribution is also called as regression. After we have trained the model, a new issue $n+1$ is entered in the issue tracking system. Estimating lifetime for the new issue $\mathbf{x_{n+1}}$ for which we do not know the lifetime is now straightforward by using the learned approximation and f and calculating

$$\hat{y}_{n+1} = f(\mathbf{x_{n+1}})$$

Regression is used to predict some continuous value. In many other settings, we are interested in dividing observations into categories and predicting for an observation which category it belongs to, i.e., y comes from a discrete distribution. This act is known as classification, or learning the class labels based on feature vectors. Lets assume we are interested in predicting whether an issue report gets closed within one month of being reported or not. We have two classes, a positive class, denoted by 1 which represents the issues that get closed, and a negative class, which represents the issues that do not get closed in one month, denoted by -1. For the classification problem, the dataset can be represented as

$$\mathscr{D} = \{(\mathbf{x_i} \in \mathbb{R}^n, y_i \in \{1, -1\})\}$$

Similarly to regression, we define a loss function for classification $L(y, \hat{y}) = max(0, 1 - y \cdot \hat{y})$, also known as Hinge loss [4]. The Hinge loss value is zero if class labels are equal, and one, if they are different.

3.1.1. Logistic regression

Logistic regression is a linear classifier [40, 70] that can learn binary class assignments with class probabilities. For binary classification, we can estimate the probability of the positive class as a conditional probability of features $P(y = 1|\mathbf{x_i})$. Formulating the prediction as linear regression would give us

$$p(X) = \beta_0 + \beta_1 x_1 + \cdots + \beta_m x_m$$

where $p(x) = P(y = 1|\mathbf{x_i})$ and β_j denotes the learned model parameters. However, such a model would generate probabilities in the whole real value range as it is not bounded. To estimate probabilities in the range of $[0, 1]$, log-odds (logit) transformation is used

$$log(\frac{p(X)}{1 - p(X)}) = \beta_0 + \beta_1 x_1 + \cdots + \beta_m x_m$$

Transforming with exponential function gives us

$$p(x) = \frac{e^{\beta_0 + \beta_1 x_1 + \cdots + \beta_m x_m}}{1 + e^{\beta_0 + \beta_1 x_1 + \cdots + \beta_m x_m}}$$

which is the logistic regression. As we have two classes, the final class labels can be derived by setting a threshold, e.g., if $p(x) > 0.5$ then the class label is positive, and otherwise it is negative.

Several approaches are available for training the logistic regression model such as using gradient descent or stochastic gradient descent to maximize the likelihood function [27, 129]. The stochastic gradient descent (SGD) uses small batches of data to update the gradients in an on-line fashion. The advantages of using SGD is that it is fast and scalable to a large number of training samples and features, making it suitable for text classification with large number examples. In this thesis, we use logistic regression with stochastic gradient descent training.

3.1.2. Decision trees

A decision tree can be regarded as a simple rule-based classifier. The classifier uses a learned tree where each node in the tree corresponds to a rule that follows conditional branching known from programming languages such as *if-then-else*. Following the tree down from the root node, answering the questions, and following the corresponding branches, the path will end at the leaf node that gives the prediction value.

There are several algorithms for learning the tree from the data, such as CART [28] or C4.5 [114]. We describe the CART algorithm here only as the Random Forests algorithm used in thesis relies on the CART algorithm internally. The CART algorithm starts to build a tree by selecting a feature from a dataset as a root node. It considers all variables and chooses the best variable and split condition. The split divides the dataset into different subsets based on some condition on the selected variable. Gini impurity determines the suitability of the split for CART. Gini impurity measures the error rate if one of the class labels from the selected subset is randomly applied to one of the observations in the subset [119]. By considering all variables and splits, the final tree minimizes the classification error rate. The algorithm continues to split nodes recursively until no improvement can be made. Decision tree outputs a tree that can be visualized and interpreted by a person.

3.1.3. Random Forests

The Random Forests [29] classifier trains multiple decision trees on the same data, but for each tree uses different random subsets of the data and random subsets of features when creating splits for individual trees. The final predictions are created from finding the frequency of classes output by each individual tree. The randomization combined with multiple trees helps to avoid over-fitting. Random Forests have shown very good performance on different datasets, even when compared to other well-known methods such as logistic regression, support vector machines or gradient-boosted decision trees [54].

The Random Forests classifier lends itself to measuring individual feature importance via *mean decrease in impurity* [28]. For each feature, this method calculates how much it decreases the Gini impurity of a node in a tree and averages this quantity across all trees in the forest. This method enables us to rank the important features in a model.

3.2. Predictive model evaluation and selection

There are many algorithms available that can be used for classification. In addition to picking a suitable classifier for a task, these algorithms themselves have parameters that need to be decided. For example, decision tree height can be limited to keep a model simple, the Random Forests has a parameter for the number of trees to train, and the SGD algorithm needs to have number of iterations specified. When picking an algorithm and parameter set combination, we need a way to estimate the suitability of the selected combination.

The overall training process consists of multiple steps. First, we have our initial labeled data that we divide into two parts – the training set and the test set. We use the training set only for training, and the test data for evaluating the predictions obtained from the model. The test set can be used for evaluation as it has the original labels. Such evaluation should give an approximation of how good the model is when predicting data for which labels are not known. Next, we describe some of the model performance measures that are used to evaluate suitability, and approaches for splitting up the training and test data.

3.2.1. Model performance measures

Let's consider the issue lifetime classification example presented in Section 3.1, where the positive class denotes issues that will be closed in a specified period and the negative class denotes issues that will not be closed in a specified period. Typically the positive class is denoted with 1 and the negative class with -1 or 0. For the classification setting, we have the predicted class $\hat{y}_i$ and the output from the classifier. For evaluation purposes, we know the actual class label y_i. Next, we define the concepts needed to measure the suitability and correctness of the classification.

- TRUE POSITIVE (TP) – The predicted class is P, the actual class is P
- FALSE POSITIVE (TP) – The predicted class is P, the actual class is N
- FALSE NEGATIVE (FN) – The predicted class is N, the actual class is P
- TRUE NEGATIVE (TN) – The predicted class is N, the actual class is N

Next, we define precision and recall:

- PRECISION $= \dfrac{TP}{TP+FP}$
- RECALL $= \dfrac{TP}{TP+FN}$

Precision measures the fraction of correctly classified positive class items over all items predicted to have positive class. Recall measures the fraction of correctly classified positive class items over all positive class items. The ideal precision and recall scores are 1, while the worst case is 0.

The F1-measure is the harmonic mean of precision and recall, defined as $F_1 = 2 \cdot (precision \cdot recall)/(precision + recall)$. The F1-measure allows us to quantify precision and recall with a single number.

Many classifiers give output in the form of a score or a probability of the prediction being positive class, ranging between 0 and 1. To get the actual class labels for positive and negative classes, we need to pick a threshold from the same range. A threshold value of t would assign negative class label to all predictions with a probability less than t and all others will get the positive class labels. Varying the threshold can give different outputs regarding the true positive and the false positive rates.

The receiver operating characteristic (ROC) plot displays the true positive rate ($\frac{TP}{TP+FN}$, recall) and the false positive rate ($\frac{FP}{FP+TN}$) change with changing the threshold value. Analyzing the ROC curve can help one find the optimal threshold in terms of desired error rates. The area under curve (AUC) measures the area under the ROC curve, describing the ROC plot in a single number. The AUC measure helps to compare classifiers over all the possible threshold values.

In other terms, the AUC measures the probability that a classifier will rank a randomly chosen positive instance higher than a randomly chosen negative instance. In the context of this work, AUC measures what the probability is with which we can rank a randomly chosen closed issue higher than a randomly chosen issue that will not close. For a random classifier, the value of AUC will be 0.5, for an ideal classifier it will be 1. An AUC score of less than 0.5 could indicate that the classification problem has not been designed properly or the classifier picks labels from the opposite class.

Different metrics have different purposes. The precision and the recall measure the quality of the classifier if we are interested in predicting the class labels. The AUC, on the other hand, conveys the quality of the classification if the threshold can be varied or we are interested in finding the most probable positive class elements.

3.2.2. Model validation and selection

The main goal when training a model is to obtain good predictive power and prevent over-fitting. Over-fitting happens when the model can classify with a good performance on the training data and might be unable to generalize on a dataset that was not used for training. To minimize such risk and get a good approximation of the performance on future unseen data, specific procedures need to be followed. The labeled dataset is split into three parts: a training set, a validation set, and a test set. The training set is used to train the model. The validation set is used to tune the hyper-parameters of the model or select the model. The test set is only used for the final evaluation. Calculating performance measures on the test set gives an approximation of what the model performance would be for unseen data.

Splitting the data into different subsets is typically done randomly, so that 80% of the initial data is used for training and the remaining 20% is used for the test set. The training set can again be divided into actual training and the validation set. The amount of labeled data decreases with the splitting and to reuse the labeled data, cross-validation can be used. Cross-validation is the process where training data is split into multiple subsets. For example, if it is split into ten subsets, for every parameter combination that needs to be evaluated, the model is trained on the nine subsets, and evaluated on the remaining subset. The final performance can be obtained by averaging over all the folds. This enables reusing all the training data, but never actually evaluating the data that was set aside for testing.

Random partitioning sometimes is not enough when the dataset contains temporal information. Consider a dataset of issue reports, where reports originate from different time periods between 2014 and 2015. Let's assume that the project introduced a new release of a specific platform in 2015 and subsequently there were a lot of bug reports about that release. Randomly splitting the data might divide some issue reports created in 2015 into both the training and test sets. Leaking future data into the training set could make the model perform better as both training and test data contained training examples from the same period. However, if we want to apply such an approach in practice, we cannot use features from future issue reports.

Temporal-splitting of the dataset divides training and test so that all test set data points are temporally placed strictly after training set data points. To extract the validation set from the training data, the same temporal ordering criteria must be used. The downside of this approach is that it prevents the usage of cross-validation and reduces the amount of data for evaluation.

Model selection and validation is part of any statistical modeling process. Model selection has also received attention in the context of software engineering tasks. Tantithamthavorn et al. found that hyper-parameter tuning can cause up to a 40% improvement in AUC for defect prediction [122]. Kocaguneli and Menzies [88] recommend to use cross-validation (and especially leave-one-out validation) for software effort estimation modeling to obtain models with the lowest bias and variance.

3.3. Network analysis

Some datasets are not easily represented by tabular data, for example, if we need to model relationships between entities. In relational settings, we are not only interested in linked entities, but also the connections that are formed by following multiple links. When a software project includes a library, a dependency relationship is formed. Similarly, the dependency itself can have dependencies, leading to a connected system of software packages. How do we analyze and model such a system to understand its properties? In this thesis, we use the concept of dependency networks. Networks are composed of nodes and connected edges and have some additional meta-data, such as node or edge attributes.

Networks can be modeled with graphs. A Graph G is defined as an ordered pair of $G = (V, E)$, where V is a set of nodes (or objects) and E is the set of edges (relationships) formed between the set of nodes. In the software dependency settings, the nodes represent software packages and the edges between them denote if one package depends on another. In our setting, the edges are directed, i.e., (v_1, v_2) is a directed edge from v_1 to v_2 but there is no edge the other way around, indicating that the relationship is only valid in one direction. The edges can also be undirected. Edges can be given weights or additional attributes to denote the properties of the relationships being modeled.

Modeling a network with graphs enables us to study its properties and compare it with other networks.

3.3.1. Paths and components

In a graph, we say that two nodes are connected if there is an edge between them or there exists a path between them. A path between two nodes is a sequence of edges that connect them and are distinct. In a directed graph, all the edges of a path have to be in the same direction.

A component in the graph is a maximal set of nodes where there exists a path between each pair of nodes in the component. A subgraph of a graph G is a graph composed only of subset of G vertices and edges.

A weakly connected component in a directed graph is a subgraph where each node is connected with every other node in the subgraph via an undirected path. Similarly, a strongly connected component is a subgraph where there exists a directed path between every pair of nodes.

3.3.2. Centrality

A typical question in network analysis is which node is important in a given network. The importance can be interpreted and defined in many ways, but typically it quantifies the connectedness of the node or how many paths pass through the node. Node (or edge) centrality is a measure of the importance of the node (edge) in the network. Some of the commonly used node centrality measures are degree centrality, betweenness centrality, and closeness centrality [107]. The degree centrality measures the number of edges connected to a node. The betweenness centrality measures the number of shortest paths in the graph that pass through that node. The closeness centrality measures the average length of the shortest path from the corresponding node to any other node in the network. All these measures express node connectedness in the graph. Alternatively, we can say that if a node with high centrality is removed from the graph, it will break many paths and connections.

4. UNDERSTANDING ISSUE DYNAMICS IN GITHUB PROJECTS

4.1. Introduction

This chapter studies the extent to which open-source projects cope with the inflow of issues they are subjected to throughout their lifetime. Based on a sample of more than 4,000 GitHub projects, we analyze the temporal dynamics of issues regarding how often they are created (arrival rate), the number of pending issues, and their lifetime. Specifically, we address the following research questions:

- RQ1.1: What is the issue-arrival rate and how does it change over time?
- RQ1.2: How do opened and pending issue numbers evolve over time?
- RQ1.3: What is the average issue lifetime and how does it change over time?

4.2. Dataset and method

The dataset for issue lifetime analysis is extracted from GHTorrent [60]. At the time of the data extraction (April 2, 2015), GitHub had more than 7 million project repositories (not counting forked ones). Not all repositories in GitHub are software projects [81], and many of them use GitHub for code hosting but not for issue tracking.

4.2.1. Filtering

In order to avoid analyzing non-software projects (e.g., pure documentation projects), projects that do not use GitHub for issue tracking, and other special cases such as one-man projects or projects with little issue activity, we filtered the dataset using the following rules:

- Projects must have been created between January 1, 2012 and December 31, 2014. We limited our observation period to this interval, because the data of older projects is only partially available in GHTorrent. Even though the dataset also contains events up to April 2015, we chose 2014 as the ending date due to the delayed crawling behavior of GHTorrent in which not all changes are instantly visible.
- Projects must not be forks of existing GitHub projects. The pull-based contribution mechanism encourages forking repositories only for the purpose of committing a change and then opening a pull request to the base repository. In these settings, the base repository's issue tracker is used as the main issue tracker. However, forks are sometimes made for other purposes as well, such as when the development has stopped in the base repository and is continued by a new team in the forked repository. In these cases, the issue tracker activity might also be present in the forked repository.

- Projects must have at least 100 opened issues and one closed issue. This criterion guarantees that we only include projects that actively use the issue tracker. The idea of setting this criteria is that with 100 issues it is unlikely that somebody is using the repository as a personal project or testing GitHub capabilities.
- Projects must have at least five commits to the main repository. This criterion guarantees that we only analyze projects where there is some development activity. Although five might seem like a small number of commits, developers can use commit squashing when merging into the master branch, hence five commits can include multiple features etc. Dabbish et al. [45] discovered through interviewing GitHub users that developers use commits as a measure for deciding whether a project is active or not.
- Projects must not show any activity before the repository creation date. In GitHub, it is possible to fork a repository and therefore inherit an already existing code base which technically shows up as code committed before the project creation.

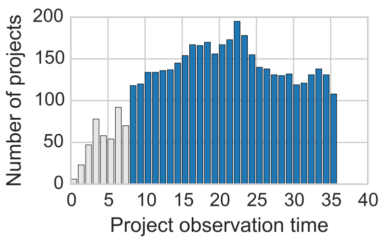
An examination of the selected data revealed that some projects had unexpectedly high issue-creation activity over short periods, such as several thousand issues created in a single day. This phenomenon indicates a data import from an older tracking system or the automatic creation of issues via GitHub's API. To get rid of possible import behavior, we additionally filtered out projects that created or closed more than 2,000 issues in any single month or created more than 500 issues in any single day.

As the data ranged between 2012 and 2014, the selection includes projects with a maximum of 3 years of history and a minimum of 1 month of history. We decided to remove projects shorter than 8 months to have enough time to observe issue closing in every retained project.

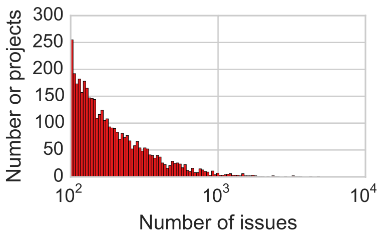
Issues can also be reopened and closed multiple times. This affects about 4% of the issues in our sample. We decided to remove issues that were reopened and focus on the "first-closing time" as it is the most common behavior. The phenomenon of issue reopening is a question that deserves separate treatment. Note also that an issue being closed in the dataset does not necessarily imply that it has been "fixed" to the satisfaction of the issue creator. An issue may be closed for a variety of reasons, such as it being a duplicate issue or because someone in the project team deems it irrelevant or unresolvable.

4.2.2. Descriptive statistics

After filtering, 4,452 projects met our criteria. Figure 2a shows the distribution of the projects' observation time lengths. Our sample contains projects with observation times ranging from 0 to 35 months. We observed that there were relatively few projects with a short observation time. We wanted to have at least 100 projects within each observation time interval, in order to have samples with comparable



(a) Number of projects per project observation bucket (in months). Dark colored bars represent the sample used in the analysis.



(b) Number of projects per number of opened issues bucket (logarithmic scale).

Figure 2. Basic properties of the dataset.

sizes. This resulted in the removal of all projects with observation less than eight months, i.e., projects created after April 2014. In Figure 2a, projects that were filtered out due to their observation time are marked in gray.

The dataset obtained after the above filtering contains 4,024 projects, comprising 967,037 issues in total of which 675,970 (69.9%) were closed and 291,067 (30.1%) were not closed during the observation period.

The number of issues per project (Figure 2b) varies by a factor of almost 50, the smallest project having 100 issues and the largest project having 4,885 issues in total. The mean number of issues per project is 240 and the median is 163.

4.2.3. Terminology

The centerpiece of our analysis is issues, i.e., bug reports, new feature requests, and development-related changes such as refactoring. In GitHub, issues are typically free text, can be submitted by anyone, support commenting, and can be referenced from other issues. The collected dataset records when an issue was created and by whom, and a set of events associated such as closing, reopening, being commented, or being referenced from another issue. All these events are marked with the time of the action and which user is responsible for it.

We distinguish the following issue states:

- **Opened issue** – Newly created issue. Each issue is opened only once during its lifetime.
- **Pending issue** – Issue that has been opened but not yet closed. These issues denote unresolved cases that need attention or actual work.
- **Sticky issue** – Issue that did not get closed during our observation period. Sticky issues are a subset of pending issues.
- **Closed issue** – Issue that is marked closed in the issue tracking system. In practice an issue might be reopened and closed again, but here we use only the first closing event. We do not distinguish closed issues based on the resolution type, meaning that a closed issue might have been closed after the bug was fixed or closed without any activity.

One might consider our notion of pending issues as too simplistic since we do not take into account re-opening and re-closing. The justification for this is the fact that re-opening and re-closing affects only 4% of all issues.

Our dataset contains projects that were created at different points in time during our observation period and therefore have varying time periods during which we could observe project behavior. Below we list our time-related terminology:

- **Issue lifetime** – Time from the first opening of the issue to the first closing of the issue.
- **Project observation time** – Number of months between project creation and the end of the observation period (December 31, 2014). This number is obtained by calculating the number of days between the two dates, dividing by 30.4 (the average number of days in a month), and rounding down to the nearest integer.
- **Relative time** – Each project is transformed into a relative timescale. The relative timescale starts from repository creation, and after every 30.4 days, a new relative month starts. This results in the final month typically not being a full month, as projects can start on any day during a month, but our observation period ends with the 31st day of a month. Relative time ”zero” represents the first month of the project observation time, relative time month ”one” represents the second month of the project observation time.
- **Observation period** – From January 2012 until end of December 2014. This is the period for while we have data about projects and can use for the analysis.

4.2.4. Notations

In our analysis, we focus on the following metrics over time: opened issues (newly created issues), sticky issues (issues that do not get closed), and pending issues (open issues that will get closed). In the following section, we give the definitions of these metrics.

Let N be the set of projects and $\mathscr{T}$ the set of all possible project observation times. Each project $i \in N$ has an observation time of $T_i \in \mathscr{T}$.

We denote a single issue as a tuple (a_j, b_j) where j is a unique issue identifier, and a_j and b_j denote opening and closing times since project creation (measured in minute resolution). For each issue, it must hold $(a_j \leq b_j) \vee (a_j \leq T_i \wedge b_j = nil)$. Let PI_i denote the set of issues associated with project i. Even though T_i has discrete values, we assume that a_j and b_j are continuous and have minute-level resolution in order to be able to derive exact ordering between closing and opening. Let $m(a_j)$ denote the corresponding relative month of a_j and $m^{-1}(t)$ the value in minutes for the corresponding month end date.

Let $o_{i,t}$ denote the number of total newly opened issues for a project i at a relative time $t \in \mathscr{T}$, then

$$o_{i,t} = |\{(a_j, b_j) | (a_j, b_j) \in PI_i, m(a_j) = t\}|.$$

We use $s_{i,t}$ to denote sticky issues, i.e.,

$$s_{i,t} = |\{(a_j, b_j) | (a_j, b_j) \in PI_i, m(a_j) \leq t \wedge b_j = nil\}|.$$

It represents sticky issues as the total number of sticky issues by the end of month t.

Let $p_{i,t}$ denote the number of pending issues at time t. The number of pending issues is the number of opened – but not yet closed – issues at a certain point of time (measured in minute resolution). Thus, we devise the number of pending issues $p_{i,t}$ for a project i during a month t as follows:

$$p_{i,t} = \frac{1}{m^{-1}(t-1) + 1 - m^{-1}(t)} \sum_{d=m^{-1}(t-1)+1}^{d \leq m^{-1}(t)} \delta_{i,d}$$

where $\delta_{i,d}$ denotes the number of open issues for project i at minute resolution d, i.e.

$$\delta_{i,d} = |\{(a_j, b_j) | (a_j, b_j) \in PI_i, a_j \leq d \wedge (b_j = nil \vee b_j > d)\}|.$$

Finally, issue lifetime for issue j, denoted by LT_j, is the number of days between issue creation and closing and can be calculated only for closed issues, i.e., $b_j \neq nil$:

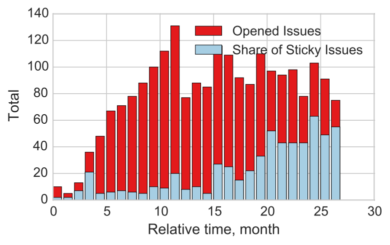
$$LT_j = (b_j - a_j)/(60 * 24).$$

4.2.5. Examples

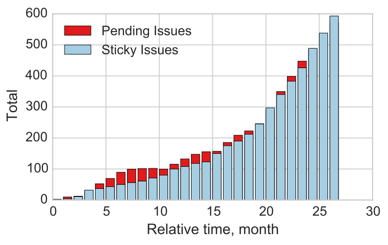
We illustrate our concepts with the help of an example project, Bootstrap[1], a front-end framework for creating user-interfaces in browsers.

Figure 3a shows the numbers of opened and sticky issues observed per month. Note that here we only show the share of sticky issues that correspond to issues opened in the month of observation. Each bar on the plot corresponds to $o_{i,t}$ and $s_{i,t} - s_{i,t-1}$ (except for the case $t = 0$, the sticky issues is equal to $s_{i,t}$). We see that Bootstrap had increasing numbers of opened issues during the first year of observation, then the number of opened issues leveled off. The monthly share of sticky issues started to rise around month 20. One reason for this could be that our observation period limits the available time for observing issues opened after month 20 being closed. Figure 3b plots the number of pending and sticky issues observed over time. We see that the number of pending issues increases and the majority of pending issues is made up by sticky issues. When comparing

[1] `https://github.com/twbs/bootstrap`

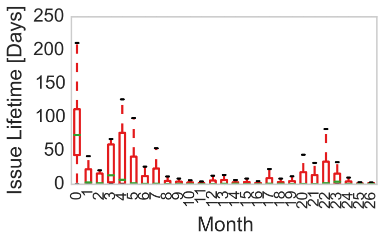


(a) Opened issues for Bootstrap.

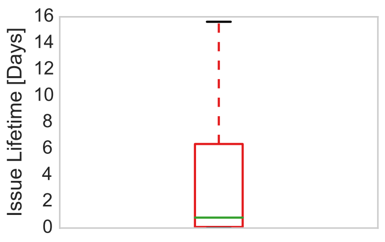


(b) Pending issues for Bootstrap.

Figure 3. Opened, pending, and sticky issues for Bootstrap.



(a) Issue lifetime distributions over time for Bootstrap.



(b) Issue lifetime distribution for Bootstrap (all issues).

Figure 4. Issue lifetime for Bootstrap. For both figures, outliers are removed. The maximums correspond to $1.5*(75p-25p)+75p$, where 25p and 75p denote corresponding percentiles.

the relative share of sticky issues with opened issues per month, we observe that the majority of opened issues get closed, but the amount of work still to be done, represented by the amount of pending issues, is continuously increasing due to the number of sticky issues.

Figure 4a shows issue lifetime distributions as boxplots for groups of issues opened in a specific month of the project observation time (relative time in months). Note that in Figure 4a month 0 is an abbreviation for the time up to the beginning of month 1, i.e., representing the observation time interval (0, 1) months. Issue lifetimes remain stable on average over the project observation time (mean lifetime equals 12.9 days, median lifetime equals 0.78 days). We see, however, that issues created in month 0 have a considerably above-average lifetime than those created in later months. One possible explanation is that during the first month issues are entered that require additional development and this usually takes more time than simply correcting a bug. In this particular example, however, there were only 8 opened issues in month 0. Therefore, this is not an important phenomenon. The overall issue-lifetime distribution, shown in Figure 4b, indicates a small median (0.78 considering all issues) but large variation (standard deviation 33.19, maximum value 351.94, number of total issues 1,566).

4.3. Results

In the following subsections we answer the research questions outlined in the introduction. First, we look at the opened issue rates, then we analyze the pending issues, and finally we analyze the issue lifetime distributions.

4.3.1. Issue arrival rate (RQ1.1)

To answer RQ1.1 we investigate the arrival rates of opened issues in our set of GitHub projects. We analyze projects with different project observation times separately because we suppose that the length of the observation time has an effect on the opened issues. For example, projects with short observation times might (on average) have different numbers of opened issues during the first months of the observation time due to the increased popularity of GitHub and the size and type of projects hosted in GitHub.

We classify projects based on observation times into buckets and calculate the average number of opened issues per month for all projects in a bucket separately. Figure 5 shows a line for each group of projects in the same bucket. There are $\mathscr{T}$ buckets in total and for each line l represents a bucket, while a point t on the line represents the average number of opened issues at relative time since creation of the projects in the bucket (i.e., the start of the project observation time), given by the following formula:

$$\mathscr{O}_{t,l} = \frac{1}{\sum_{i \in N \wedge T_i = l \wedge t \leq l} 1} \sum_{i \in N \wedge T_i = l \wedge t \leq l} o_{i,t}$$

Looking at Figure 5, we observe a relatively higher average number of opened issues right after project creation as compared to a few months after project creation. This tendency is visible for all project buckets but most explicitly for buckets of projects with shorter observation times. Overall, we see that the average number of opened issues is stable or shows a slight negative trend over the project observation time. For the first month, a project in GitHub receives an average of 19.7 opened issues, but one year later, during the 12th month, it receives 10.3 opened issues, on average. The relative decline of opened issues after the first few months might be explained as a start-up effect, i.e., at the beginning many issues are submitted but never worked on because they represent unrealistic features to be included in the project. Furthermore, we observe that projects in buckets with shorter observation times seem to have significantly higher numbers of opened issues than those in buckets with longer observation times. For example, in the first month after project creation, projects in buckets with the shortest observation time have on average more than five times more opened issues than projects with the longest observation time. This might be explained by the relative growth of GitHub and the increasing size of projects in recent times. The drop-offs in the last months are caused by a technical artifact. Namely, our relative time line starts with repository creation, and due to this, the last month

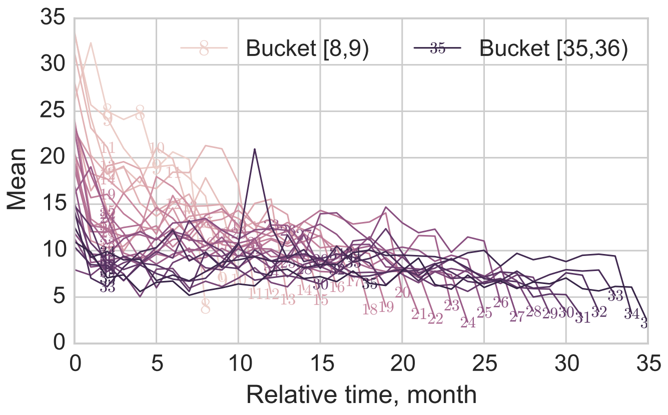


Figure 5. Opened issues for projects with different lifetime. Color intensity varies with observation period length.

is probably not a full month, as the observation period ends at the end of the month. In addition, we observe that for some groups, there are outlier months, such as the outlier at month 12. The outlier is caused by two different projects opening 818 and 694 issues in a single month (repositories `glasklart/hd` and `MrNukealizer/SCII-External-Maphack` respectively). We observe stable average numbers of opened issues for all buckets with relative times after month 28.

To understand what the ratio (or share) of sticky issues to opened issues for projects with different observation times is, we calculated the ratio of sticky to opened issues for different buckets. In Figure 6, we display the ratios for each bucket and display the corresponding lines in four different graphs, each containing a subset of buckets. The purpose is to make patterns between groups of buckets more visible. The ratio of sticky to opened issues varies mostly between 0.2 and 0.6, meaning that still more issues get closed than stay open during the observation time. We observe that the ratio of sticky issues is higher for early months, then levels off, and finally starts to rise due to the technical effect of project observation time ending. The exception is the set of recent projects with short observation times. Compared to other buckets, projects in buckets with observation times 8–14 have fewer sticky issues in the early months followed by a steady growth of the ratio. One possible explanation for this phenomenon might be that issue submitters in recent projects are more realistic in their issue management and do not fill the issue tracker with issues that will never be worked on or are resolved only after a long time. Alternative explanation could be that, projects with longer observation times are not using the repository any more and the development has stalled, causing accumulation of sticky issues. Another possible explanation could be that due to the growth of the projects maintained in GitHub, more development capacity is available to work on issues and thus issues receive more attention and are resolved more quickly. This explanation, however, would

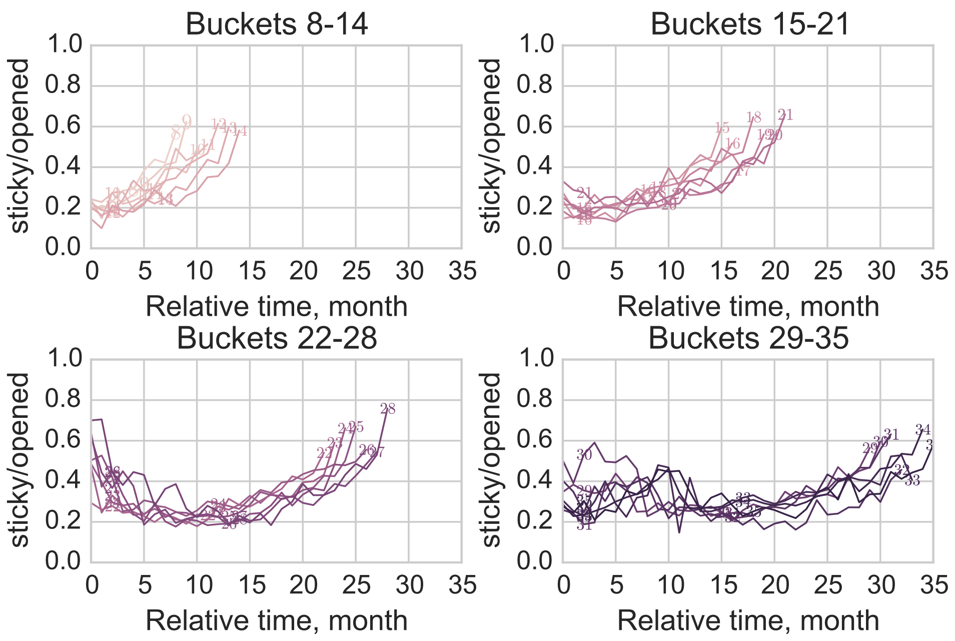


Figure 6. Ratio of (shares of) sticky issues to opened issues.

not explain why the ratio for projects with short observation times strongly grows after a few months.

To answer RQ1.1, we can conclude that the average monthly rate of opened issues for projects in our data set decreases over time. During the first 12 months, the average number of issues opened drops roughly by a factor of two. The ratio of sticky issues to opened issues changes differently over time for projects with a shorter observation time as compared to projects with a longer observation time.

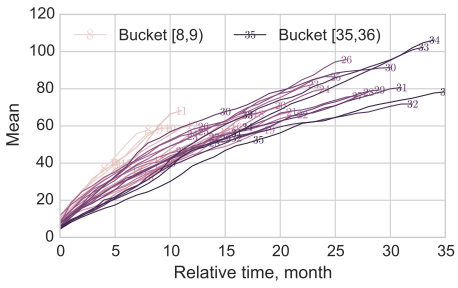
4.3.2. Pending issue growth (RQ1.2)

To answer RQ1.2 we analyzed the dynamics of pending issues and sticky issues over time. Again, we consider projects with different observation times separately. In Figure 7a, displaying the average number of pending issues over time, we classified projects with different observation times into buckets. For each bucket we show the average of pending issues over time as different lines T_i, the values of each line defined as:

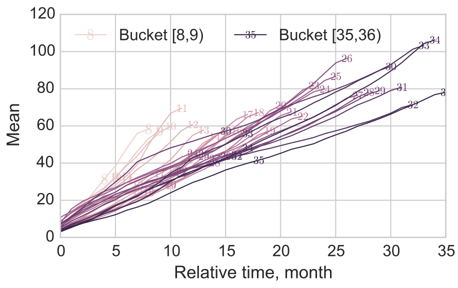
$$P_{t,l} = \frac{1}{\sum_{i \in N \wedge T_i = l \wedge t \leq l} 1} \sum_{i \in N \wedge T_i = l \wedge t \leq l} p_{i,t}$$

where l denotes the project observation time and t is the relative time for projects with observation time l, and thus it must always hold that $t \leq l$. Similarly, in Figure 7b we plot the total number of sticky issues over time for each bucket.

One can see that pending issues are growing at approximately constant rates for all buckets. This phenomenon is underpinned by the growth pattern of sticky



(a) Pending issues.



(b) Sticky issues.

Figure 7. Pending and sticky issues. Even though, sticky issues are a subset of pending issues, we have plotted them on separate Figures for clarity.

issues, issues that have not been resolved within our observation period.

The growth rates for both pending and sticky issues are different between buckets. Generally, there seems to be a tendency that more recent projects (shorter project observation time) have on average higher growth rates in the early months of observation time. To quantify this, we divided projects into two groups. First group was composed of projects with observation time less or equal than 12 months and the other group composed of projects with observation time longer than 12 months. We compare the number of pending issues for both groups at month 10 using a one-sided Mann-Whitney's U test [97]. For the test, the null hypothesis indicates that the distributions of the groups are equal. Alternative hypothesis indicates that values of one group are larger than the other's. The test rejected null hypothesis (p-value $< 10^{-15}$) indicating that the first group has more pending issues at the month 10. Comparing the number of pending issues for the months 8, 9, 11, 12 also yielded in the first group having more pending issues than the second group (p-value $< 10^{-4}$ for all tests). On the other hand, there exists strong differences between growth rates of buckets with just a one month difference of project observation time. For example, projects with an observation period of 34 months have in time interval (34, 35) 25 percent more pending (and sticky) issues than projects with 35 months of observation period in the same time interval.

As an answer to RQ1.2, we can say that pending issues are growing constantly and this comes mostly from the sticky issues that do not get resolved during our observation period.

4.3.3. Issue lifetime (RQ1.3)

So far, we have observed that there is a steady arrival of new issues and an increasing number of pending (and sticky) issues. Although the number of pending issues grew, most opened issues actually got resolved (closed) during the project observation time. In this section, we answer RQ1.3 by analyzing issue lifetimes in Github projects.

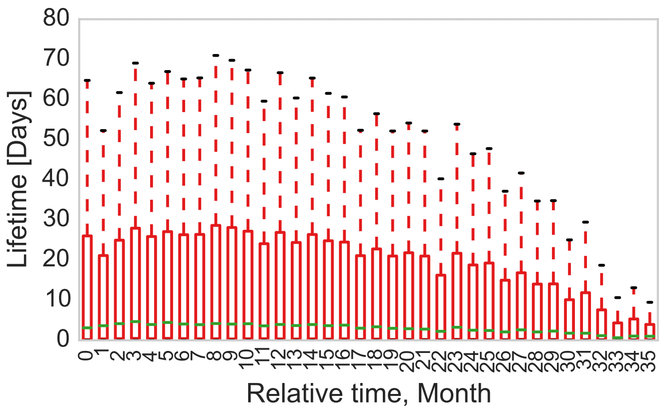


Figure 8. Issue lifetime distribution depending on issue creation month, each month t lists distribution lifetime of ILT_t. Outliers have been removed.

Figure 8 shows issue lifetime distributions for issues opened during each month (relative time from the start of project observation time) of all projects in our data set, i.e., we do not classify data into buckets with the same project observation times. We consider all issues that are not sticky. Each boxplot represents issue lifetimes for the corresponding ILT_t group, defined as $ILT_t = \{LT_j|(a_j,b_j) \in PI_i, i \in N, b_j \neq nil, m(a_j) = t\}$.

In Figure 8 one sees variation in maximum values but medians vary little over time. The median issue lifetime is 3.1 days for issues opened in month 0 (time interval [0, 1]), 4.1 days for issues opened in month 10 (time interval [9, 10]), 2.89 for issues opened in month 20 (time interval [19, 20]), and 1.78 for issues opened in month 30 (time interval [29, 30]). Thus, one can observe a slight increase from month zero until month 10 in the median value, and then a slight decrease. Around month 20, the median starts to drop more, but this might be a technical effect, caused by the fact that issues opened towards the end of our observation, in order to be included in the lifetime measurement, must have been closed before the end of the observation period, thus, leaving out all issues that might be closed after a longer lifetime.

Figure 9 shows distributions of issue lifetimes for groups of projects with the same project observation times l. Each boxplot represents the issue lifetime distribution for the set ILL_l, defined as $ILL_l = \{LT_j|(a_j,b_j) \in PI_i, i \in N, b_j \neq nil, T_i = l\}$. The observation that variation of issue lifetimes is growing for projects with longer observation time is not surprising, as in projects with longer observation time there is more time available to solve an issue. On the other hand, the median values seem to be stable. The median for projects with an observation time of 8 months is 2.64 days, for projects with an observation time of 20 months, it is 3.4 days, and for projects with an observation time of 30 months, it is 3.61 days. The variation over all projects is small, considering that the theoretical maximum difference can be more than four times for projects with an observation time of 8

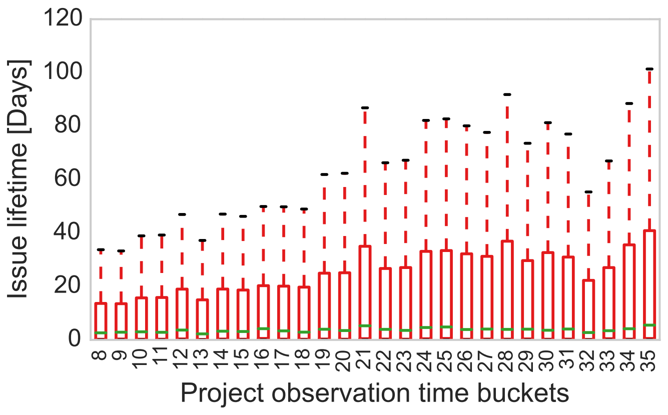


Figure 9. Issue lifetime distribution depending on the project lifetime, each lifetime bucket l lists distribution of lifetime for issues in ILL_l.

months and projects with an observation time of 35 months.

As in the previous section, we also analyzed different project buckets separately. In Figure 10, we show median issue lifetimes for each group of projects with different observation times. Formally, there are again T lines in total and for each line l, we calculate a median for a set of issues $MLT_{l,t}$, defined as $MLT_{l,t} = \{LT_j | (a_j, b_j) \in PI_i, i \in N, b_j \neq nil, T_i = l, m(a_j) = t\}$. We observe that a few outliers distort the big picture. Ignoring those outliers, we observe that medians are stable. The drop in the last months of the observation times is due, to some extent, to the fact that issues requiring a longer time for closing are excluded from the analysis. Interestingly, median values for projects in different buckets are very similar, which is especially true for projects with observation times between 15 and 35 months. To answer RQ1.3, we can conclude that issue lifetimes are stable over project observation times.

4.3.4. Discussion

Our results partly confirm and partly contribute to published research. For example, our results related to RQ1.1 confirm the results of Kenmei et al. [82] who studied trends in newly opened issues. They found that for some systems (e.g., JBoss) there is an increasing trend and for others (e.g., Mozilla) there is not. Similarly, our results, which are averages over large numbers of projects, do not show a trend of increasing numbers of opened issues over time for projects with comparable observation times. However, we found that more recent projects, i.e., projects with a shorter observation time in our study period, tend to have generally higher volumes of opened issues than older projects, i.e., projects with longer observation times. Furthermore, more recent projects seem to have a decreasing trend in numbers of opened issues, while older projects show a more stable behavior.

Garousi [56] analyzed three open-source projects and found evidence of increasing work and short issue lifetimes. He showed that for jEdit and DrPython

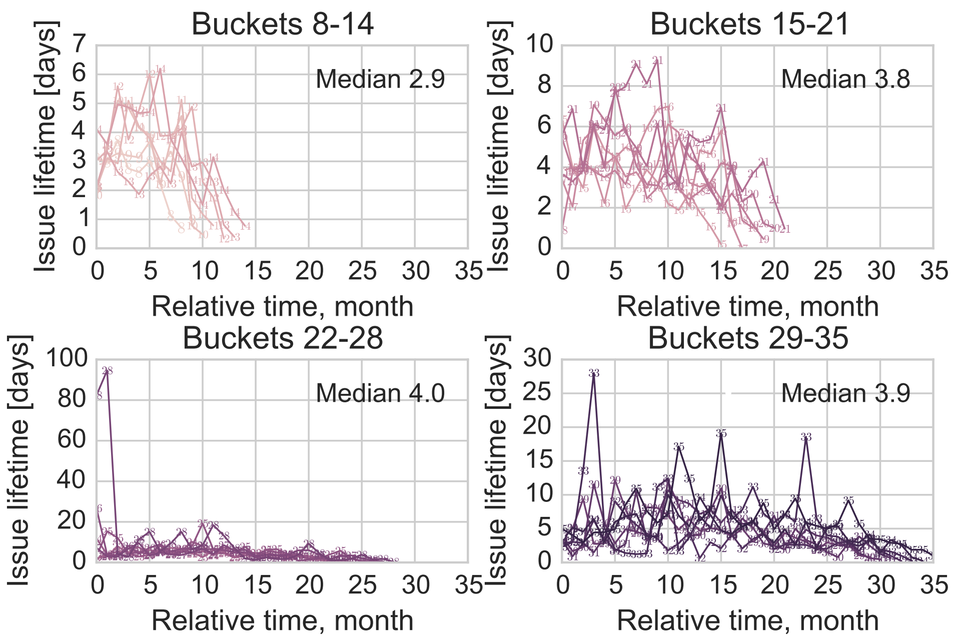


Figure 10. Median issue lifetimes. For each group, we have calculated the median issue lifetimes over all issues over all months in that group.

projects, the fraction of issues that are closed within the first day is 23% and 42% percent, respectively. Related to RQ1.2, we found that for Github projects, the median of issue lifetimes varies between 2 and 4 days. Thus more than 50% of issues get closed after 4 days at the latest , and in many cases earlier. Thus, our results are closer to those of Garousi [56] than to those of Marks et al. [99] who found that for Mozilla 46% of the bugs are closed after three months of bug creation, while for Eclipse 76% are closed. The findings of Grammel et al. [63] regarding closed-source IBM Jazz projects suggest that community-created issues can be valuable, but they are handled differently than those created by project members. The average issue lifetime for community-created issues is 39 days, but for the team issues it is 5.9 days. These results are comparable to the 12.9 average issue lifetime we observed for the Bootstrap project. However, we also saw that median values are much smaller (e.g., 0.78 days for Bootstrap) and issue lifetime distributions are extremely long-tailed. Thus, reporting mean values of issue lifetime might not be useful.

Lujiten et al. [95] studied issue report handling in GNOME projects, focusing on bug reports (software defects). They found evidence of more new bug reports being created than resolved over time, resulting in the growing backlog of issues. They developed Issue Churn View, a method for visualizing issue backlog contents separated into groups of issues based on the lifetime. The visualization revealed that majority of the backlog is composed of issues that have been open for more than 26 weeks and the absolute numbers for long-living issues is growing. Yet, at the same time new issues are being opened and resolved. Our findings align with theirs, confirming the increasing number of sticky and pending issues.

Our results regarding the trend of increasing pending and sticky issues over time (RQ1.2) seems to be related to the observation of highly positively skewed distributions of issue lifetimes.

4.3.5. Design implications

The exploratory analysis of issue lifetime reveals many potential areas for improvement in issue management and issue tracking systems in open-source projects.

The growing number of pending issues indicates that projects should make fine-grained statistics about issue resolution visible, such as the average issue resolution time and number of issues closed in recent days. This helps potential users to get an overview of how active the project is and incorporate resolution time into their project selection criteria, if they are considering adopting a project.

Another aspect would be to evaluate automated issue triaging, severity estimation methods to rank important issues higher. Even though several methods have been proposed [58, 65, 125, 143], there is still a lack of empirical evidence about how these methods should be applied in practice. Even automatically closing issues after some period of inactivity could bring clarity and transparency, as we observed issues are left open for long periods and those that get closed, get resolved relatively fast. If an issue is still relevant after closing, the stakeholders can automatically reopen it. Today there exists a bot on GitHub [113] that can be configured to automatically close stale issues after some period of inactivity. Searching for the phrase *"This issue has been automatically marked as stale because"* on GitHub in October 2018 revealed 49,865 issues, of which 7,741 are open and 42,124 closed. This illustrates that there is a need for such functionality.

4.4. Threats to validity

In this section we briefly discuss threats to validity that may affect our results. *Construct validity* threats concern the relationship between theory and observation. In our study, these threats can be mainly due to the way we measure the various types of opened, closed, pending, and sticky issues as well as to the quality of the data extracted from GitHub, and also due to the fact that we neither distinguish between types and sizes of projects nor issue categories like bug fixes, enhancements, refactoring, and so on. We tried to address the issue of data quality by defining exclusion criteria that filter out projects with certain data anomalies, e.g., low activity, small size, and issue imports due to changes in the issue tracker system used.

External validity concerns the generalization of the findings. Different from most of the studies presented in related work, our results rely on the analysis of more than 4,000 GitHub projects over a time period of three years. We believe that our results are to some degree representative of open-source projects in general. However, we noticed that there is some variation between projects depending on the length of the observation time. Also, we do not distinguish between types of projects and application domains. Finally, we would like to point out that closed-source projects might have different issue behaviors due to the more controlled environment in which these projects are conducted.

4.5. Summary

Issue trackers are important for software projects to manage bugs and as a general task list indicating development actions needing to be done.

We analyzed the issue dynamics of more than 4,000 GitHub projects. Understanding issue volumes and issue lifetimes can be a source for understanding project performance and planning project resources. Once typical evolution patterns for issues are better understood, they might become an indicator of the state of a project and its future outlook.

The primary finding about the increasing number of pending issues over project lifetime indicates the need for better issue management tooling to prevent issue creep up in open-source projects.

5. PREDICTING ISSUE LIFETIME IN GITHUB PROJECTS

5.1. Introduction

In the previous chapter, we observed that GitHub repositories would have pending issues accumulate over time. In this chapter, we build a model to predict whether an issue will be closed during a specified time frame.

Within the scope of the problem of issue lifetime prediction, this chapter seeks to answer the following research questions:

- RQ2.1: What level of accuracy is achieved by classification models trained to predict issue lifetime at different calendric time points in an issue's lifetime and for different calendric periods (one day, week, one month, one quarter, one semester, and one year) using both static and dynamic features of an issue as well as contextual features?
- RQ2.2: What features are most important when predicting issue lifetime?

5.2. Approach

In this setting, we address the problem of predicting, at a given time point during an issue's lifetime, whether or not the issue in question will close after a given time horizon, e.g. predicting whether an issue that has been open for one week will remain open one month after its creation. The general problem of issue (or bug) lifetime prediction has received significant attention in the research literature. The focus of this study differs from previous work in four respects. First, the bulk of previous work has focused on analyzing a small number of hand-picked projects. In contrast, we study the prediction problem based on a large sample of projects hosted in GitHub. Second, most previous work has focused on exploiting static features, i.e. characteristics extracted for a given snapshot of an issue – typically issue creation time. In contrast, the present study combines static features available at issue creation time with dynamic features, i.e. features that evolve throughout an issue's lifetime. Third, previous approaches focus on predicting lifetime based on characteristics of the issue itself. In contrast, the present study combines characteristics of the issue itself with contextual information, such as the overall state of the project or recent development activity in the project. Finally, most previous studies do not employ temporal splits to construct prediction models. In other words, models are trained on future data and then evaluated on past data. In this study, we construct models predictively using strict temporal splits such that predictions are always made based only on past data, which reflects how such predictive models would be used in practice.

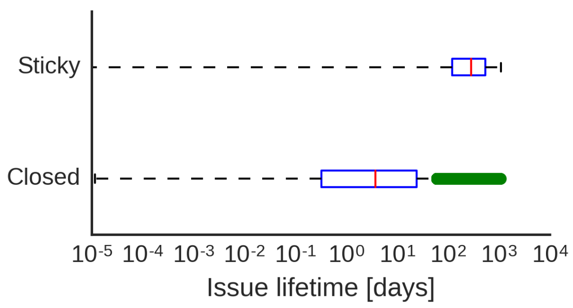


Figure 11. Issue lifetime box-plots for closed and sticky issues. The green-filled line represent outliers not falling into the inter-quantile range.

5.3. Dataset

The dataset in this study is the same as in Chapter 4 and follows the same dataset extraction process as described in Section 4.2.1. In addition, we used GHTorrent's MongoDB service [60] to query the issue title and body for all the issues (queries issued in January 2016).

5.3.1. Analysis of issue lifetime

Figure 11 shows the issue lifetime distribution for the 69.9% of issues that get closed in the observation period (bottom box-plot) and for the set of remaining "sticky issues" (top box-plot). We use the term *sticky issue* to refer to issues that do not get closed in our observation period. For the sticky issues, the lifetime is calculated with the assumption they were all closed on January 1, 2015 (recall that we only retained issues created in 2014 or before).

The median lifetime for the closed issues is 3.7 days, the mean lifetime is 32.6 days, and 90% of issues get closed in 96.4 days or less. We observe that the median lifetime for sticky issues is 280 days, which is approximately 75 times longer than for closed issues. This long lifetime gives us confidence that most of the sticky issues are indeed long-lived issues rather than issues that will be closed shortly after the end of the observation period. Note that the maximum theoretical lifetime of any sticky issue is 1,092 days (i.e. this is the number of days between the start of the observation period and Jan. 1, 2015).

5.4. Model Construction

The overall predictive model construction method involves extracting features to characterize issues in the dataset, training the model, and evaluating the model. In the following subsections, we give detailed information about the first two steps.

5.4.1. Features

Our approach is based on applying supervised machine learning. The input for learning algorithms is a set of features that describe each issue in as much detail as possible. Next we list the features we extracted from each issue and the justification for doing so (the features are listed in Table 2).

During the feature engineering process, we tried to come up with features that would capture the properties of issues as well as the activity of the project and issue submitter around the time of issue creation. The assumption is that besides individual issue factors, the surrounding context also determines whether an issue will be closed.

We initially came up with 36 features. We identified correlated features by calculating Spearman's rank correlation [89] between all pairs of features and manually removed a feature from the each pair of those with a correlation value larger than 0.8. The decision of which feature to remove was done manually, but if a feature was correlated with multiple other features, it was removed first. For the remaining 32 features, we calculated the chi-squared (χ^2) statistic [93] between the training label and each of the features. We then removed features that were ranked to the last third of all the features based on the χ^2 statistic value. The χ^2 statistic between a training label and a feature measures statistical dependence between them, thus enables us to remove features that are most likely to be independent of the class label. This gave a final feature set with 21 features. Table 3 lists features that were removed from the initial set. Most of the filtered features are related to issue features and actions that could be done in the issue tracker, such as updating issue milestone (`nMilestonedByT`, `nDeMilestonedBy`), renaming the issue (`nRenameT`) and removing the label (`nUnlabeledT`). These actions in GitHub are not frequently used and the resulting features were sparse and only had assigned value in a small subset of issues.

To capture the dynamic aspects of open issue reports, we calculate the evolving features at different time points. For example, the number of comments changes over time, but the issue title does not. In Table 2, the dynamic features have suffix T in their name.

Issue Features. The first group of features describes the issue itself. For example the number of comments (`nCommentsT`) can be regarded as a measure of engagement on the issue. Guo et al. have found that more commenting on the bug report can lead to a faster fix [65] and Tsay et al. [128] have shown that the more comments on a pull-request, the more likely it is to be accepted. Besides the comments themselves, the number of persons interacting with the issue might impact the issue resolution time. Number of actors (`nActorsT`) is the total number of persons who have had interactions with the issue – opening, closing, commenting, and referencing. These features are dynamical in nature – the number of comments can be different at each observation point.

Other features in this group, such as the number of times an issue has been assigned or mentioned in connection with other issues, reflect the overall activity of the issue and are dynamical. Besides dynamical features, we extracted the issue content text length (`issueCleanedBodyLen`) to represent the length or possible complexity of the issue.

The issue reports' unstructured textual content has been shown to have predictive power for estimating the issue lifetime [134]. The typical approach for analyzing text data would be to convert issue reports into a bag of word representations. This typically leads to a large sparse representation, as some words are only present in a small set of documents. Adding all these features to our previously defined features would make the classification task harder as the number of parameters can become very large. In addition, it makes it harder to understand what the important features are.

We decided not to include textual features directly into our model. Instead, we transformed the textual content into a single score, representing the likelihood that an issue report with such text will be closed within a time period. A similar approach has been previously used for bug classification [143] and clustering [10].

For each issue, we joined issue title and content text into a single text.[1] We parsed the markdown representation and completely removed all source code blocks, tables, and links. Next we removed the remaining markdown markup and kept only the textual content. We converted all the text into lowercase and removed the punctuation and English stop words (such as a, the, that, etc.). We also applied the Porter stemming algorithm to extract the stem for each word. For each issue, we kept single words and n-grams of size 2 and we constructed the bag of words representation using feature hashing [133], with 2^{20} features. Each vector is normalized with l_2 norm and is non-negative. Using the hashing trick, we can keep all the words and n-grams and do not have to construct a dictionary during training that contains all allowed words, which is helpful when deriving the score for different parts of the data as the usage of dictionary could also leak information about the target label.

We divided the training data into two random sets. We use the first subset to train a model on text vectors and predict the score on the second subset. The predicted scores will be the corresponding textual score (`textScore`) for the second subset. We repeat the process the other way around – we train on the second subset and predict on the first subset and attach a prediction score to the features of the first subset. The scores will be appended to the overall feature set. The process is illustrated in Figure 12.

For the test set, we train on the whole training data and predict for the test set's textual content and add the score as a new column. Note that in any case, we do not leak any information about the target as we always derive the score using different subsets and do not compare them to actual labels.

To train the `textScore` classifier, we used the Stochastic Gradient Descent-based classifier [129] incorporating logistic error, l_2 regularization with a penalty of 0.001, and shuffling before iterations and 5 iterations. This model is suitable for problems with a large number of features as the regularization helps to control

[1] We use the latest version of title and body as GHTorrent only keeps the last version if the fields are updated.

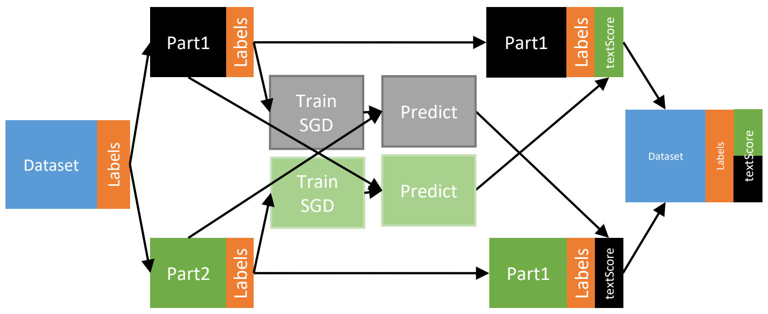


Figure 12. Deriving `textScore` feature for the training data.

over-fitting by constraining coefficient values and is fast to train. Other alternatives to consider would be the linear support vector machines (obtaining the probabilities is more costly) and Naive Bayes. We briefly experimented with the latter one and the results were approximately in the same order.

Issue submitter features. An individual's reputation has been shown to have an impact on the time in which an issue will be fixed [65]. The idea of this group of features is to capture the previous interactions that the issue submitter has had prior to the submission in the context of the project. The features extracted reflect the prior activities done by the submitter in the past three months in the context of this project, such as the number of issues created (`nIssuesByCreator`) and their number of commits (`nCommitsByCreator`).

Participant's features. Open-source projects have different levels of participation, ranging from core team to irregular contributors. To study the possible effects of individual influence, we extract the number of commits made (`nCommitsByActorsT`) by the people participating in the issue, or in other words, actors. Actors are all people who comment on an issue, change any of its properties such as tags, milestones, and/or assignments. In addition, we count a person to be an actor if they reference it from another issue or commit message. These features are dynamic and we calculate them over a period of two weeks before the observation point, to see whether the persons who have had interactions with the issue are still active.

Project (contextual) features. The aim of project features is to capture the overall state of the project. Our hypothesis is that if the project is not active, i.e., there has not been coding activity recently or no issues have been closed, then it is also likely that new issues will not receive attention. We calculate the total number of commits in the past three months (`nCommitsInProject`), the total number of new issues in the past three months (`nIssuesCreatedInProject`), and the same activity in the past two week with respect to the observation point (`nCommitsInProjectT`, `nIssuesCreatedInProjectT`). We use different period ranges of three months and two weeks to prevent possible overlap and correlated features, as in some cases the dynamic features can be calculated close to the issue submission and therefore have overlap with each other.

Note that we do not use any identifier of the specific project to which an issue belongs, as our goal is to study the performance of cross-project models built on large project repositories. However, information about the project characteristics and its state is captured via the above contextual features.

5.4.2. Model training

Often for prediction tasks, cross-validation is used for evaluating the suitability of the model and making sure the model performance is reliable on different subsets of data. Our goal is to train a predictive model that also takes into account the temporal information of issues. This prevents us from using traditional cross-validation. The idea is that for training data, we can only use data from a period that is prior to the period in which issues contained in the test dataset have been opened. This corresponds to a real-world scenario – we cannot use future data for training and then test on past data.

Our dataset covers three years: 2012, 2013, and 2014. We split the data in two from September 1, 2013. Everything before September 2013 is for training data, and everything after that point in time is for testing. This split leaves 424,004 (43.9%) issues into the training set and 543,033 (56.1%) into the testing set. In addition, the final number of issues that can be used for training and testing depends on the task as the number of issues that could be used for estimating whether an issue will be closed within a year is smaller than the number of issues used for estimating whether an issue will be closed within a month.

We trained classification models for different combinations of an observation point (i.e. the point in an issue's lifetime when the prediction is made) and a prediction horizon (i.e. the timeframe after issue creation by which we predict that an issue will already be closed). For example, an observation point of 7 days means that we make a prediction for an issue that has already been open for 7 days. Meanwhile, a prediction horizon of 30 days means we predict whether an issue will be closed within 30 days of its creation or not (note that this is a binary classification task).

The observation points and prediction horizons are chosen to match calendric periods (one day, one week, one fortnight, one month, one quarter, one semester, and one year) and of course, the issue creation time itself is taken as one of the observation points. This leads to seven observation points (0, 1, 7, 14, 30, 90, and 180 days) and seven prediction horizons (1, 7, 14, 30, 90, 180, and 365 days).

Note that the models with a zero-day observation point are such that the dynamic features are not meaningful. A small caveat, however, is that the dataset also has issues where the first comments arrive at exactly the same time as the is-

Table 2. Features extracted for each issue. Suffix "T" (short for "Time") in the feature name denotes that this feature is dependent on the observation point.

Issue features	
nCommentsT	Number of comments issue has received before the observation point T.
nActorsT	Number of unique persons who have commented, referenced or subscribed to the issue before the observation point T.
nAssignmentsT	Number of assignment events before T.
nLabelsT	Number of labels added before T.
nMentionedByT	Number of times issue was mentioned from other issues before T.
nReferencedByT	Number of times issue was mentioned in commit messages using the issue id, before T.
nSubscribedByT	Number of persons subscribing to receive updates on the issue before T.
meanCommentSizeT	Average comment size of the comments received before the observation point T.
issueCleanedBodyLen	Length of the combined title and body with markdown parsed and tags removed.
textScore	Classification score obtained from cleaned issue title and content.
Issue submitter features	
nIssuesByCreator	Number of issues created by the issue submitter in the three months prior to issue opening.
nIssuesByCreatorClosed	Number of issues created by the issue submitter that were closed in the three months prior to issue opening.
nCommitsByCreator	Number of total commits to the issue repository by the issue submitter in the three months before the issue opening.
Participant's features	
nCommitsByActorsT	Total number of commits done by actors who committed code to the project repository during the period from two weeks before the issue creation to observation point T.
nCommitsByUniqueActorsT	Number of unique actors who committed code to the project repository during the period from two weeks before the issue creation to observation point T.
Project features	
nIssuesCreatedInProject	Number of issues created in the project during the three months prior to issue creation.
nIssuesCreatedInProjectClosed	Number of issues created and closed in the project in the three months prior to issue creation.
nCommitsInProject	Number of commits created in the project in the three months prior to issue creation.
nIssuesCreatedProjectT	Number of issues created in the project during the period of 2 weeks before the issue creation until the observation point T.
nIssuesCreatedProjectClosedT	Number of issues created and closed in the project during the period of 2 weeks before the issue creation until the observation point T.
nCommitsProjectT	Number of commits in the project during the period of 2 weeks before the issue creation until the observation point T..

Table 3. Features removed from the initial feature set due to correlations or low predictive power.

Feature	**Description**	Reason
Issue features		
nDemilestoningT	Number of times issue milestone tag was removed	chi-squared
nMilestonedByT	Number of times milestone was set	chi-squared
nRenamedT	Number of times issue was renamed	chi-squared
nUnassingedByT	Number of times issue was unassigned	chi-squared
nUnlabeledT	Number of times a label was removed	chi-squared
issueBodyLen	Length of the raw contents (markdown) of the issue content body text	Correlated with issue-Cleaned-BodyLen
issueTitleLen	Length of the raw contents (markdown) of the issue title text	chi-squared
sumCommentSizeT	Total comment size of the comments received before the observation point	Correlated with meanCom-mentSizeT
nPersonsMentionedBody	Number of persons mentioned (using @mentions) in the issue body text	chi-squared
nCommitsMentiondBody	Number of commits mentioned in the issue body text	chi-squared
nIssuesMentiondBody	Number of issues referenced (using #issueId or user/project#issueId convention) in the issue body text	chi-squared
nCodeBlocksInConet	Number of code sections in the issue body text	chi-squared
Issue submitter features		
nCommitsByCreatorProjects	Number of different repositories committed to	Correlated with nCommits-ByCreator
Participant's features		
nPersonCommitingInProject	Number of unique persons committing in the project	chi-squared
nCommitsProjectUniqueT	Number of unique committers in the project within two week of observation point	Correlated with nCom-mitsProjectT

sue itself. We do count such comments when calculating the zero-day features in order to keep the feature calculation method consistent. Thus the dynamic feature nCommentsT is meaningful for zero-day models.

For each pair (observation point and prediction horizon), we trained a classifier to predict whether an issue will be closed before or after the end of the prediction horizon. Naturally, such a predictive model only makes sense when the prediction horizon ends after the observation point. Hence, there are only 28 valid combinations of an observation point and a prediction horizon, and this is the number of models we trained.

In line with the predictive setting, when evaluating a model for a given observation point, we only make predictions for issues that were not yet closed at the observation point in question. Hence, the sizes of the training and the testing sets are different for each combination of observation period and prediction horizon.

We approach the issue lifetime prediction problem using binary classification. The problem could also be approached by training a multi-class classifier, for example where different classes denote whether an issue will be closed in a corresponding time range. We choose binary classification as it helps us to understand at which points in time it is feasible to make predictions and how the accuracy of the models changes depending on the chosen observation point and prediction horizon. In addition, regression analysis could be used to estimate issue lifetime. However, regression analysis would not allow to use sticky issues as they have not been closed and the lifetime would be undefined. Regression analysis would be suitable only for projects where all the issues get closed eventually.

5.4.3. Classification method

We use the Random Forest method [29] for classifier construction. As we don't have a separate validation set to select the best hyper-parameter combination, we did not do hyper-parameter selection. For the hyper-parameters, we set the number of trees to 1,000 and limit the maximum tree depth to 5. The number of trees parameter for Random Forests is typically regarded as the larger the better, but more trees requires more computational cost to calculate. We limited the maximum tree depth to 5 to keep the learned trees simpler and reduce the training error to prevent over-fitting on the training data.

As we train in total 28 classifiers on the different observation and prediction horizon combinations, we do not perform any additional hyper-parameter optimization in order to use the same classifier for all different prediction horizon values and avoid optimizing each task separately. The results reported below should thus be construed as lower-bounds that can be further improved via hyper-parameter optimization.

5.4.4. Evaluation

To evaluate the classifiers, we use the following technical measures: precision, recall, F1-score and area under the receiver operating characteristic (ROC) curve (AUC) (Introduced in Section 3.2.1). In our classification task, the positive class denotes issues that will be closed in the specified period and the negative class denotes issues that will not be closed in the specified period. We calculate the precision and recall for the positive class.

In the context of our classification task, precision measures the fraction of correctly classified closing issues over all issues predicted to close. Recall measures the fraction of correctly classified closing issues over all closing issues. The ideal precision and recall scores are 1, while the worst case is 0. AUC measures the probability with which we can rank a randomly chosen closed issue higher than a randomly chosen issue that will not close. For a random classifier, the value of AUC will be 0.5, while for an ideal classifier it will be 1.

We use these different measures for two reasons. Firstly, we want to compare our results with existing work and all these measures have been used in previous work. Secondly, different measures help us to prove the usefulness of the model in different scenarios. If our goal is to make an individual prediction for a single issue, it is important to obtain good precision and recall scores in order to get a correct prediction for each item. If we were interested in finding the ranked list of most likely issues to be closed in a project, then the AUC reflects how well this can be done.

We will use random forest feature importance (Section 3.1.3) to understand which features are useful in making the decision and also compare feature ranking for different observation point scenarios.

5.5. Results

In this section we report the evaluation results and analyze them with respect to the questions posed in Section 1.

5.5.1. Classifier performance (RQ2.1)

We analyze model performance for different observation points to answer our research question RQ1 (*What level of accuracy is achieved by classification models trained to predict issue lifetime using static, dynamic and contextual features?*) Table 4 shows the obtained AUC scores for each combination of observation points and prediction horizons. The scores all fall into the range of 0.636 to 0.694. The results show that long-term predictions can be made with higher AUC than short-term predictions. For each observation point, the best AUC score corresponds to the 180 day prediction horizon. The AUC scores are increasing until 180 day prediction horizon and the 365 day prediction horizon has lower AUC score than the 180 day prediction horizon.

Table 4. AUC scores for different prediction horizon and observation point (OP) values.

	Prediction horizon(days)						
OP	1	7	14	30	90	180	365
0	0.650	0.658	0.662	0.670	0.676	0.679	0.658
1		0.640	0.643	0.651	0.659	0.669	0.654
7			0.639	0.646	0.654	0.672	0.645
14				0.649	0.657	0.674	0.636
30					0.657	0.676	0.639
90						0.688	0.663
180							0.694

Table 5. F1 scores for different prediction horizon and observation point (OP) values.

	Prediction horizon(days)						
OP	1	7	14	30	90	180	365
0	0.449	0.614	0.669	0.725	0.800	0.855	0.887
1		0.406	0.497	0.573	0.660	0.769	0.853
7			0.250	0.422	0.573	0.692	0.785
14				0.296	0.518	0.651	0.751
30					0.406	0.578	0.730
90						0.362	0.654
180							0.498

We also calculated F1-scores for each model as shown in Table 5. The F1-scores show the cases of models that fail overall, meaning that while they do correctly identify the issues that are most likely to close (i.e. that have a certain level of ranking accuracy as measured by AUC), they have either low precision or low recall or both. We observe the lowest F1-scores when the gap between the observation point and the end of the prediction horizon is the smallest (cf. the diagonal values in the table). The reason for this phenomenon is that in these cases, the class imbalance is the highest. The best F1-scores are obtained when the gap between the observation point and the prediction horizon is the largest. In other words, longer-term predictions are more accurate. One possible explanation for this is that there are more issues with a smaller lifetime (note that half of the issues have a lifetime of less than 3.7 days) and making predictions for them is harder as there are more varied reasons for closing them. Meanwhile, there is a smaller number of issues with a longer lifetime and therefore the features can better capture the corresponding reasons for closing them.

The raw experimental results give us a broad view of how well the models perform. But since the sample sizes vary considerably across different (observation point, prediction horizon) combinations, we cannot directly compare performance across models. Accordingly, in Table 6, we report the results for experiments in which the testing set is always of the same size for all models with a given prediction horizon. This makes it more meaningful to compare models across different observation points. For each prediction horizon, the table gives the performance across all observation points that are before the prediction horizon. For exam-

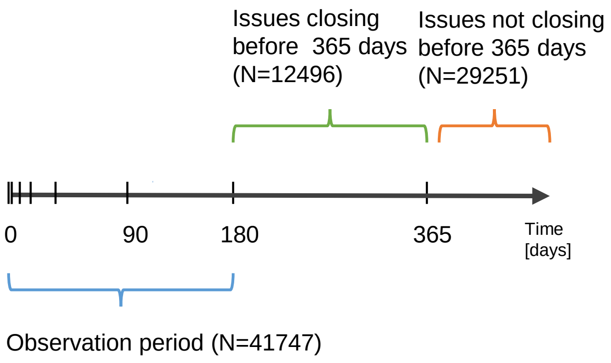

Issues closing before 365 days (N=12496)
Issues not closing before 365 days (N=29251)
0
90
180
365
Time [days]
Observation period (N=41747)

Figure 13. Observing a set of issues for prediction whether they will close before 365 days or after (Observation point 180 days, prediction horizon 365 day). The sample size (N) values correspond to the first group in Table 6.

ple, for the prediction horizon of 365 days, we only include issues that have not been closed in the first 180 days. For these issues, we can perform the prediction at different observation points. Figure 13 illustrates the concept of observing the same set of issues over different observation points (and thus different observation periods) in order to predict whether or not the issue will close before 365 days or not. Similarly, we do the analysis for other prediction horizons, with each time one less possible observation point and a larger test set.

When looking at the AUC and precision values (Table 6), we observe that in many cases (specifically for prediction horizons of 365, 180 and 90 days) the scores increase as the observation point increases. For the prediction task of whether an issue will be closed within 365 days, we observe an 28.7% increase in AUC (from 0.499 to 0.694) and a 28.5% increase in precision (from 0.301 to 0.421) across the seven corresponding observation points. This supports the hypothesis that observing an issue over an extended period can lead to better predictive power. In contrast, the recall scores are more fluctuating and show a slight negative trend when the observation point increases. The reason for this is that the number of true positives (correctly classifying closing issues) decreases only slightly with an increasing observation point, while the number of true negatives (correctly classified issues that will not close) keeps increasing.

To summarize the findings with respect to RQ1, we conclude that when making repeated predictions for issues as they evolve over time, predictions made at later observation points yield higher AUC and precision scores, but lower recall scores.

5.5.2. Feature importance (RQ2.2)

In order to address RQ2 (*What features are most important when predicting issue lifetime?*), we analyze the mean decrease in impurity for each feature as discussed in Section 3.1.3. We are particularly interested in understanding the role

Table 6. Prediction performance of models tested with a constant test size (N) for any given prediction horizon.

Observation point	AUC	Precision	Recall	F1	TP	FP	FN	TN
	Prediction horizon of 365 days (N=41747)							
0	0.499	0.301	0.623	0.406	7790	18125	4706	11126
1	0.529	0.312	0.631	0.418	7879	17346	4617	11905
7	0.557	0.339	0.581	0.428	7263	14191	5233	15060
14	0.570	0.360	0.574	0.442	7169	12757	5327	16494
30	0.597	0.368	0.599	0.456	7485	12866	5011	16385
90	0.658	0.382	0.635	0.477	7938	12826	4558	16425
180	0.694	0.421	0.608	0.498	7594	10425	4902	18826
	Prediction horizon of 180 days (N=136926)							
0	0.569	0.199	0.769	0.317	18657	74947	5606	37716
1	0.594	0.217	0.663	0.327	16088	58178	8175	54485
7	0.634	0.234	0.644	0.343	15617	51169	8646	61494
14	0.649	0.231	0.651	0.341	15784	52500	8479	60163
30	0.664	0.229	0.663	0.341	16098	54107	8165	58556
90	0.688	0.242	0.719	0.362	17446	54609	6817	58054
	Prediction horizon of 90 days (N=235195)							
0	0.592	0.267	0.755	0.395	41674	114185	13527	65809
1	0.605	0.293	0.566	0.386	31230	75506	23971	104488
7	0.625	0.291	0.570	0.385	31449	76578	23752	103416
14	0.639	0.292	0.599	0.392	33091	80357	22110	99637
30	0.657	0.296	0.645	0.406	35595	84508	19606	95486
	Prediction horizon of 30 days (N=318153)							
0	0.612	0.175	0.757	0.284	35234	166590	11288	105041
1	0.616	0.191	0.597	0.290	27795	117533	18727	154098
7	0.632	0.189	0.608	0.289	28266	121068	18256	150563
14	0.649	0.192	0.651	0.296	30295	127771	16227	143860
	Prediction horizon of 14 days (N=372749)							
0	0.613	0.142	0.757	0.239	33088	199788	10606	129267
1	0.616	0.152	0.627	0.245	27375	152226	16319	176829
7	0.639	0.155	0.650	0.250	28400	155246	15294	173809
	Prediction horizon of 7 days (N=486691)							
0	0.639	0.275	0.785	0.408	86055	226352	23584	150700
1	0.640	0.288	0.692	0.406	75883	187941	33756	189111

played by dynamic features, hence we compare the feature importance for the models constructed at creation time (the zero-day models) versus one week after issue creation (the 7-day models). This approach allows us to understand the role played by dynamic features early on during the lifetime of an issue. Regarding the prediction horizon, we look at models constructed to predict if an issue will close after 30 days (short-term predictions) or after 180 days and 365 days (long-term predictions).

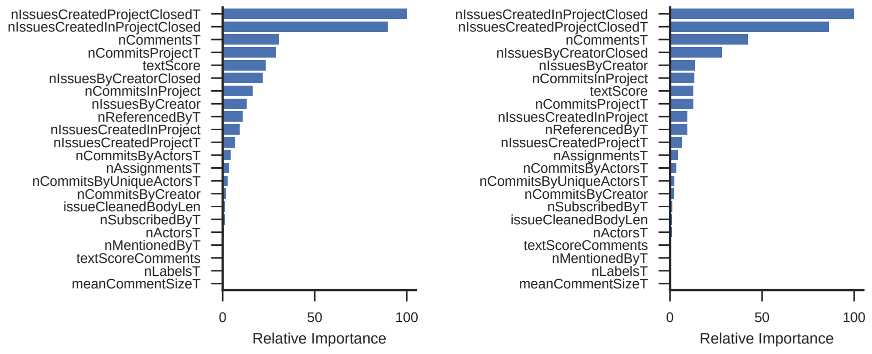
The ranking of feature importance for the zero-day models is given in Figure 14. In the case of the zero-day model with a 30-days prediction horizon (cf. Figure 14a), the top-ranked features are `nIssuesCreatedProjectClosedT` and `nIssuesCreatedInProjectClosed` (i.e. issue closing activity in the two weeks and the three months before the issue submission respectively). The presence of these features at the top of the ranking (including `nCommitsProjecT` as the fourth feature) suggests that contextual features play an important role when making predictions at creation time. The third feature in the ranking is the number of comments (`nCommentsT`), which in the zero-day models only has two possible values (0 or 1) – some issues come without any attached commentary at issue creation time, while others come with a comment having the same timestamp as that of issue creation. Expectedly, the presence of this initial comment carries some information about issue lifetime. We also observe that `textScore` is highly ranked, stressing the potential value of extracting information from the text attached to issues.

In the case of the zero-day model with a 180-day prediction horizon (Figure 14b), we observe that the top 3 features are the same as in the model with a 30-day prediction horizon. On the other hand, the importance of `textScore` when predicting for 180 days is lower than when predicting for 30 days. In other words, the text attached to the issue is useful for short-term prediction, but becomes less important for longer-term prediction.

In both cases (zero-day models with 30-day and 180-day horizons), the features that have least importance are those related to the size of comments and number of issue labels. This is simply because these features are dynamic and hence not meaningful for zero-day models.

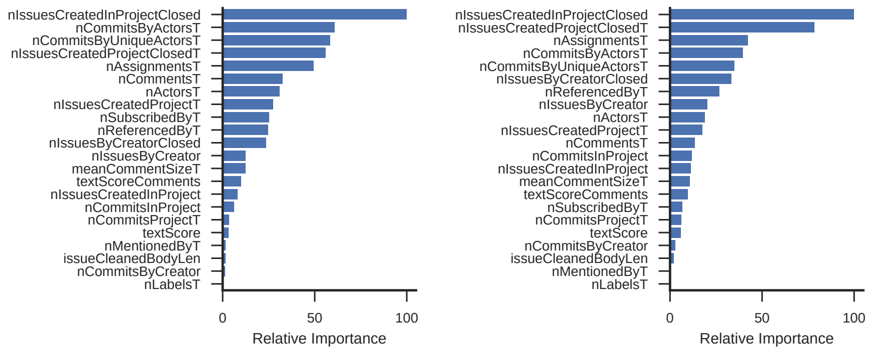
Let us now compare the feature importance ranking of the zero-day models (Figure 14) against the 7-day models (Figure 15) with a 180-day prediction horizon. The top feature remains the same `nIssuesCreatedInProjectClosed`. We observe that feature `textScore` has less importance in the 7-day models, and instead dynamic features overtake it in importance, e.g. number of commits (`nCommitsByActorsT`), number of unique actors committing (`nCommitsByUniqueActors`), and number of assignments of the issue (`nAssignmentsT`). This observation reinforces the hypothesis that dynamic features carry information that complements static features, particularly when making long-term predictions.

One can wonder if a longer-term prediction horizon affects feature importance



(a) 30-day horizon (b) 180-day horizon

Figure 14. Feature importance for the zero-day models with 30-day and 180-day horizons.



(a) 180-day horizon (b) 365-day horizon

Figure 15. Feature importance for the 7-day models with 180-day and 365-day horizons.

significantly. To this end, Figure 15b displays the feature importance ranking of the 7-day model with a 365-day prediction horizon. It turns out that this ranking is very similar to the one for the 7-day model with a 180-day horizon. Although not shown here, we observed a similar ranking in the 7-day model with a 90-day horizon, suggesting that the task of predicting closing time with a horizon of a few months is similar to that of predicting it with a one-year horizon.

In summary, with reference to RQ2, we can say that contextual features complement static features both for short-term and long-term predictions. Dynamic features in turn complement both static and contextual features and their inclusion explains the observed increase in accuracy of models built for later observation points.

5.6. Discussion and limitations

The observed accuracy of our models (AUC and precision scores) suggests that predictive models of issue lifetime across large sets of open-source projects could potentially be used in practice if users were willing to tolerate some fluctuation in their predictive accuracy. Compared to previous research, Giger et al. [58] obtain AUC scores between 0.649 and 0.823 when increasing the dynamic feature observation period from 0 days to 30 days (Eclipse JDT project). Their results also show fluctuations in performance, i.e., observing features for a longer period does not lead to a monotonic increase in model performance. Their model precision scores are also better, ranging from 0.635 to 0.885, but recall values are lower than in our experiments, ranging from 0.485 to 0.661. They also experience high variation across projects, especially with Gnome Gstreamer project dataset where performance decreases with longer post submission data, with the AUC values mostly decreasing from 0.724 to 0.586. Francis & Williams [55] similarly show that the same issue lifetime prediction method can perform differently with an open-source versus a closed-source private project. This confirms that issue lifetime prediction varies across projects, and suitable accuracy can not be always obtained.

The experimental setup used by Giger et al. [58] is not directly comparable to ours, as they used a different prediction task and their dataset properties were different, such as a considerably larger median issue lifetime. The main obstacle for using their study as a baseline is that they have a different set of features with richer meta-data about issue reports such as reporter, milestone, outcome, and platform. Another aspect that might work in their favor is that they perform cross validation without using any temporal information about the creation of issue reports (i.e. no temporal split), so that "future data" may be used to classify an issue at a given time point. Nevertheless, their results with using only static features are comparable in terms of AUC, where they had scores ranging from 0.649 to 0.724 across projects.

With respect to the use of temporal splitting, the reported findings are in line with those of Assar et al. [10], who observe that when using temporal splits and observing issues for a longer period, the prediction error becomes lower compared to shorter observation periods.

Our work uses issues from more than 4,000 projects. The projects have different development practices, backgrounds, resources and goals. Hence the heterogeneity in project properties can affect our results as the models cannot make sound generalization based on issues from different projects. It has been shown that features, such as developer reputation, which can be important in projects determining the issue lifetime, are not important in other scenarios [22]. Another shortcoming is that we do not distinguish between different types of issues (e.g. bugs vs. feature requests), although this can have an effect on the resolution time [11, 105].

Another limitation of the study is the lack of cross-validation in the evaluation of the classification models. We have chosen to use temporal splits between training and testing data, as this is exactly how the models are trained when used in real world scenarios, and should give a better estimate of issue lifetime. The drawback of this choice is that it does not leave much room for performing multiple test splits, since the period covered by the dataset is exactly the length required for one split.

In addition, the precision scores of our models are low – i.e. issues that we predict will close before the horizon often remain open beyond it. This restricts their direct practical applicability. A critical direction for future work is thus to investigate the reasons for low precision and to improve the models, for instance, by introducing additional features (e.g. from the code commits or from the text of the comments).

A potential threat to validity is that our dataset contains non-software development projects since removing all of them manually would be impractical. To estimate the extent of non-software development projects in the dataset, we manually checked a random sample of 100 projects. We found that 89 of them can be classified as software projects (i.e. projects containing code and build files or deployment guides). Two projects contained only documentation, two contained specifications, two data, one was used purely as an issue tracker for an externally hosted project, and four had been since deleted and thus their nature could not be ascertained.

5.7. Replication package

Replication of the experiments was successfully carried out by Mittal [101] as a course project. Replication revealed minor issues in our code regarding fixing random seeds, and a bug which resulted in subset of training data not being used. The results presented in thesis therefore differ numerically from the results presented in the original paper [85]. However, the conclusions and findings remain unchanged. The dataset and the code for training and evaluating the model is available at
`https://github.com/riivo/github-issue-lifetime-prediction`.

5.8. Summary

We studied the problem of predicting issue lifetime in GitHub projects for different calendric periods, using a combination of static (creation-time), dynamic, and contextual features. Based on the issues extracted from a sample of 4,000 projects, we show that such predictive models exhibit better accuracy when trained with one-day-old or one-week-old issues to predict whether or not an issue will remain open after a one month or longer period. This study highlights the importance of dynamic and contextual features in such predictive models.

6. STRUCTURE AND EVOLUTION OF PACKAGE DEPENDENCY NETWORKS

6.1. Introduction

We observed that projects have an increasing number of pending issues that linger over time. The pending issues can also contain bug reports including security vulnerabilities. With software reuse becoming more prevalent, these bugs could propagate through dependencies with other projects. This led us to study dependency networks formed between software packages.

The goal of this chapter is to study the current state of dependency networks, to understand their characteristics, and to predict their future evolution. We have formulated the following research questions to guide our research:

RQ3.1: *What are the static characteristics of package dependency networks?*
RQ3.2: *How do package dependency networks evolve?*
RQ3.3: *How vulnerable are package dependency networks to the removal of a random project?*

The answers to these questions can help to quantify the state of the ecosystems, give an overview of the trends in dependency management, and inform the development of improved dependency management tools.

6.2. Background and terminology

The current study analyses dependencies between software projects. We distinguish between two types of software projects: packages and applications. We define a *package* as a reusable code or a set of components that can be included in other applications by using dependency management tools. Packages are published in *repositories* and are available to everyone. *Applications* are projects that make use of packages, are not published as a package, and thus cannot be used in other projects as a dependency. Packages and applications can have multiple versions distinguished by version numbers.

One package can depend on another package. If package `C` depends on package `D`, we say that `C` has a *dependency* (`C` is a dependent of `D`) and `D` has a *reverse dependency* (`D` has a *dependent*). Applications can have dependencies but since they are not published as reusable packages they cannot have reverse dependencies. A project has a *direct dependency* if a package on which the project depends, and which it needs in order to be built, is directly included in the project. A project can have *transitive dependencies* on packages that are not needed for the project itself but are needed for the direct dependencies included in the project to work. Transitive dependencies can be included through multiple levels of dependencies. Figure 16 illustrates the concepts of dependency relation types from the perspective of package `C`.

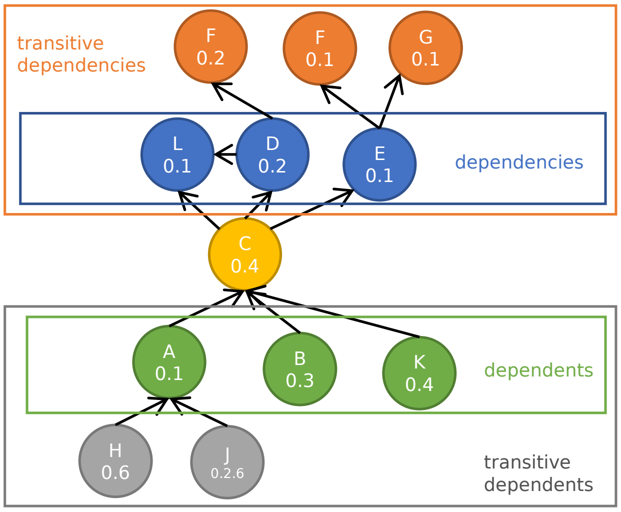


Figure 16. Dependency relationship types between projects from the perspective of package *C*.

A *dependency network* is composed of packages, applications, and the dependency relations between them. An *ecosystem* is a set of packages and applications involved in a dependency network.

6.3. Research questions

Our overall goal is to analyze the structure and evolution of dependency networks to gain insight into current dependency usage and possible issues that arise from them. Next, we explain the motivation behind each research question in more detail.

RQ3.1 (Structure). Currently, not much is known about the static properties and topologies of programming language package ecosystems. For example, we know to what extent dependencies are used in packages only [47, 136]. However, we do not know if there are differences in dependency usages across published packages and applications. Modern package managers allow different conventions for specifying dependency version numbers such as the exact version or version range. However, we do not know what the most popular way of specifying dependencies is. Answers to these questions would enable us to understand the current state of the dependency ecosystem and would be the starting point for analyzing ecosystem evolution.

RQ3.2 (Evolution). Software projects can add new dependencies and update existing dependencies. Changes in dependencies in a new release of a single package will also be reflected in the overall dependency network. Studying the dependency network's evolution since its creation can explain its current state and also provide knowledge to help explain it and make predictions about its future evolution. The need for such analysis was outlined by respondents to a recent survey on software ecosystems challenges [120]. One of the answers given by a respondent stated: *if an ecosystem is not able to evolve quickly it is going to die* [120]. Similarly, our goal is to understand the current evolutionary state of the studied ecosystems and analyze whether they are growing or not.

RQ3.3 (Vulnerability). When selecting a package to use, several factors are important besides the functionality it provides. Developers ideally would like to be sure that the package quality is good, well-maintained, and trustworthy. As these properties are not explicitly visible, developers might end up using packages of varying quality. For example, if an attacker publishes packages with names very similar to the names of popular packages, developers making a typo could end up using them unwillingly [75]. The *left-pad* incident happened because the developer decided to remove the package. How vulnerable are ecosystems to such scenarios? We define vulnerability as the number of projects that are affected if we remove a package or a specific version of it. This scenario also helps us estimate the proportion of the dependency network that is impacted if a package contains a bug, or is stopped being maintained. The vulnerability measure is a proxy for a centrality or importance of the package in the ecosystem. Note that we lend the term vulnerability from the complex systems domain, where attack vulnerability denotes the decrease of network performance due to a removal of vertices or edges [3, 74].

6.4. Method

In the following section, we describe the data collection method, preprocessing steps, and our approach to modeling dependency networks using graphs.

6.4.1. Context

We study three package ecosystems for the programming languages: JavaScript, Ruby, and Rust. Majority of the packages and applications are hosted on GitHub for the chosen programming languages. These languages have central repositories for distributing packages, namely `npm`, `RubyGems`, and `Crates`. Developers specify required packages in their projects' dependency files (`package.json, Gemfile, Cargo.toml`) and packages are retrieved by the dependency manager (*npm*, *Bundler*, *Cargo*). The packages contain source code and developers can use functionality from packages in their project. In addition to packages, we study applications downloaded from GitHub. By adding applications, we can analyze package usage from the end-user's viewpoint.

We chose to study JavaScript and Ruby, both dynamically typed languages, which are popular choices for web application development. Rust, on the other hand, is a multi-paradigm language that supports static typing primarily meant for system programming. JavaScript and Ruby have been used since the 1990s and their corresponding central package managers appeared in 2010 and 2004, respectively. Rust first appeared in 2010 and its central package management appeared in 2014. Our analysis of JavaScript revolves around the packages used in the *node.js* environment and managed through the `npm` tool, but also includes packages only needed for web development, such as front-end frameworks. JavaScript differs from the other languages used in this study as it supports multiple versions of a project in its dependency chains. For example, if package A depends on package B version 1.0 and package C depends on version 2.0, while package B depends again on package C version 3.0, then `npm` downloads both versions of the package C. Rust and Ruby do not allow such a scenario and a single version of package C is required. In practice, JavaScript developers can have more freedom in including dependencies, but Rust and Ruby developers need to make sure their dependencies do not conflict.

6.4.2. Data collection

We used multiple sources to compose the dataset. For JavaScript and Ruby, we downloaded the full list of packages, release dates, dependencies, and other relevant meta-data from their central repositories, `npm` and `RubyGems`, respectively. To extract data from `npm`, we used the public API [108]. For `RubyGems`, we used a copy of their meta-data database available online [117].

Central repositories such as `npm` and `RubyGems` host projects that are typically only libraries, frameworks, command line applications, or resource bundles for web development. We also include end-user applications from GitHub in our study to understand the package usage in practice. We used the GHTorrent [60] database from March 2016 to select projects whose repository language identified by GitHub was either Rust, JavaScript, or Ruby, were not forks, and the project GitHub repository did not appear in the `npm`- or `RubyGems`-hosted project list. After composing the initial list of projects, we made an HTTP request to every repository to check if it had a dependency file in the root folder of the latest revision. We only cloned repositories that had a dependency file present in the latest revision. For Rust, we cloned all projects listed in GHTorrent, but for JavaScript and Ruby, we only cloned those that either had at least one fork or at least one star, to minimize the number of projects to be collected. We acknowledge that we were not trying to collect all the projects from GitHub.

Rust has a central repository called *Crates.io*, but its meta-data is not available in a structured machine-readable format. Therefore, for Rust, we only rely on the packages from GitHub by first selecting all Rust language projects from the GHTorrent database and then filtering out those that do not have a dependency file named *Cargo.toml*. The Rust data can be considered as a sample of the whole package universe of Cargo and additional applications written in Rust.

Data collection took place between April 2016 and May 2016. We collected the package repository data after collecting applications from GitHub. We excluded all updates and changes after April 2016 in order to get a comparable time scale for all ecosystems.

6.4.3. Parsing GitHub projects

The projects obtained from GitHub have their dependency information recorded in dependency files. To extract dependencies, we consider all revisions of the dependency files to recover the dependency history. We used the *git log* command to extract all changes to the dependency file. For accurate modeling, we had to know when each version of a project was released. JavaScript's *package.json* and Rust's *cargo.toml* provide explicit version information of the project. Ruby's dependency files (.gemspec and Gemfile) are written in Ruby code and sometimes the version number is expressed as a variable or read in from a file. This makes reading the exact version numbers hard, as there is no general pattern. Extracting this is therefore not feasible, as it would require manual inspection or executing the code. In cases where we could not extract explicit version numbers, we used the time of the last modification of the dependency file. This only affects applications and does not impact the dependency network structure as dependency files do not have dependents. The limitation of this approach is that there might be many more revisions than actual releases. If multiple revisions of a dependency file exist with the same version number, we use the latest revisions for the version. Developers might change the contents of the file during development with the new version number already entered but after the release the contents will not change.

6.4.4. Resolving dependencies

When parsing dependency files, we encountered situations where some of the dependencies were not available. A dependency might not be available in a case where a single revision of a dependency file committed to the repository contained typos or incorrect version constraints – thus a dependency did not exist. We only kept those dependencies that we could match in the central repositories for JavaScript and Ruby. For Rust, we kept all dependencies we could match between the projects as we did not use official package repository data. If a dependency is specified as a reference to a git source code repository, we only kept this if it was a Rust project and the repository was in the list of collected projects.

Dependency version constraints can be specified in different ways, for instance as an exact version, a latest version, or a pattern-based matching using the semantic versioning notation. A version number is typically written in the format of `MAJOR.MINOR.PATCH`. An increase in the `MAJOR` number denotes incompatible API changes, an increase in the `MINOR` number indicates an addition of backward compatible changes, and an increase in the `PATCH` number indicates a bug fix. A version requirement specification has specific notations for describing valid version. JavaScript and Rust support similar notation formats. To obtain *any* version or the *latest* version, the requirement should be specified as the wild-card (*) or with an explicit condition (≥ 0). The *tilde* operator (~) matches the most recent `MINOR` version. For example, ~3.0.3 matches the highest version in the range $(3.0.3, 3.1)$, but will not match 3.1. The *caret* (^) will select the most recent `MAJOR` version (the first number). For example, ^1.2.3 matches the highest version in the range $(1.2.3, 2.0)$. Ruby does not support the *tilde* and the *caret* directly, but has something similar called the pessimistic operator, expressed by $\sim>$. For example, $\sim>$ 3.0.3 is equivalent to ~3.0.3. Requirement $\sim>$ 1.1 is equivalent to ^1.2, i.e., matches the highest version in the range $(1.2.3, 2.0)$.

For network construction, we must be able to represent the state of dependencies as they were at the time a package was released or an application was committed to the repository. With inexact version requirements, the actual version that might be included in the project might differ every time the project is built, as a more up-to-date version of a dependency that satisfies the requirements might have become available. We resolved all dependency version requirements to the version that would have been used when the package was released or a GitHub commit was made. Therefore, we knew when the release was made and also could trace back which packages and versions were available at that time. For JavaScript projects, we used the package *semver* to find for each dependency the highest version candidate available. For Ruby projects, we used *Gem* library code for finding the latest revision among all the matching candidates. For Rust, we implemented our own dependency resolution.

Dependency version resolution did not take into account transitive dependencies and possible version conflicts. We are aware that in practice, some other version might have been chosen. To resolve all dependencies we would have needed to re-implement the corresponding language dependency resolution algorithm because dependency management tools do not support resolving dependencies as they would have been resolved at any arbitrary time in the past.

6.4.5. Network construction

When modeling a system within a network, we need to define what nodes and edges represent. A straightforward approach to representing dependency relations in networks is to model projects as nodes, and directed edges between them denote dependencies between projects. The limitation of this solution is its lack of differentiation between project versions and thus this modeling approach could give misleading information about the network. Figure 17 illustrates three different approaches for network modeling. Packages A and B depend on different versions of C, but only C version 0.4 depends on D. The aggregated network model would indicate that package B is dependent on package D, which is not true. The number of different packages dependent on D is two (A and C) in the actual network, but aggregated version would give us three projects (C, A, and B). We also studied an approach where we annotate network edges with attributes. We have a list of pairs (source version, target version) for which this edge is valid.

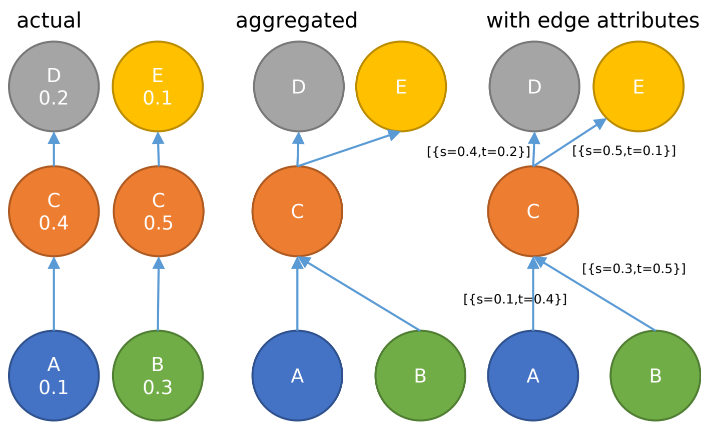


Figure 17. Dependency network construction approaches.

When traversing the network, we have to make sure that the target version on the edge that was used to access the node has a corresponding source node for taking the next step. For evolution analysis, both the aggregated network and aggregated network with attributes are unsuitable. If we want to answer questions such as what the number of transitive dependencies is, we have to consider all project versions. A new release of a project can update its dependencies, thus increasing the connectivity in the aggregated graph. For example, all versions of the aggregated graphs (Figure 17) would indicate that project C has two dependencies; however, at any one time, it can have only one. Considering this, it might affect all the projects and we would get a more connected graph than the actual project and the number of dependencies would not reflect the actual value.

We chose an approach where a node represents a specific project version. The edges denote dependency relations between specific versions (Figure 17, *actual*). With this modeling approach, we can find the correct answers to queries such as how many different versions depend on a project and how many of these are unique projects.

In our analysis, we sometimes used the aggregated modeling version with edge attributes for some calculations. Whenever we did so, we mention it explicitly in the following. By analyzing the top 10 projects for JavaScript based on the number of dependencies, we confirmed that the aggregated network without edge attributes overestimates the dependency counts. Therefore we decided to use edge attribute information when analyzing dependency chains.

Our choice of dependency network model makes it hard to compare our results with existing research, which uses the aggregated network without attributes [136]. Only Hejderup [71] uses a similar approach to our actual network. The difference is that Hejderup also keeps meta-nodes in the network to represent a project. Each meta node has links to the corresponding project's version node.

We only use projects that have at least one dependency or one reverse dependency. If a project neither had dependencies nor is a dependency for others, it

Table 7. Summary of datasets.

	Projects in the network					
	Projects	Dependencies	Applications	Packages	Versions	**Version dependencies**
Rust	7,978	25,144	0	7,978	22,105	66,055
JS	246,670	1,182,114	78,657	168,013	1,319,919	7,260,426
Ruby	147,449	776,061	69,544	77,905	1,231,480	10,747,737

	Initially collected	
	Applications	Packages
Rust	0	11,037
JS	84,987	254,466
Ruby	62,133	122,786

does not appear in the network. As soon as a project adds a dependency, it appears in the network. Due to this filtering, single isolated nodes cannot exist in the network, while isolated clusters of connected nodes can.

We kept snapshots of the network for each month. A snapshot records how the ecosystem looked at the end of the corresponding month. Snapshots are cumulative, adding new projects and dependency links. Neither projects nor links are ever removed. All analyses involving the temporal evolution are also cumulative, i.e., if we calculate some property at a specific time, we calculate the property for all the projects published up to that point.

We manually removed three projects from our dataset that appeared to be outliers. Two JavaScript applications and one Ruby package had been engineered so that they would contain all possible packages in their dependency file.

6.5. Results

6.5.1. Description of dependency networks (RQ3.1)

In this subsection, we describe the datasets and basic properties of the dependency networks.

Static properties. Table 7 lists basic properties of the language ecosystems used in our study, the number of projects initially collected, and different releases in the network.

We initially collected 11,037 Rust, 339,453 JavaScript, and 184,919 Ruby projects. However, not all packages have dependencies or are used as a dependent, and therefore we exclude those projects in the network-based analysis. The exclusion was based on the latest snapshot and included projects that never had any dependencies. The final dataset comprises 7,978, 246,670 and 147,449 projects for Rust, JavaScript, and Ruby, respectively.

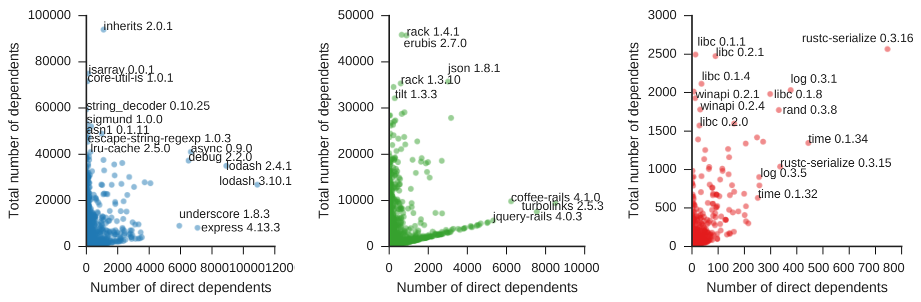
Table 8. Mean (median) number of dependencies and dependents.

	Transitive		Direct	
	Dependencies	Dependents	Dependencies	Dependents
JS	54.6 (17)	15.5 (0)	5.5 (3)	1.3 (0)
Ruby	34.1 (22)	6.4 (0)	8.7 (4)	1.2 (0)
Rust	9.3 (5)	7.4 (0)	3.0 (2)	1.6 (0)

Table 8 lists the number of dependencies and dependents (reverse dependencies) per release. Comparing languages, we see that Ruby projects have more direct dependencies on average (8.7) than JavaScript (5.5) and Rust (3.0). The differences in the number of direct dependents are smaller, i.e., 1.2, 1.3, and 1.6, respectively. However, we again see larger differences across transitive dependencies and transitive dependents (the average number of projects that depend on a project). JavaScript has the largest number of transitive dependencies and dependents, 54.6 and 15.5, respectively, while Ruby has 34.1 and 6.4, and Rust 9.3 and 7.4, respectively. The number of transitive dependents for JavaScript is almost two times larger than for the other languages. Ruby has the highest average number of direct dependencies and Rust has the highest number of direct dependents. Differences in the number of dependencies across ecosystems indicates that for different the criterias for deciding when to use a dependency can be different. JavaScript's large dependency count can possibly be attributed to fact that developers are preferring third-party code more for some reason or that the standard library lacks some required functions. The latter argument can be illustrated by the popularity of simple, short packages such as left-pad (padding a string to specified length) or isarray (a single line function for checking if an object is an array, the second most frequently used `npm` package according to Table 11).

Direct and transitive dependents. The *left-pad* incident had a high impact not because it was directly used in many projects but indirectly, through transitive dependencies (in our dataset, 33 unique projects depend directly on the left-pad, but 175,377 unique projects have left-pad as a transitive dependency). Figure 18 shows the relationship between the total number of dependents (direct and transitive dependents) and direct dependents for all projects at the beginning of April 2016. For all ecosystems, we can see that there exist projects that have a small number of direct dependents (less than 100) and a large number of transitive dependents. We can see that this pattern is stronger in JavaScript (Figure 18a) and Ruby (Figure 18b) than for Rust. Ruby also exhibits a clear pattern with a package having an equal number of direct and transitive dependents, meaning that a package is only involved in direct dependencies but not transitive ones.

Weakly connected components. Even though we limited our analysis to projects that have at least one dependency relation, the ecosystems under study are not fully connected for Rust and JavaScript. We calculated the number of weakly connected components in the dependency graphs for all languages. A



(a) JS (b) Ruby (c) Rust

Figure 18. Relationships between the number of direct dependents and total dependents in April 2016. A sample of project names are plotted.

Table 9. Distribution of version update counts.

		Type	5p	median	mean	95p	max
explicit	JS	Package	1.0	1.0	1.06	1.0	69.0
		Application	1.0	1.0	1.37	3.0	253.0
	Ruby	Package	1.0	1.0	1.19	2.0	96.0
		Application	1.0	1.0	1.53	3.0	343.0
	Rust		1.0	1.0	1.19	2.0	62.0
implicit	JS	Package	1.0	1.0	1.11	2.0	66.0
		Application	1.0	1.0	1.70	4.0	280.0
	Ruby	Package	1.0	1.0	1.91	6.0	95.0
		Application	1.0	1.0	2.17	6.0	344.0
	Rust		1.0	1.0	1.44	4.0	63.0

weakly connected component in a directed graph is a subgraph where each node is connected with every other node in the subgraph via an undirected path. We observed the emergence of a giant weakly connected component in each of the three analyzed ecosystems. For Rust, JavaScript and Ruby, 96.14%, 98.2%, 100% of projects, respectively, belong to the largest weakly connected component in the latest snapshot. Many real-world networks such as social networks exhibit the giant component property [52]. The remaining projects are part of components with a small number of projects. The existence of a giant component illustrates the fact that existing packages, even being developed by different developers, can be used together in applications. Their ability to be used together makes the ecosystem valuable, illustrating that the packages in the ecosystem follow standards and any random set of packages could be possibly used together.

Dependency updates and constraint notation practices. We define explicit dependency version change as a manually changed version constraint for a dependency by a developer. The number of explicit changes is similar across ecosystems (Table 9). The number of implicit changes denotes the number of times a dependency was resolved to a different version after each project release or dependency file commit, but without modifying the dependency requirement specification. An implicit update happens when dependencies are specified with flexible constraints, and there are newer versions released matching the constraints. The number of implicit updates has a larger variation across projects, with the highest mean of 2.17 for Ruby, compared with 1.7 for Rust, and 1.44 for JavaScript. The mean number of implicit updates for the published packages is smaller than for applications: 1.91 and 1.1 for Ruby and JavaScript, respectively. We also see that the maximum values for both explicit and implicit updates are larger for applications, which can be explained by higher velocity in development as these projects do not have dependents. For both types of projects, packages and applications, Ruby seems to have higher update counts which can be explained by its longer history. Another insight is that there are more implicit than explicit updates, indicating dependencies are updated more often than a developer would manually do this. Even though implicit updates can be considered a good practice, in case the implicit update fixes a possible bug or security vulnerability. On the other hand, implicit updates can prevent reproducible builds and the same source code configuration can yield in different outcome. In the following section, we analyze more closely the popular ways of specifying dependency version requirements that lead to implicit updates.

Table 10 lists the relative popularity of each requirement specification scheme in each ecosystem. Note that we also distinguish here between published packages and applications. The different ways to specify versions are: any or latest version (*any*), exact version (*exact*), explicitly specified version range such as $[2.0, 4)$, and one-sided ranges (*range*), the most recent minor version (*tilde*), the most recent major version (*caret*) or anything else, such as manually specified git version (*other*).

The dominating approaches for Rust version specifications are *exact* and *any* versions, used in 32% and 47.8% of the cases, respectively. Besides these, all different possible schemes for specification are used by developers. Rust developers prefer to specify specific versions or latest versions, as the ecosystem is growing.

Among the most popular approaches for JavaScript are the *caret*, *exact*, and the *tilde* notation. Exact versions are used only in 22% of the cases for different JavaScript projects. The difference between JavaScript GitHub projects and published packages is non-existent, whereas for Ruby, there are differences in the fraction of exact versions and range-based specifications. We looked more into range usage in packages and found that a majority of range specifications in published packages come from specifications that require a larger version than the specific major version. We used Pearson's chi-squared test to confirm that Ruby's applications and published packages have different preferences in specifying version requirements ($\chi^2 = 884540$, $df = 5$, p-value $< 2.2 \cdot 10^{-16}$). Ruby also has the lowest number of exact dependencies, which in turn can explain our observation of Ruby having the highest number of implicit version updates on average (Table 9). In the end, we used Pearson's chi-squared on the full contingency table (Figure 10 with absolute values) to confirm that dependency management preferences vary across languages ($\chi^2 = 8025600$, $df = 20$, p-value $< 2.2 \cdot 10^{-16}$).

Table 10. Relative popularity of dependency specification notations.

Ecosystem \ Type	any(*)	caret(^)	exact	other	range	tilde(~)
JS Application	0.047	0.498	0.221	0.005	0.019	0.210
JS Package	0.037	0.536	0.217	0.007	0.029	0.174
Ruby Application	0.583	0.157	0.135	0.000	0.063	0.062
Ruby Package	0.360	0.178	0.070	0.000	0.249	0.143
Rust	0.320	0.034	0.478	0.109	0.007	0.052

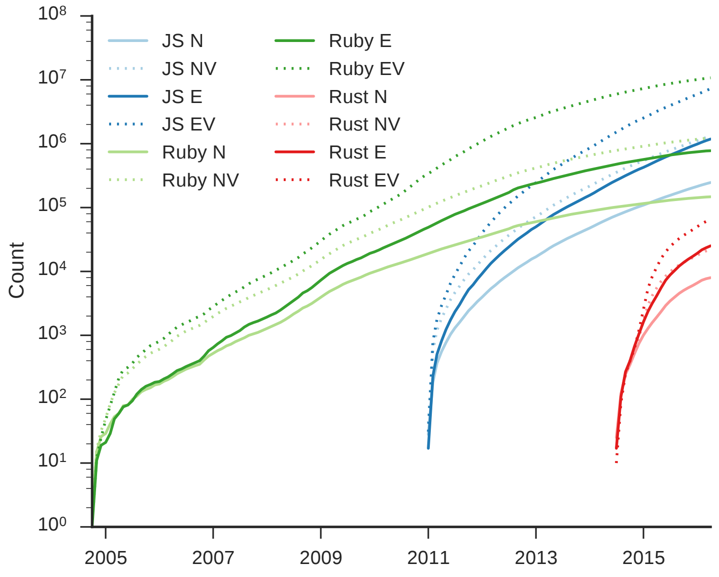


Figure 19. Number of unique projects (N), dependencies between projects (E), the number of versions (NV) and dependencies between versions (EV).

6.5.2. Dependency network evolution (RQ3.2)

In this subsection, we investigate the evolution of dependency networks in more detail.

General growth. To understand how the ecosystems are growing, we first analyzed the number of projects and dependency relations between them. Figure 19 shows the number of projects and unique relations in the dependency network. We also show the number of releases and the number of dependency links between them. We see that in almost all cases, the speed at which the number of relations is growing is getting faster compared to the number of nodes in the network, especially visible for JavaScript (JS N on Figure 19), where the difference between the number of projects and dependencies is tenfold. The figure also indicates that the growth of Rust is still continuing. JavaScript has become larger than Ruby, both in terms of versions and dependencies between versions, and its growth speed is faster than Ruby. The growth of Ruby has been been continuing at a constant steady rate since 2012, whereas JavaScript is in fact growing at an accelerated rate.

Figure 19 highlights the size differences when analyzing actual networks and aggregated networks with annotated edges. There is a more than tenfold difference between the number of nodes and edges in both networks and the difference is growing. Therefore, there are differences in the network structure, which confirms our initial discussion on the choice of network modeling approach.

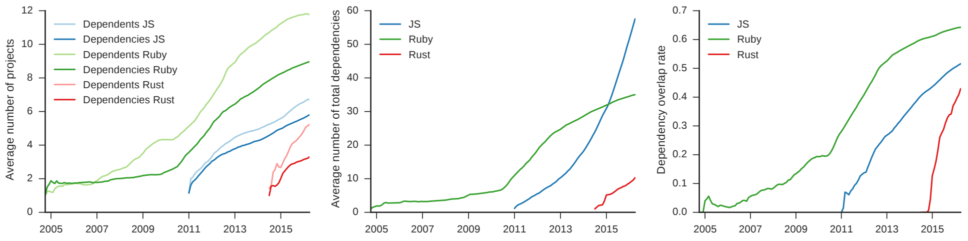
As the ecosystem is composed of multiple projects, we next analyzed the project-level changes in dependencies. We sought to determine what the number of dependencies and dependents for projects and the full size of the transitive dependency chain was. Figure 20a shows the number of dependencies and dependents for each project release. We see a faster growth for the number of dependents in Ruby and JavaScript. The number of dependencies has been growing at a slower rate. When comparing JavaScript and Ruby, we see that the difference between the number of dependents is larger than the number of dependencies. One possible explanation could be that the overall number of packages published in `RubyGems` is smaller than in `npm` and there are fewer alternatives for packages, leading to a higher number of dependents.

Figure 20b shows the total number of dependencies for each project release. We observe fast growth for JavaScript projects and slower, steadier growth for Ruby and Rust projects. The average size of total dependencies for JavaScript was 34.3 in April 2015 but grew to 54.6 in April 2016, more than 60% yearly growth. Growth at such a speed is unlikely to continue and most likely will be lower in the future.

When comparing JavaScript's and Ruby's numbers of direct dependencies (Figure 20a) and the total number of transitive dependencies, we see that JavaScript projects have more transitive dependencies, but fewer direct dependencies. This behavior indicates differences between these two ecosystems. Ruby has packages that are used mostly by applications and do not have dependencies, but packages published by JavaScript do have dependencies, making the ecosystem more connected and complex. One possible explanation for JavaScript's larger number of transitive dependencies can be that JavaScript developers are more open to third-party code usage and the standard library lacks functionality, resulting in usage of micro-packages (See Section 6.5.1 for examples).

Judging by these observations, it is hard to predict the exact number of transitive dependencies for Rust as both Ruby and JavaScript have shown different behavior. We argue that this may be because Rust is a very new ecosystem at its initial stages of evolution.

Conflict evolution. The ecosystems keep growing and the number of dependencies between projects is also growing. We next analyze projects that have a single dependency included through multiple packages, which could lead to conflicts if the package version requirement specification does not match.



(a) Evolution of the number of direct dependencies and dependents for each project version. Average over those that have at least one dependency or one dependent.

(b) Average number of total dependencies, including the full transitive closure. Average calculated over projects that have at least one dependency.

(c) Dependency overlap evolution. Fraction of projects having at least one dependency required through multiple projects.

Figure 20. Dependency network evolution.

We define a dependency overlap as a situation where a project appears as a dependency through multiple different paths for a single project. In practice, overlap could lead to conflict, which would occur only if the version specification did not match and it was not possible to find the best matching version. Dependency overlap illustrates how much dependencies are co-used in projects. On the other hand, it illustrates the need for consistent usage of version number specification by package maintainers. Increasing dependency overlap should give developers a signal to look at their dependency version requirements and use as loose of criteria as possible in order to allow dependency managers to find a suitable version.

Figure 20c lists the proportion of projects that have dependency overlap in their dependency chains. The general trends are similar to the overall growth of the ecosystems. More than two-thirds of Ruby and half of JavaScript projects have a single dependency appear through multiple dependency chains. The result indicates package reuse, but the chance of dependency version conflicts might become more likely. Increasing overlap can lead to issues which prevent different packages from being used together due to dependencies that cannot be satisfied. Similar behavior has been observed for Debian software packages [2].

6.5.3. Fragility and vulnerability (RQ3.3)

Next we analyze the tolerance of dependency networks to the removal of a single project or a single release. We define the vulnerability of a package as the fraction of the network nodes that are impacted by the removal of a single package or a single package version. This approach enables us to analyze the impact of incidents such as the *left-pad* project removal. While complete removal of a project removes all versions from the dependency networks, we can also study the removal of a single version. For example, bugs or security vulnerabilities might not impact all project versions as only specific ones might contain the bug.

We first calculate the vulnerability of the network where each node denotes the different version. For each package version, we calculate the total number of dependents. Next, we have the list with the total number of dependents for all packages. Within this list, we look at the maximum value and the 90th percentile value. We chose these values as the distribution of the number of dependents is skewed and the median value is typically either 0 or 1 depending on the snapshot date.

Figure 21a shows the maximum and 90th percentile vulnerability score normalized with respect to the full network size at each snapshot. We see that the maximum is fluctuating and having a positive trend, which means that there is a version in the network whose importance is growing. Looking at the 90th percentile value, we see a decreasing trend, which indicates that most of the other packages in the ecosystem are not central and are not included in the majority of dependency paths.

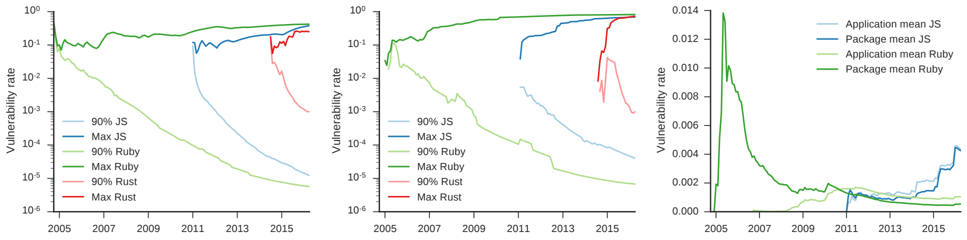
We also look at the vulnerability on the aggregated graph. Figure 21b shows the same vulnerability calculation on the aggregated network, meaning we remove a project and all its versions. It is evident that the maximum score is growing and the impact of every project is growing. This is even interesting in the context of growing ecosystems, whose absolute values are also increasing. The 90th percentile vulnerability is again decreasing.

To find the differences between packages and applications, we analyzed the mean vulnerability rate for different types of JavaScript and Ruby projects. Figure 21c shows the average number of projects affected by a single package removal. The figure illustrates the dependence on a single package. We see that right after the creation of the package ecosystem, it starts to decrease. In a later phase, the positive trend of JavaScript is more visible. The average number of impacted applications remains larger than the packages.

Table 11 lists the top five releases based on unique dependent projects and unique dependent releases. For JavaScript, the list is composed of unique utility packages, such as array or inherits. For Ruby and Rust we see that multiple versions of single packages have made it to the top lists. The top five packages for Ruby are related to webserver (rack) or templates (erubis, tilt). Rust packages are an interface to system-level types and libraries (libc), a serialization library (rustc-serialize), and a logging library (log).

6.6. An example of a critical bug-fix release adoption

To demonstrate how dependency network analysis could help to understand the adoption of a bug-fix release for an existing package, we conduct a case study on a reported vulnerability for the Ruby package Rack. Rack is a library for building web applications in Ruby and is used by most Ruby web frameworks and web servers. It is also one of the most popular packages according to Table 11. The vulnerability CVE-2015-3225 [43] enables an attacker to craft request parameters that would crash the web application, essentially enabling denial-of-service-type



(a) Relative vulnerability with respect to the actual dependency network size. The network size is the number of releases.

(b) Relative vulnerability with respect to the total number of unique projects in the network.

(c) Relative vulnerability among published packages and GitHub projects.

Figure 21. Vulnerability of the ecosystems.

Table 11. Top packages based on the number transitive dependents (unique project releases).

Package	**Direct dependents**	**Transitive dependents**	**Vulnerability**
	JavaScript		
inherits 2.0.1	8131	499254	0.38
isarray 0.0.1	727	384907	0.29
core-util-is 1.0.1	524	371871	0.28
string_decoder 0.10.25	39	303116	0.23
sigmund 1.0.0	256	283319	0.21
	Ruby		
erubis 2.7.0	9014	519555	0.42
rack 1.4.1	4707	490329	0.40
rack-test 0.6.2	1566	386937	0.31
rack 1.3.10	3248	362810	0.29
tilt 1.3.3	2084	359862	0.29
	Rust		
libc 0.1.1	44	5520	0.25
rustc-serialize 0.3.16	1651	5379	0.24
libc 0.2.1	141	4840	0.22
libc 0.1.4	79	4598	0.21
log 0.3.1	1030	4415	0.20

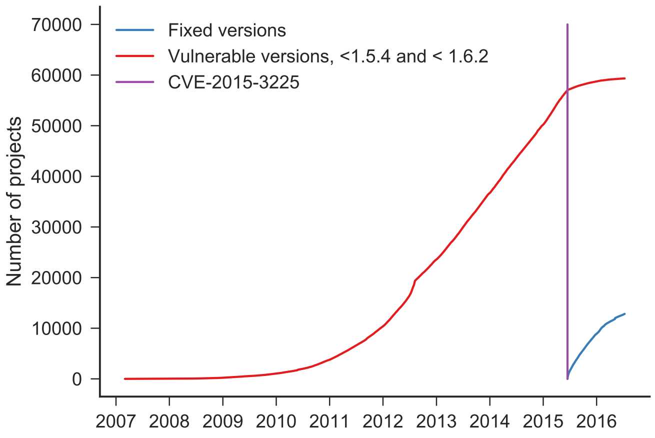


Figure 22. Adoption of vulnerable `Rack` versions through direct and transitive dependencies. The vertical purple line indicates release of the public Common Vulnerabilities and Exposures (CVE) notification.

attacks.

The vulnerability disclosure was released to the public on 16 June 2015, but the fixed versions were released on 12 June 2015. Figure 22 shows the adoption of vulnerable and fixed versions through direct and transitive dependencies. We can see that the adoption of fixed versions started right after their release. At the same time, vulnerable versions were still being adopted by projects, although at a lower rate than compared to the fixed versions. In total, 2,324 projects that had not used Rack before, adopted one of the vulnerable versions after the CVE was published. By the end of our observation period in July 2016, one year after the vulnerability disclosure, only about 20.7% of the projects had migrated away from the vulnerable versions (12326 projects out of 59336 projects that had adopted the vulnerable version before).

6.7. Discussion

Below, we discuss our results, their practical implications, compare them with related work, and outline the limitations of our research. The results differ to some extent for all studied languages, but generalizations can still be made.

6.7.1. Results

Network modeling. Previous research on package dependency networks has not reached a consensus on how to model dependencies using graphs. We propose an approach for modeling and constructing the network from dependency data. We believe that the chosen approach captures the actual network most accurately, enabling us to analyze dependencies on their version level. Although the analysis of aggregated network can yield similar conclusions, the real dependencies are still using version information and in future evolution stages this might not be sufficient anymore. We believe that our contribution in network modeling is a single step forward towards a more unified system of software dependency network modeling.

Structure. Analysis of dependency network structure reveals differences between ecosystems. Although this has been observed before [47] for the dependencies, we have also shown differences in dependency version constraint specifications across ecosystems. The findings complement previous research [26] that found that different ecosystems approached API changes differently, which could impact dependency management. Our findings indicate that there are more implicit than explicit version updates, which suggests that there may be a need for tools to automatically monitor the dependencies that are included through implicit updates and reveal possible breaking API changes. Alternatively, monitoring implicit updates could be incorporated into the build process to notify developers about the changes in the dependencies to raise awareness.

Evolution. Our evolution analysis revealed that the number of transitive dependencies for JavaScript projects has grown over 60% over the past year. A large number of dependencies can lead to issues such as extended build time because of fetching the dependencies and increased software package size. Exponential growth has been observed inside the Apache ecosystem as well [15]. Recently, a newer dependency management tool compatible with `npm` was introduced [141]. One of its main new functions is an improved concurrent dependency downloading ability. The tool tries to solve the dependency abundance problem by providing a faster download. Alternatively, a future solution could be to study how to reduce dependencies through better static code analysis. Function-call level analysis could help to eliminate unneeded dependencies if the corresponding code is not invoked. Our finding illustrates that by observing network evolution, such troubles can be anticipated. Analysis of trends and number of transitive dependencies over time could be useful for other package-based language ecosystems.

Vulnerability. Our vulnerability analysis, inspired by the *left-pad* incident [91], reveals that each studied ecosystem has packages whose removal could impact up to 30% of the other packages and applications. We showed that ecosystems have a few central packages that they depend on, which could enable bug spreading if they are not up to date. The high vulnerability score of a package should also alert developers and maintainers to make sure all security bugs are fixed quickly. A package with a high vulnerability score can be of interest to attackers as an opportunity to exploit projects depending on it.

6.7.2. Design implications

By using our findings, one could design a better package ecosystem and better dependency management tooling. First, we would propose making dependency relations explicitly visible to understand the importance of packages in the ecosystems. Having an up-to-date view of which packages are most popular and important in the ecosystem can ensure they receive maintenance and support effort from the community. Repositories such as `npm` and `Crates` give statistics about the popularity of packages and list dependents for each package already today. But they still lack fine-grained information about transitive dependents and the list of other packages that a package is frequently installed together with. This information could make developers aware of the package usage-patterns and update dependencies accordingly, i.e., to support smooth updates for its most popular dependents and to reduce the likelihood for co-installability issues [131].

We would also investigate alternatives to semantic versioning. Our findings revealed that exact version notation is used frequently for Rust packages. Specifying exact versions guarantees reproducible builds, but does not allow automated updates. The versioning system and package manager could automatically suggest overriding strict versioning, if it detects it is possible to update packages without introducing breaking changes. Overall, the ecosystem and tooling should improve awareness of what dependencies are used, make dependency listing explicit, help to minimize irrelevant dependencies and at the same time support automated upgrading of dependencies for non-breaking changes.

6.7.3. Limitations

The limitation of our dependency network construction approach is that it will not compose the exact representation that the build tool would have. When resolving wildcard version specifications with a matching version, we look at all dependencies separately for given projects. In practice, when using build tools, the whole transitive closure of dependencies would be resolved if a package is included through multiple paths and a matching version was calculated that shares all requirements. To recreate the exact dependencies for a project historically is complicated as dependency management tools do not support backdated retrieval.

6.8. Replication package

Datasets and source code used in this chapter are available at
https://github.com/riivo/package-dependency-networks

6.9. Summary

The main contributions of this chapter are: (i) an approach to extract dependency networks from public (open-source) repositories; and (ii) an analysis of the dependency networks of JavaScript, Ruby, and Rust. The latter analysis shows that these ecosystems are alive and growing, with JavaScript having the fastest growth. JavaScript also exhibits the largest number of transitive dependencies per project among the studied languages. All ecosystems have a subset of popular packages used in the majority of projects. Yet, over time, these ecosystems have become less dependent on a single popular package such that the removal of a random project would not impact the whole ecosystem, but still impact up to one third of the ecosystem for some languages.

7. CONCLUSION AND OUTLOOK

Modern software development relies on open-source software to facilitate reuse and reduce redundant work. Developers use open-source packages in their projects. Even if developers are aware of the possible risks, they still might be lacking full insights into how these components are being developed and maintained. The main goal of this thesis has been to bring new insight into issue management and dependency management in the context of open-source software projects.

This thesis has analyzed data from GitHub projects following the mining software repositories approach. Empirical analysis of issue report data from more than 4,000 GitHub projects revealed trends about the growth of pending issues. To understand which issues get closed, we developed a machine-learning-based approach for predicting issue closing time. Inspired by the discovery of a growing number of pending issues, we analyzed dependency management with respect to dependency updates. We also developed an approach for the whole ecosystem-wide dependency network analysis to quantify possibilities of vulnerabilities spreading through non-maintained dependencies.

The approaches and findings developed here could help to bring transparency into open-source projects with respect to how issues are handled or dependencies are updated. The thesis highlights the risk of acquiring vulnerabilities through software packages.

7.1. Contributions and findings

In the introduction we formulated three broad research questions. In this section, we summarize our contributions and findings.

7.1.1. Dynamics of issue lifetime

In this thesis we empirically analyzed issue handling for more than 4,000 open-source projects on GitHub. Our findings revealed that a fraction of issues opened in open-source projects remain open for long periods of time. At the same time, the mean issue resolution time remains stable over the project lifetime. Stable issue resolution time and growth of pending issues indicates that some issues do not get attention and those that do get it are usually resolved in a matter of days. The approach developed for analysis is generalizable for analyzing and quantifying issue accumulation in software projects.

7.1.2. Predicting issue lifetime

Our findings about pending issues led us to build a model to predict issue lifetime. We developed a machine-learning-based method for temporal prediction of issue lifetime in GitHub projects. We demonstrated that predictive models exhibit better accuracy when trained with one-day-old or one-week-old issues to predict whether or not an issue will remain open after a one month or longer period. Our approach utilizes dynamic and contextual features of the project and issues, and shows how recalculating these features over issue lifetime can lead to better results. In addition, we identified that different sets of features are important for different prediction tasks.

7.1.3. Characteristics of open-source package ecosystems

We developed a method for analyzing package ecosystems based on network analysis. Software projects include packages to reuse code and thereby form a dependency relationship. We model dependency relationships as a network and analyze how the networks have evolved for three large ecosystems: JavaScript, Ruby, and Rust. By including both packages from respective package manager and end-user projects, we have an overview of ecosystem growth and package update frequencies. We developed a notion of ecosystem vulnerability – what fraction of the ecosystem is vulnerable to a removal or infection of a single package. We show that in the `npm` package ecosystem the amount of transitive dependencies is increasing and the vulnerability to the removal of a single package is in general decreasing. The developed approach is generic and usable for monitoring the structure and evolution of different software packaging ecosystems.

7.2. Opportunities for future work

This work opens up multiple possibilities for lines of future work, which we outline below.

7.2.1. Issue lifetime prediction

Our issue analysis did not distinguish between issue types. Future work should look more into the semantics behind issues and distinguish between issue categories such as bugs or feature requests and how they might impact results. There is evidence that the issue type is often incorrectly entered into issue tracking systems [72], yet at the same time the issue type can make a difference in prediction models [11]. Future work should also clarify if there is a need for issue type information in such kind of analysis and if it is possible to infer it via machine-learning approaches. The next step would be to study how issue growth actually impacts projects in terms of code changes, new releases or even project popularity. Furthermore, future work should look into how to detect reasons why issues are not being resolved. Issues might not be worked on because there is lack of resources, their priority or severity is low or the issue report is lacking details. Incorporating the reason into the output of the issue life-time prediction model could give stakeholders more context and make it more useful.

The predictive models studied in this paper benefit from being trained on a large dataset. The drawback of this asset is that the set of projects in the dataset are very heterogeneous, making the prediction problem more difficult. One possible direction for future work would be to study the performance of predictive models trained for specific types of projects (i.e. partial classifiers), such as to strike a tradeoff between volume of projects and homogeneity.

Another direction for future work would be to extend the feature set with more dynamic and contextual features, such as features extracted from the text of the comments added during the lifetime of an issue, or features capturing how busily the developers are handling other concurrent issues in the same or in other projects.

7.2.2. Dependency analysis

The dependency management process should be studied qualitatively to understand issues developers are facing. There is lack of studies on how a measure quantifying *dependency health* in an ecosystem should be developed, by combining network data with data about testing efforts, code analysis, number of maintainers, etc.

This work takes a high-level view of dependencies. Future work should go into lower-level details, such as function calls. When including a dependency, only subset of its code and transitive dependencies would be used depending on which functions are invoked. Function-call level analysis would enable to reveal which dependencies and code-paths are actually invoked in a project. This would enable to analyze ecosystem-wide data-flow and would give information to package maintainers about how their packages are used. In addition, it could help developers understand if they are impacted by a possible security issue if security notices indicate impacted functions. Furthermore, this would enable to enforce proper semantic versioning, which in turn could enable automated updates for bug fix releases that do not change APIs.

7.2.3. Better tooling for issue and dependency management

The methods developed in this thesis for issue lifetime characterization and prediction, and dependency analysis can be turned into tools for supporting developers. There are challenges that need to be solved, such as tailoring the tools such that they fit into the development workflow and study how to make stakeholders adopt the tooling. For example, our issue lifetime prediction model can give support for issue prioritization and insights for the submitter. Alternatively, future research could lean towards automated actions based on issue lifetime predictions, e.g., closing the issue if the model predicts it will not be resolved in the upcoming year and has not seen any activity recently. The computation cost for such automated tooling is feasible, the biggest limitation is the requirement of having real-time access to data, which is typically only possible for organizations running the repository, such as GitHub and `npm`.

Future research should also focus on dependency management tooling. The general goal of future research is to support developers with tools in dependency management and maintenance and provide analytics to maintainers about their packages and the overall ecosystem trends. Automated dependency updates are a reality today, but understanding what they might break is still unclear to developers. Automated tooling to understand test coverage and potential pitfalls for upgrading dependencies is needed.

7.3. Closing remarks

Software development is a predominantly human-centered activity and will remain so in the near future. Providing tools that support developers and provide insights into projects enables developers to deliver complex software projects faster and with better quality. Project and product improvement, however, can only be achieved through a better understanding of the underlying processes. This thesis contributes to a better understanding of issue and dependency management in open-source projects.

BIBLIOGRAPHY

[1] Pietro Abate, Roberto Di Cosmo, Jaap Boender, and Stefano Zacchiroli. Strong dependencies between software components. In *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*, ESEM '09, pages 89–99, Washington, DC, USA, 2009. IEEE Computer Society.

[2] Pietro Abate, Roberto Di Cosmo, Louis Gesbert, Fabrice Le Fessant, Ralf Treinen, and Stefano Zacchiroli. Mining component repositories for installability issues. In *Mining Software Repositories (MSR), 2015 IEEE/ACM 12th Working Conference on*, pages 24–33. IEEE, 2015.

[3] Réka Albert, Hawoong Jeong, and Albert-László Barabási. Error and attack tolerance of complex networks. *Nature*, 406(6794):378, 2000.

[4] Ethem Alpaydin. *Introduction to machine learning*. MIT press, 2014.

[5] Sven Amann, Sebastian Proksch, Sarah Nadi, and Mira Mezini. A study of Visual Studio usage in practice. In *Software Analysis, Evolution, and Reengineering (SANER), 2016 IEEE 23rd International Conference on*, volume 1, pages 124–134. IEEE, 2016.

[6] Giuliano Antoniol, Kamel Ayari, Massimiliano Di Penta, Foutse Khomh, and Yann-Gaël Guéhéneuc. Is it a bug or an enhancement?: A text-based approach to classify change requests. In *Proceedings of the 2008 Conference of the Center for Advanced Studies on Collaborative Research: Meeting of Minds*, CASCON '08, pages 23:304–23:318. ACM, 2008.

[7] John Anvik, Lyndon Hiew, and Gail C Murphy. Coping with an open bug repository. In *Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, pages 35–39. ACM, 2005.

[8] John Anvik, Lyndon Hiew, and Gail C. Murphy. Who should fix this bug? In *28th International Conference on Software Engineering (ICSE 2006), Shanghai, China, May 20-28, 2006*, pages 361–370, 2006.

[9] Jorge Aranda and Steve Easterbrook. *Anchoring and Adjustment in Software Estimation*. ESEC/FSE-13. ACM, New York, NY, USA, 2005.

[10] Saïd Assar, Markus Borg, and Dietmar Pfahl. Using text clustering to predict defect resolution time: A conceptual replication and an evaluation of prediction accuracy. *Empirical Software Engineering*, pages 1–39, 2015.

[11] Dimitrios Athanasiou, Ariadi Nugroho, Joost Visser, and Andy Zaidman. Test code quality and its relation to issue handling performance. *IEEE Transactions on Software Engineering*, 40(11):1100–1125, 2014.

[12] Alberto Bacchelli and Christian Bird. Expectations, outcomes, and challenges of modern code review. In *Proceedings of the 2013 international conference on software engineering*, pages 712–721. IEEE Press, 2013.

[13] Alberto Bacchelli, Michele Lanza, and Romain Robbes. Linking e-mails and source code artifacts. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 375–384. ACM, 2010.

[14] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. The evolution of project inter-dependencies in a software ecosystem: The case of Apache. In *ICSM*, pages 280–289, 2013.

[15] Gabriele Bavota, Gerardo Canfora, Massimiliano Di Penta, Rocco Oliveto, and Sebastiano Panichella. How the Apache community upgrades dependencies: An evolutionary study. *Empirical Software Engineering*, 20(5):1275–1317, 2015.

[16] Olga Baysal, Reid Holmes, and Michael W Godfrey. Developer dashboards: The need for qualitative analytics. *IEEE software*, 30(4):46–52, 2013.

[17] Olga Baysal, Reid Holmes, and Michael W Godfrey. No issue left behind: Reducing information overload in issue tracking. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 666–677. ACM, 2014.

[18] Moritz Beller, Georgios Gousios, Annibale Panichella, and Andy Zaidman. When, how, and why developers (do not) test in their IDEs. In *Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2015, pages 179–190, New York, NY, USA, 2015. ACM.

[19] Moritz Beller, Georgios Gousios, and Andy Zaidman. Oops, my tests broke the build: An explorative analysis of Travis CI with GitHub. In *Proceedings of the 14th International Conference on Mining Software Repositories*, pages 356–367. IEEE press, 2017.

[20] Moritz Beller, Georgios Gousios, and Andy Zaidman. Travistorrent: Synthesizing Travis CI and GitHub for full-stack research on continuous integration. In *Mining Software Repositories (MSR), 2017 IEEE/ACM 14th International Conference on*, pages 447–450. IEEE, 2017.

[21] Dane Bertram, Amy Voida, Saul Greenberg, and Robert Walker. Communication, collaboration, and bugs: The social nature of issue tracking in small, collocated teams. In *Proceedings of the ACM Conference on Computer Supported Cooperative Work, CSCW*, pages 291–300, Savannah, Georgia, USA, 2010. ACM.

[22] Pamela Bhattacharya and Iulian Neamtiu. Bug-fix time prediction models: Can we do better? In *Proceedings of the 8th Working Conference on Mining Software Repositories*, MSR '11, pages 207–210. ACM, 2011.

[23] Christian Bird, Alex Gourley, Prem Devanbu, Michael Gertz, and Anand Swaminathan. Mining email social networks. In *Proceedings of the 2006 international workshop on Mining software repositories*, pages 137–143. ACM, 2006.

[24] Tegawende F Bissyande, David Lo, Lingxiao Jiang, Laurent Reveillere, Jacques Klein, and Yves Le Traon. Got issues? Who cares about it? A large scale investigation of issue trackers from GitHub. In *Software Reliability Engineering (ISSRE), 2013 IEEE 24th International Symposium on*, pages 188–197. IEEE, 2013.

[25] C. Bogart, C. Kästner, and J. Herbsleb. When it breaks, it breaks: How ecosystem developers reason about the stability of dependencies. In *2015 30th IEEE/ACM International Conference on Automated Software Engineering Workshop (ASEW)*, pages 86–89, Nov 2015.

[26] Christopher Bogart, Christian Kästner, James Herbsleb, and Ferdian Thung. How to break an API: Cost negotiation and community values in three software ecosystems. In *Proceedings of the ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE)*, FSE '16. ACM Press, 11 2016.

[27] Léon Bottou. Large-scale machine learning with stochastic gradient descent. In *Proceedings of COMPSTAT'2010*, pages 177–186. Springer, 2010.

[28] L. Breiman, J. Friedman, R. Olshen, and C. Stone. *Classification and Regression Trees*. Wadsworth and Brooks, Monterey, CA, 1984.

[29] Leo Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[30] Bugzilla. `https://www.bugzilla.org/`. Last accessed 12.05.2017.

[31] Raymond PL Buse and Thomas Zimmermann. Analytics for software development. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 77–80. ACM, 2010.

[32] Jordi Cabot, Javier Luis Canovas Izquierdo, Valerio Cosentino, and Belén Rolandi. Exploring the use of labels to categorize issues in open-source software projects. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 550–554. IEEE, 2015.

[33] M. Cadariu, E. Bouwers, J. Visser, and A. van Deursen. Tracking known security vulnerabilities in proprietary software systems. In *2015 IEEE 22nd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, pages 516–519, March 2015.

[34] Robert N Charette. This car runs on code. *IEEE spectrum*, 46(3):3, 2009.

[35] Morakot Choetkiertikul, Hoa Khanh Dam, Truyen Tran, and Aditya Ghose. Predicting delays in software projects using networked classification. In *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*, pages 353–364, 2015.

[36] Morakot Choetkiertikul, Hoa Khanh Dam, Truyen Tran, Aditya Ghose, and John Grundy. Predicting delivery capability in iterative software development. *IEEE Transactions on Software Engineering*, 2017.

[37] Maelick Claes, Tom Mens, Roberto Di Cosmo, and Jérôme Vouillon. A historical analysis of Debian package incompatibilities. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 212–223. IEEE Press, 2015.

[38] Mike Cohn. *User stories applied: For agile software development*. Addison-Wesley Professional, 2004.

[39] Valerio Cosentino, Javier L Cánovas Izquierdo, and Jordi Cabot. A systematic mapping study of software development with GitHub. *IEEE Access*, 5:7173–7192, 2017.

[40] David R Cox. The regression analysis of binary sequences. *Journal of the Royal Statistical Society. Series B (Methodological)*, pages 215–242, 1958.

[41] J. Cox, E. Bouwers, M. v. Eekelen, and J. Visser. Measuring dependency freshness in software systems. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 109–118, May 2015.

[42] Kevin Crowston, Hala Annabi, and James Howison. Defining open source software project success. *ICIS 2003 Proceedings*, page 28, 2003.

[43] Cve-2015-3225: Potential denial of service vulnerability in rack. `https://www.cvedetails.com/cve/CVE-2015-3225/`.

[44] Jacek Czerwonka, Nachiappan Nagappan, Wolfram Schulte, and Brendan Murphy. Codemine: Building a software development data analytics platform at microsoft. *IEEE software*, 30(4):64–71, 2013.

[45] Laura Dabbish, Colleen Stuart, Jason Tsay, and Jim Herbsleb. Social coding in GitHub: Transparency and collaboration in an open software repository. In *Proceedings of the ACM 2012 conference on Computer Supported Cooperative Work*, pages 1277–1286. ACM, 2012.

[46] Marco D'Ambros, Michele Lanza, and Romain Robbes. An extensive comparison of bug prediction approaches. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 31–41. IEEE, 2010.

[47] Alexandre Decan, Tom Mens, and Maelick Claes. On the topology of package dependency networks: A comparison of three programming language ecosystems. In *European Conference on Software Architecture Workshops*, 2016.

[48] Alexandre Decan, Tom Mens, and Maëlick Claes. An empirical comparison of dependency issues in OSS packaging ecosystems. In *IEEE 24th International Conference on Software Analysis, Evolution and Reengineering, SANER 2017, Klagenfurt, Austria, February 20-24, 2017*, pages 2–12, 2017.

[49] Alexandre Decan, Tom Mens, Maelick Claes, and Philippe Grosjeanm. When GitHub meets CRAN: An analysis of inter-repository package de-

pendency problems. In *23rd IEEE International Conference on Software Analysis, Evolution, and Reengineering*, 2016.

[50] Roberto Di Cosmo, Berke Durak, Xavier Leroy, Fabio Mancinelli, and Jérôme Vouillon. Maintaining large software distributions: New challenges from the FOSS era. In *Proceedings of the FRCSS 2006 workshop.*, pages 7–20. EASST, 2006.

[51] Roberto Di Cosmo, Stefano Zacchiroli, and Paulo Trezentos. Package upgrades in FOSS distributions: Details and challenges. In *Proceedings of the 1st International Workshop on Hot Topics in Software Upgrades*, HotSWUp '08, pages 7:1–7:5. ACM, 2008.

[52] David Easley and Jon Kleinberg. *Networks, crowds, and markets: Reasoning about a highly connected world.* Cambridge University Press, 2010.

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

[54] Manuel Fernández-Delgado, Eva Cernadas, Senén Barro, and Dinani Amorim. Do we need hundreds of classifiers to solve real world classification problems? *J. Mach. Learn. Res.*, 15(1):3133–3181, 2014.

[55] P. Francis and L. Williams. Determining 'Grim Reaper'; Policies to prevent languishing bugs. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 436–439, 2013.

[56] Vahid Garousi. Evidence-based insights about issue management processes: An exploratory study. In *Trustworthy Software Development Processes*, pages 112–123. Springer, 2009.

[57] Daniel M German, Bram Adams, and Ahmed E Hassan. The evolution of the R software ecosystem. In *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*, pages 243–252. IEEE, 2013.

[58] Emanuel Giger, Martin Pinzger, and Harald Gall. Predicting the fix time of bugs. In *Proceedings of the 2nd International Workshop on Recommendation Systems for Software Engineering*, RSSE '10, pages 52–56. ACM, 2010.

[59] Alessandra Gorla, Ilaria Tavecchia, Florian Gross, and Andreas Zeller. Checking app behavior against app descriptions. In *Proceedings of the 36th International Conference on Software Engineering*, pages 1025–1035. ACM, 2014.

[60] Georgios Gousios. The GHTorrent dataset and tool suite. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 233–236, 2013.

[61] Georgios Gousios, Martin Pinzger, and Arie van Deursen. An exploratory study of the pull-based software development model. In *Proceedings of the*

36th International Conference on Software Engineering, ICSE 2014, pages 345–355. ACM, 2014.

[62] Georgios Gousios, Andy Zaidman, Margaret-Anne Storey, and Arie van Deursen. Work practices and challenges in pull-based development: The integrator's perspective. In *Proceedings of the 37th International Conference on Software Engineering - Volume 1*, ICSE '15, pages 358–368. IEEE Press, 2015.

[63] Lars Grammel, Holger Schackmann, Adrian Schröter, Christoph Treude, and Margaret-Anne Storey. Attracting the community's many eyes: An exploration of user involvement in issue tracking. In *Human Aspects of Software Engineering*, page 3. ACM, 2010.

[64] Philip J. Guo, Thomas Zimmermann, Nachiappan Nagappan, and Brendan Murphy. "Not my bug!" and other reasons for software bug report reassignments. In *Proceedings of the ACM 2011 Conference on Computer Supported Cooperative Work*, CSCW '11, pages 395–404. ACM, 2011.

[65] P.J. Guo, T. Zimmermann, N. Nagappan, and B. Murphy. Characterizing and predicting which bugs get fixed: An empirical study of Microsoft Windows. In *Software Engineering, 2010 ACM/IEEE 32nd International Conference on*, volume 1, pages 495–504, 2010.

[66] Emitza Guzman and Walid Maalej. How do users like this feature? A fine grained sentiment analysis of app reviews. In *Requirements Engineering Conference (RE), 2014 IEEE 22nd International*, pages 153–162. IEEE, 2014.

[67] Shi Han, Yingnong Dang, Song Ge, Dongmei Zhang, and Tao Xie. Performance debugging in the large via mining millions of stack traces. In *Proceedings of the 34th International Conference on Software Engineering*, pages 145–155. IEEE Press, 2012.

[68] Ahmed E Hassan. The road ahead for mining software repositories. In *Frontiers of Software Maintenance, 2008. FoSM 2008.*, pages 48–57. IEEE, 2008.

[69] Ahmed E Hassan and Tao Xie. Software intelligence: The future of mining software engineering data. In *Proceedings of the FSE/SDP workshop on Future of software engineering research*, pages 161–166. ACM, 2010.

[70] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The Elements of Statistical Learning*. Springer New York, 2009.

[71] JI Hejderup. In dependencies we trust: How vulnerable are dependencies in software modules? Master's thesis, TU Delft, Delft University of Technology, 2015.

[72] Kim Herzig, Sascha Just, and Andreas Zeller. It's not a bug, it's a feature: How misclassification impacts bug prediction. In *Proceedings of the 2013*

International Conference on Software Engineering, ICSE '13, pages 392–401, Piscataway, NJ, USA, 2013. IEEE Press.

[73] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *Software Engineering (ICSE), 2012 34th International Conference on*, pages 837–847. IEEE, 2012.

[74] Petter Holme, Beom Jun Kim, Chang No Yoon, and Seung Kee Han. Attack vulnerability of complex networks. *Physical review E*, 65(5):056109, 2002.

[75] How a student fooled 17,000 coders into running his 'sketchy' programming code. `https://fossbytes.com/typosquatting-technique-used-by-student-tricks-17000-coders/`. Accessed: 2016-06-19.

[76] https://help.github.com/articles/about-pull-request-reviews/. `https://help.github.com/articles/about-pull-request-reviews/b/`, 2017. [Online; accessed 22-September-2017].

[77] Javier Luis Cánovas Izquierdo, Valerio Cosentino, Belén Rolandi, Alexandre Bergel, and Jordi Cabot. GiLA: GitHub label analyzer. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 479–483. IEEE, 2015.

[78] Yujuan Jiang, Bram Adams, and Daniel M. German. Will my patch make it? and how fast?: Case study on the linux kernel. In *Proceedings of the 10th Working Conference on Mining Software Repositories*, MSR '13, pages 101–110. IEEE Press, 2013.

[79] Zhen Ming Jiang, Ahmed E Hassan, Gilbert Hamann, and Parminder Flora. Automatic identification of load testing problems. In *Software Maintenance, 2008. ICSM 2008. IEEE International Conference on*, pages 307–316. IEEE, 2008.

[80] Jira. `https://www.atlassian.com/software/jira`. Last accessed 12.05.2017.

[81] Eirini Kalliamvakou, Georgios Gousios, Kelly Blincoe, Leif Singer, Daniel M. German, and Daniela Damian. The promises and perils of mining GitHub. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, MSR 2014, pages 92–101. ACM, 2014.

[82] Benedicte Kenmei, Giuliano Antoniol, and Massimiliano Di Penta. Trend analysis and issue prediction in large-scale open source systems. In *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*, pages 73–82. IEEE, 2008.

[83] Hammad Khalid, Emad Shihab, Meiyappan Nagappan, and Ahmed E Hassan. What do mobile app users complain about? *IEEE Software*, 32(3):70–77, 2015.

[84] Riivo Kikas, Marlon Dumas, and Dietmar Pfahl. Issue dynamics in GitHub projects. In *Product-Focused Software Process Improvement*, volume 9459

of *Lecture Notes in Computer Science*, pages 295–310. Springer International Publishing, 2015.

[85] Riivo Kikas, Marlon Dumas, and Dietmar Pfahl. Using dynamic and contextual features to predict issue lifetime in GitHub projects. In *Proceedings of the 13th International Conference on Mining Software Repositories*, MSR '16, pages 291–302, New York, NY, USA, 2016. ACM.

[86] Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. Structure and evolution of package dependency networks. In *Proceedings of the 14th International Conference on Mining Software Repositories*, MSR '17, pages 102–112, Piscataway, NJ, USA, 2017. IEEE Press.

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

[88] Ekrem Kocaguneli and Tim Menzies. Software effort models should be assessed via leave-one-out validation. *Journal of Systems and Software*, 86(7):1879–1890, 2013.

[89] Stephen Kokoska and Daniel Zwillinger. *CRC standard probability and statistics tables and formulae*. Crc Press, 1999.

[90] Raula Gaikovina Kula, Daniel M. Germán, Takashi Ishio, and Katsuro Inoue. Trusting a library: A study of the latency to adopt the latest maven release. In *22nd IEEE International Conference on Software Analysis, Evolution, and Reengineering, SANER 2015, Montreal, QC, Canada, March 2-6, 2015*, pages 520–524, 2015.

[91] left-pad issue #4. `https://github.com/stevemao/left-pad/issues/4`. Last accessed 25.01.2017.

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

[93] Huan Liu and R. Setiono. Chi2: feature selection and discretization of numeric attributes. In *Tools with Artificial Intelligence, 1995. Proceedings., Seventh International Conference on*, pages 388–391, 1995.

[94] Benjamin Livshits and Thomas Zimmermann. Dynamine: Finding common error patterns by mining software revision histories. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-13, pages 296–305, New York, NY, USA, 2005. ACM.

[95] B. Luijten, J. Visser, and A. Zaidman. Assessment of issue handling efficiency. In *Mining Software Repositories (MSR), 2010 7th IEEE Working Conference on*, pages 94–97, May 2010.

[96] Mircea F Lungu. *Reverse engineering software ecosystems*. PhD thesis, University of Lugano, 2009.

[97] Henry B Mann and Donald R Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50–60, 1947.

[98] Andrian Marcus, Jonathan I Maletic, and Andrey Sergeyev. Recovery of traceability links between software documentation and source code. *International Journal of Software Engineering and Knowledge Engineering*, 15(05):811–836, 2005.

[99] Lionel Marks, Ying Zou, and Ahmed E. Hassan. Studying the fix-time for bugs in large open source projects. In *Proceedings of the 7th International Conference on Predictive Models in Software Engineering*, Promise '11, pages 11:1–11:8. ACM, 2011.

[100] T. Mens and S. Demeyer. *Software Evolution*. SpringerLink: Springer e-Books. Springer Berlin Heidelberg, 2008.

[101] Dishant Mittal. Paper replication report: Using dynamic and contextual features to predict issue lifetime in github projects. *Private communications*, 2017.

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

[103] Parastoo Mohagheghi and Reidar Conradi. Quality, productivity and economic benefits of software reuse: A review of industrial studies. *Empirical Software Engineering*, 12(5):471–516, 2007.

[104] Nuthan Munaiah, Steven Kroh, Craig Cabrey, and Meiyappan Nagappan. Curating GitHub for engineered software projects. *Empirical Software Engineering*, 22(6):3219–3253, 2017.

[105] Alessandro Murgia, Giulio Concas, Roberto Tonelli, Marco Ortu, Serge Demeyer, and Michele Marchesi. On the influence of maintenance activity types on the issue resolution time. In *Proceedings of the 10th International Conference on Predictive Models in Software Engineering*, PROMISE '14, pages 12–21. ACM, 2014.

[106] Kevin P Murphy. *Machine Learning: A Probabilistic Perspective*. MIT Press, 2012.

[107] Mark Newman. *Networks: An introduction*. Oxford university press, 2010.

[108] NPM API. `https://registry.npmjs.org/-/all`. Accessed: 2016-05-01.

[109] Lucas D. Panjer. Predicting Eclipse bug lifetimes. In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, page 29. IEEE Computer Society, 2007.

[110] Gustavo Pinto, Fernando Castor, and Yu David Liu. Mining questions about software energy consumption. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 22–31. ACM, 2014.

[111] Luca Ponzanelli, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, and Michele Lanza. Mining StackOverflow to turn the IDE into a self-confident programming prompter. In *Proceedings of the 11th Working Conference on Mining Software Repositories*, pages 102–111. ACM, 2014.

[112] Simone Porru, Alessandro Murgia, Serge Demeyer, Michele Marchesi, and Roberto Tonelli. Estimating story points from issue reports. In *Proceedings of the The 12th International Conference on Predictive Models and Data Analytics in Software Engineering*, page 2. ACM, 2016.

[113] Probot stale. `https://github.com/probot/stale`. Last accessed 1.10.2018.

[114] J. Ross Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1993.

[115] Steven Raemaekers, Arie van Deursen, and Joost Visser. Semantic versioning versus breaking changes: A study of the Maven repository. In *Proceedings of the 2014 IEEE 14th International Working Conference on Source Code Analysis and Manipulation*, SCAM '14, pages 215–224, Washington, DC, USA, 2014. IEEE Computer Society.

[116] Marc J Rochkind. The source code control system. *IEEE Transactions on Software Engineering*, pages 364–370, 1975.

[117] RubyGems API. `https://rubygems.org/pages/data`. Accessed: 2016-05-01.

[118] R.K. Saha, S. Khurshid, and D.E. Perry. An empirical study of long lived bugs. In *Software Maintenance, Reengineering and Reverse Engineering (CSMR-WCRE), 2014 Software Evolution Week - IEEE Conference on*, pages 144–153, 2014.

[119] Toby Segaran. *Programming collective intelligence: building smart web 2.0 applications*. O'Reilly Media, Inc., 2007.

[120] Alexander Serebrenik and Tom Mens. Challenges in software ecosystems research. In *Proceedings of the 2015 European Conference on Software Architecture Workshops*, ECSAW '15, pages 40:1–40:6. ACM, 2015.

[121] Emad Shihab, Akinori Ihara, Yasutaka Kamei, Walid M. Ibrahim, Masao Ohira, Bram Adams, Ahmed E. Hassan, and Ken-ichi Matsumoto. Studying re-opened bugs in open source software. *Empirical Software Engineering*, 18(5):1005–1042, 2012.

[122] Chakkrit Tantithamthavorn, Shane McIntosh, Ahmed E Hassan, and Kenichi Matsumoto. Automated parameter optimization of classification techniques for defect prediction models. In *Software Engineering (ICSE),*

2016 IEEE/ACM 38th International Conference on, pages 321–332. IEEE, 2016.

[123] Patanamon Thongtanunam, Chakkrit Tantithamthavorn, Raula Gaikovina Kula, Norihiro Yoshida, Hajimu Iida, and Ken-ichi Matsumoto. Who should review my code? A file location-based code-reviewer recommendation approach for modern code review. In *Software Analysis, Evolution and Reengineering (SANER), 2015 IEEE 22nd International Conference on*, pages 141–150. IEEE, 2015.

[124] Ferdian Thung, David Lo, and Julia Lawall. Automated library recommendation. In *Reverse Engineering (WCRE), 2013 20th Working Conference on*, pages 182–191. IEEE, 2013.

[125] Yuan Tian, D. Lo, and Chengnian Sun. Drone: Predicting priority of reported bugs by multi-factor analysis. In *Software Maintenance (ICSM), 2013 29th IEEE International Conference on*, pages 200–209, 2013.

[126] Walter F Tichy. Design, implementation, and evaluation of a revision control system. In *Proceedings of the 6th international conference on Software engineering*, pages 58–67. IEEE Computer Society Press, 1982.

[127] P. Tripathy and K. Naik. *Software Evolution and Maintenance*. Wiley, 2014.

[128] Jason Tsay, Laura Dabbish, and James Herbsleb. Influence of social and technical factors for evaluating contribution in GitHub. In *Proceedings of the 36th International Conference on Software Engineering*, ICSE 2014, pages 356–366. ACM, 2014.

[129] Yoshimasa Tsuruoka, Jun'ichi Tsujii, and Sophia Ananiadou. Stochastic gradient descent training for l1-regularized log-linear models with cumulative penalty. In *Proceedings of the Joint Conference of the 47th Annual Meeting of the ACL and the 4th International Joint Conference on Natural Language Processing of the AFNLP: Volume 1 - Volume 1*, ACL '09, pages 477–485. Association for Computational Linguistics, 2009.

[130] Muhammad Usman, Emilia Mendes, Francila Weidt, and Ricardo Britto. Effort estimation in agile software development: A systematic literature review. In *Proceedings of the 10th International Conference on Predictive Models in Software Engineering*, pages 82–91. ACM, 2014.

[131] Jérôme Vouillon and Roberto Di Cosmo. On software component co-installability. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(4):34, 2013.

[132] Xiaoyin Wang, Lu Zhang, Tao Xie, John Anvik, and Jiasu Sun. An approach to detecting duplicate bug reports using natural language and execution information. In *Software Engineering, 2008. ICSE'08. ACM/IEEE 30th International Conference on*, pages 461–470. IEEE, 2008.

[133] Kilian Weinberger, Anirban Dasgupta, John Langford, Alex Smola, and Josh Attenberg. Feature hashing for large scale multitask learning. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 1113–1120. ACM, 2009.

[134] Cathrin Weiss, Rahul Premraj, Thomas Zimmermann, and Andreas Zeller. How long will it take to fix this bug? In *Proceedings of the Fourth International Workshop on Mining Software Repositories*, MSR '07, page 1. IEEE Computer Society, 2007.

[135] Peter Weißgerber, Daniel Neu, and Stephan Diehl. Small patches get in! In *Proceedings of the 2008 International Working Conference on Mining Software Repositories*, MSR '08, pages 67–76, New York, NY, USA, 2008. ACM.

[136] Erik Wittern, Philippe Suter, and Shriram Rajagopalan. A look at the dynamics of the JavaScript package ecosystem. In *Proceedings of the 13th International Workshop on Mining Software Repositories*, MSR '16, pages 351–361. ACM, 2016.

[137] Edmund Wong, Jinqiu Yang, and Lin Tan. Autocomment: Mining question and answer sites for automatic comment generation. In *Automated Software Engineering (ASE), 2013 IEEE/ACM 28th International Conference on*, pages 562–567. IEEE, 2013.

[138] Xin Xia, D. Lo, Xinyu Wang, Xiaohu Yang, Shanping Li, and Jianling Sun. A comparative study of supervised learning algorithms for re-opened bug prediction. In *Software Maintenance and Reengineering (CSMR), 2013 17th European Conference on*, pages 331–334, 2013.

[139] Xin Xia, David Lo, Emad Shihab, Xinyu Wang, and Bo Zhou. Automatic, high accuracy prediction of reopened bugs. *Automated Software Engineering*, 22(1):75–109, 2015.

[140] Tao Xie, Suresh Thummalapenta, David Lo, and Chao Liu. Data mining for software engineering. *Computer*, 42(8), 2009.

[141] Yarn: A new package manager for javascript. `https://code.facebook.com/posts/1840075619545360/yarn-a-new-package-manager-for-javascript/`. accessed 2016-10-27.

[142] Yue Yu, Huaimin Wang, Vladimir Filkov, Premkumar Devanbu, and Bogdan Vasilescu. Wait for it: Determinants of pull request evaluation latency on GitHub. In *Proceedings of the 12th Working Conference on Mining Software Repositories*, MSR '15, pages 367–371. IEEE Press, 2015.

[143] Yu Zhou, Yanxiang Tong, Ruihang Gu, and Harald Gall. Combining text mining and data mining for bug report classification. In *Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution*, ICSME '14, pages 311–320. IEEE Computer Society, 2014.

[144] Thomas Zimmermann, Rahul Premraj, and Andreas Zeller. Predicting defects for Eclipse. In *Proceedings of the third international workshop on predictor models in software engineering*, page 9. IEEE Computer Society, 2007.

[145] Thomas Zimmermann, Andreas Zeller, Peter Weissgerber, and Stephan Diehl. Mining version histories to guide software changes. *IEEE Transactions on Software Engineering*, 31(6):429–445, 2005.

ACKNOWLEDGEMENTS

I would like to express my gratitude and appreciation to my supervisors Marlon Dumas and Dietmar Pfahl. I am thankful for their constant encouragement, suggestions and feedback over all these years.

I am thankful to Georgios Gousios for hosting me at Radboud University, finding time to discuss ideas and guiding me in my research.

I would also like to thank my thesis reviewers Tom Mens and Andy Zaidman for their insightful feedback and comments.

I want to thank Tõnu Soots for helping to improve the readability of the thesis.

I would like to thank my parents for supporting me and encouraging me to work on my thesis. Their regular inquiries about the progress of the thesis was the best motivation to complete it.

I would also like to acknowledge those who awarded me with scholarships and provided funding for my research: IT Akadeemia Scholarhip, Estonian Doctoral School of Information and Communication Technology (IKTDK) Scholarhip, Estonian Research Council, and ERDF via the Software Technology and Applications Competence Centre.

SISUKOKKUVÕTE

Avatud lähtekoodiga tarkvaraprojektide vearaportite ja tehniliste sõltuvuste haldamise analüüsimine

Nüüdisaegses tarkvaraarenduses kasutatakse avatud lähtekoodiga tarkvara komponente, et vähendada korratava töö hulka. Avatud lähtekoodiga projektide hulga kasvuga kaasneb uusi katsumusi kogukonna panuste ja soovide haldamisel. Kasutajad saavad esitada uusi nõudeid, vearaporteid ja tööülesandeid, sundides arendajad nendega tegelema. Teisalt soovivad arendajad enda tööd lihtsustada, kasutades projektides kolmanda poole tarkvarateeke. Selles töös uuritakse kahte probleemi avatud lähtekoodiga arenduses: tööülesannete eluiga ja avatud lähtekoodiga tarkvara komponentide vaheliste sõltuvuste võrgustiku analüüsimist.

Veahaldussüsteeme kasutatakse lisaks tarkvaravigade haldamisele ka selleks, et kirjeldada arendusülesandeid ja uusi kasutuslugusid. Avatud lähtekoodiga projektides saab iga kasutaja lisada uue vearaporti või nõude, mis viib olukorrani, kus nõuete esitajaid on rohkem kui projektis osalisi. Selles töös uuritakse vearaporti käsitlemise ajalist dünaamikat 4000 avatud lähtekoodiga projekti näitel, mis on kogutud ühisarendusplatvormist GitHub. Konkreetsemalt uuritakse vearaportite loomise sagedust, avatud olekus raportite arvu, keskmist vearaporti eluiga ning nende muutumist ajas. Tulemused näitavad, et vearaporteid luuakse vahetult pärast projekti loomist tavapärasest rohkem ning avatud vearaportite arv kasvab aja jooksul pikalt lahendamata vearaportite tõttu. Keskmine raporti eluiga on projekti eluea vältel stabiilne.

Vearaporti lahtiolekuaja ennustamise meetodid aitavad projektijuhtidel prioriseerida tööd ning planeerida ressursse. Selles töös uuritakse, kuidas kaasata vearaporti eluea ennustamise mudelisse erinevat tüüpi andmeid, näiteks kommentaare, arendajate ja projekti aktiivsust. Töös arendatakse välja masinõppel baseeruv mudel, mille abil ennustatakse erinevatel ajahetkedel, kas antud vearaport suletakse mingis etteantud ajavahemikus. Tulemused näitavad, et dünaamilised ja kontekstipõhised tunnused on eriti olulised pikema perioodi ennustamisel ning erinevat tüüpi tunnuste olulisus muutub ennustusperioodi pikkuse järgi.

Doktoritöös uuritakse ka seda, kuidas tarkvaraprojektides tehnilisi sõltuvusi kasutatakse. Tarkvaraarendajad kasutavad varem väljaarendatud komponente, et kiirendada arendust ja vähendada korratava töö hulka. Samamoodi kasutavad spetsiifilised komponendid veel omakorda teisi komponente, misläbi moodustub komponentide vaheliste seoste kaudu sõltuvuste võrgustik. Selles töös arendatakse välja meetodid komponentidevahelise sõltuvuste võrgustiku struktuuri ja ajalise kasvu uurimiseks. Meetodit rakendatakse kolme laialt levinud tarkvarateegi ökosüsteemi peal, milleks on JavaScript, Ruby ja Rust. Tulemused näitavad, et uuritud ökosüsteemid erinevad. JavaScripti transitiivsete sõltuvuste hulk on viimase uuritud aasta jooksul üle 60% kasvanud, mis viitab kasvavale keerukusele. Töös demonstreeritakse, kuidas võrgustiku struktuuri analüüsi abil saab hinnata

tarkvaraprojektide riski hõlmata sõltuvusahela kaudu mõni turvaviga.

Doktoritöös arendatud meetodid ja tulemused aitavad avatud lähtekoodiga projektide vearaportite ja tehniliste sõltuvuste haldamise praktikat läbipaistvamaks muuta.

CURRICULUM VITAE

Personal data

Name:	Riivo Kikas
Date of Birth:	15.08.1987
Nationality:	Estonian
Language skills:	Estonian (native), English
E-mail:	riivokik@ut.ee

Education

2012–	University of Tartu, Ph.D. in Computer Science
2009–2011	University of Tartu, M.Sc. in Computer Science
2006–2009	University of Tartu, B.Sc. in Computer Science

Employment

2016–	Senior Software Engineer, Twilio Estonia OÜ
2009–2015	Junior Researcher, Software Technology and Applications Competence Centre (STACC)
2007–2009	Junior Developer, AS Nortal

ELULOOKIRJELDUS

Isikuandmed

Nimi:	Riivo Kikas
Sünniaeg:	15.08.1987
Kodakondsus:	Eesti
Keelteoskus:	eesti keel, inglise keel
E-post:	riivokik@ut.ee

Haridus

2012–	Tartu Ülikool, informaatika doktorant
2009–2011	Tartu Ülikool, M.Sc. informaatikas
2006–2009	Tartu Ülikool, B.Sc. informaatikas

Teenistuskäik

2016–	Tarkvaraarenduse insener, Twilio Estonia OÜ
2009–2015	Nooremteadur, Tarkvara Tehnoloogia Arenduskeskus OÜ
2007–2009	Nooremarendaja, AS Nortal

LIST OF ORIGINAL PUBLICATIONS

- Riivo Kikas, Marlon Dumas, and Dietmar Pfahl. Issue dynamics in GitHub projects. In Product-Focused Software Process Improvement, volume 9459 of Lecture Notes in Computer Science, pages 295–310. Springer International Publishing, 2015.
- Riivo Kikas, Marlon Dumas, and Dietmar Pfahl. Using dynamic and contextual features to predict issue lifetime in GitHub projects. In Proceedings of the 13th International Conference on Mining Software Repositories, MSR'16, pages 291–302, 2016. ACM.
- Riivo Kikas, Georgios Gousios, Marlon Dumas, and Dietmar Pfahl. Structure and evolution of package dependency networks. In Proceedings of the 14th International Conference on Mining Software Repositories, MSR '17, pages 102–112, 2017. IEEE Press.